Characterization and Parameterized Generation of Digital Circuits

BY

Michael D. Hutton

A thesis submitted in conformity with the requirements
for the Degree of Doctor of Philosophy in the
Graduate Department of Computer Science,
University of Toronto

Abstract

Characterization and Parameterized Generation of Digital Circuits

Michael D. Hutton
Ph.D. Thesis 1997
Department of Computer Science
University of Toronto

The development of new architectures for Field-Programmable Gate Arrays (FPGAs) and other forms of digital circuits, and the computer-aided design (CAD) software tools for these devices is greatly hampered by the lack of realistic test circuits or *benchmarks* that exercise them properly. Benchmarking is a crucial process in the design of CAD algorithms, as layout problems are typically NP-hard and heuristic algorithms are required.

This thesis investigates combinatorial structure in digital circuits. We define and analyze a series of graph-theoretic properties of combinational and sequential circuits, including a theoretical characterization of reconvergent fanout and metrics to capture the inherent locality found in hand-made or synthesized circuits, and propose a new model for describing sequential and hierarchical circuits. By measuring these characteristics on public and proprietary industrial circuits, we determine a realistic *profile* of circuits.

From our set of new characteristics, we define the new combinatorial problem of *parameterized random circuit generation*, advancing a new paradigm for benchmarking in computer-aided design. We then present a heuristic algorithm which solves it, fully implemented in a publicly available tool, GEN. Heuristic methods can only be judged on their actual results, and a key feature of the research is the empirical validation of the generated circuits. We compare standard post-layout metrics for the circuits produced by GEN with existing benchmark circuits and with random graphs, showing conclusively both that the generated circuits are very good proxies for real circuits and that random graphs are not.

Acknowledgments

Many people have contributed their advice and support to this effort.

First and foremost, I would like to thank my supervisors, Jonathan Rose and Derek Corneil. They have conveyed upon me not only their knowledge and technical skills, but have provided me with superior guidance and motivation. Good supervision is crucial to one's motivation and personal happiness in graduate school, and I have been blessed with having two outstanding supervisors, both technically and personally. I am pleased to count Derek and Jonathan not only as mentors and advisors but as my good friends.

Thanks to the other members of my committee, Steve Brown, Dave Lewis, Rudi Mathon, Mike Molloy, Ken Sevcik, and to my external examiner Andrew Kahng for their advice and direction. I also benefited from discussions with other students in Jonathan's group: Vaughn Betz, Steve Wilton and Mohammed Khalid. J. P. Grossman wrote some of the initial code for GEN as a summer student.

I received generous financial support from NSERC. A grant targeting this work was also received from the Hewlett Packard Corporation, and I gratefully acknowledge their help. Thanks to the Altera Corporation for providing me with a summer internship in 1996, where I was able to both continue with the work and to gain valuable insights into the FPGA industry. I am grateful to the Department of Computer Science for providing me with the opportunity to teach for the past several years, both for the financial benefits and because I simply enjoyed doing it.

Thanks to my parents, Barbara and David, and the rest of my family for their endless support and love. Thanks also to Donna MacIsaac, for her love, and for being there when I needed her.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In an ideal world, a field-programmable gate array (FPGA) vendor would use hundreds or thousands of benchmark circuits in determining the architecture of a next generation device, and in 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. Similarly, a commercial developer of general computer-aided design tools would require quality benchmark circuits to evaluate the effectiveness and efficiency of various new algorithmic techniques. The use of benchmarks is crucial for all facets of computer-aided design because the vast majority of interesting and practical problems are NP-hard and can only be solved by heuristic or approximate techniques. Similarly, FPGA design is inherently inexact, so architectural questions must also be answered empirically.

A fundamental problem exists for CAD and FPGA research: Because the device and its tools are new, there are few large (or correctly sized) designs available to perform these kinds of exploration and evaluation. In the case of FPGAs, some circuits will always exist by purchasing benchmarks from customers migrating from larger gate-arrays or synthesis from high-level design languages but these rarely suffice. Cutting-edge CAD vendors have no such luxury, and are forced to expend considerable effort creating benchmarks internally.

The proprietary nature of benchmarks, and the rarity of commonly accepted benchmark standards means that it is often very difficult to compare competing heuristic solutions for a given problem. In an effort to address this issue, researchers at the Microelectronics Centre of the University of North Carolina (MCNC) [74] have collected approximately 200 public

benchmarks and have made them freely available by anonymous ftp. These circuits are very popular for empirical validation in academic research, but largely spurned by industry as too small (about half are 100 nodes or fewer).

A related effort by the PREP Corporation [56] has defined a number of small representative benchmarks, with the goal of evaluating the logic capacity and speed of FPGAs. The metric is "how many" of the individual (but joined) circuits can be packed in a given device, and how fast the resulting compound circuit will run. Most researchers believe that this method does not address how logic characteristics change with size, especially with respect to interconnect usage, nor does it yield interesting test-cases for CAD software.

Random graphs are another possibility, particularly attractive because there is an infinite supply. Random graphs have often been used for the evaluation of partitioning algorithms for large circuits (where there are no available benchmarks). In particular, a number of classic partitioning papers [45, 46, 48] have done empirical validation with random graphs. One of the contributions of this thesis is to show that arbitrary random graphs are *not* realistic proxies for real circuits, and exhibit increasingly bad behaviour as the problem size increases.

A traditional graph-theoretic approach to NP-hard problems is to restrict the input domain, then identify an efficient deterministic algorithm for a subclass of graphs. For example, it is NP-hard to exactly determine the minimum number of colours $\chi(G)$ required to form a "proper colouring" of an arbitrary input graph G [29]. But, if G is known to be P_4-free—for any path $xyzw$ in G it is always the case that one of the edges xz, xw or yw also exists in G—it has been shown [14] that a straightforward greedy algorithm exists to determine $\chi(G)$ exactly in linear time.

One could claim that domain restriction is not directly applicable to practical CAD problems because a boolean network really is just an arbitrary graph: "for any G, an orientation of its edges and labeling of its nodes with primitive boolean functions (e.g. $\wedge$, $\vee$, $\neg$) provides a boolean network computing *some* function." However, our fundamental belief, as we will discuss further, is that such an arbitrary labelling of a general graph does not result in a *practical* or *realistic* boolean network as would be produced by a human designer or an automated synthesis tool. Without necessarily ruling out certain types of graphs as *possible* inputs to a software tool, we can perform data analysis to identify the *expected* structure of realistic inputs, and tune our tools to the distribution of expected

inputs.

It is a well known fact that relatively simple heuristics can often perform well—in practice the difficulty associated with random or arbitrary graphs does not occur, because real circuits exhibit much more structure than would be found randomly. For example, channel routing is known to be NP-hard [33, 50], but the search for more complicated algorithms or a guaranteed approximation scheme for the basic algorithmic problem is no longer "interesting" because existing heuristic algorithms work well and quickly [50] for all known data. This situation is analogous to the conclusions of Shew [63] who studied the application of graph colouring to scheduling with a conflict graph. He found that, even though arbitrary conflict graphs are always possible, real-life input *tends* to have P_4-free or nearly P_4-free structure: the heuristic algorithm was working well in practice because it was optimal for large subgraphs of the input it was given.

In the design and evaluation of good inexact architectures and heuristic algorithms it is crucial to understand the type of data that the FPGA or algorithm will be required to handle and thus to trust the test data that are used in its creation. The goals of this research are to provide a greater understanding of the graph-theoretic structure of real-life digital circuits and to apply this knowledge to the generation of high quality benchmark circuits.

In this thesis, we present a careful methodology for dealing with the benchmarking problem. We define a number of new graph-theoretic properties of combinational and sequential circuits. These properties are based on well known and important features of digital logic such as combinational delay, fanout, and reconvergent fanout. We also propose metrics that capture the inherent local structure of circuits not seen in random graphs. Given this better understanding of the combinatorial structure of circuits, we define the new problem of "parameterized circuit generation" and solve this problem by proposing and fully implementing a new algorithm. Since both of these efforts contain a large body of empirical and heuristic work, the final proof is in the resulting circuits themselves. We give conclusive evidence that the circuits we produce are realistic benchmarks by contrasting them both to existing benchmarks and to random graphs. As a byproduct of this *validation* step, we show the non-viability of purely random graphs as benchmarks.

The software tools CIRC and GEN arising from this work are freely available, and themselves form an important contribution to the community. CIRC is a tool for performing

analysis on an input circuit, and producing statistical and structural information about it. GEN takes a list of parameters (discussed in Chapters 3 and 4) and produces a circuit which satisfies the user's specification.

CIRC and GEN have been downloaded under an academic license by more than 30 persons representing more than 20 companies and academic institutions, and have been installed by the author for use at Xilinx, Altera, Actel, and Hewlett Packard Corporations.

1.1 Overview of the Thesis.

The research described in this thesis has three distinct aspects: *characterization* of digital circuits, *generation* of parameterized random benchmarks, and *validation* of circuit quality.

In Chapter 2, we provide further context and motivation for this work, and discuss previous work on circuit characterization and wireability, and circuit generation.

Chapters 3 and 4 address the characterization issue, asking the question "What is a circuit?" Chapter 3 deals with combinational circuits, introducing new characteristics of circuits based on combinational delay, and proposing a new theoretical characterization of reconvergent fanout and metrics for capturing the inherent local structure in combinational circuits. In Chapter 4, we investigate the more complex sequential circuit. We give an abstract model of a sequential circuit, defined in terms of combinational building blocks, and add a number of new characteristics specific to sequential circuits.

In Chapter 5 we formally define the parameterized circuit generation problem for combinational and sequential circuits, and give an algorithm to solve it. The algorithm has been fully implemented in the tool GEN, and we discuss a number of implementation details from this experience.

Chapter 6 deals with the final research topic, empirical validation. Using GEN to "clone" existing benchmarks from their parameterization, we can compare post-place and global route metrics of wireability between real circuits, their GEN-clones, and random graphs of the same size. We use this method to give strong empirical evidence both that our algorithm and tool provide good benchmarks, and that standard models and methods for random graphs do not.

We conclude and describe areas for future work in Chapter 7.

The historic development of the research differs from how it will be presented herein.

Chapter 2

Background and Previous Work

2.1 Terms and Definitions.

A graph $G = (V, E)$ has n nodes (vertices) and m edges, unless otherwise specified.

A *boolean network* G is a directed graph whose nodes, also called *gates*, are labeled as primitive boolean functions: typically $\wedge$ (and), $\vee$ (or) and $\neg$ (not). Edges are also referred to as *wires*. A boolean network is *combinational* if it is acyclic. A *sequential* network is traditionally defined as a circuit with memory. We will assume that memory is implemented by atomic *flip-flop* nodes in the representation of the circuit as a graph, rather than built from gates. All sequential circuits discussed in this thesis will be single-clock synchronous networks, unless stated otherwise, which means that all directed cycles must be "broken" by one or more flip-flops. We will ignore the issue of pipelining, whereby flip-flops are added for timing reasons but logically function as buffers, so we assume that all sequential circuits have back edges[1]. When referring to the graphical representation of a practical boolean network, we will use the term *circuit graph* or *circuit*. The term *graph* will refer to an arbitrary graph which may or may not arise from a boolean network. A *random graph* is one drawn from some natural distribution by a stochastic process. For example, a random graph $G(n, p)$ is a graph on n nodes such that each potential edge exists with independent probability p.

In a circuit graph G, nodes with no incoming edges are called *primary inputs* and nodes with no outgoing edges *primary outputs*. The *fanin* (*fanout*) of a node is the number of

[1] A back edge is a "feedback" edge which goes from one sequential level to a previous level. Sequential levels are formally defined later.

incoming (outgoing) edges. The *depth* of a circuit is the longest input to output directed path. In a combinational circuit, this distance is the *unit combinational delay*, delay(G), of the circuit. The length of a shortest directed path from an input to a particular node x defines the unit combinational delay for x, delay(x). In a sequential circuit, the combinational delay of a node is the length of the shortest directed path from either an input *or* a flip-flop, and the combinational delay of the circuit is the maximum combinational delay over all nodes.

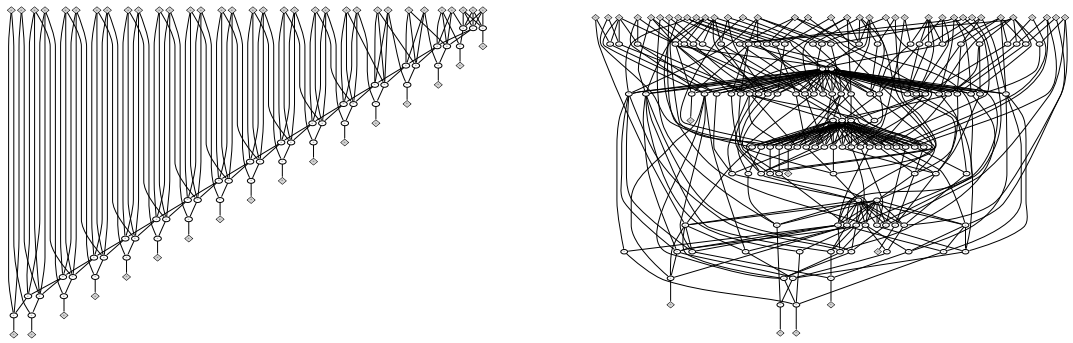
A circuit is often modeled as a *hypergraph*, $H = (V_H, E_H)$, particularly for the partitioning problem. $V_H = V_G$ and each node and its set of fanouts collectively form a hyperedge in E_H, usually called a *net*. Electrically, this is the more correct model of a circuit, but most problems are more easily defined in terms of graphs.

The *recursive fan-in* (*fan-out*) of a node, also called a *cone*, is the set of all preceding (following) nodes in the partial order underlying G (undefined for sequential networks). When two disjoint directed uv paths exist in G, we say that G is a *reconvergent* network and that G is "reconvergent at v." In a non-reconvergent combinational network every fanout-cone is a tree. The increasing presence of reconvergence is known to introduce difficulty into many CAD problems, as the input graphs become less and less "tree-like."

Circuits are often classified into two distinct types. *Datapath* circuits are repetitive, simple sequences where each node is often connected only to immediate physical neighbours. Arithmetic functions such as an adder or multiplier are typical datapath circuits. *Random logic* or *control* circuits are loosely defined as everything else. They typically lack the regularity of datapath circuits. Since the structure of a datapath circuit is usually well-known to the designer, and the type of functions computed are typically more generic (rather then application specific), they are often treated as special cases for layout. It is relatively easy to synthesize a datapath using commercial CAD tools, so we will be primarily interested in circuits in the random-logic category. Figure 2.1 shows examples of datapath and random logic, taken from the MCNC benchmark collection.

We will occasionally refer to qualitative size of circuits (small, medium, large). Current generation FPGAs have one to four thousand 4-input lookup tables (or LUTs)[2], and next

[2]A k-input lookup table is a logic element which can be programmed to implement *any* single-valued boolean function on k inputs. Though an FPGA could have a more restrictive type of logic block, or have different logic functions available throughout the architecture, the industry standard is to use the k-input LUT uniformly across the chip.



Datapath logic
MCNC circuit **my_adder**

Random logic
MCNC circuit **C432**

Figure 2.1: Datapath vs. random logic.

generation devices will have double that. So we will use "small" to refer to circuits with 500 LUTs or fewer, "medium" for 500 to 5,000 LUTs, "large" for up to 10,000 LUTs, and very large for beyond 10,000 LUTs. Gate-array technology typically quantifies logic in terms of standard 2-input gates. Industry typically translates one LUT/flip-flop pair as comprising about 12 such gates. State of the art gate arrays are currently in the 1,000,000 gate range, or about ten times the capacity of current FPGAs.

2.1.1 Computer-Aided Design for Digital Circuits.

It is important to have a common view of what is implied by a particular computer-aided design software problem. We give enough detail here to be self-contained, and refer the interested reader to Lengauer's comprehensive book [50] for more detail.

Technology-independent optimization refers to the manipulation of a network to achieve some common basic requirements for all technologies (such as constraining fan-in/out [32, 38]) or to effect a result deemed to be of value for any destination technology; e.g. isolating and merging common boolean expressions to reduce the size of the network.

Partitioning refers to separating the nodes of a graph into two or more disjoint sets or *modules* to minimize some graph-theoretic measure, usually the number of edges or hyperedges which cross the partition boundaries, subject to such constraints as the minimum or maximum module size. An equivalent notion to the number of inter-module edges is the number of vertices in each module that have external connections, often referred to as the number of logical *pins* in the module. Standard formulations of the problem are NP-hard. Various heuristic algorithms exist. One popular approach is the Kernighan-Lin-Fiduccia-

Mattheyses (KLFM) [46, 26] algorithm, which performs incremental improvements (swaps) from an initial solution until some predetermined tolerance is reached. In practice, such algorithms can perform reasonably well, despite theoretical proofs of pathological non-optimality.

Though primitive boolean functions ($\wedge$, $\vee$, $\neg$) are the basic blocks of the abstract description of a circuit, hardware implementation typically draws from a larger library of available basic functions, often determined by physical design concerns. *Technology mapping* is the process of converting from a circuit whose nodes are basic blocks of one (e.g. the generic) type into one whose basic blocks are of another (the technology specific) type. For field-programmable gate arrays the basic block is usually a k-input lookup table (LUT), in which case the problem of finding a *size* or *depth* optimal mapping is somewhat different from the subgraph matching problem of typical library based mapping. Existing software to compute such a mapping includes flowmap [13], chortle [27], rmap [60], xnfmap [73] and mis/sis-pga [54]. For a general reference, see the textbook by Brown *et. al.* [11].

Placement is the embedding of a graph G into the physical, geometric, world. This is often abstracted as a mapping of the nodes of G into the nodes of the $N \times N$ grid-graph $G_{N,N}$ (the "host") to minimize some approximation of channel width (see below) or a total wirelength metric (e.g. sum of Manhattan distances between adjacent nodes in G). In this thesis we will use both this model, which closely resembles a number of Xilinx FPGAs, and hierarchical variations such as occur in the Altera 10K programmable device. Once placement has taken place, we can define the *length* of a particular edge and the total *wire-length* R , *average wire-length* $\bar{R}$, and distribution of wire lengths $\mathcal{R} = \{R_l\}$, with respect to the placement. *Wireability*, which refers to the types and distributions of wire-lengths which can be supported by a given host (e.g. $G_{N,N}$) independent of any particular circuit, will be discussed in more detail in Section 2.2.2[3]. By the term *routability* we refer to the "ease" of successfully placing and routing a specific network G into a host, using these and other related metrics.

Given the placement, a *global routing* is an assignment of the edges of G to paths in $G_{N,N}$. Then we have the notion of *channel width*, W, defined as the maximum over all

[3]Note that the term "wireability" refers mostly to the process of determining statistical relationships on the connectivity and distribution of wires once circuits are already placed on a grid-like architecture. It is not usually used in the sense of a quality judgement on a network. In general, we don't use the terms wireable and unwireable in that sense, rather we use routable and unroutable.

edges in $G_{N,N}$ of the number of paths using that grid edge. The *optimal channel width* over all placements is denoted W^*. A *detailed routing* assigns the paths of the global routing to realizable electrical connections with respect to the technology. In the case of mask-programmable technology this means physical wires which can interact (cross) other wires in only specific ways. Field-programmable gate arrays have preexisting wires laid out in *tracks* in each channel, each track is broken up into *segments* (actual wires) connected by *programmable connections* with which to select a given path within a track or between tracks. A detailed routing is then a refinement of the global routing which specifies the settings of the programmable switches to code a physical path in the segments of the coarse routing (channels only). It is possible to consider global and detailed routing together as a single problem. For some FPGA architectures the concept of detailed routing makes less sense, and this approach is taken.

Since known deterministic algorithms for NP-hard problems are considered infeasible, existing practical algorithmic solutions often have no provable performance (correctness or quality). For evaluation of competing techniques the community uses various "standard" benchmark suites. By running a new algorithm on the benchmark circuits a quantitative measure (run-time, channel width, percent of routable connections, speed of the circuit) can be obtained for comparison with existing algorithms. The currently accepted standard in academia is to use the the MCNC benchmarks [74]. We will occasionally refer to and take examples from this collection of circuits; two such examples have already been shown in Figure 2.1. Industry would typically use proprietary benchmark sets, and would not announce results of their experiments.

Some terms with respect to implementation technologies: *full-custom* VLSI refers to the layout of a design (transistors and wires) on a totally empty and unconstrained space. *Standard-cell* refers to a technology where the basic blocks come from a library of predetermined logic elements which can be placed in rows at the specification of the designer. Detailed routing then reduces to channel routing (with feed-through cells) in the horizontal channels between the rows. A *gate array* technology constrains the logic elements to lie on a rectangular grid with both horizontal and vertical routing channels. *Mask programmable* gate array (MPGA) technology then allows the wires to be freely placed on a separate fabrication layer at manufacturing time.

2.1.2 Field-Programmable Gate Arrays.

The design of an application specific integrated circuit (ASIC) using either gate array or standard cell technology requires that the wiring is added as one step in the fabrication process. A recent technological alternative to this type of ASIC is the field-programmable gate array, which has both programmable logic elements and a programmable routing network to connect the logic. FPGAs can be programmed using just a personal computer and simple hardware interface, giving them flexibility and time-to-market advantages over traditional ASICs, which must have all wiring completed in a fabrication plant. However, programmability typically incurs a factor of ten in decreased chip density and a factor of three in decreased speed for the resulting hardware. This tradeoff is increasingly more acceptable to designers, and the FPGA industry has grown from an insignificant portion of the ASIC business in 1984 to a 1.4 billion US dollar industry today.

The advent of FPGAs spawns a host of new problems for CAD designers. Because FPGAs have a fixed routing network instead of "open real estate" the layout problem becomes more graph theoretic than geometric in nature. For rapid prototyping, it is common to implement a single design on multiple FPGAs or even boards of FPGAs, creating new variations on the partitioning problem which do not arise in higher capacity, more finely grained, ASICs. While the routing problem for gate arrays is one of minimizing channel width, CAD software for FPGAs deals with a binary fit/no-fit problem. Because of the programming logic, FPGAs also produce new challenges for timing estimation.

In addition to these new software problems, there is the issue of the FPGA *architecture*. Numerous choices exist in the design of an FPGA: Do I organize the logic and routing architecture hierarchically, or in a flat grid? How big should logic elements be? How many tracks should be placed in each row/column and how should they be connected together? Should the programming be permanent, or stored in a way which is reconfigurable? All of these issues must be addressed in the context of device cost, routability, timing, power consumption, noise, and the ability to write efficient CAD software. The architectural design process is inherently approximate, so many of these questions can only be answered empirically with benchmarks.

It is by no means clear which architectural choices are correct, or even if there are correct choices. The Actel Corporation manufactures FPGAs using a standard-cell like architecture,

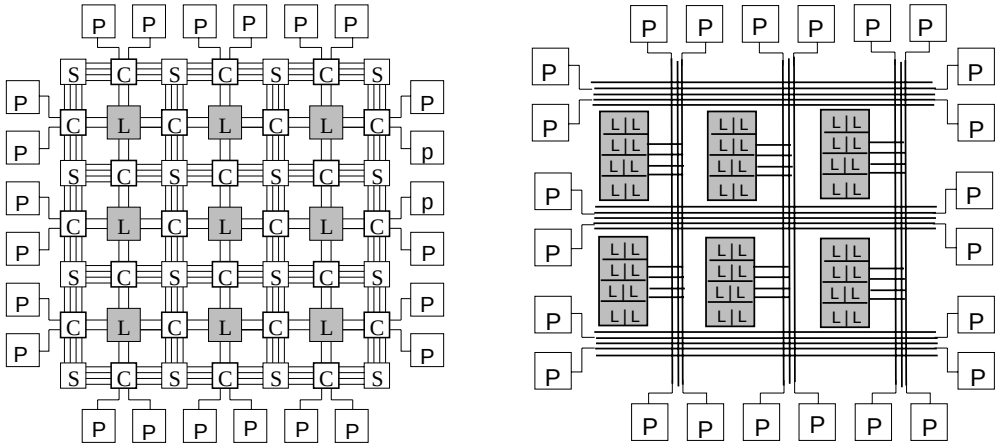


Figure 2.2: Different FPGA architectures.

and uses anti-fuse technology for permanent programmability (the only major vendor to do so). Altera's 10K series of devices organize logic elements into a shallow hierarchy: cliques of fully connected logic and a more sparse interconnection structure between cliques. Xilinx uses a "flat" architecture reminiscent of a gate array, with a routing architecture consisting of multi-track channels with "switch" (S) block modules at the intersection of channels, and "connection" (C) block modules where logic-block pins enter the routing network (P and L stand for pin and logic block, respectively). Abstract representations of Xilinx and Altera architectures are shown in Figure 2.2. Both Altera and Xilinx use SRAM bits to program the parts, which means the logic can be re-programmed repeatedly, in some cases during the computation itself though this is not commonly done.

The research described in this thesis applies both to the ASIC and the FPGA world, but it is of particular interest for FPGAs. As mentioned previously, hardware and software architects of a "new" 1,000 LUT FPGA have to deal with the discrete fit/no-fit issue rather than more fine-grained optimization problems. Typically this means a large number of circuits in the 900-950 LUT range would be required to exercise the device, while neither a 400 LUT circuit nor a 1,200 LUT circuit would be an interesting test case. The circuits must also be representative enough to deal with the vastly different types of circuits that a user might wish to implement. Thus FPGA vendors consider their benchmark suites to be closely guarded proprietary information, and universally feel that there are "never enough benchmarks."

2.1.3 Graph Classifications.

A *class* of graphs is a set of graphs which are related in some way. A class can be defined by some specific graph-theoretic property, for example "A graph is *regular* if all vertices have the same degree." The members of the class can sometimes be defined by recursive construction: "A single vertex is a *tree*, a tree T with a new vertex x and an edge from x to some vertex v of T is also a tree." The class can be determined by virtue of what it does not contain, for example a forbidden configuration or subgraph: "A *tree* is a connected graph with no cycles."; "A *planar* graph is a graph which contains no subgraph "homeomorphic" to the graphs K_5 or $K_{3,3}$." Any other well defined mathematical definition would also be appropriate. Note that often a graph class can be defined in multiple equivalent ways. For example a planar graph is commonly defined geometrically as "a graph for which there exists a embedding which maps vertices to points in the plane and edges to Jordan curves connecting their respective endpoints that do not intersect except at those endpoints."

If $G = (V_G, E_G)$ is a graph then any graph $H = (V_H, E_H)$ where $V_H \subseteq V_G$ and $E_H \subseteq E_G$ is a *subgraph* of G. If xy is an edge of G, and xy is in H whenever both x and y are in H, then we say that H is an *induced* subgraph of G, otherwise it is a *partial* subgraph. It is often interesting when the definition of a class is closed under the taking of subgraphs; that is, the definition of the class is *hereditary*. Planarity is hereditary, because any subgraph of a planar graph is clearly planar.

2.2 Previous Work.

2.2.1 Rent's Rule.

The commonly accepted relationship called "Rent's rule" dates back [49] to E. F. Rent of IBM, who made an empirical observation regarding the partitioning problem:

Rent's rule: Let G be a circuit with n blocks (nodes) and m wires (edges). Consider a "reasonable" partition of the blocks of G into modules $M_1, M_2, \cdots, M_l$ where the modules each satisfy a pin constraint: the number of external vertices in any M_i is constrained to some value P^*, and the number of modules is no less than five. Then the empirical relationship

$$P = kB^r \tag{2.1}$$

is found to hold in general, where

k = the average number of edges incident on a block,

P = the average number of pins (external vertices) in a module,

B = the average number of blocks in a module, and

r = the "Rent exponent", empirically $0.5 \leq r \leq 0.8$.

Satisfyingly enough, for the trivial "total" partition of G into modules of one block each Rent's rule with $B = 1$ correctly gives the average number of pins as the average degree over the blocks in the network. This does not hold empirically for partitions into only a few modules, as stated in the definition and discussed later.

The algorithm for separating the circuit into modules is undefined in the standard formulation of Rent's rule. Later refinements by Feuer [25] specify that placement provides such a "good" partition into modules in terms of geometric proximity in the sense that from any circle (closed set of grid points of Manhattan distance r from a fixed centre point) the number of external connections will follow Rent's rule on average. So we can think of Rent's rule as both a law that holds for a given partition, on average, and at the same time as the *expected* relationship for a specific module in terms of its terminal and non-terminal vertices.

It is crucial to note how closely the notion of Rent's rule is tied to that of a good empirical modularization. For example, it is possible to self-embed $G_{N,N}$ (equivalently, give a partition) badly so that every wire is of length $\frac{N}{2}$, yielding a channel width of $O(N)$ and Rent exponent $r = 1$, even though the trivial embedding has $W = 1$ and $r = 0.5$. Thus any discussion of Rent's rule holding in an abstract sense must capture somehow the *existence* of some modularization, either in a non-constructive sense or by exhibiting the modularization directly. Hagen *et. al.* [35] investigated this in detail, and defined the *intrinsic Rent parameter* of a circuit as the minimum possible Rent parameter over the set of all partitioning algorithms. They gave empirical evidence to show that different algorithms do yield different values for the Rent parameter. We also stress that Rent's rule applies to modules *on average* and does not address maximum or minimum behaviour for a particular module.

Empirical calculations.

Landman and Russo [49] discuss the historical origins of Rent's rule. They calculate $P = 4.17B^{0.65}$ for Rent's initial data and cite various other independent confirmations: A study by Meade and Geller [52] yields $P = 4B^{0.7}$. Notz *et al.* [55] find $P = kB^{\frac{2}{3}}$ where k is one plus the average fan-in of the network. Radke [57] also mentions the "common knowledge" of the rule (attributing it to Rent), noting observations of p varying from 0.5 to 0.7 with values of k between 3 and 5.

The typical method for calculating r is to perform an empirical partition, sample modules for values (P_i, B_i), and perform a linear regression on $\log P_i = \log k' + r \log B_i$ $(P_i = c^{k'} B_i^r)$, usually constraining $k = c^{k'}$ as a constant. Landman and Russo specifically point out that Rent's Rule is unstable when the number of modules is less than 5. One reason that this would be true is that the number of pins on a chip is usually a hard constraint in practice, and the engineer must build the design within the given number of I/Os. Hierarchy inside the chip does not suffer from these hard constraints, and should exhibit more consistent behaviour.

Russo [58] notes that more parallel "high performance" circuits tend to exhibit larger r, because they tend to have a higher pins-per-gate ratio, hence Rent exponent.

Theoretical Issues.

In an attempt to understand the determinants underlying Rent's rule, and also to investigate the tradeoff between logic (control) and memory, Donath [17] developed a model of the process of designing computer hardware. He models the modular decomposition process of hardware design, and argues that Rent's Rule is a natural consequence of a structured design methodology. Donath also investigated the "information content" involved in trading memory bits for logic (i.e. implementing logic functions as lookup-tables in ROM), and derived a rough rule of thumb which states that one basic logic element (gate) is equivalent to 8.5 bits of memory[4].

Landman and Russo cite an old unpublished manuscript of Donath [20] in which he proves that a random graph G, defined as "a graph with edges distributed randomly among

[4]This suggests that a 2K ROM, used as a lookup table, would be expected to implement a boolean function comprising about 1900 2-input NAND gates (on average). Similarly, a 2K truth table could be expected (on average) to optimize into about 1900 gates of combinational logic.

its vertices," exhibits a linear relationship between P and B (i.e. $r = 1$). The statement is difficult to interpret without a concrete random graph model, but the basic property will also be visible in the theoretical wirelength studies of the next section.

Discussion.

Rent's Rule works well as a predictor of I/O to logic ratios for internal connections to a chip. Most researchers would use a safe overestimation of r to predict the number of pins required for a chip, or to generate a theoretical "envelope" on the number of tracks or wirelength but, in practice, would combine this with empirical analysis.

For our purposes, Rent's Rule is not a good "characterization" of circuits, because of its reliance on an existing parameterization and on its reference to the average-case behaviour of the partition hierarchy. However, Rent's Rule is a well accepted guideline in the community, and important to keep in mind as a general rule of thumb about circuits.

2.2.2 Stochastic Wireability Models.

Routability refers to estimating the wirelength or fittability of a circuit on a given host graph or architecture. Early research on gate-arrays gave us a number of statistical properties and distributions which can be used to predict routability for circuits.

Wire length distributions.

Using random placements [16, 36, 59], or assumptions about stochastic properties of placement [24, 61] and Rent's rule [18, 19, 25] various theoretical models of wire-length have been proposed.

Donath [16] studied the statistical properties of randomly placing a random graph on a grid. He developed a lower bound on the average wire-length $\bar{R}$, over *all* placements, for an embedding of a given G into $G_{N,N}$. He showed that this lower bound is dependent only on n and m (the number of edges), and is independent of the structure of the graph:

$$\bar{R} = \frac{m n^{\frac{1}{2}-\frac{n}{2m}}}{e^{1-\frac{n}{2m}}} \tag{2.2}$$

$$= \frac{m n^{\frac{1}{2}-\frac{1}{2k}}}{e^{1-\frac{1}{2k}}} \qquad \text{(average vertex degree } k\text{)}. \tag{2.3}$$

The bound provides some information, since we expect that any reasonable algorithm does better than the expected random placement. However, the bound implies an $\bar{R}$ of approximately $\frac{N}{3}$ and $W = O(N)$ [18], so the bounds are too loose to have any practical utility: studies have shown both $\bar{R}$ and W^* to be roughly proportional to $\log N$ in practice. Note that this result also implies the unit Rent exponent for the random graphs in Donath's construction.

Donath [18] later developed a formula for the upper bound on expected average wire length $\bar{R}$ based on a "pseudo-random" placement. The placement is partly stochastic, but attempts to "reflect both the characteristics of logic complexes as they are designed by engineers and the effect of the placement procedure." By assuming that Rent's rule holds recursively, he developed a new upper bound for the expected average wire-length.

$$\begin{aligned} \bar{R} &\sim B^{r-\frac{1}{2}}, && r > \tfrac{1}{2} \\ \bar{R} &\sim \log B, && r = \tfrac{1}{2} \\ \bar{R} &\sim f(r), && r < \tfrac{1}{2} \qquad \text{(independent of } B.) \end{aligned} \tag{2.4}$$

An important note is that the estimator under the Rent assumption differs from that of a purely random placement, which yields $\bar{R} \sim \sqrt{B}/3$ as mentioned earlier. Donath compares his upper bound to experiments on five real circuits and finds that the estimate is about double the average wire length found in practice. The dependence on both B and p is supported by the experiments.

Feuer [25] does a similar analysis to develop wire length estimators from Rent's rule, and also calculates the distribution of wire lengths. He derives, from Rent's rule and several simple geometric assumptions about the placement, an expression

$$R_l = c(r, B) l^{2r-4} \tag{2.5}$$

for the expected number of connections between any two grid points of Manhattan distance l apart in a placed circuit. The parameters r and B are from Rent's rule, and c is a constant function of these parameters only, hence constant for a given graph.

This distribution leads to expressions for the average wire length of connections *internal* and *external* to a region of radius d:

$$\bar{R}^i = \sqrt{2}\frac{\alpha(5-\alpha)}{(3-\alpha)(4-\alpha)}\frac{B^{r-0.5}}{(1-B^{r-1})}, \tag{2.6}$$

and

$$\bar{R}^e = \sqrt{2}\frac{(1-\alpha)(5-\alpha)}{(3-\alpha)(4-\alpha)}B^{r-0.5} \tag{2.7}$$

where $\alpha = 2 - 2r$.

The overall average wire length predicted by the model is

$$\bar{R} = \sqrt{2}\frac{(2-\alpha)(5-\alpha)}{(3-\alpha)(4-\alpha)}\frac{B^{r-0.5}}{(1+B^{r-1})}. \tag{2.8}$$

Since $(1+B^{r-1})$ vanishes for large B, the latter is proportional to $B^{r-\frac{1}{2}}$; exactly as derived by Donath for $r > \frac{1}{2}$ (Equation 2.4).

Feuer's analysis yields justification that geometric proximity after placement is itself a "good modularization" for application of Rent's rule, as mentioned earlier, because the derivation from the proximity assumptions generates Rent's rule, which is then itself assumed for the derivation of the wire-length estimators.

El Gamal and Syed [24] define a purely stochastic model in which wire lengths are distributed Poisson(λ) and wire trajectories are parameterized by $\gamma, \alpha, \beta, p, u$. They develop a formula for average wire length in terms of these parameters, and estimate the parameters using empirical data. As an application of their model they vary the parameter u, the percentage of utilized gates, holding other parameters fixed and find that "it is better to use an array of size $\frac{n}{.8}$ with more tracks (channel width) than a larger array of size $\frac{n}{.5}$ with fewer tracks." It is stated that 100% utilization ($u = 1$) is unrealistic, and implied that the model bears this out as well.

Sastry and Parker [59] show that "any placement which satisfies Rent's rule, or any similar pin-to-block relationship," will have a wire-length distribution which is Weibull:

$$R_l = \alpha\beta l^{\beta-1}e^{-\alpha l^\beta} \tag{2.9}$$

with mean

$$\bar{R} = \frac{1}{\beta}\left(\frac{1}{\alpha}\right)^{\frac{1}{\beta}}\Gamma\left(\frac{1}{\beta}\right). \tag{2.10}$$

The parameters α and β are calculated from the empirical data by log-linear regression on empirical data. Though there is definitely a relationship between r and these parameters, the authors do not develop a closed formula, and instead rely on the regression to give empirical values.

Channel Width.

Wire length alone does not capture the effect of difficult areas or "hot-spots" with high channel width. What we often would like is to have a prediction of the greatest channel width in the array. This, of course, would have to be less than the available channel width if routing is to take place.

El Gamal [23] gives such a model. He calculates W in terms of $\bar{R}$ assuming that the distribution of lengths is geometric and adding the additional assumptions of trajectory along a minimum (Manhattan) distance path in the array: each lattice point emits X_i wires of length L_{ij}—given an initial trajectory (up-right, up-left, down-right, down-left) flip L_{ij} coins and move up or down on heads and left or right on tails, as appropriate.

The conclusions to be drawn vary with $\bar{R}$. If $\bar{R}$ is finite, then the distribution of channel densities is Poisson:

$$W_t = \mathcal{P}\left(\frac{\lambda \bar{R}}{2}; t\right) \tag{2.11}$$

where W_t is the number of channel segments with width t. The expected maximum channel density converges to $O(\ln N)$ (almost always) when $\bar{R} \leq O(\ln N)$ and $O(\bar{R})$ (almost always) otherwise. Since the former seems the most reasonable occurrence the primary conclusion is that channel densities are distributed Poisson with a mean channel density of $\frac{\lambda \bar{R}}{2}$. Brown *et. al.* [10, 11] find the accord between this prediction and several actual circuits to be very good. They note, however, that the model becomes less accurate if the FPGA model is expanded to give segments of more than unit length.

Applying Routability.

Chan, Schlag and Zien [12] recently combined several of the results just discussed to predict routability for a Xilinx 3000 series FPGA. Circuits are classified as "unroutable", "marginally routable" and "easily routable" based on the Feuer's expectation of channel width for the circuit vs. the available channel width, an estimator for the Rent parameter

r using mincut partitioning and El Gamal's estimator for W.

Another (earlier) model for routability was given by Brown [10]. Here, routing is a stochastic process with parameters specifying the network for the FPGA (e.g. the number of connections in the switch and connection blocks, the channel width), several model-parameters (event probabilities) and basic properties of the circuit to be routed (size, connections and expected wire length $\bar{R}$ (from El Gamal)). An expression for the expected percentage of unrouted connections is generated. The model has been used both as an indicator of routability and as a vehicle for determining good settings for the parameters which specify the FPGA; e.g. to determine how much flexibility (how many switches) to put in a Xilinx C-block or S-block.

Discussion.

The stochastic results cited in this section are the traditional approaches to characterizing circuits and determining theoretical bounds for architectural parameters. The goals of this thesis are quite different, in that we want to determine graph-theoretic characteristics taken from analyzing the circuit *graph* itself. The purpose of including this previous work is more to provide context for the current research, and because the terms introduced here are used elsewhere in the thesis.

2.2.3 Other Generation Efforts.

Random Graphs.

In this thesis, the term a *random graph* will refer to graphs generated by stochastic methods which do not take into account the properties of digital circuits. Such random graphs are drawn uniformly from the set of all graphs, or uniformly from a partially restricted set of all graphs, such as "all regular graphs." The most common such model is the random undirected graph $G(n, p)$, defined as a graph on G nodes in which each potential edge exists with uniform independent probability p. These graphs can be easily generated, but are not realistic as circuits: for $p < n \log n$ they are disconnected and otherwise they contain $O(n \log n)$ edges and have most nodes with degree $\log n$, *almost always*. The former makes the graph uninteresting, and the latter makes it electrically infeasible as a circuit.

There are also well-known methods for generating random degree-constrained graphs

uniformly. We use one such method (with modifications) for comparison to our circuits in Chapter 6. However, there are no known methods to uniformly generate directed I/O constrained graphs, or to generate directed graphs with restricted path-lengths, or which properly satisfy the electrical constraints of a synchronous sequential circuit.

There is a long history of using random undirected graphs as benchmarks. Kernighan and Lin [46], Johnson *et. al.* [45], Krishnamurthy [48] and Hagen and Kahng [34] (for others see [50]) used random graphs to compare and evaluate partitioning algorithms. Vargese *et. al.* [68] and Hauk *et. al.* [37] also used random graphs to study architectural parameters and algorithmic issues for logic emulation systems with FPGAs, which require very large circuits. Random graphs are currently unavoidable for experimentation beyond the size of existing circuits.

Generating Circuits by Transformation.

Iwama *et. al.* [43], in independent work, discuss how to apply transformation rules to a initial seed circuit and create a different structural circuit with the same logic function. This work applies only to combinational circuits, and is limited to generating variations on the initial circuit. In a paper to appear later this year [44], they will discuss an improvement on the work which generates seed circuits from random truth-tables, rather than requiring an input circuit.

This work is primarily aimed at benchmarking for logic synthesis (logic independent optimization) algorithms. The authors do not describe any applications of the approach to dealing with physical design algorithms or architectural issues.

Generating Circuits with Rent's Rule.

In independent work, Darnauer and Dai [15] have recently given an algorithm for generating random undirected graphs to meet a given Rent parameter. The basic idea is to generate a random partition hierarchy, and recursively generate a graph from it. The approach has an obvious attraction for partitioning, which was its primary application. Darnauer and Dai showed the empirical validity of their algorithm for relatively small combinational circuits on partitioning problems.

The primary drawbacks of the method are that the tool loses control over combinational delay and does not have the ability to generate sequential circuits with the properties which

we concentrate on in this thesis and which are important for FPGA architectures and all other physical-design CAD problems. We believe it would be possible to incorporate the most important aspects of the Rent-based method into the high-level hierarchy of our sequential generation; one area for further work would be to investigate combining our approach for generating medium-size circuits with a high-level partition hierarchy. We will remark further on this in Chapter 7.

2.3 "Obvious" Properties of Circuit Graphs.

Based on common knowledge of digital circuits, we can make a number of preliminary observations about their combinatorial structure.

One obvious property is that the class of circuit-graphs is hereditary: if G is a circuit then any induced subgraph of G will also be a circuit. Because electrical fanin and fanout are constrained in all but special circumstances, we observe that the number of edges should be linear in the number of nodes. For convenience, we will assume that any complete circuit is connected, since a CAD algorithm could easily check connectivity and significantly decrease the problem complexity on disconnected graphs.

From Rent's rule, we can expect that circuits exhibit some type of "hierarchical" structure. However, this is an abstract notion only, since Rent's rule and the wireability studies mentioned earlier do not give any applicable graph-theoretic restrictions which we can use directly.

Also from empirical studies of Rent's rule, we note that the number of inputs and outputs in a circuit is sub-linear in the number of nodes (unless $r = 1$ for the circuit, which is not seen empirically). For a chip with a reasonable aspect ratio and packaging constraints this follows independently of Rent's rule, since the number of I/Os can only be a small constant multiple of the perimeter.

Chapter 3

Characterization of Combinational Circuits

This chapter describes the statistical and structural characteristics that we have identified for combinational circuits.

Parts of this work are directly motivated by the generation problem. In order to generate benchmark circuits, we will need a *default* parameterization file, so we want to develop a statistical profile for relationships between parameters. For example, if the user asks simply for a circuit with 1000 nodes, we will need to choose a reasonable number of primary inputs and outputs, and a reasonable value for combinational delay. The complete set of default equations is in the file "comb.gen" shown in Appendix A of this dissertation.

Characterizations which describe the combinatorial structure of circuits, however, are of interest in their own right, and we propose a number of them here. Combinational shape, reconvergence and locality are all structural characterizations that are introduced in this thesis, and deal with the inherent structure in circuits which separates them from arbitrary graphs. In addition to becoming data for the circuit profile, the structural ideas will form the basis for the generation algorithm of Chapter 5.

For the empirical work here, we use the MCNC circuits. However, it is important to point out that the tool CIRC that we have produced to extract the characterization of a circuit is independent of the data; the user could use it on any collection of benchmark circuits, then redefine the default profile accordingly. CIRC is implemented to read circuit netlists in the Berkeley BLIF format [74], and output numerous statistical and structural characteristics.

As well, CIRC is able to do netlist translation, and output circuits in a number of other netlist formats (including Actel ADL [1], Altera AHDL/TDF [4] and Xilinx XNF [73]).

3.1 Empirical Data.

A large portion of the work in Chapters 3 and 4 is empirical, and for this we use the MCNC benchmark circuits. The use of the MCNC circuits is largely unavoidable, since they are the only large set of public benchmarks. We note that a user of the tools could profile their own (internal) circuits as the basis for an alternative defaults file. (See Appendix A.)

The MCNC benchmark circuits are a well-known set of combinational and sequential benchmarks available from http://www.cbl.ncsu.edu/. The circuits were converted from EDIF[1] to BLIF[2] using a modified conversion tool from MCNC. We did technology-independent optimization with SIS [62] (keeping the better result of script.rugged and script.algebraic) then technology mapped using FLOWMAP [13] into k-input lookup tables, for $k = 2..8$. Specifically, each circuit was mapped 7 times, into 2-input LUTs, 3-input LUTs up to 8-input LUTs. We chose to use lookup-tables because of their simplicity, functional completeness and the ease of changing to different LUT-sizes. We believe that the structural properties of circuits are sufficiently captured by the use of LUTs to determine valid characterizations without the added complexity of more technology-dependent libraries.

One issue that we do not fully explore in this work is the effect of this early optimization (CAD flow) on the exact statistical characterization which follows. For example, FLOWMAP is a delay based technology mapper, and it is not clear whether a different mapper would have changed some of our statistical results. Similarly, due simply to the volume of data, we spend most of our analysis on 4-LUT mapped MCNC circuits, largely because this is the most popular choice in the FPGA industry.

3.2 Basic Parameters of Combinational Circuits.

The characteristics in this first section are more for statistical purposes than to provide any new structural information about circuits.

[1] EDIF is a "standard" netlist format used in industry.
[2] BLIF is a format used by the Berkeley SIS tool, and commonly used in academia.

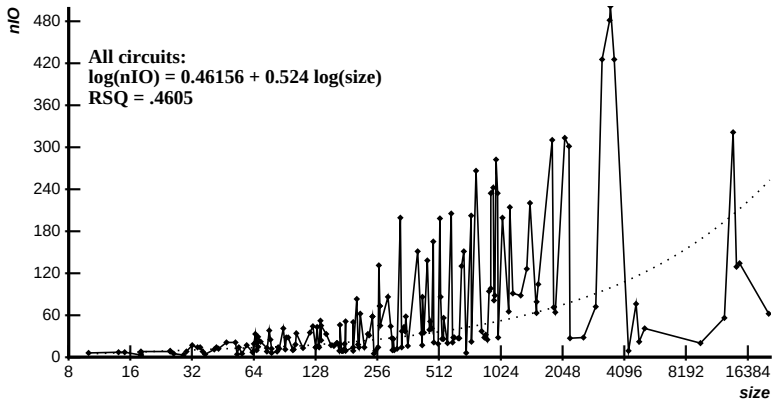


Figure 3.1: Size (2-LUTs) vs. I/O for MCNC circuits.

3.2.1 Circuit Size and I/O.

The most basic characteristic of a circuit is the relationship between the size of the circuit (number of LUTs n) and the number of primary inputs (n_{PI}) and outputs (n_{PO}). (Define $n_{\mathrm{IO}} = n_{\mathrm{PI}} + n_{\mathrm{PO}}$.) Using linear regression and experimentation, we have determined that a Rent-like functional relationship, $\log(n_{\mathrm{IO}}) = a + b \cdot \log(n)$ best captures the relationship between IOs and circuit size[3]. A simple linear relationship best describes the division of I/Os between inputs and outputs: $n_{\mathrm{PI}} = c + d \cdot n_{\mathrm{PO}}$. Figure 3.1 shows a plot of n vs. n_{IO}, and a least-squares regression line for the log-linear Rent relationship[4]. We note that simply determining values for the coefficients a, b, c, and d does not capture the increase in variance with n so we model these coefficients as truncated[5] Gaussian distributions around the best-fit line[6]. The actual equations are shown in the IOFrame section of comb.gen in Appendix A.

[3]Note that Rent's Rule explicitly does not apply uniformly for the circuit as a whole (i.e. to predict I/O given n), so we use different functional forms for ranges of n, determined empirically. The actual relationship is a piecewise combination. See Appendix A for the exact equations.

[4]Notice that the X-axis is shown with a log scale so that all points can be displayed with reasonable precision. Thus the visual variance around the regression line is deceptively large.

[5]Though the mean and variance can be determined exactly from the data, we shield ourselves from outliers by truncating the distribution before unrealistic values (in particular, negative values). It is also necessary for us to generate reasonably tame values, because a circuit which is an outlier in one parameter is often an outlier in all parameters, and choosing the parameterization independently cannot model this well.

[6]The regression line itself is not a strong predictor of the relationship between size and I/O, but this is not the point. Together with the Gaussian distribution of variance, we get a good probabilistic sample of a reasonable number of I/Os for a given size. Given the actual variance in the data, this is all that can be expected.

3.2.2 Nodes and Edges.

Two other dependent parameters of a circuit are the number of edges and the average fanin of the circuit. Looking at the data for 4-LUT mapped circuits, we see that average fanin varies from 2 to nearly 4, with a close to (truncated) Gaussian distribution centered around 3, and this is how we model it in the default profile. It is well known from technology-mapping literature that a circuit mapped to k-LUTs will not use all the inputs in each LUT unless $k = 2$, so this is to be expected.

As a byproduct of our experiments, we have observed that the final wirelength of a circuit after placement and global routing is much more highly correlated to the number of edges (equivalently average fanin) in the circuit than it is to the number of nodes. Though this might be easy to believe, it is quite interesting that utilization results for FPGAs are almost always specified in terms of the typical gate size of circuits which fit completely independent of the number of wires in the circuit. This suggests that a more accurate metric of "typical utilization" in an FPGA might be the *wire utilization* used, rather than the *logic utilization*, meaning that n_{edges} is probably a more indicative measure of circuit size than the number of nodes n.

3.2.3 Fanout Distribution.

Recall fanout(x) is the number of edges leaving a node x. A circuit's *maximum fanout* and *fanout distribution* (the number of nodes with fanout 0, 1, 2, etc.) is an important structural parameter which cannot be modeled by known methods in the theory of random directed acyclic graphs. Note that the *fanin* distribution is less interesting for technology-mapped circuits because they have an *a priori* constraint on fanin.

The maximum fanout and the fanout distribution for a selection of MCNC circuits is shown in Table 3.1. The first component gives the number of fanout zero nodes, which is less than or equal to the number of primary outputs (a primary output is not necessarily of fanout zero). A large proportion of the remaining nodes are fanout 1, with decreasing incidence as the fanout value gets higher. Most circuits with a reasonable number of nodes have some higher fanout values. Since these circuits are combinational, we do not have high-fanout clock, clear or reset signals to deal with, but even when discussing sequential circuits later we will ignore these special signals.

Using data from the entire benchmark set, we have developed a simple heuristic algorithm to generate reasonable fanout distributions given the circuit size, number of edges, max_fanout and number of I/Os. Essentially, we choose the n individual fanout values probabilistically from a discretized exponential distribution which is modified online to ensure that $\sum_i i \cdot fanout[i] = n_{edges}$ at completion.

Name	Size	Max_out	Fanout Distribution
cht	102	46	36 32 28 1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 1 ...
9symml	106	34	1 94 2 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 2 ...
C1355	115	16	32 24 8 32 8 0 0 0 1 8 0 0 0 0 0 0 2
bw	137	66	25 72 17 9 1 4 1 2 0 0 0 0 0 0 0 1 0 0 0 0 ...
C1908	178	25	25 51 31 33 7 11 5 2 3 2 1 1 0 1 0 0 1 1 2 0 ...
C3540	481	66	21 235 88 37 11 21 15 3 9 5 1 1 2 0 1 1 14 2 3 4 ...
x3	512	122	99 250 80 29 12 3 7 2 6 3 3 0 0 0 1 1 3 1 1 1 ...
ex4p	514	26	14 360 27 16 15 11 22 2 5 2 5 5 4 2 5 4 0 1 0 1 ...
C6288	559	43	32 35 450 8 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
alu4	1536	249	8 1267 67 41 32 33 14 13 11 3 2 9 9 5 4 0 0 1 1 0 ...

Table 3.1: Fanout distribution for selected MCNC circuits.

Though we take a relatively simplistic approach to modeling the fanout distribution, we note that this type of distribution is nothing like what is seen for random graphs. For random directed acyclic graphs of the same size (nodes and edges) as the MCNC circuits **cht** and **ex4p**, we see fanout distributions of (23 19 18 23 19) and (79 67 75 66 83 77 67) respectively, which are nearly uniform. We point out that this is largely by construction, since natural models for such random directed graphs result in bounded fanin + fanout in order for the graphs to both be connected and to have a linear number of nodes. However, there are no known ways of generating random directed graphs having exponentially distributed fanout vectors which are connected and have a reasonable number of edges.

The heuristic algorithm mentioned above is the model for fanout distribution that we use in the default profile.

To some extent, the average fanout and the distribution of fanout values is dependent on the LUT size k used in technology mapping. A circuit mapped to 2-LUTs will have a much lower average fanout than a circuit mapped into 7-LUTs, in general: though more logic is stored in a LUT (reducing the overall number of edges), the computed value is then used by more other LUTs in the netlist, increasing the fanout value. As a basic rule, the

average fanout follows the average fanin, with variations occurring based on the distribution of I/Os and flip flops.

CIRC outputs a number of other degree-related statistics about a circuit, such as the average fanin and fanout for each combinational delay level, and the average fanout for primary inputs (and later flip-flops) as opposed to internal nodes. These are not used in the default profile, but we note that the information they provide is useful in the debugging of CAD tools, and in analyzing place and route anomalies occurring when the tool encounters outliers in the input.

3.3 Delay-Based Parameters of Combinational Circuits.

For a combinational circuit, define $d(x)$, the *delay* of node x, as the maximum length over all directed paths beginning at a PI and terminating at x, corresponding to the unit delay model. The delay, $d(G)$ (or just d), of a circuit is the maximum delay over all nodes in G. Using a similar empirical analysis to that previously mentioned, we have determined a stochastic relationship between delay d and circuit size n in which d is roughly $\log n$ on average.

Figure 3.2 shows a plot of size vs. combinational delay for 83 combinational MCNC circuits. The dashed function is the line $d = \log(n)$, representing the expected delay for a circuit with n nodes. The lower dotted line is $d = \log(\log(n))$, and the upper dotted line is $d = 3 \cdot \log(n) + \log(\log(n))$. Together these represent the lower and upper bounds on delay as modelled in the circuit profile[7].

3.3.1 Circuit Shape.

Combinational delay is very important in the characterization of circuits, precisely because it is so important in the design and synthesis process. Define the *shape distribution*, shape(G), of a circuit as the number of nodes at each combinational delay level. Figure 3.3 shows a small example circuit (**cm151a**), and its shape distribution (12, 4, 2, 2) displayed as a

[7] The dashed line is a best-fit regression line for the expected delay, and the default is to choose from a Gaussian distribution centered on this line. The two dotted lines represent the imposed truncation on the Gaussian distribution, i.e. the imposed upper and lower bound on the values which will be chosen. The imposed lower bound is $\log(\log(n))$ and the imposed upper bound is $n/3$. These upper and lower bounds given above and shown pictorially in the graph are chosen to include a majority of the points which are feasible, while excluding outliers (such as negative delay) which might otherwise occur. Note that modeling in this way underestimates the number of outliers often seen in practice, as evidenced in Figure 3.2.

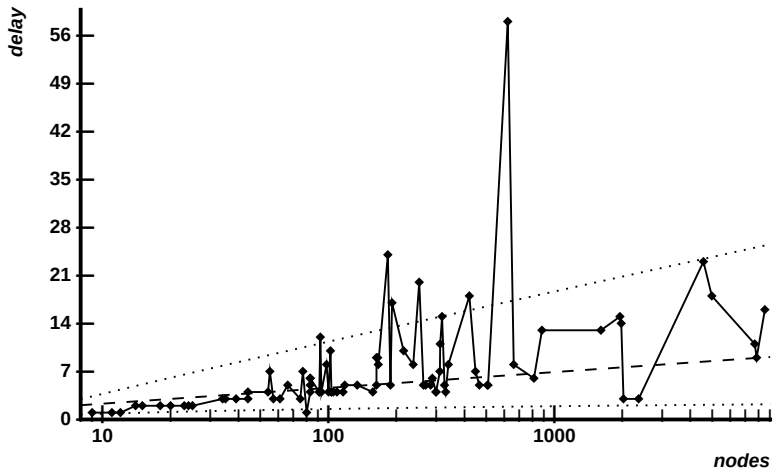


Figure 3.2: Size vs. combinational delay for MCNC circuits.

histogram. Note that even though the primary outputs are shown in circuit drawings we do not count them in determining delay or the shape distribution. Rather, we define "primary output" as a property on the fanin node. While these examples are mapped to 4-LUTs, the basic form of the distribution changes only proportionately for different LUT-sizes.

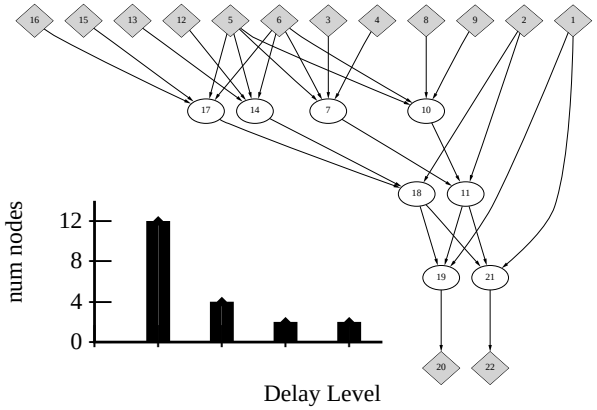
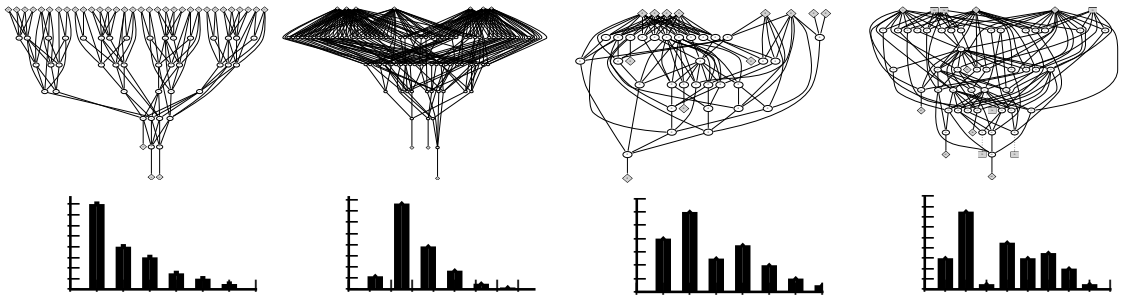


Figure 3.3: Shape distribution.

A characterization such as shape is not an obvious one to a circuit designer, who typically thinks of a design in terms of block diagrams, physical layout, or a set of boolean equations. However, looking at circuits from a graph-theoretic point of view, it is natural to try to draw the circuit in the plane with nodes divided into delay levels, and the importance of shape becomes clear.

The interesting thing about shape is that most circuits tend to have similar shapes. Random directed acyclic graphs from natural distributions tend, as a group, to have a different typical shape. Table 3.2 shows a sample of shape distributions for MCNC circuits,



comp	rd73	sqrt8ml	dk14 (L0)
(36)	(53)	(12)	(8)
"decreasing"	"conical histogram"	"two maxima"	"remaining cases"

Figure 3.4: Different shape distributions.

along with a qualitative classification of different shape functions. Figure 3.4 shows four shape classes and an example of each. Of the 109 combinational multilevel circuits in the MCNC set, 36 have a shape which is strictly decreasing from the primary inputs (as **comp**), 53 have a *conical* shaped histogram, fanning out from the inputs to an extreme point, then strictly decreasing (as **rd73**), 12 have the conical shape with a "bump" (as **sqrt8ml**) and only 8 did not fit into these categories. This distribution of shapes is fundamentally different from degree-constrained random graphs (discussed earlier in Section 2.2.3 and in Chapter 6) which tend, as a group, to almost always have a basically "flat" shape.

We performed experiments to determine whether there is any relationship between shape and routability metrics such as wirelength per edge. However, no significant correlation was found to exist for the MCNC data.

Name	Size	Delay	Shape Distribution
cht	102	2	47 44 11
9symml	106	6	9 57 24 7 6 2 1
C1355	115	4	41 24 8 10 32
bw	137	4	5 57 46 17 12
C1908	178	10	33 23 13 14 22 27 20 6 10 8 2
C3540	481	12	50 82 104 76 44 29 24 22 17 16 10 5 2
x3	512	5	135 202 123 40 10 2
ex4p	514	5	84 245 124 42 14 5
C6288	559	28	32 76 30 30 30 30 30 30 30 30 30 30 30 30 30 29 7 2 2 2 2 2 2 2 2 2 2 3 2
alu4	1536	7	14 692 518 198 80 21 11 2

Table 3.2: Shape distribution for selected MCNC circuits.

Though the example of Figure 3.3 shows both primary inputs at the last combinational

delay level and having zero fanout, neither is typical. We also extract and use the shape distribution of primary outputs (POShape) in the default profile of circuits. POShape is a vector of the number of output nodes at each combinational delay level.

3.3.2 Edge-Length Distribution.

Since nodes have a well-defined delay, we can define the length of a directed edge by length$(x, y) = d(y) - d(x)$. Clearly, the edge length is always between 1 and delay(G), and we define a related *edge length distribution.*

In the example of Figure 3.3 there are 24 edges of length 1, and 2 each of length 2 and 3, so the edge length distribution is (0,24,2,2,0). (Note the placeholder for absent length-0 edges; this is just so that we can have all vectors indexed similarly from 0).

Table 3.3 shows a sample of edge-lengths from the MCNC circuits. We find that almost all circuits have an edge-length distribution with a very similar structure: a large number of edges of length 1, and a quickly falling distribution over the combinational delay of the circuit. This type of distribution is not at all what one would expect of a random graph where the probability of any two pairs of edges being connected is the same. Empirically, such an edge length distribution is not common for random directed graphs arising from natural models (see Section 6.1.).

In the default profile, we model the edge length distribution by probabilistically sampling a discretized exponential distribution, which closely approximates this behaviour[8]

Name	Edges	Delay	Edge-Length Distribution
cht	102	2	0 202 0
9symml	106	6	0 271 41 6 6 0 0
C1355	115	4	0 216 32 0 32
bw	137	4	0 349 93 11 6
C1908	178	10	0 319 78 37 14 11 11 8 16 15 0
C3540	481	12	0 1017 317 143 28 18 13 14 13 6 0 5 1
x3	512	5	0 1071 139 49 8 2
ex4p	514	5	0 1248 167 8 3 0
C6288	559	28	0 1094 70 66 66 66 66 66 66 66 66 66 68 70 65 62 63 2 0 0 0 0 0 0 0 0 0 0 1 0
alu4	1536	7	0 4494 757 125 23 1 0 0

Table 3.3: Edge-length distribution for selected MCNC circuits.

[8]There are no appropriate statistical techniques to formalize this, so "closely approximates" means that the distributions appear reasonable when compared by hand.

3.3.3 Fanout Shape.

Another natural shape characterization is the distribution of *total* fanout by combinational delay level. The fanout distribution by delay level for our sample circuits is shown in Table 3.4. It is interesting that fanout by delay level is close to a strictly decreasing function for all of the circuits sampled. (However, note the exceptions in **C1908** and **C6288**.) Since the shape is conical for many of these circuits, we make the observation that the average fanout of primary inputs is higher than for other nodes. In fact, this is largely true when the number of nodes on a level is smaller than that of its succeeding level in the shape function.

Even though this distribution provides interesting information about the structure of combinational circuits, in practice it results in over specification in the profile. This is because the shape, edge-length and fanout distributions already mentioned constrain the delay-fanout enough that we can calculate tight bounds algorithmically. Thus, we do not currently generate delay-fanout as part of the statistical profile of combinational circuits.

Name	Edges	Delay	Delay-Fanout Distribution
cht	102	2	157 1 0
9symml	106	6	226 60 24 7 6 1 0
C1355	115	4	112 32 72 64 0
bw	137	4	267 124 56 12 0
C1908	178	10	167 53 36 87 67 51 17 19 10 2 0
C3540	481	12	558 338 204 142 85 74 53 42 37 21 18 3 0
x3	512	5	874 297 82 14 2 0
ex4p	514	5	868 376 135 33 14 0
C6288	559	28	1056 119 58 58 58 58 58 58 58 58 58 58 58 58 62 61 6 2 2 2 2 2 2 2 2 2 3 2 0
alu4	1536	7	2867 1700 529 197 79 20 8 0

Table 3.4: Delay-fanout distribution for selected MCNC circuits.

3.4 Reconvergence in Combinational Circuits.

Reconvergence occurs when multiple fanouts from a single node x in the circuit branch back together at a later point y—we say the circuit is *reconvergent at* y. Many circuits exhibit reconvergent fanout, but in widely varied degree, so an appropriate characterization is to quantify this amount.

Define the *out-cone* of a node x (in a circuit with no directed cycles) to be the recursive

fanout of x: the subgraph induced by all nodes reachable on a directed path from x. Figure 3.5 shows out-cone(a). Edges which are not in the out-cone, but are incident with nodes which are, are shown as bold dashed lines.

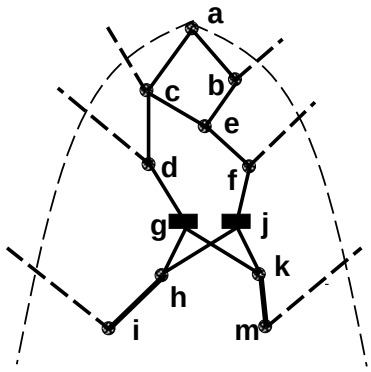


Figure 3.5: Reconvergence in combinational circuits.

For circuits mapped to 2-LUTs, define the *reconvergence number of node* x, $R(x)$, as the ratio of the number of fanin-2 (i.e. "reconvergent") nodes in out-cone(x) to the size of out-cone(x):

$$R(x) = \frac{|\ \{y \in \text{outcone}(x) \text{ s.t. } y \text{ has fanin 2 in outcone}(x)\}\ |}{|\text{outcone}(x)|} \tag{3.1}$$

This value arises from its combinatorial interpretation. By Kirchoff's theorem [31, pp. 49-54], the numerator counts the $\log_2 t$ where t is the number of *spanning out-trees*[9] rooted at x in the directed graph representation of the circuit. Essentially, each reconvergent node represents a choice of two alternatives in the construction of a spanning out-tree, which multiplies the number of trees by two (adds 1 to $\log_2(t)$). Each non-reconvergent node represents a "required" in-edge, hence does not affect the number. The purpose of taking the logarithm is simply to obtain tractable numbers when dealing with large graphs. The denominator then scales that value with the size of the out-cone so that different graphs can be compared based on their relative amount of reconvergence, which otherwise would be dominated by the size of the circuit[10].

The intuitive argument for counting spanning out-trees is clear: a single spanning out-tree has zero reconvergence, and the number of spanning out-trees scales with the number

[9] A spanning out-tree rooted at r is a spanning tree such that each node, except the designated root node, has exactly one fanin. Hence each node lies on a unique directed path from the root.

[10] Analysis shows that there is no significant statistical correlation between R and n, so this adjustment is sufficient.

of ways that reconvergent fanout occurs in the circuit. This is even more compelling when we generalize the reconvergence calculation to sequential circuits in the next chapter.

For circuits mapped to k-LUTs, $k > 2$, the reconvergence calculation generalizes, both algorithmically and combinatorially, if we set the numerator as the sum, over all nodes y in the out-cone of x, of $\log_2(\text{fanin}(y))$. Thus $0 \leq R(x) \leq \log_2(k)$.

$$R(x) = \frac{\sum_{y \in \text{outcone}(x)} \log_2(\text{fanin}(y))}{|\text{outcone}(x)|} \tag{3.2}$$

To identify the reconvergence $R(G)$ present in an entire *circuit* G, we compute the weighted (by out-cone size) average of $R(x)$ for all primary inputs x in G. Thus $0 \leq R(G) \leq \log_2(k)$ continues to hold for circuits. In this way, highly reconvergent small portions of a circuit will not unduly affect the overall quantification.

The observed reconvergence numbers for the 198 combinational and sequential 2-LUT-mapped MCNC circuits vary between 0.0 and 0.92, with a relatively even distribution of circuits through the range 0.0 to 0.85. R is somewhat a measure of complexity of the logic—we find that intuitively simple, tree-like, logical functions have low R (e.g. **parity**: $R = 0.00$, **decod**: $R = 0.00$, **mux**: $R = 0.15$), and more complex functions have higher R (e.g. **alu2**: $R = 0.52$, **sqrt8ml**: $R = 0.53$). Combinational logic and the combinational parts of sequential arithmetic logic fall mostly in the range 0.0 to 0.6, whereas the combinational parts of finite state machines are mostly in the range 0.5 to 0.85 (9 of the 10 most reconvergent circuits are finite state machines). Table 3.5 shows the reconvergence numbers for a sample of combinational MCNC circuits for which we have some functionality information. Note that this information is inherently biased, because most circuits have no listed description and were left out of the table. Thus we can make only the vague observations about relative complexity of the logic.

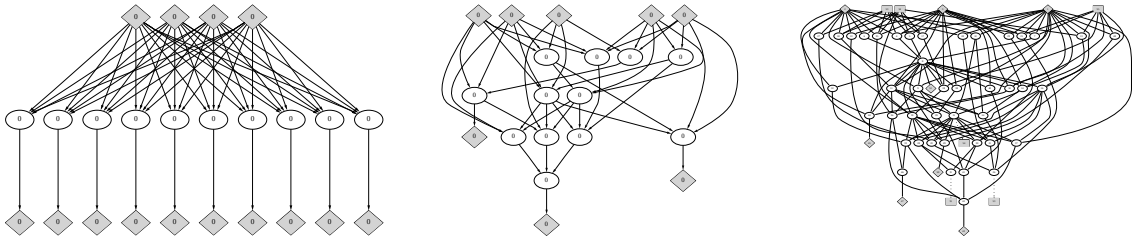
In a physical sense, there is a high degree of correlation between R and the other characteristics of a circuit; in particular, the number of edges (when $k > 2$), and the shape and out-degree functions. Using the examples of Figure 3.4, circuits which have an exaggerated conical shape, such as **rd73** ($R = 0.40$) and **sqrt8ml** ($R = 0.53$) tend to have higher reconvergence values, whereas circuits like **comp** ($R = 0.22$) are lower. This also tends to explain the difference between combinational and sequential circuits because the first "sequential level" of most finite state machines tends to be very conical. A conical shape arises because

Name	R	Description
parity	0.00	parity tree
decod	0.00	simple decoder
count	0.15	counter
mux	0.15	multiplexor
C1355	0.19	error correcting
my_adder	0.21	adder
C5315	0.26	ALU and selector
dsip	0.27	sequential encryption
des	0.30	data encryption
z4ml	0.30	2-bit adder
9symml	0.31	count ones in inputs
C2670	0.32	ALU and control logic
C7552	0.34	ALU and control logic
C880	0.36	ALU and control logic
s208	0.38	sequential multiplier
s838	0.41	sequential multiplier
s1196	0.41	sequential "logic"
C1908	0.44	error correcting
i10	0.47	combinational "logic"
sbc	0.47	sequential snooping bus controller
C3540	0.50	ALU and control logic
alu2	0.52	ALU
sqrt8ml	0.53	square root function
mult16a	0.54	sequential 16 bit multiplier
mult32b	0.54	sequential 32 bit multiplier
C432	0.58	priority controller
C6288	0.63	16 bit multiplier
apex4	0.63	combinational logic from a PLA
s400	0.63	sequential FSM: traffic light controller
clma	0.63	sequential bus interface
bbtas	0.76	finite state machine
pdc	0.79	finite state machine
s1488	0.83	finite state machine (controller)
dk16	0.89	finite state machine

Table 3.5: Reconvergence for selected MCNC circuits.

of a low I/O to logic ratio, natural because I/Os are "reused" over time in a sequential circuit.

Figure 3.6 shows examples of three different small circuits. The first, **cm42a** is a decoder, and has no reconvergence at all. The second, **rd53**, is combinational control logic, and has a reconvergence number of 0.40. The third is the first level of a finite state machine (we just converted flip-flops to primary inputs and primary outputs and dropped any logic past the flip flops). Its computation of reading the inputs and producing an encoded state has a reconvergence number of 0.69, the largest of the three.



cm42a R=0.00 | rd53 R=0.40 | dk14 (L0) R=0.69

Figure 3.6: Circuits with Varying Reconvergence.

3.5 Locality in Combinational Circuits.

To this point we have concentrated on *delay* as the fundamental characteristic of a circuit. Both the shape and edge length functions are delay based. This differs from previous work on wireability analysis, outlined in Section 2.2.2, which uses Rent's Rule and other stochastic measures of wirelength to describe the physical characteristics of a circuit.

In the generation process, it is clearly necessary to introduce some form of local clustering into a synthetic circuit. In this section we visit the issue of local structure in combinational circuits, with the goal of better understanding wirelength issues in the context of our existing delay based combinational model. Specifically, we will define metrics for wirelength and edge connections between delay levels and give an algorithm for ordering and positioning nodes within their combinational delay which allows us to calculate these metrics.

The best method of measuring the real wirelength and other routability parameters would likely be to execute placement and global routing on a gate array and measure the Manhattan wirelength, as would be performed by layout tools such as VPR [8], Altera's MAX+PLUS2 [4] or Xilinx PPR [73]. However, our purpose is to quickly determine a small amount of information necessary to characterize the locality in a circuit, not to do a complete and expensive physical layout.

Our process for extracting locality information is to determine an ordering of the nodes within each combinational delay level, and then an integer x-coordinate positioning for each node which respects the order: in other words, an embedding of the circuit graph on the integer grid, where the y-coordinate is constrained to be the node's combinational delay.

Given such a positioning $u.x$ for each node u, we can establish a number of metrics: Define *spread(i)* as the difference between the maximum and minimum x coordinates of

nodes on level i (i.e. the "width" of level i). Define *span(u)* for node u as the maximal distance between the coordinates of its fanins. Define *wirelength(u,v)*, for edge $e = (u, v)$ to be $|u.x - v.x|$, *wirelength(u)*, for node u, to be the sum over all fanins u of v of wirelength(u, v), and *wirelength(C)* for a combinational circuit C to be the sum, over all nodes u in C, of wirelength(u).

We note that the wirelength of a circuit in this sense is a layout into a shape structure. Thus it would be related to, but not necessarily the same value as, wirelength after embedding into a standard cell array, a gate array, or an FPGA. Empirically, though there is a strong linear relationship between the two forms of wirelength, the variance is large enough that the version based on shape would not, in itself, be a valid predictor of wirelength or routability in a gate array or FPGA.

To order and position the nodes for these wirelength and span calculations, we use an approach similar to that used by Gasner, North and Vo in the DOT package [30], used to draw many of the pictures in this thesis. The basic approach for ordering is to use the barycentric heuristic [22] to iteratively reduce crossing number between delay levels. We then diverge from the DOT approach to perform a more straightforward method of positioning nodes with integral coordinates which maintain the ordering but reduce wirelength. Sections 3.5.1 and 3.5.2 discuss these two aspects of the algorithm, then Section 3.5.3 discusses the results of executing the algorithm on combinational MCNC circuits.

3.5.1 Node Ordering Within Delay Levels

The problem of node ordering on a DAG G with delay d is to compute "good" orderings of the nodes at each level i, $0 \leq i \leq d$. The word "good" in the context of graph drawings is itself a new area of research, and there is no uniformly accepted metric of goodness. However, previous research [5, 22, 47] has determined that minimizing the *crossing number* not only yields drawings which are more viewable, but it also tends to illustrate symmetry and minimize the length of the drawn edges. Furthermore, since our ordering problem is similar to the placement problem of standard-cell layout, minimizing the crossing number is clearly desired. The crossing number of a graph and a given ordering is the number of pairwise crossing edges in the straight-line drawing of the graph when nodes are constrained in the y coordinate to their delay level and in the x coordinate to the determined ordering.

Figure 3.7 shows a drawing (by DOT [47]) of the MCNC circuit **comp** which illustrates

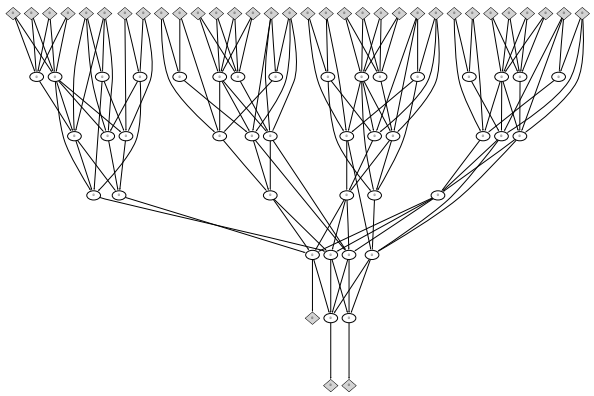


Figure 3.7: Minimizing crossings for a better "drawing."

the local effects of minimizing the crossing number. Though this drawing retains many crossings, a natural effect of the algorithm is to separate the loosely connected portions of logic. Our goal in this section is to take advantage of this separation in order to determine the amount of local structure present in a circuit. Because a drawing in this way corresponds to our delay-based model of circuits, this is also a natural way to impose local structure on a circuit later in the generation algorithm.

The problem of layout to minimize the crossing number is known to be NP-hard [21], even when $d = 1$ (i.e. the graph is bipartite and has two levels). Thus only heuristic algorithms are possible.

We will use a method similar to that originally used by Sugiyama *et. al.* [65], analyzed by Eades and Wormald [22] and used by Gasner *et. al.* [30] for the DOT program.

The basic algorithm is as shown in Figure 3.8. Note that "current_order" and "best_order" refer to data structures which hold the ordering for all levels of the graph.

On even passes, we treat level $i-1$ as fixed, and resort the nodes at level i based on the average ordinal value of their fanins. For odd passes we use level $i+1$ as fixed and look at fanouts. The initial order is simply a random ordering of nodes for each level.

The algorithm converges very quickly—about 10 iterations suffice for even large circuits (this is pointed out by Gasner *et. al.* [30] as well). The crossing number typically decreases by about a factor of 10 from the randomized to the "placed" version. We note that in random graphs generated as per Section 6.1, the crossing number decreases only by a factor of about 3.

```
best_order = a random order
compute crossing(best_order)
current_order = best_order
iter = 0
loop
    /* Compute a new current order */
    for (j = 0; j < d; j++)
        /* Working on combinational delay level j */
        if (iter is even)
            compute the average fanin index of each node at level j
            resort nodes at level j based on average fanin index
        else
            compute the average fanout index of each node at level j
            resort nodes at level j based on average fanout index
        end if
    end for
    iter++
    compute crossing(current_order)
    exit loop if crossing(current_order) > crossing(best_order)
    best_order = current_order
end loop
```

Figure 3.8: Algorithm to compute the crossing number.

Computing the crossing number.

In order to execute the heuristic algorithm above, we need to calculate the crossing number for edges between two combinational delay levels. The obvious approach is to examine each pair of edges to see if they cross, which can be accomplished in $O(n^2)$ time—we have $O(n)$ nodes and also $O(n)$ edges between any two delay levels, under the assumption of constant fanin (otherwise the obvious algorithm becomes $O(n^4)$). For large circuits, a quadratic algorithm is too expensive. Fortunately, we can give an easy to implement $O(n \log n)$ algorithm. To our knowledge, no such algorithm has been previously given for computing the crossing number.

Problem: Given a bipartite graph $G(X, Y)$ and sorted orders $1..|X|$ and $1..|Y|$ for the nodes of X and Y, determine the number of pairwise crossing edges, that is the number of pairs of edges (x_1, y_1) and (x_2, y_2) such that $x_1 < x_2$ and $y_2 < y_1$ in the respective orderings of X and Y.

Our solution uses a divide and conquer approach which, interestingly enough, actually allows us to count more crossings than we examine (i.e. we can count $O(n^2)$ crossings in $O(n \log n)$ time.

Let $n_x = |X|$ and $n_y = |Y|$, and let x_i (y_i) denote the i'th node in the sorted order of X (Y).

Define the following sets of nodes:

A: nodes x_i in X such that $i \leq \frac{n_x}{2}$

B: nodes x_i in X such that $i > \frac{n_x}{2}$

C: nodes y_i in Y such that $i \leq \frac{n_y}{2}$

D: nodes y_i in Y such that $i > \frac{n_y}{2}$

Then we can classify each edge as AC, AD, BC or BD. There are $4 \cdot 4 = 16$ types (combinations) of edge crossings.

We calculate the number of crossings from X $(A + B)$ to Y $(C + D)$ by dividing the edges into their categories (trivially in $O(n)$ time) and decomposing the problem as follows:

$$
\begin{aligned}
&\text{crossings}(A + B, C + D) = \\
&\qquad \text{crossings}(A + B, C) && \text{/* recursive call */} \\
&\qquad + \text{crossings}(A + B, D) && \text{/* recursive call */} \\
&\qquad + \text{num\_cross}(AC - AD) && \text{/* separate computation */} \\
&\qquad + \text{num\_cross}(BC - BD) && \text{/* separate computation */} \\
&\qquad + |AD| \cdot |BC| && \text{/* sizes only */}
\end{aligned}
$$

The recursive call "crossings$(A + B, C)$" refers to the sub-problem on the nodes (and induced edges) not incident on D. The call to num_cross$(AC - AD)$ will be a separate routine to count all crossings between an AC and an AD edge, and no others.

Since C and D partition Y evenly, each recursive call is on at most one half of the maximum edges to the preceding call. Thus, as long as we can find a linear time algorithm for num_cross() the entire algorithm will be $O(n \log n)$.

The cases for num_cross() are symmetric, so we will work on num_cross$(AC - AD)$ only. Assume that the edges have been divided into AC and AD edges already (easily $O(n)$ time), and are still sorted by x_i value in the ordering. Then an AC edge (x_i,y_j) and AD edge (x_k,y_l) cross if and only if $i > k$. We know that $j < l$ from the edge classes.

We take a single pass through A, and count the number of AC edges at each location i. Then we scan again, summing, to calculate the number of AC edges *to the right of* location

i. In a third pass, we look at every AD edge, which necessarily crosses the number of AC edges with x coordinate to the right of the current location, which is the previous sum vector. This correctly calculates num_cross($AC - AD$) in linear time. Note that we do *not* count the $AC - AC$ edges here, or we would be double counting them.

The proof that the algorithm works is a simple case analysis.

AC-AC	–	counted only in crossings($A + B, C$)
AC-AD = AD-AC	–	counted only in num_cross($AC - AD$)
AC-BD = BD-AC	–	cannot cross
AC-BC = BC-AC	–	counted only in crossings($A + B, C$)
AD-AD	–	counted only in crossings($A + B, D$)
AD-BD = BD-AD	–	counted only in crossings($A + B, D$)
AD-BC = BC-AD	–	must cross; counted in the product
BC-BD = BD-BC	–	counted only in num_cross($BC - BD$)
BC-BC	–	counted only in crossings($A + B, C$)
BD-BD	–	counted only in crossings($A + B, D$)

We conclude that crossings($A + B$,$C + D$) can be calculated in $O(n \log n)$ time.

3.5.2 Coordinate Positioning of Nodes

To position nodes, we perform another iterative step.

From the previous step, the order of nodes within each delay level is fixed. Define *width* to be equal to the maximum size of any delay level, and coordinates $u.x$ for every node u, which equally proportions the nodes at level i across *width*.

The iterative step is similar to the ordering algorithm, except that we do not exchange nodes, just move them closer together or further apart within the ordering. On even iterations we define the *centre* of a node u as the average x coordinates of its fanins. On odd iterations we use its fanouts.

For each node u at level i, we compute centre(u), and move $u.x$ as far as possible to center(u), without going past u's neighbour.

Wirelength, as previously defined, is the sum of the lengths of each edge. The length of an individual edge is the difference in the x coordinates of its endpoints.

As in the ordering step, it takes only a small number of iterations for the wirelength to

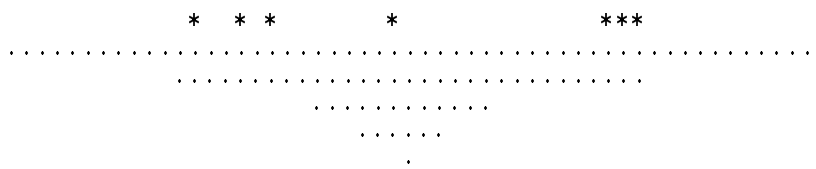


Figure 3.9: Locality placement for **rd73**.

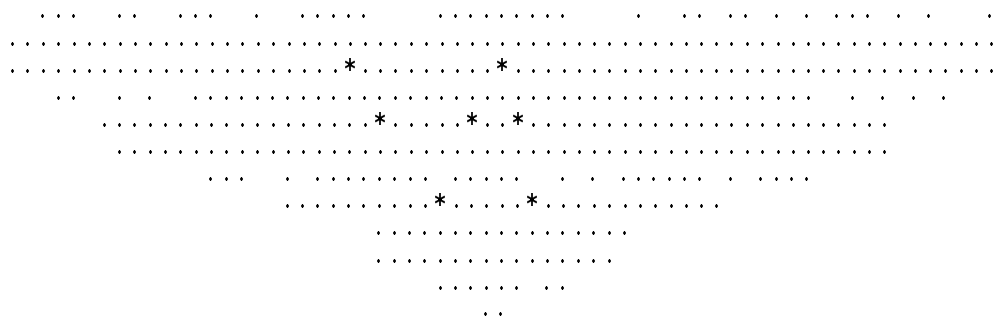


Figure 3.10: Locality placement for **C432**.

hit a minimum. At that point, we can also calculate the other metrics (span and spread) mentioned earlier.

3.5.3 Discussion

Though our goal in doing this pseudo placement is to extract metrics like spread and average span, it is interesting to see the effects of the placement on real circuits. We note that a complete algorithm that takes into account room for edges to be drawn would result in node coordinates that mimic the results of DOT.

Figure 3.9 shows a drawing of the nodes in **rd73**. A '.' indicates a node position, and a '*' indicates a node which has fanout greater than ten. We see a very balanced local structure below the inputs level, but a wide spreading of the primary inputs.

Figure 3.10, on the other hand, shows a circuit which has a slightly less balanced structure. In addition to high-fanout nodes, we see "holes" in the layout which indicate areas where nodes are drawn apart from their neighbours by local structure.

Figure 3.11 shows a structure which is further from balanced. We observe the wide spread of nodes at delay 1, for example.

Our final example is shown in Figure 3.12. This is a circuit which exhibits a great deal of local structure. We observe the tree-like way that terms are collected from the inputs to

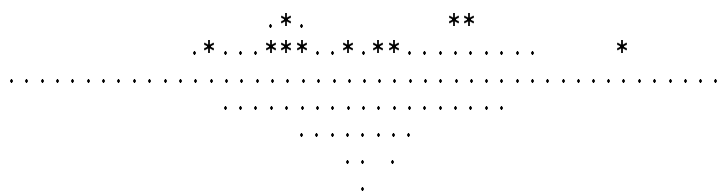


Figure 3.11: Locality placement for **rd84**.

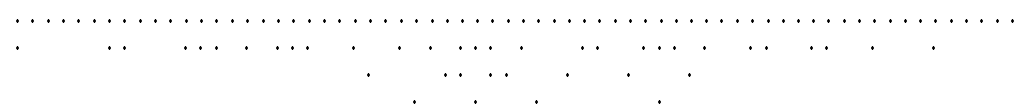


Figure 3.12: Locality placement for **i3**.

outputs: a wide spread, with lots of holes, as the delay increases. This indicates that nodes are more closely related to their close neighbours in index value.

The metrics of average in span and spread for each delay level values can be seen as quantifications of the locality present in the circuits: A high average node span indicates that nodes draw from a wider range of fanins, and that the circuit is less local than would otherwise be the case. The spread of a level, compared to the number of nodes it has, gives information about how closely the nodes at a level share fanins and are pushed together by the wirelength minimization process.

An important aspect of locality that these particular metrics (span and spread) do *not* capture is edge crossings, in particular the distribution of crossings over x coordinate "slices" of the drawing. It would be very useful in the generation algorithm to have more information of this type, but we leave this particular topic to future work. As well, though we can extract this information from specific existing circuits, we have not yet investigated methods to model this type of locality in the default profile (though this could be done). Thus it is currently useful only for generating "clone" circuits, as will be discussed in later chapters.

It is important that the locality algorithm is fast. Extraction of local information from a medium sized circuit such as **alu4** uses one tenth the cpu time of a complete place and route[11].

[11] This does not necessarily make the method a competitor for standard placement algorithms, because we are not restricting the placement to a minimal size square grid.

Chapter 4

Characterization of Sequential Circuits

Combinational circuits have limited application, and any general CAD tool or FPGA must be able to deal with sequential circuits. In this chapter, we expand our characterization of combinational circuits towards this goal.

Before we can proceed it is necessary to have a more detailed model of what we mean by a sequential circuit. In Section 4.1 we describe such a model, defining sequential circuits in terms of combinational building blocks. Section 4.2 describes the basic statistical characterizations arising from the model and our empirical analysis with the MCNC benchmark circuits, in particular the issue of "ghost" inputs and outputs arising in the decomposition of a sequential circuit. Section 4.3 extends the combinational characterization of reconvergence from the previous chapter to sequential circuits.

4.1 The Sequential Model.

We model a sequential circuit as a hierarchy of two or more combinational circuits connected with flip-flops and "back-edges." A single level sequential circuit is simply a combinational circuit.

For this work we consider only synchronous sequential circuits with a single global clock. This ensures that there is a well-defined notion of time in the logical operation of the circuit, and we can define "sequential levels" on the basis of time increments.

In addition to a single clock restriction, we ignore reset/preset/clear lines, assume uni-

directional I/O pins, and do not allow internal tristate buffers. None of these are major restrictions in a theoretical sense: our model can be generalized to allow for circuits to be analyzed or generated hierarchically, so multiple clocks could be hidden within sub-circuits generated separately without a great deal of difficulty. (The generalized model has not yet been implemented in CIRC and GEN.) Similarly, bidirectional pins and tristates can be simulated in standard logic.

In a practical sense, however, we point out that bidirectional pins and the correct physical layout of busses in a design are important, and a commercial system would certainly deal with them explicitly. This is particularly true for FPGA architectural experiments, as tristates or other buffers could be consumable resources and bidirectional pins may (depending on the architecture) introduce greater stress on the routing network than do separate input and output pins. Also, though clocks are often special resources, FPGAs have a limited number of them, and the software may have to deal with some clocks or reset lines as ordinary logic signals. We leave implementation of these detailed features for future work.

For simplicity we assume that the only registers allowed are D-type flip-flops (as is common for most commercial FPGAs). Thus all nodes are of type PI (primary input), LOG, (logic) or DFF (flip-flop). Recall from the previous chapter that PO (primary output) is a property of a logic node, not a separate node type.

Our abstract model of a sequential circuit is shown in Figure 4.1. The figure shows a 3-level sequential circuit. The definitions of primary input, primary output, and all measures of fanout remain as described in Chapter 3. 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 $level(x) > level(y)$. By definition, any other edge is necessarily from a node at sequential level i to a DFF at level $i + 1$, and we call it a *FF-edge*.

It is important to point out that, though this model could appear to apply only to certain types of circuits which have a pipelined appearance, it does not actually preclude other views of sequential connections. Rather we just *define* sequential levels in this way.

With the introduction of sequential levels, we have to modify the definition of combinational delay: for node x, delay(x) = 0 if x is a PI or a DFF and one greater than the maximum delay over its fanins, otherwise. The definition of edge-length is as before, even if

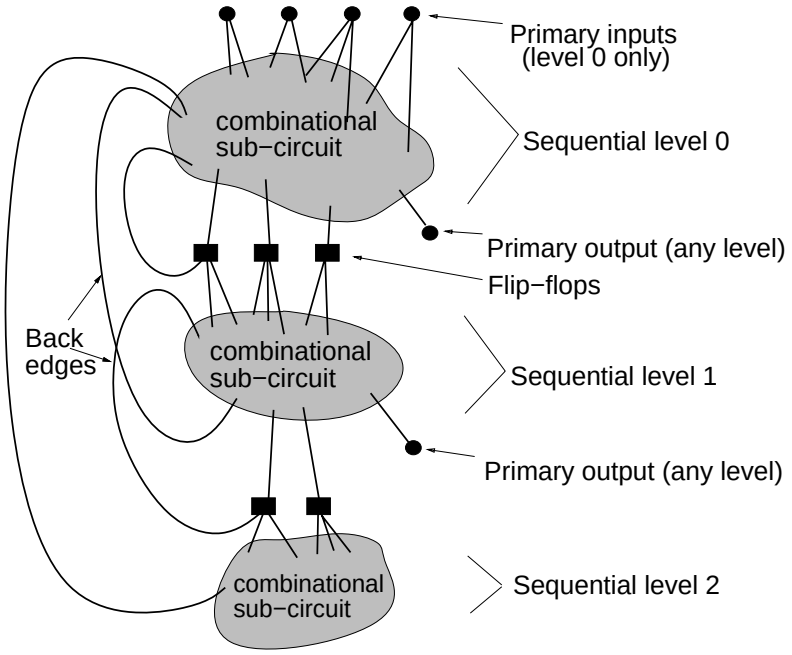


Figure 4.1: Abstract model of a 3-level sequential circuit

the nodes are at different sequential levels, except that edges to a DFF are always of length one. The *size* of the circuit is $n = n_{LOG} + n_{PI} + n_{DFF}$.

4.2 Characteristics of Sequential Circuits.

There are a number of new sequential characteristics arising directly from the model, and we describe them here. Note that all empirical results are based on the MCNC circuits, as mentioned previously in Section 3.1.

4.2.1 Basic Characteristics

The division of a circuit into its combinational sub-circuits introduces the concepts of *sequential shape*, the number of nodes in each successive sequential level, and the number of sequential *levels*. We also have counts of the numbers of flip-flops and back-edges. Table 4.1 shows this information for a sample of sequential MCNC circuits.

The number of I/Os is greatly decreased for sequential circuits, and we find that the Rent-like parameterization that we used before is no longer an adequate reflection of the I/O consumption for the circuit. In fact, we find that there is no real statistical correlation between the size of the circuit and the number of I/Os. In the default profile for sequential

Name	Nodes	IOs	nDFF	Edges	nBack	Levels	Seq. Shape
s838	167	37	32	556	256	2	169 65
s953	214	39	29	739	184	3	191 65 3
styr	238	19	5	814	219	2	207 45
planet	266	26	6	910	300	2	169 110
sbc	372	96	27	1273	300	2	388 51
mm30a	467	63	90	1697	235	2	500 90
dsip	1362	425	224	5440	896	2	1590 224
s298	1930	9	8	6944	2218	2	1636 305
bigkey	1699	425	224	6108	1344	2	1591 560
clma	8361	127	31	30114	5596	3	5810 2640 3

Table 4.1: Sequential circuit characteristics for selected MCNC circuits.

circuits, we use one quarter of the combinational I/O calculation as an upper bound on the number of I/Os, then choose the number of PI and PO for the circuit uniformly between 2 and the upper bound. In practice this yields reasonable values.

We find that the number of sequential levels is a small constant. Recall that a circuit with one sequential level is a combinational circuit. Of 78 sequential MCNC circuits, 69 have two sequential levels, 6 have three levels, and there is one circuit each of 4, 7, and 8 sequential levels. In all cases we saw, the majority of the combinational logic lies in the zeroth sequential level. We typically see successive sequential levels of logic having less than half the logic of the preceding level.

The number of flip-flops in a circuit also has little correlation to the amount of logic in the circuit. This can occur for many reasons. For example, the designer of a state-machine has the choice of encoding the state directly or in logarithmic size with extra decoding logic. Thus the number of flip-flops in the defaults file is also calculated with a wide degree of variation. We use a Gaussian distribution around a constant-deflated square root of the number of nodes as an approximation. See Appendix A. Note that this roughly models the number of flip-flops as the number of I/Os in a combinational circuit, not an unreasonable thing looking at the model. In FPGAs, the number of *available* flip-flops is usually proportional to the number of LUTs (often 1:1), but this is more to do with the design cost of adding the flip-flops and with logic block homogeneity than with the raw numbers of flip-flops required by circuits.

The number of back-edges varies between one and two times the number of nodes at the first sequential level, and we model it as such.

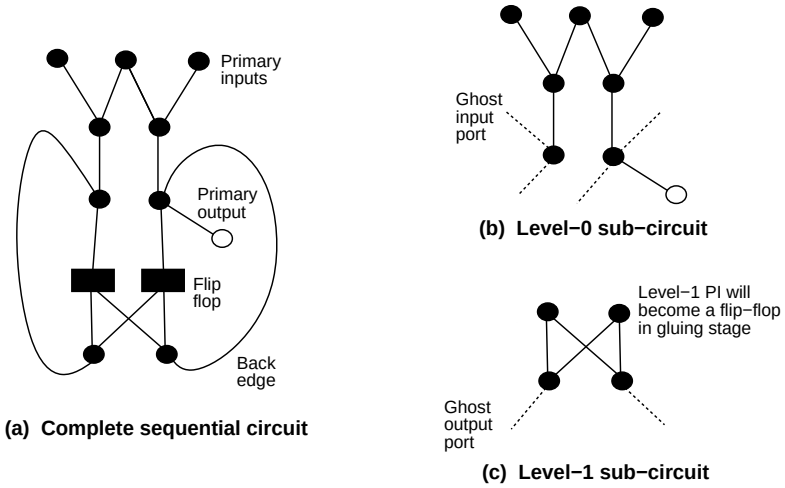


Figure 4.2: Example decomposition of a 2-level sequential circuit.

4.2.2 Decomposing Sequential Circuits.

Our model defines a sequential circuit based on its combinational sub-circuits, back-edges, and flip-flops. As part of the characterization process, we want to decompose a sequential circuit into its component parts. To describe the sequential interface within combinational sub-circuits we introduce the concepts of a *ghost input port* and *ghost output port*. Intuitively, these are points that connect different sequential levels (combinational sub-circuits).

These are best understood with a small example. Figure 4.2(a) shows a sequential circuit with three primary inputs, one primary output, two flip-flops (hence two flip-flop edges) and two back-edges. The decomposition of this circuit is shown to the right: Figure 4.2(b) shows the level-0 sub-circuit with 3 primary inputs and one primary output. We have two *ghost input ports* (GI) which record the existence of back-edges from a succeeding level, and two *ghost output ports* (GO) which record the location of back-edges connected to a preceding level or, as in this case, edges to flip-flops at a succeeding level. Similarly, Figure 4.2(c) shows the level-1 sub-circuit with two primary inputs (which used to be flip-flops) and two ghost outputs. Note that GI and GO ports correspond more closely to edges than nodes, since a single node can have up to $k-1$ ghost inputs, and *max_out* ghost outputs. We note that in any sub-circuit, a zero-fanout node must have at least one GO or PO attached to it.

In the parameterization of the combinational sub-circuits, it is not sufficient simply to record the number of ghost inputs and outputs, as this ignores a great deal of information about the interface between sub-circuits. In particular, if we are to use this model as the

basis for a generation algorithm, it is important to ensure that sub-circuits are *compatible*. For a single GO and GI to be compatible, the combinational delay of the node with the GO must be less than that of the node with the GI (i.e. it is legal/sensible to connect the GI to the GO in the context of combinational delay). For two sub-circuits to be compatible, there must exist a matching of GI and GO between them, all of which are compatible.

To deal with compatibility issues between sub-circuits, we introduce the GI and GO *shape* within sub-circuits. Define the vector *GIshape[i]* as the number of ghost inputs at combinational delay i, i=0..d, and $GOshape[i]$ 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, 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.

For the circuit in Figure 4.2(b) we have GIshape = (0,0,2) and GOshape = (0,2); Figure 4.2(c) has GIshape = (0,0) and GOshape = (0,2). We note that flip-flops are not included in the GIshape of a level, because they are already recorded in n_{DFF} (a purely semantic detail).

As an example, the circuit **clma** has 3 sequential levels:

```
Level 0:
    GIshape = (  526 1245 664 354 451 429 860 502 295 48 37 25 22 2 4 0 0 )
    GOshape = (    0    0   0   8   4   7   1   2   0  2  0  0  0 1 1 2 1 )
Level 1:
    GIshape = (   74   45   3   8   2   0   0   0   0  0 )
    GOshape = ( 1289 1282 412 671 372 364 555 360 151  4 )
Level 2:
    GIshape = (    0    0 )
    GOshape = (  136    2 )
```

We find that the GI and GO shapes of MCNC sequential circuits do not statistically show any common shape beyond GIshape[i] being roughly proportional to shape[i] within sub-circuits. We have a heuristic process for generating reasonable GI and GO shapes which are compatible, and the interested reader is referred to the GEN source-code for details.

Note that the shape distributions for the combinational sub-circuits of sequential circuits differ from those of purely combinational circuits. This is because the second sequential level often has many more flip-flops (inputs) than is typical for a combinational circuit of

the same size.

4.2.3 Extensions to the Sequential Model.

With ghost input and output ports now defined, it is worth pointing out that the sequential model can be generalized to describe arbitrary levels of hierarchy, rather than just the interface between multiple levels in a simple sequential circuit.

For example, we can define a purely combinational circuit as a hierarchy of combinational sub-circuits simply by combinational specifications and a compatible GI and GO interface (without requiring that the circuit have flip-flop or back-edges). In combination with a partitioner this would allow us to form a partition tree model of an input circuit.

It would also be interesting to use this mechanism to describe an interface to other forms of circuits (e.g. memory), or to deal with circuits at the block diagram level.

The ability to generalize the use of ghost inputs in generation and outputs would open the door to a hierarchical generation process.

In this dissertation, however, we will restrict ourselves to simple sequential circuits.

4.3 Generalizing Reconvergence.

In Section 3.4 we defined the reconvergence number of a node r in a combinational circuit as the proportion of reconvergent nodes to total nodes in the out-cone of r. We also pointed out the combinatorial significance of the numerator as the $\log_2 t$, where t is the number of spanning out-trees rooted at r.

Recall a *spanning out-tree* $T(G)$ of a directed graph G with respect to a designated root node r is a spanning tree of G such that, for all x in G there is a (necessarily unique) directed r-x path in T.

Recall that the combinational *out-cone* of r in G (which we now denote G_r^c) is defined recursively as follows: r is in G_r^c and if x is in G_r^c and xy is a *forward* edge of G then the node y and the edge xy are also in G_r^c. Define the *sequential out-cone*, G_r^s of r to be identical, but *without* the restriction of xy being a forward edge. Then G_r^c is always a subgraph of G_r^s.

Using the sequential out-cone, the numerator in our reconvergence calculation no longer corresponds exactly to the number of spanning out-trees. Consider the sequential circuit

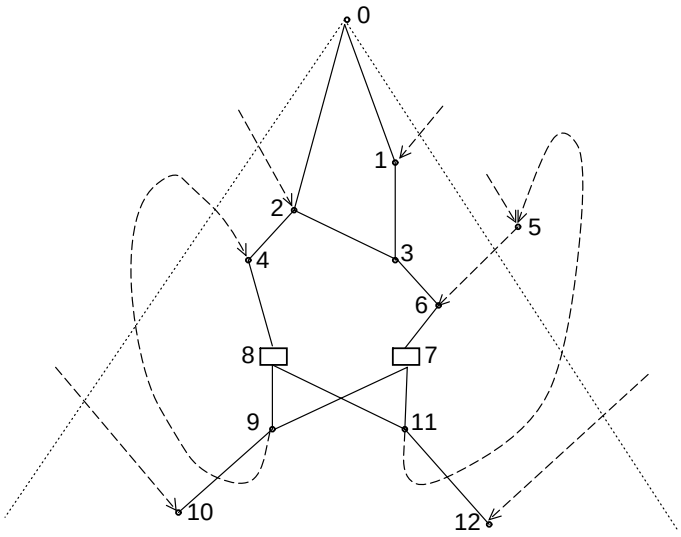


Figure 4.3: Reconvergence in a circuit.

represented in Figure 4.3. The combinational out-cone of node 0 is shown within dotted lines from 0. The number of reconvergent nodes in the combinational out-cone of node 0 is 3 (nodes 3, 9 and 11), and there are 2^3, or 8, spanning out-trees. However, the sequential out-cone of node 0 additionally includes vertex 5, and edges (11,5), (5,6) and (9,4). The number of reconvergent nodes in the sequential out-cone of node 0 is five, (nodes 3, 4, 6, 9, and 11), but the number of spanning out-trees is 15, not 32. The reason for this is that the choice of edges is no longer independent: no spanning out-tree can contain both (5,6) and (7,11).

Define the n by n matrix K with respect to a digraph G as follows:

$$K_{ij} = \begin{cases} \text{in-degree}(i) & i = j \\ -1 & i \neq j, (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

We note that K_{ii} is 0 if and only if i is a source in G, and that the sum of the entries in any column i is 0. Furthermore, if the vertices are in topological order[1], K is upper-triangular if and only if G is acyclic.

Now consider the graph G_r^s (with n' nodes) for digraph G with root r. Let K_r be the minor with respect to r of the Kirchoff matrix of G_r^s (i.e. the matrix formed by removing

[1]A *topological order* on the vertices of a directed acyclic graph G is any order σ such that the existence of edge xy implies that $\sigma(x) < sigma(y)$. Such an order always exists for an acyclic digraph.

row and column for r, resulting in a square matrix of dimension $n' - 1$). They we can apply the following to count the number of spanning out-trees from r in G_r^s.

Theorem (Kirchoff, c.f. [31]) *The number of spanning out-trees rooted at r in a finite digraph G is equal to the determinant of K_r.*

The basic idea of the proof is that as the determinant of this matrix is broken into terms by a standard linear algebra decomposition, the number of leaf corresponds to the number of trees from a given root vertex. The combinatorial justification for this process is explained fully in the book by Gibbons [31][pages 49-54], and the interested reader is referred there for the details.

For our purposes here, it is sufficient to explain the process with our example (Figure 4.3).

The intuition is clearer for acyclic graphs. Ignoring all back edges, the out-cone of node 0 consists of the 12 nodes and solid edges shown inside the cone. The Kirchoff matrix of the out-cone of 0 is then

$$K = \begin{pmatrix} 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Combinatorially, the number of spanning out-trees from G can be calculated as the product of the in-degrees of the vertices of G (not including the root)—if the in-degree of vertex x is 1, then that edge must be present in any spanning out-tree. If x has two or more inputs then any one can be chosen independently of other choices of edges in $T(G)$. Since K_r is upper triangular, its determinant is the product of the diagonal elements. (Note, because we chose the out-cone, the value is always at least 1.) Thus, the number of spanning out-trees in the out-cone of 0, ignoring back edges, is 2^3 or 8.

The situation is more complicated when we allow cycles. Adding the vertex 5 and edges (11,5), (5, 6) and (9,4) increases the dimension of K by 1, and makes K_r no longer

upper triangular; correspondingly, the choice of edges is no longer independent; no spanning subtree can contain both (5,6) and (7,11). Thus we utilize the thorem of Kirchoff, with

$$K = \begin{pmatrix}
0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 2 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 2 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 2 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 2 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}$$

and $|K_0| = 15$. There are 15 spanning out trees from 0, not 32—more than the 8 in the combinational out-cone, but significantly less than the 32 obtained from counting reconvergent nodes in the sequential out-cone as if they were independent.

It should be clear that the number of spanning out trees can be seen as a true measure of the reconvergence of r, more so than the counting method. With this in mind, we define the sequential reconvergence number, R^s of a vertex v in G as

$$R^s(v) = \log_k \det(K_r(v))/|G_r^s|,$$

where K is calculated on G_r^s, and k is the maximum in-degree (LUT-size) of the circuit G.

So the reconvergence of any node v is the logarithm of the number of spanning out-trees normalized by the size of the out-cone G_v^s. The purpose of taking the logarithm, as before, is to scale the number to within a comprehensible range for large graphs; this, with the normalization by the size of the out-cone, generates $0 \leq R^s(v) < 1$ for G mapped into 2-LUTs and $0 \leq R^s(v) < k$ for G mapped into k-LUTs.

Note that $R^s = 0$ if and only if the out-cone is already a tree.

To calculate the sequential reconvergence number of a graph we take, as in the combinational case, the weighted average of the reconvergence numbers of its primary inputs.

Note that the combinational reconvergence number of a sequential circuit is still well-defined. It is equivalent to performing the calculation on the circuit G with all back-edges removed or ignored. It is often, but not always, true that $R^c < R^s$; it depends on the

relative growth of the out-cone compared to the additional reconvergence in it.

Implementation Details.

Calculating the determinant of an n by n matrix uses $O(n^3)$ time. In CIRC we use a sparse-matrix implementation, which greatly decreases the required computation time. However, it is still not practical to calculate R^s for circuits with more than about 5,000 LUTs.

We take care to deal with the numerical stability of the determinant calculation with row pivoting, but above 2,000 LUTs, we sometimes encounter ill-conditioned matrices. CIRC will warn the user in these cases.

Empirical Calculations of R^c and R^s.

We calculated the combinational and sequential reconvergence numbers for all MCNC circuits. A sample of these is shown in Table 4.2. For comparison, we give R^c for both 2-LUT and 4-LUT mapped circuits, and R^s for 4-LUT mapped circuits.

We note, as in the combinational case, that any results here are biased by the contents of the MCNC benchmark set, which has limited documentation and could be missing large classes of logic. Thus our comments can only be based on the data that is available.

Observe that, as in the combinational case, there is a reasonable amount of grouping among the different types of logic. The arithmetic logic falls in the lower part of the spectrum, and finite state machines in the higher end. Within these bands, we notice, for example, the closeness of the reconvergence numbers for different implementations of multipliers, and for multipliers of different sizes. This data indicates that the reconvergence number is useful information, and captures some part of the fundamental nature of circuits. It would be very interesting to do these comparisons with greater information about the circuit functionality than we have, but the MCNC circuits have no documentation beyond these brief one-line descriptions.

There are several reasons that the finite state machines tend to have large reconvergence numbers: they often have very few I/Os, and their first sequential level often has a very exaggerated conical shape. Because of the small number of I/Os we often see that the sizes of the combinational and sequential out-cones are close, also explaining why they tend to have large R^c.

We point out the growth in the amount of reconvergence in a circuit as k increases, which

Name	R^c (k=2)	R^c (k=4)	R^s (k=4)	Circuit Description
elliptic	0.47	0.85	0.26	elliptic eqn solver
dsip	0.27	0.81	0.28	encryption
mult16b	0.36	0.56	0.30	16-bit multiplier
mult16a	0.54	0.62	0.36	16-bit multiplier
mult32a	0.54	0.61	0.36	32-bit multiplier
s208.1	0.38	0.39	0.45	digital fractional multiplier
s344	0.38	0.59	0.57	4-bit multiplier
ecc	0.66	0.96	0.58	error-correcting
lion	0.52	0.79	0.63	fsm
bbtas	0.76	0.84	0.73	fsm
traffic	0.50	0.73	0.78	fsm, traffic light
bigkey	0.53	0.89	0.83	key encryption
sbc	0.47	0.53	0.83	snooping bus controller
dk27	0.77	1.00	0.88	fsm
dk15	0.65	0.88	0.92	fsm
s382	0.63	1.04	0.92	fsm, traffic light
bbara	0.70	0.94	0.93	fsm
mm30a	0.69	1.12	0.95	min-max
mark1	0.58	0.76	1.03	fsm
s526n	0.63	1.04	1.03	fsm, traffic light
mm4a	0.66	1.12	1.04	min-max
tseng	0.49	0.78	1.04	bus-controller
keyb	0.72	0.99	1.14	fsm
opus	0.66	0.85	1.14	fsm
dk14	0.65	1.17	1.15	fsm
ph-dcd	0.61	1.02	1.19	phase decoder
diffeq	0.57	0.97	1.20	differential eqn solver
gcd	0.37	0.66	1.22	compute gcd
s832	0.65	0.90	1.22	fsm
bbsse	0.69	1.04	1.23	fsm
ex6	0.66	1.11	1.23	fsm
mm9b	0.68	1.15	1.23	min-max
sse	0.69	1.04	1.23	fsm
s820	0.67	0.96	1.24	fsm from a PLD
dk17	0.71	1.09	1.25	fsm
sand	0.77	1.06	1.25	fsm
styr	0.77	1.09	1.25	fsm
s510	0.77	0.75	1.33	fsm controller
dk512	0.80	1.39	1.34	fsm
s1	0.76	1.29	1.44	fsm
s1488	0.83	1.32	1.45	fsm controller
pma	0.80	1.32	1.46	fsm
s1494	0.82	1.37	1.47	fsm controller
planet	0.84	1.36	1.50	fsm
s298	0.84	1.60	1.60	fsm from a PLD
bbrtas	0.93	1.65	1.65	fsm

Table 4.2: Reconvergence for selected MCNC circuits.

is as one would expect: the number of nodes in an out-cone decreases, but the number of reconvergent paths remains unchanged (except when entirely "consumed" by a larger LUT).

There is a reasonably strong correlation between R^c and R^s, however not enough that one can predict the other. There are a number of cases where the two are drastically different. We reiterate our belief that R^s has a more theoretically pleasing value because of its combinatorial interpretation. However, as noted earlier, we can only effectively calculate it up to about 5,000 LUTs. Beyond that point R^c becomes the only available value.

Reconvergence and Routability

It is interesting to compare the routability of circuits with their reconvergence numbers. However, routability is (obviously) sensitive to both the number of nodes and the number of edges in the circuit so we need a large number of circuits which are very close in size. Such a subset does not exist in the MCNC circuits.

Some such experiments were possible with the Altera benchmark circuits, where we do have large numbers of similarly sized circuits. We find that R can be used in combination with other parameters to form a model of routability, but that any predictions are still dominated by other parameters which prevent us from isolating reconvergence. Further details of this particular study constitute proprietary information, but we leave the direction of research for future work.

The combinational characterization of circuits from Chapter 3, and a predecessor of the algorithm of Section 5.2 for combinational generation, though without locality characteristics, (GEN 1.0) first appeared in the 1996 Design Automation Conference [41]. This work since been submitted for journal publication [42]. The model of Chapter 4 for sequential circuits, excluding sequential reconvergence (Section 4.3) and the updated algorithm for sequential generation (GEN 3.0) was presented at the 1997 ACM Symposium on Field-Programmable Gate Arrays [39]. A journal version is in preparation. Sequential reconvergence (Section 4.3), the work on locality analysis (Section 3.5) and its effects on the generation algorithm have not yet been published outside of the thesis.

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.

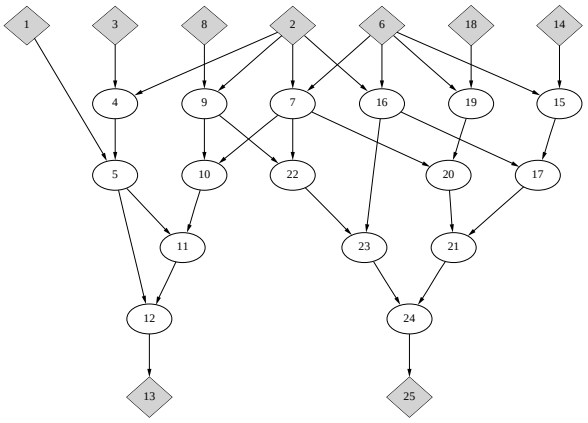
In the next section, Section 5.2, we discuss how to generate purely combinational circuits. We model combinational circuits using the descriptions of delay, shape and fanout from Chapter 3, and build combinational circuits to that model.

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 $N = \bigcup\{N_i\}$,

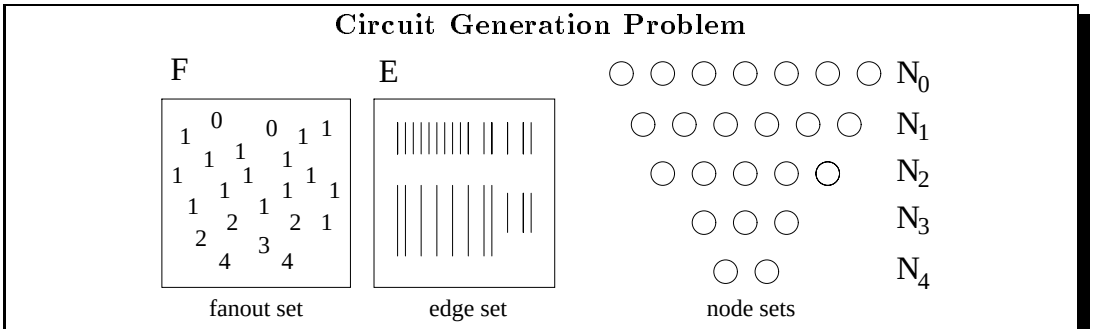
(b) $n_i = |N_i|$,

(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

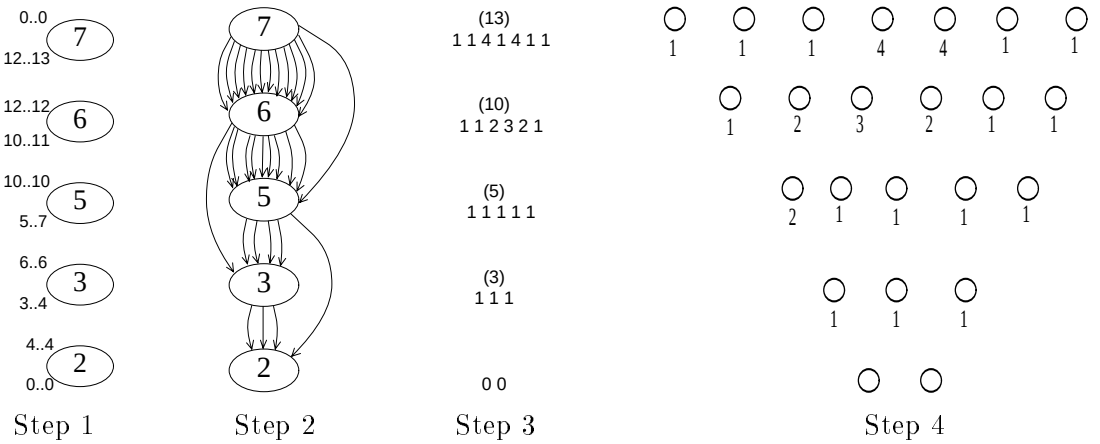


Figure 5.3: Example at the conclusion of Steps 1 to 4.

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

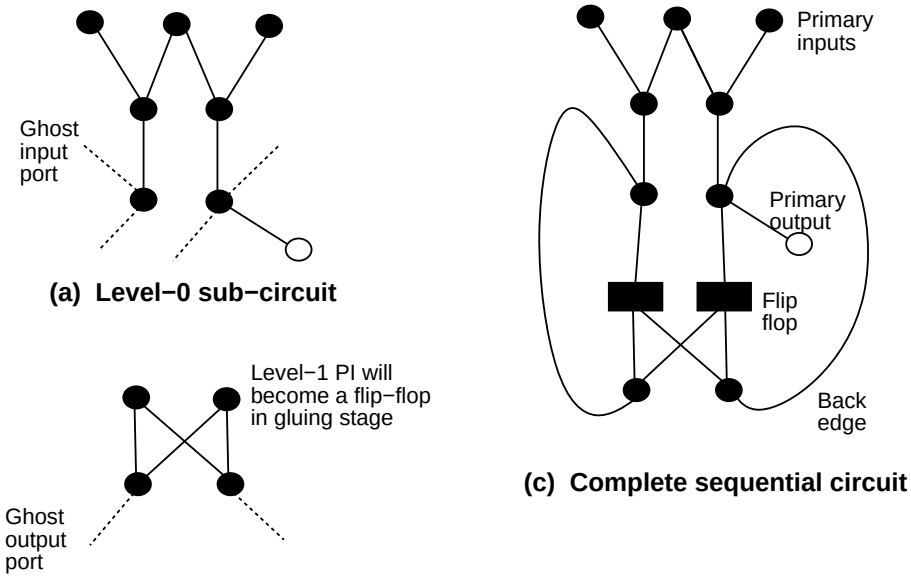


Figure 5.4: Example construction of a 2-level sequential circuit.

$0, n_{PI} = 3, n_{PO} = 1, n_{edges} = 6, n_{GI} = 2, n_{GO} = 2, n_{latch} = 2, shape = (3,2,2), GIshape = (0,0,2), GOshape = (0,0,2)\}$. 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

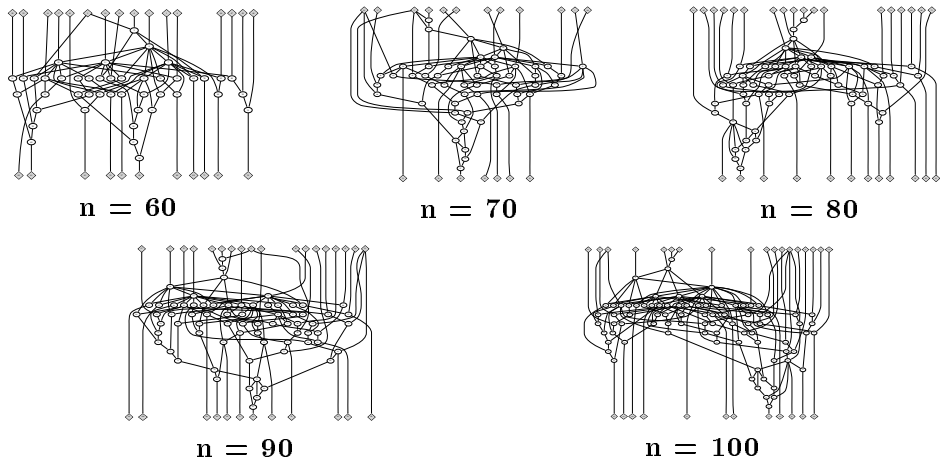


Figure 5.5: A GEN circuit family ({`k=2; n=60..100 by 10`}).

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.

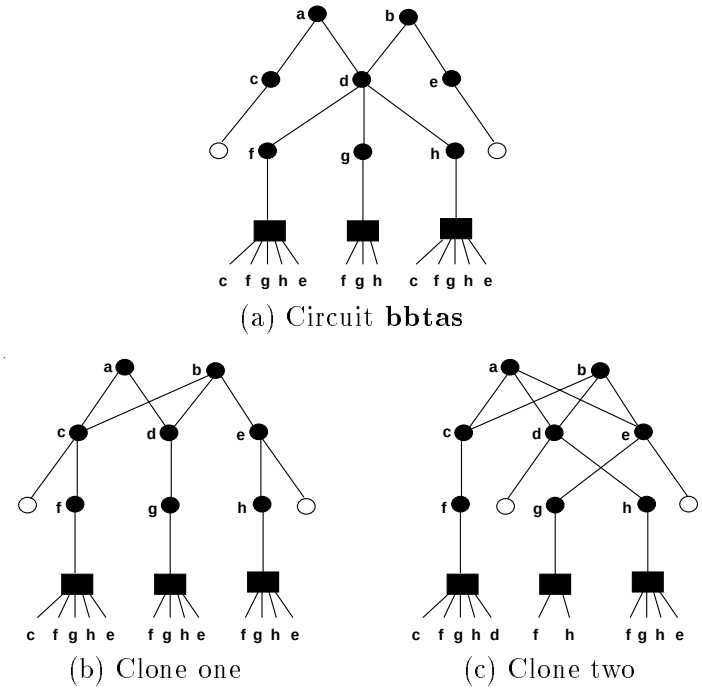
```
/* CIRC 3.1, compiled Wed Aug 28 15:36:17 PDT 1996. */
X = {
    name="bbtasclone";
    L0=(@.comb_circ) {
        name="L0";
        n=8; kin=4; nPI=2; nDFF=0; level=0; delay=2;
        shape=(2,3,3);
        nEdges=7; edges=(0,7,0);
        nGI=13; GIshape=(4,9,0);
        nGO=3; GOshape=(0,0,3);
        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.

Chapter 6

Validation of Circuit Quality

As discussed earlier, heuristic algorithms such as GEN are best compared on the basis of their actual results. The primary applications of the benchmark circuits produced by GEN are FPGA architectural exploration and software tools for computer-aided design. Thus, our method of validation will use well accepted metrics of routability to compare "real" benchmark circuits with clone circuits produced by GEN. Because the GEN algorithm contains a number of random and probabilistic techniques, it is also interesting to compare the GEN-circuits against standard random graphs of the same size.

In the case of combinational circuits, we use the MCNC benchmarks as our real circuits. For sequential circuits, the author was able to use industrial circuits provided by the Altera Corporation while employed there on an internship.

Our validation process is outlined in Figure 6.1. We take a real circuit, its clone circuit from GEN, and a random graph of the same size. These are individually placed and routed, and comparisons are made based on reconvergence number (from CIRC), track-count and wirelength (from VPR), and the "wiring resources" used on an Altera 10K20RC240 commercial FPGA (from MAX+PLUS2).

In Section 6.1 we show how to create reasonable random graphs for this comparison. Section 6.2 then gives a number of examples to visually indicate the differences between random graphs, GEN-circuits and real benchmarks, itself a form of validation. In Section 6.3 we discuss the empirical results for the combinational MCNC circuits, and in Section 6.4 we discuss empirical results for sequential industrial circuits.

Because GEN creates circuits using only a small parameter list, the goal is to show how

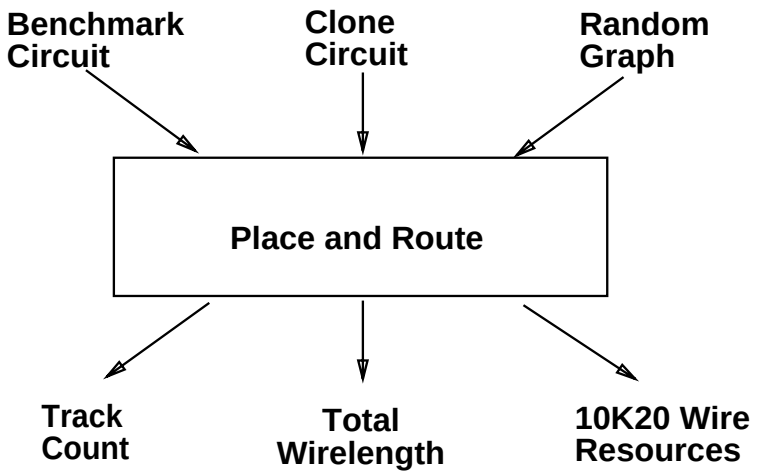


Figure 6.1: The validation process.

close they are to existing circuits. For a method such as proposed by Iwama *et. al.* [43] (See Section 2.2.3), where new circuits are generated by repeated small transformations or mutations, it would be equally important to show that the result was significantly *different*.

It is important to point out that the default parameterizations and the benchmark circuits produced by GEN are not at all restricted to the existence of an initial circuit to clone, other than for this validation process. We are able to generate benchmark circuits of up to 200,000 LUTs, well beyond the level of current FPGAs or ASIC circuits, but we can only *validate* the process up to the largest circuits in the MCNC and industrial collections, currently about 4500 LUTs.

6.1 Generating Comparison Random Graphs.

As mentioned earlier in Section 2.2.3, there are several natural models under which it is relatively easy to generate uniform random graphs. The most common model used is $G(n,p)$: a graph on n nodes where each edge exists independently with probability p. However, these graphs have either too many edges, or are disconnected (depending on p—see Section 2.2.3), so they are too unrealistic even to form a basis for comparison. The closest form of random graph that we can generate as a fair comparison is a random t-regular undirected graph, which we then force to be directed.

6.1.1 Random Directed Acyclic Graphs.

To generate a random graph the same size as a combinational circuit with n nodes and m edges, we calculate the largest t such that $t \cdot n < 2m$, generate a t-regular graph, then add the required number of leftover edges by random sampling. We direct the edges by taking a random ordering of the nodes and directing each edge from the lower to the higher numbered vertex.

A random t-regular graph can be generated as follows[1]:

1. Create a random permutation σ of size $2 \cdot t \cdot n$, to represent $2 \cdot t \cdot n$ nodes of a new graph (with no edges).

2. Join the nodes σ_{2i} and σ_{2i+1} with a new edge, $i = 0..(t \cdot n) - 1$. This creates a graph on $2 \cdot t \cdot n$ nodes with $t \cdot n$ edges, where each node is connected to exactly one other, i.e. a random matching.

3. Collapse (i.e. "identify") all nodes labeled $\sigma_{ti}..\sigma_{(t+1)i-1}$ into a single node x_i, for $i = 0..n - 1$.

The result of this process is an n node undirected graph where the degree of each node is exactly t. The algorithm does not, however, guarantee that the graph is *simple* (contains no double-edges or self-loops). The expected number of loops (edges from vertex v to itself) is given by

$$\begin{aligned}
\mu_1 &= \Pr(\text{edge is loop}) \cdot \text{edges in } G \\
&= \frac{\#\text{pairs which produce a loop}}{\#\text{ pairs}} \cdot \text{edges in } G \\
&= \frac{\binom{t}{2} \cdot n}{\binom{nt}{2}} \cdot \frac{nt}{2} \\
&= \frac{t(t-1)n}{(nt)(nt-1)} \cdot \frac{nt}{2} \\
&= \frac{t-1}{2} \qquad (n >> 1)
\end{aligned}$$

[1]Thanks to Mike Molloy [53] for showing me this construction and the analysis of it. The configuration model was introduced in this form by Bollobás[9] and motivated in part by the work of Bender and Canfield[7]. This model arose in a somewhat different form in the work of Bekessy, Bekessy and Komlós[6] and Wormald[71, 72].

and the expected number of double connections (multiple uv edges) is given by

$$\begin{aligned}
\mu_2 &= \frac{\text{possible double edges}}{\text{possible edge-pairs}} \cdot \text{edges in } G \\
&= \frac{\binom{n}{2}\binom{t}{2}\binom{t}{2} \cdot 2}{\binom{nt}{2}\binom{nt-2}{2}} \cdot \left(\frac{nt}{2} \cdot \frac{nt-2}{2} \right) \\
&= \frac{n(n-1)t^2(t-1)^2}{nt(nt-1)(nt-2)(nt-3)} \cdot \frac{nt(nt-2)}{4} \\
&= \frac{(t-1)^2}{4} \qquad (n >> 1).
\end{aligned}$$

It is quite interesting that the expected number of loops and double edges is a function purely of t, independent of n. The distribution of the events is Poisson, so the probability that a given G generated by the construction is loop-free, double-edge-free is $e^{-(\mu_1+\mu_2)}$. For $t = 5$ the probability that G is simple is 0.006. Thus we can expect to find a simple t-regular graph within a couple of hundred iterations. In practice, however, we can (and do) just delete the loops and multi-edges and choose new edges when adding the $m - tn$ other edges. Constructions due to Frieze [28] and McKay and Wormwald [51] allow this to be done without sacrificing perfect uniformity, but this is not necessary for our purposes.

For the direction of edges, we just use the (natural) ordering which comes from the random permutation σ. To add the extra edges, we uniformly choose a node with low fanin, uniformly choose a node from those with lesser numbers, and add an edge. We repeat this process until the number of edges in the graph is m.

One problem with these random graphs is that they have an overly high number of I/Os. For any random ordering of the nodes used to choose the edge directions, the probability that the i'th node x has all its edges directed forward (i.e. is a PI) is approximately $(\frac{i}{n})^t$, so the expected number of PIs is

$$\begin{aligned}
E[n_{PI}] &= \sum_{i=1}^{n} \left(\frac{i}{n} \right)^t \\
&= \frac{O(n^{t+1})}{n^t} \\
&= O(n).
\end{aligned}$$

Empirically, we calculate that for $t = 5$, about 8% of nodes are primary inputs, and by symmetry 8% are primary outputs.

6.1.2 Random Directed Graphs with Cycles.

For sequential circuits, we also want to have a given number of flip-flops and back edges. The introduction of back-edges also offers the opportunity to "repair" the I/O bias in the acyclic circuits.

We generate a random directed graph on n nodes and m 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-bounded fanin. The algorithm is as follows.

1. Generate a t-regular graph as in the combinational case.
2. Randomly label n_{PI} fanin-zero nodes as PI (similarly n_{PO} fanout-zero nodes as PO).
3. Randomly connect unlabeled fanout-zero and fanin-zero nodes by new edges until they are exhausted. 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 ignore this choice and restart the search for an alternate connection that does not involve a back-edge. This reduces the number of unwanted I/Os in the circuit, while also adding back edges and flip-flops.
4. Continue randomly connecting random nodes to random nodes with fanin less than k until the graph contains exactly m edges.

The graphs generated by this 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, and thus represents an extremely fair comparison of GEN circuits to "random graphs." 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.

6.2 Visual Validation: Examples.

For smaller circuits, we can observe the output of GEN pictorially. One command-line option of CIRC causes a DOT script to be output. The DOT program [47] takes this description of the graph and generates a drawing in postscript. We show a number of these drawings here. Further examples are shown in Appendix C.

6.2.1 GEN Circuits from Defaults.

Figure 6.2 shows four different combinational circuits produced by GEN using the default parameter distributions. We note that these circuits appear to be "normal" circuits, and include many features such as areas of high fanout. The visual "quality" of the circuits is most striking when one observes the similarity to MCNC circuits, shown in Figure 6.3, and the contrast between MCNC circuits and the random graphs shown in Figure 6.4.

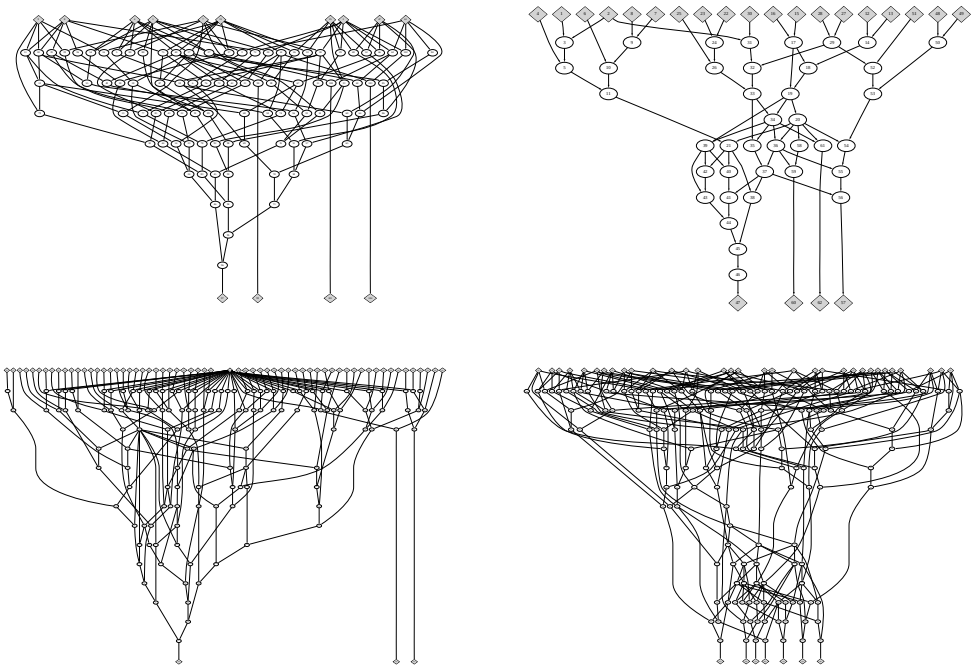


Figure 6.2: Varied circuits produced by GEN, using the default profile.

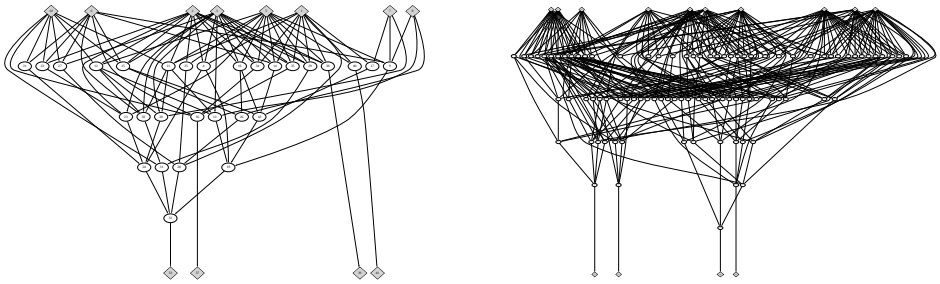


Figure 6.3: MCNC combinational circuits **sqrt8** and **sa02**.

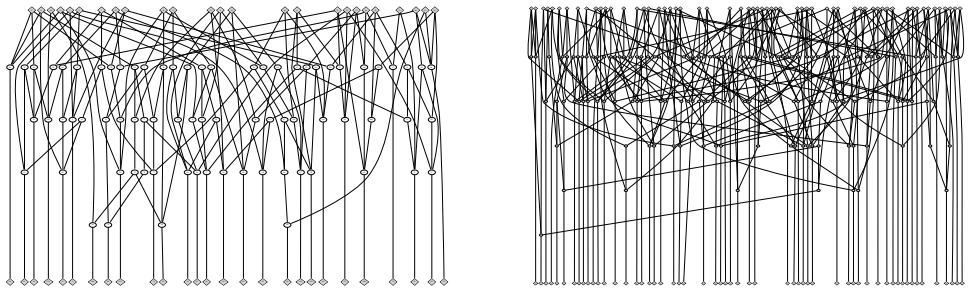
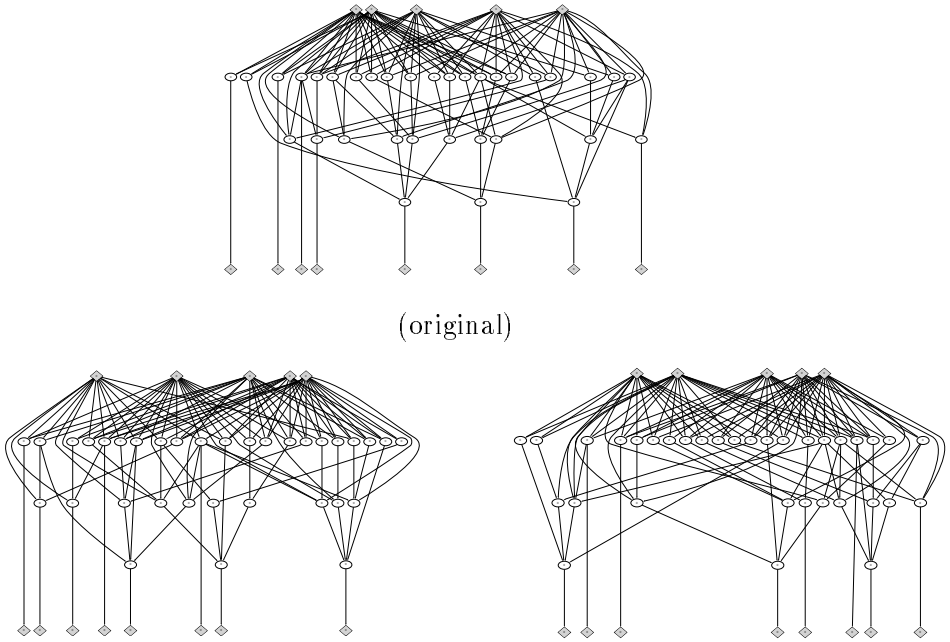


Figure 6.4: Random 4-regular digraphs

6.2.2 GEN Clone-Circuits.

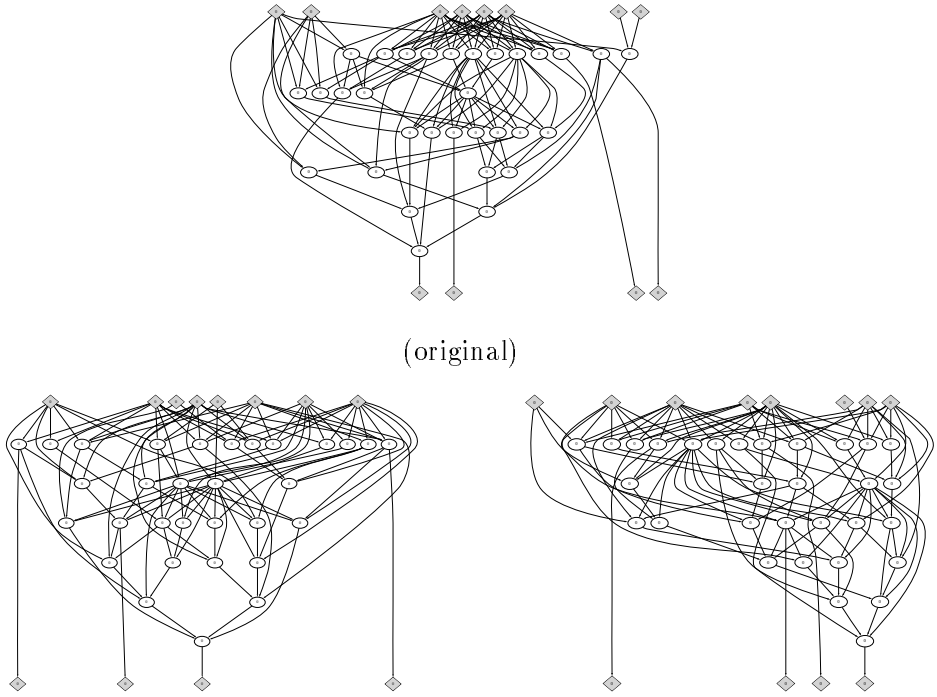
Figures 6.5 and 6.6 show two MCNC circuits, each original circuit pictured with two different clone circuits generated from its characterization by CIRC. Notice that the clones have a similar structure in terms of the parameters given to GEN, but are different in the implementation of that structure, just as they are different from the original.



(original)

Figure 6.5: MCNC combinational circuit **squar5** and two clone circuits from GEN.

Figure 6.7 shows the MCNC sequential circuit **dk15** and two clone circuits produced by GEN. Unfortunately, DOT is only designed to display directed acyclic graphs, so we are unable to automatically display graphs according to our sequential model. To generate



(original)

Figure 6.6: MCNC combinational circuit **sqrt8ml** and two clone circuits from GEN.

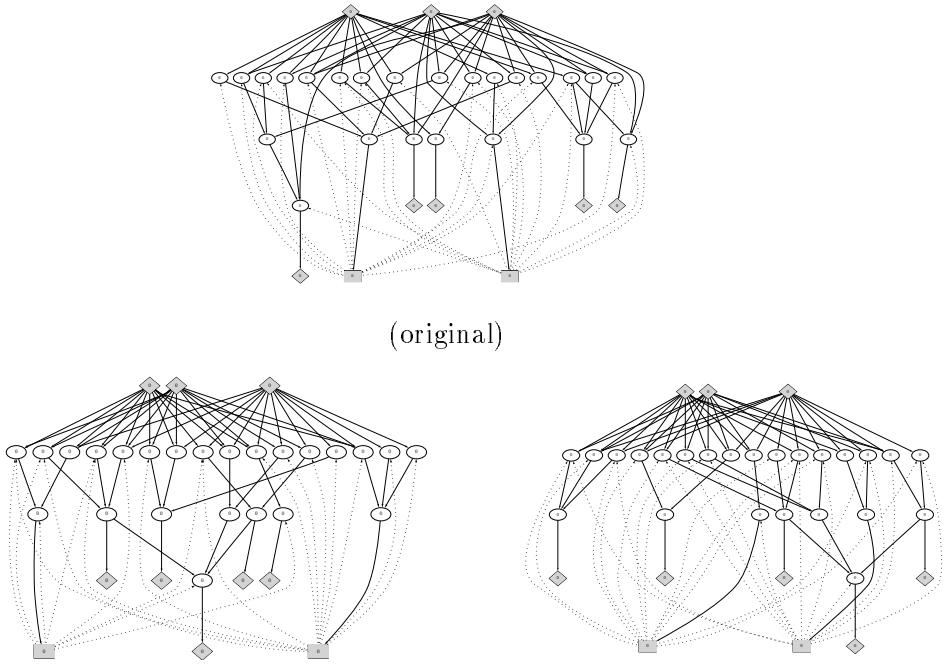
acceptable input for DOT, CIRC reverses all back-edges and gives instructions for DOT to display them as dotted in the drawing.

6.3 Combinational MCNC Circuits.

In this section we deal with the validation question for combinational circuits. We judge the quality of the generated circuits with respect to parameters not specified in generation: reconvergence, and post-placement and routing wirelength and track count. We note that a validation process for other characteristics such as node activity in simulation or timing analysis could also be performed; we leave this for future work.

We constructed the clone scripts (See Section 5.4.3) for 42 combinational MCNC circuits[2] with CIRC (i.e. n, n_{PI}, n_{PO}, d, shape, fanout and edge length distributions), and generated corresponding circuits meeting those profiles with GEN. Our method of validation is to compare unspecified characteristics of the MCNC circuits against those of the corresponding

[2]There are actually 109 combinational circuits in the LGSynth93 benchmark suite, but the majority are too small to be useful. We have restricted the experiments to circuits with 100 LUTs or more.



(original)

Figure 6.7: MCNC sequential circuit **dk15** and two clone circuits by GEN.

generated circuits and against random graphs of the same size (as discussed in the previous section).

Validating Reconvergence.

Reconvergence (from Section 3.4), R, is not a parameter to GEN. Reconvergence captures numerous properties of a circuit, including high fanout, and the interaction between shape, edge length and fanout distribution, all of which affect the ability to place and route the circuit. We calculated R for the generated circuits and compared them to those of the original circuits from which the generation profiles were extracted and to those of random graphs of the same size. The results for the MCNC circuits and their corresponding GEN-clones and random graphs are shown in Table 6.1. Recall that $0 \leq R \leq 2$ for 4-LUT mapped circuits.

We found that, for over half of generated circuits, R was within 0.1 of the value for the corresponding MCNC circuit. On average R differed by 22% in absolute value (if cancellation is allowed the difference is only 9%). This indicates that the correlation for an important descriptive parameter, R, did carry through the generation process.

		Reconvergence			Tracks			Wirelength		
	size	MCNC	GEN	RND	MCNC	GEN	RND	MCNC	GEN	RND
sao2	100	0.48	0.57	0.45	4	4	6	616	602	879
cht	102	0.10	0.17	0.10	3	3	5	353	445	572
9symml	106	0.41	0.57	0.44	4	4	7	606	582	867
C1355	115	0.80	0.56	0.21	5	4	6	677	655	825
C499	115	0.80	0.56	0.22	5	4	6	668	655	831
bw	137	0.67	0.66	0.67	4	4	9	842	794	1342
clip	149	0.59	0.63	0.79	4	4	9	978	896	1579
9sym	153	0.45	0.51	0.44	4	4	8	950	858	1424
C432	160	0.96	0.95	0.15	4	4	7	855	895	1347
rd84	165	0.53	0.78	0.60	5	4	9	1171	999	1927
o64	176	0.00	0.00	0.05	3	3	5	395	375	1204
C1908	178	0.84	0.95	0.28	5	6	8	1196	1249	1777
i3	178	0.00	0.00	0.05	3	3	6	332	344	1209
alu2	207	0.88	0.97	0.64	5	5	10	1425	1425	2591
i5	221	0.00	0.16	0.06	3	3	5	655	1180	1620
exmpl2	223	0.36	0.30	0.05	4	4	6	1053	1289	1523
toolrg	225	0.31	0.46	0.37	5	5	9	1520	1417	2494
t481	230	0.62	0.76	0.62	6	6	10	1763	1728	3071
C880	234	0.57	0.64	0.16	5	6	7	1419	1655	2233
duke2	273	0.56	0.56	0.36	6	5	10	2169	2008	3277
i2	275	0.02	0.06	0.02	3	3	6	727	716	2203
i4	290	0.00	0.01	0.03	3	3	6	592	639	2393
vda	305	0.72	0.77	0.55	7	5	12	2787	2557	4613
i6	320	0.24	0.21	0.05	3	3	7	1181	1262	2501
i7	402	0.20	0.20	0.03	3	3	6	1352	1403	4114
i9	464	1.07	0.72	0.22	5	5	12	2770	3072	6913
C3540	481	0.86	0.84	0.38	6	8	15	3726	4887	8321
cordic	489	0.80	0.89	0.39	7	7	15	4279	4859	8891
table3	494	0.73	0.87	0.49	8	6	15	5442	4847	8840
table5	500	0.78	0.86	0.39	8	7	15	5612	5018	9159
x3	512	0.26	0.24	0.08	4	5	10	3454	4289	7029
ex4p	514	0.41	0.25	0.23	4	5	12	3425	3914	8604
apex6	528	0.25	0.21	0.08	4	6	10	3217	4331	7115
C6288	559	0.90	1.16	0.45	4	8	16	2900	6207	10287
k2	559	0.60	0.60	0.18	7	7	14	5190	5191	9139
misex3c	563	0.53	0.63	0.37	6	5	15	4841	4493	10989
dalu	575	0.46	0.48	0.19	5	6	13	3827	4871	9547
i8	614	0.77	0.43	0.18	5	7	15	5729	6391	10181
apex1	740	0.67	0.56	0.36	8	7	19	8124	7725	15326
apex3	921	0.66	0.59	0.30	8	7	19	10658	9831	34423
C7552	945	0.53	0.45	0.05	5	6	13	5751	10384	15918
ex5p	1072	1.12	1.20	0.27	10	8	21	14343	12615	27904
i10	1252	0.72	0.55	0.09	6	8	19	15085	23915	28738
apex4	1270	0.90	0.69	0.23	9	8	23	16312	14279	34423
misex3	1411	0.55	0.77	0.24	8	7	24	16139	14799	40152
alu4	1536	0.50	0.62	0.22	7	6	26	15818	13561	45177
seq	1791	0.48	0.67	0.21	8	7	27	21348	19796	57040
des	1847	0.50	0.39	0.07	6	9	23	17898	33925	50294
apex2	1916	0.47	0.64	0.20	8	8	29	23203	22742	63418
spla	3706	0.97	1.07	0.13	10	9	19	49724	52583	167832
pdc	4591	1.01	1.27	0.10	11	10	19	74553	66131	225679
signed difference			9%	-45%		3%	123%		10%	119%
absolute difference			22%	48%		14%	123%		17%	119%

Table 6.1: Empirical validation using combinational MCNC circuits.

In contrast, the reconvergence numbers of the random graphs did not match the MCNC circuits well at all. We observe that these random graphs also exhibit diminishing R as n increases. This is partly due to the two factors mentioned earlier: the absence of high-fanout nodes and the large number of I/Os. Thus any generator which does not take these factors into account will fail to emulate crucial behaviour of real circuits.

Validating Routability.

To test the "routability" of our output circuits, we used a locally available tool, VPR [8], to place and global route the sets of MCNC circuits, generated circuits, and random graphs described above. The circuits are compared on two different metrics: the maximum number of tracks per channel required to successfully route, and the total wirelength of the global routing. VPR is a high-quality tool, currently the best academic global router available, so it provides a good quality solution for our comparisons.

VPR [8] chooses a minimal square grid to support the size of the circuit, and minimizes both maximum track-count per channel and total wirelength (by re-routing with successively fewer tracks per channel until failure occurs).

Table 6.1 also shows the routing statistics for the MCNC circuits, clones and random graphs with summary statistics (percentage pairwise differences) on the last line. We see that the track count for the generated circuits differed by 14%, on average, from the corresponding MCNC circuit, whereas the random graphs differed by 123%. Wirelength differed by 17% for the generated circuits and 119% for random graphs.

For both track-count and wirelength, we note that the variation for GEN clones lies in both directions whereas random graphs were universally harder to place and route. Thus, the *signed* differences for the GEN clones were only 3% in track-count and 10% in wirelength, meaning that the difference applies as much to the variance of GEN circuits as to an inherent specification bias. The random graphs, on the other hand, showed an obvious and consistent bias.

Though not shown in the table, we note that there is a corresponding increase in the cpu time required place and route the GEN circuits and random graphs, which is roughly proportional to the increase in wirelength (i.e. small for GEN circuits, and double or more for random graphs).

These results clearly show the circuits produced by GEN are very similar to the MCNC originals and significantly more realistic than random graphs as benchmark circuits.

Locality Revisited.

The above empirical results are all for the original method of producing locality—using the locality parameter L described in Chapter 5.

As mentioned in the description of the algorithm there, our hope is to eventually use a form of locality generation which is based on the locality characterization in Section 3.5. We are pursuing ongoing work to that end, and have made changes to GEN which use *spread* and *span* to parameterize edge connections in Step 5 of the generation algorithm.

Unfortunately, these efforts have not yet shown any numerical improvements over simply using the locality parameter L. There are several possible explanations for this. One is that, although the span of a node models the *distance* between nodes in the delay based layout, it does not model the interaction between edges, i.e. crossings. It is possible that edges are at the correct distance, but exhibit a balanced (rather than clustered) distribution of crossing numbers across horizontal slices of the layout, making the place and route problem more difficult.

Though the empirical results show that we already have an excellent method of producing circuits, it is theoretically displeasing to not tie the issue of locality characterization into the generation algorithm. For this reason, we feel that further characterizations of locality, especially the ability to parameterize locality generation in ways such as described in Section 3.5, are an important direction for ongoing and future work.

6.4 Sequential MCNC Circuits.

We validate the sequential GEN-circuits by generating clones of 22 industrial benchmark circuits (provided by the Altera Corporation), and comparing the post-placement and routing statistics from VPR and Altera's *max+plus2* for the original circuit with that of the clone circuit and a equivalently sized (in terms of nodes, edges, flip-flops and I/O) random graph.

The benchmark circuits[3] were output as BLIF after synthesis and fitting with Altera's commercial place-and-route tool MAX+PLUS2 into an Altera 10K20RC240 FPGA, and all analysis by CIRC, including the extraction of clone scripts, takes place from that point. Given industrial criticisms of the MCNC circuits, it is extremely useful to be able to compare our results with real industrial circuits.

Table 6.2 shows the comparison between the original, GEN and random circuits after placement and global routing by VPR and implementation on an Altera 10K20-RC240 FPGA [4] by MAX+PLUS2. The benchmarks used are all of the appropriate size (between 60 and

[3] Use of Altera circuits was made while the author was a summer intern there and had access to proprietary data and software.

	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 6.2: Empirical validation using sequential circuits from industry.

100% logic utilization, with most in the higher end of the range) for exercising this 10K20 part, which has 1152 LCELLS (logic blocks or LUT+FF combinations) and 240 user I/O pins.

The first column identifies the circuit. 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.

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" (see Section 2.1.2). Though all original circuits do fit in the 10K20, one of the clone circuits and thirteen 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, which means that the numbers for random circuits are deceptively low.

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 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 Section 4.3 we gave the definition of a measure quantifying generalized reconvergence for sequential circuits. By this measure, GEN circuits differ by about 0.19 on average, while random graphs differ by 0.28 on average. The difference in the average wirelength and track count between the original and clone circuits likely results from as yet unknown parameters. We hope to address the issue with future work on local structure in circuits.

These empirical results show that the GEN circuits are significantly more realistic than even carefully generated random graphs. Though not perfectly close, the GEN is able to generate circuits which are quite similar to the original benchmark circuits. We remark that, due to the proprietary nature of the circuits, we are not able to update the empirical results to take into account the new locality characterizations discussed in Sections 3.5, 5.2.2 and 6.3.

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.

Chapter 7

Conclusions and Future Work

7.1 Thesis Summary.

In this thesis we make new inroads into the understanding of digital circuits as graphs. We introduce a new method for dealing with the shortage of quality benchmark circuits for computer-aided design and for answering questions about FPGA architectures. The use of benchmarks is crucial for these applications because of their inherent heuristic or approximate nature.

Our approach to this problem involves first determining a combinatorial *characterization* of combinational and sequential circuits. We apply the new characteristics developed in this dissertation to form a statistical *profile* of circuits. Based on our abstract model of combinational and sequential circuits, we define the problem of *parameterized circuit generation* and give an algorithm to solve the problem. To bind this work together, we provide a method for the *validation* of benchmark circuit quality. In this validation process, we show both strong empirical evidence that the circuits produced by our software are good proxies for existing real benchmark circuits and that random graphs are not.

Using the methods developed here, we are able to generate large numbers of sequential benchmark circuits of up to 200,000 logic elements. The software implementation of the algorithm is fast, and can generate a circuit with 30,000 nodes, beyond the size of current FPGAs, in less than one minute of Sparc4 CPU time. The tools are practical and can output circuit netlists in the Berkeley BLIF format, or in other commercial formats such as Xilinx XNF [73], Altera AHDL/TDF [4], Actel ADL [1] and a subset of Verilog. The tools are of interest to industry as well as academia, and have already been used at a number of

companies.

7.2 Specific Contributions.

This thesis contributes to the state of knowledge in the following ways:

- We define a set of new statistical characteristics of combinational circuits: *shape*, *edge length* and *output distribution* and formalize a model and description of combinational circuits in terms of these and other parameters.
- We define a new theoretical combinatorial characterization of reconvergent fanout in both combinational and sequential circuits, and give an algorithm to extract the reconvergence parameter from a circuit.
- We define a characterization of "locality" in combinational circuits, and give an algorithm to efficiently extract locality information from a circuit.
- We define a new abstract model of sequential circuits, and a set of new characteristic parameters of sequential circuit graphs.
- Using existing benchmarks we form a *profile* of circuits in terms of the above characteristics. This profile is given in Appendix A, in the GEN specification language SYMPLE.
- We identify and formally define the problem of "parameterized random circuit generation" and set the ground rules for what type of generation tool is acceptable. We give a detailed algorithm for generating circuits using the combinational and sequential characteristics above, and incorporating the default profile just mentioned.
- We give a method of validating the quality of our benchmark circuits, and provide conclusive evidence both that this algorithm generates high quality circuits, and that random graphs produced by other means are not good for use as benchmarks.
- With the approach of this thesis, we provide a new methodological framework for approaching the design and analysis of heuristic algorithms, and the validation process for these algorithms. This paradigm is increasingly important for the algorithms

community as data sizes and execution time for hard problems continue to increase faster than algorithmic and machine speedups.

In addition, this thesis makes specific practical contributions to the community by providing two new freely-available software tools CIRC and GEN, together comprising about 50,000 lines of C source-code. CIRC is a tool for the analysis and extraction of all the circuit characteristics mentioned above. GEN implements the complete algorithm of Chapter 5, taking an input parameterization, in combination with the default profile, and producing a usable benchmark circuit which meets that parameterization. The code for CIRC and GEN can be obtained from the project web-site [40]. Copies of the source code have been downloaded under an academic license by more than 30 persons representing more than 20 companies and academic institutions, and have been installed by the author for use at Xilinx, Altera, Actel, and Hewlett Packard Corporations. Circuits produced by GEN have also been used in an academic partitioning competition held at the 1996 ACM/SIGDA Design Automation Conference.

7.3 Future Work.

The concept of circuit characterization and parameterized benchmark generation has not been studied before, and there are numerous ways in which it can be extended. We divide these into two areas: research into new understanding of circuits and better methods of generation, and suggestions for the practical improvement of the current CIRC and GEN implementations.

7.3.1 Further Research

The most interesting area of future research is to improve on the combinatorial characterization of locality in circuits. Rent's Rule provides us with a rough guide that we can apply on average, and our characterization of locality from Section 3.5 provides empirical data that we can apply directly to generation. A better understanding of local structure and hierarchy would provide new insights into all phases of computer-aided design, especially partitioning and placement. This knowledge would greatly help us with questions about how to properly scale the architectural features of FPGA architectures and whether to use flat or hierarchical architectures. Along with a study of locality, any other new

characterizations which provide us with knowledge about circuits would be useful.

We need to know more about how the structure of circuits changes with size. The circuits examined in this thesis range to 4,500 LUTs, about the boundary where circuits change from single-purpose computations to "system on a chip" designs. The next generation FPGAs will begin to implement these system level functions, and we must be prepared for them. A closely related issue would be how to validate circuits when we don't have any real circuits available to clone. It is difficult, however, to think of validating the routability and structure of a 200,000 LUT circuit when we have never seen one.

As circuits move towards system level designs, we will want to generate benchmarks which have multiple different types of logic in the same design. This means adding memory and datapath elements to the control logic that we currently generate. Our model of sequential circuits abstractly extends to an arbitrary form of hierarchy, but further investigations into how to "stitch" datapath, memory, or existing real circuits into GEN-circuits would be very interesting. Wilton [69, 70] has done investigations into the structure of reconfigurable memory for FPGAs; the marrying of this work with GEN would be particularly interesting now that FPGAs are starting to include large configurable memory blocks on-chip. Implementation of a more generalized hierarchy would also allow us to incorporate aspects aspects of Darnauer and Dai's Rent-based algorithm to the generation of very large control circuits with controllable partition trees.

7.3.2 Improvements for GEN.

An obvious effort for future work on GEN would be to fine-tune the parameters, the default profile, and the algorithm. The current profile is based on the MCNC circuits, and it would be beneficial to extend this to an empirical analysis of more, and more varied circuits. Such an exercise is as much political as academic, because it involves convincing competing vendors that there is mutual benefit to pooling their information.

Were more circuits available, it would be very useful to analyze different *types* of circuits separately. We roughly classified circuits as datapath or control logic in Chapter 2, but there are a number of distinct circuit classes within each of these: arithmetic, digital signal processing, state-machines and encryption, for example. Separate default scripts for each type of circuit would be useful.

The current version of GEN outputs a structural netlist, with all lookup-tables simply

programmed as nand gates. This means that we cannot guarantee the usefulness of circuits for the evaluation of synthesis and optimization tools. Trevillyan [67] has done some preliminary work to analyze the statistical contents of lookup-tables in technology mapped circuits, and it would be a good practical improvement to GEN to study this problem further and incorporate functionality into the output netlist.

References

[1] Actel, *Actel FPGA Data Book and Design Guide*, Actel Corp., 955 East Arques Avenue, Sunnyvale, CA 94086, 1996.

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

[3] C. J. Alpert and A. B. Kahng, *Recent Directions in Netlist Partitioning: A Survey*, Integration, the VLSI Journal, 12 (1995), pp. 1–81.

[4] Altera, *Altera Data Book*, Altera Corp., 2610 Orchard Parkway, San Jose, CA 95134-2020, 1996.

[5] G. Battista, P. Eades, R. Tamassia, and I. Tollis, *Algorithms for Automatic Graph Drawing: An Annotated Bibliography.* Tech. Report, Dept. of Computer Science, Brown University, Providence, RI. (Updated regularly.), 1989.

[6] A. Bekessy, P. Bekessy, and J. Komlós, *Asymptotic Enumeration of Regular Matrices*, Stud. Sci. Math. Hungar, 7 (1972), pp. 343–353.

[7] E. A. Bender and E. R. Canfield, *The asymtotic number of labelled graphs with given degree sequences*, J. Comb. Theory (A), 24 (1978), pp. 296–307.

[8] V. Betz and J. Rose, *Directional Bias and Non-Uniformity in FPGA Global Routing Architectures*, in IEEE/ACM International Conference on Computer Aided Design (ICCAD), 1996, pp. 652–659.

[9] B. Bollobás, *A Probabilistic Proof of an Asymptotic Formula for the Number of Labelled Regular Graphs*, Europ. J. Combinatorics, 1 (1980), pp. 311–316.

[10] S. D. Brown, *Routing algorithms and architectures for Field-Programmable Gate Arrays*, PhD thesis, University of Toronto, January, 1992.

[11] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic, *Field-Programmable Gate Arrays*, Kluwer, Norwell, Mass., 1992.

[12] P. K. Chan, M. D. F. Schlag, and J. Y. Zien, *On routability prediction for field-programmable gate arrays*, in Proc. 30th ACM/IEEE Design Automation Conference, 1993, pp. 326–330.

[13] J. Cong and Y. Ding, *FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs*, IEEE Trans. CAD, 13 (June, 1994), pp. 1–12.

[14] D. G. Corneil, H. Lerchs, and L. A. Stewart-Burlingham, *Complement reducible graphs*, Disc. Appl. Math, 3 (1981), pp. 163–174.

[15] 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 (FPGA96), Feb., 1996, pp. 66–72.

[16] W. E. Donath, *Statistical Properties of the Placement of a Graph*, SIAM J. Appl. Math, 16 (1968), pp. 439–457.

[17] ——, *Equivalence of Memory to Random Logic*, IBM J. Res. Dev., 18 (1974), pp. 401–407.

[18] ——, *Placement and average interconnection lengths of computer logic*, IEEE Trans. Comp., CAS-26 (1979), pp. 272–277.

[19] ——, *Wire length distribuition for placements of computer logic*, IBM J. Res. Dev., 25 (1981), pp. 152–155.

[20] ——, *Hierarchical structure of computers*, Tech. Rep. RC 2392, IBM T. J. Watson Research Centre, Yorktown Heights, N. Y. USA, March 1969.

[21] P. Eades, B. McKay, and N. Wormald, *On an edge-crossing problem*, in Proc. 9th Australian Computer Science Conference, 1986, pp. 327–334.

[22] P. Eades and N. C. Wormald, *Edge Crossings in Drawings of Bipartite Graphs*, Algorithmica, (1994), pp. 379–403.

[23] A. El Gamal, *Two-dimensional stochastic model for interconnections in master slice integrated circuits*, IEEE Trans. on Circuits and Systems, CAS-28 (1981), pp. 127–138.

[24] A. El Gamal and Z. A. Syed, *A New Statistical Model for Gate Array Routing*, in Proc. 20th ACM/IEEE Design Automation Conference, 1983, pp. 671–674.

[25] M. Feuer, *Connectivity of Random Logic*, IEEE Trans. Comp., C-31 (1982), pp. 29–33.

[26] C. M. Fiduccia and R. M. Mattheyses, *A linear time heuristic for improving network partitions*, in 19th ACM/SIGDA Design Automation Conference (DAC), 1982, pp. 175–181.

[27] R. J. Francis, J. Rose, and K. Chung, *Chortle: A Technology Mapping Program for Lookup Table-Based Field Programmable Gate Arrays*, in Proc. 27th ACM/IEEE Design Automation Conference, 1990, pp. 613–619.

[28] A. M. Frieze, *On Random Regular Graphs with Non-Constant Degree*, tech. rep., Research Report #88-2, Dept. of Mathematics, Carnegie Mellon Univesity, 1988.

[29] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, 1979.

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

[31] A. Gibbons, *Algorithmic Graph Theory*, Cambridge University Press, Great Britain, 1985.

[32] M. C. Golumbic, *Combinatorial merging*, IEEE Trans. Comput., 25 (1976), pp. 1164–1167.

[33] J. Greene, V. Roychowdhury, S. Kaptanoglu, and A. E. Gamal, *Segmented Channel Routing*, in Proc. 27th ACM/IEEE Design Automation Conference, 1990, pp. 567–592.

[34] L. Hagen and A. B. Kahng, *A New Approach to Effective Circuit Clustering*, in Proc. ACM/SIGDA Int'l Conference on Computer Aided Design (ICCAD), 1992, pp. 422–427.

[35] L. Hagen, A. B. Kahng, F. J. Kurdahi, and C. Ramachandran, *On the Intrinsic Rent Parameter and Spectra-Based Partitioning Methodologies*, IEEE Trans. CAD, 13 (1994), pp. 27–37.

[36] N. Harada, *A New Average Interconnection Length Prediction Method for Master-slice LSI*, in Proc. 19th ACM/IEEE Design Automation Conference (DAC), 1982, pp. 127–138.

[37] S. Hauk, G. Borriello, and C. Ebling, *Mesh Routing Topologies for FPGA Arrays*, in Proc. 2nd ACM/SIGDA Int'l Conference on FPGAs (FPGA94), 1994.

[38] H. J. Hoover, M. M. Klawe, and N. J. Pippenger, *Bounding fan-out in logical networks*, J. ACM, 31 (1984), pp. 13–18.

[39] M. Hutton, J. Rose, and D. Corneil, *Generation of Synthetic Sequential Benchmark Circuits*, in 5th ACM/SIGDA Int'l Symp. on FPGAs (FPGA97), 1997, pp. 149–155.

[40] M. D. Hutton, *CIRC/GEN Web-site.* http://www.eecg.toronto.edu/~mdhutton/gen/, 1997.

[41] 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.

[42] M. D. Hutton, J. S. Rose, J. P. Grossman, and D. G. Corneil, *Characterization and Parameterized Random Generation of Combinational Benchmark Circuits.* Submitted for journal publication.

[43] K. Iwama and K. Hino, *Random Generation of Test Instances for Logic Optimizers*, in Proc. 31st Design Automation Conference, 1994, pp. 430–434.

[44] K. Iwama, K. Hino, H. Kurokawa, and S. Sawada, *Random Benchmark Circuits with Controlled Attributes*, in To appear, Proc. 1997 European Design and Test Conference, 1997.

[45] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon., *Optimization by simulated annealing: An experimental evaluation (Part I).* Preprint, AT&T Bell Laboratories. Murray Hill, NJ, 1985.

[46] B. W. Kernighan and S. Lin, *An Efficient Heuristic Procedure for Partitioning Graphs*, Bell Systems Technical Journal, 49 (Feb., 1970), pp. 291–307.

[47] E. Koutsofios and S. C. North, *Drawing graphs with* dot, tech. rep., AT&T Bell Laboratories, Murray Hill, New Jersey 07974, USA, 1993.

[48] B. Krishnamurthy, *An improved min-cut algorithm for partitioning VLSI networks*, IEEE Transactions on Computers, C-33 (1984), pp. 438–446.

[49] B. S. Landman and R. L. Russo, *On a Pin Versus Block Relationship for Partitions of Logic Graphs*, IEEE Trans. Comp., C-20 (1971), pp. 1469–1479.

[50] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, Wiley, West Sussex, England, 1990.

[51] B. D. McKay and N. C. Wormald, *Uniform Generation of Random Regular Graphs of Moderate Degree*, J. Algorithms, 11 (1990), pp. 52–57.

[52] R. M. Meade and H. Geller, *Systems 360 Influence on the Design of Solid Logic Technology*, Solid State Design/Circuit Design Eng., (July 1965).

[53] M. Molloy, *Private communication.* University of Toronto.

[54] R. Murgai, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli, *Improved Logic Synthesis Algorithms for Table Look Up Architectures*, in Proc. Intl. Conf. on CAD, 1991.

[55] W. A. Notz, E. Schischa, J. L. Smith, and M. G. Smith, *Benefiting the System Designer*, Electronics, (Februrary 1967).

[56] Programmable Electronics Performance Corporation, *PREP PLD Benchmark Suite#1, V1.2.* 504 Nino Ave. Los Gatos, CA 95032. http://www.prep.org, 1993.

[57] C. E. Radke, *A justification of and an improvement on a useful rule for predicting circuit-to-pin ratios*, in Proc. 6th Annual SHARE/ACM/IEEE Design Automation Workshop, 1969, pp. 257–267.

[58] R. L. Russo, *On the Tradeoff Between Logic Performance and Circuit-to-Pin Ratio for LSI*, IEEE Trans. Comp., C-21 (1972), pp. 147–153.

[59] S. Sastry and A. C. Parker, *Stochastic Models for Wireability Analysis of Gate Arrays*, IEEE Transactions on Computer-Aided Design, CAD-5 (1986), pp. 52–65.

[60] M. D. F. Schlag, J. Kong, and P. K. Chan, *Routability-Driven Technology Mapping for Lookup Table-Based FPGAs*, IEEE. Trans. CAD, 13 (Jan., 1994), pp. 13–26.

[61] C. Sechen, *Average interconnection length estimation for random and optimal placements*, in Proc. IEEE International Conference on Computer Aided Design (ICCAD), 1988, pp. 190–193.

[62] E. M. Sentovich *et.al*, *SIS: A System for Sequential Circuit Analysis.* Tech. Report No. UCB/ERL M92/41. University of California, Berkeley, 1992.

[63] S. D. Shew, *A Cograph Approach to Examination Scheduling*, Master's thesis, University of Toronto, 1986.

[64] J. Spencer, *Ten Lectures on the Probabilistic Method*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1987.

[65] K. S. Sugiyama, S. Tagawa, and M. Toda, *Methods for visual understanding of hierarchical system structures*, IEEE Trans. Syst. Man, Cybern,, SMC-11 (1981), pp. 109–125.

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

[67] L. Trevillyan, *An Experiment in Technology Mapping for FPGAs using a fixed Library.*, in International Logic Synthesis (IWLS) Workshop, 1993.

[68] J. Varghese, M. Butts, and J. Batcheller, *An Efficient Logic Emulation System*, IEEE Trans. VLSI Systems, 1 (1993), pp. 171–174.

[69] S. J. E. Wilton, *Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memory*, PhD thesis, University of Toronto, 1997.

[70] S. J. E. Wilton, J. Rose, and Z. G. Vranesic, *Memory-to-Memory Connection Structures in FPGAs with Embedded Memory Arrays*, in 5th ACM/SIGDA Int'l Symp. on FPGAs (FPGA97), 1997, pp. 10–16.

[71] N. C. Wormald, *The Asymptotic Connectivity of Labelled Regular Graphs*, J. Comb. Theory (B), 31 (1981), pp. 156–167.

[72] ——, *The Asymptotic Distribution of Short Cycles in Random Regular Graphs*, J. Comb. Theory (B), 31 (1981), pp. 156–167.

[73] Xilinx, *The Programmable Gate Array Data Book*, Xilinx Inc., 2100 Logic Drive, San Jose, CA 95124, 1996.

[74] S. Yang, *Logic Synthesis and Optimization Benchmarks, Version 3.0*, tech. rep., Microelectronics Centre of North Carolina, P.O. Box 12889, Research Triangle Park, NC 27709 USA, 1991.

Appendix A

Default Parameterization Scripts

GEN has a number of command-line options, but the vast majority of a circuit parameterization is too complicated to be specified at the command line. GEN is augmented with a rich specification language called SYMPLE with which parameter programs or *scripts* can be written.

Whenever GEN is run, the defaults files are automatically read. This sets up the default frames `comb_circ` and `fsm_circ` which are then modified by the user's parameterization. The file "defaults.gen" must exist either in the current directory, or in the directory specified by the GENDIR environment variable.

1 A Brief Introduction to SYMPLE.

SYMPLE is a specification language, as opposed to a programming language. In that sense, it is more like VHDL than like a procedural language such as C. In particular, SYMPLE utilizes lazy evaluation, which means that no concept of temporal or procedural computation exists. Thus, a program such as:

```
x = 5;
output(x);
x = 6;
output(x);
```

is an error because it does not "make sense" to specify the signal x to have two different values. Similarly the statement "x = x + 1;" is an error.

SYMPLE has two main objects: A *cell* is largely analogous to a variable or a parameter, and a *frame* corresponds to a collection of cells. Used appropriately, a frame can also function as a subroutine.

Consider the following SYMPLE program, which covers most of the major concepts in frames:

```
Z = {
    a = 2;
    X = {
        a = 5;
        b = $.a + a;
    };
    out = X.b;
```

```
};
Y = (@.Z) { a = 4;};
output(Y.out);
```

The first thing to note is that *no* evaluation takes place until an **output** statement is parsed. At that point, the cell specified by the output statement must have been defined. To output Y.out, we require the frame Y, evaluated as a copy of frame Z below the top level frame (denoted "@") so Z is cloned (duplicated without evaluation). Then it is specified to modify the value 'a' in the new frame to be a 4 instead of its previous value. Now we can evaluate out, which is X.b. Within frame X, we have a=5 (note: different scope from the 'a' in the parent Y frame), and b = 5 + the parent's 'a' cell (which is 4), giving Y.X.b a value of 9. Thus the output of the program is 9.

The **eval** statement is also a command, which forces a value to be evaluated (fixed with a final value) but not output.

A SYMPLE program is parsed sequentially. As frames are defined as modifiers of previous frames, they are duplicated, and the new values specified. Then the frame is evaluated only when it is output.

Modified frames can function as a subroutine in which all parameters are optional. Note that a guarded parameter evaluation in a SYMPLE "subroutine" frame would look like the following:

```
Z = {
    a = 0;
    X = {
        out = 5 / $.a;
    };
    result = a==0 ? 0 : X.out;
};
V = Z { a = 4; };
output(V.result);
W = Z;
output(W.result);
```

which has the outputs "1.25" and "0."

SYMPLE has a set of library functions which are available to the user. Most of these are used in the obvious way, and the comb.gen defaults file is a good place to look for examples. Some are as follows:

```
/* A is taken from a Gaussian dist'n with mean 1.0546, std. dev .01 */
/* The selected value is then truncated to the range (0,2).         */
a = gauss(1.0546, .01, 0, 2);

/* Here we use standard C if...then expression syntax, and a min function */
IOmax = n < 100 ? n/3 : min(n/2, 600);

/* nearest integer, also floor and ceil, are available */
nEdges1 = nint((n-nIN)*avg_in);

/* the arity of max/min functions is infinite */
```

```
width_lower = max(wlower4a, wlower4b, wlower4c, wlower4d);

/* functions can be nested */
fmax_out = max(lower1, lower2, min(upper1, upper2, sample));

/* exp and log work with natural base */
locality = nint(2 + exp(log(n)/log(10))/exp(2));
```

Subroutines can be created to form data abstraction and hiding. For example, the we can have the following:

```
default_delay = {
    n = $.n;
    delay = nint(1 + gauss(1.2*log(n), 1, 1, n/3));
};
comb_circ = {
    n = 100;
    ...
    delayframe = (@.default_delay);
    delay = delayframe.delay;
    ...
};
```

Here we have defined a frame which evaluates delay, instantiated with a different value *each time we specify a comb_circ*, and taken comb_circ.delay as the default value for delay in comb_circ.

It is quite different to use the statement "delay = (@.default_delay).delay" rather than using the frame modifier, because this would force the delay parameter in the default_delay frame to be permanently evaluated, and we would always get the same Gaussian value from that point on in the execution of the program. In particular, if this was a hierarchical circuit then all sub-circuits sub-circuits would get have the same combinational delay.

Defining a circuit by the statement:

```
X = comb_circ {delay = 4; };
```

means that we have overridden the value of delay to be the *constant* 4 instead of the value evaluated in delayframe. This mechanism allows us to hide intermediate calculations in sub-frames. The easiest way to understand how things are being evaluated is to turn trace on (command-line option "trace") when running GEN, and to play with test scripts.

SYMPLE also has another construct with side-effects. The statement "`in("stuff.gen")`" specifies an include syntax, similar to #include in C. If there is an environment variable GENDIR, symple will look first in the current directory then in GENDIR for the include file.

2 GEN Combinational Defaults File.

```
/*
 *  GEN default script for combinational circuits.
 */

/*  $Revision: 3.0 $ */
```

```
/*  Determine a reasonable nPI, nPO.
 */
default_io = {
    n = -1;

    a = gauss(1.0546, .01, 0, 2);
    b = gauss(0.5627, .20, 0, .62);
    IOmin = 2 * log(n);
    IOmax = n < 100 ? n/3 : min(n/2, 600);
    nIO = nint(min(max(IOmin, exp(a + b * log(n))),IOmax));

    nPI = nint(gauss(1.1 * nIO/2, log(nIO), 2, nIO - 1));
    nPO = (nIO - nPI) > 0 ? (nIO - nPI) : nint(rand(1,nPI));
};


/*  Determine a reasonable delay.
 */
default_delay = {
    n = -1;
    kin = -1;
    nBot = -1;

    /* Have to have enough delay to collect terms */
    mindelay = log(log(n)) + ceil((log(n/nBot))/(log(kin)));

    delay4 = log(log(n)) + gauss(log(n), 1, 1, n/3);
    delay3 = delay4 * log(log(delay4));
    delay2 = delay4 * log(delay4);
    delay0 = kin==2 ? delay2 : kin==3 ? delay3 : delay4;

    delay = nint(max(delay0,mindelay));
};


/*  Determine a reasonable nEdges.
 */
default_num_edges = {
    n = -1;
    kin = -1;

    avg_in = kin==2 ? 2 : 2 + gauss((kin-2)/2, (kin-2)/5, .8, kin-2);
    nEdges1 = (n-nPI)*avg_in;
    lower = 2 * (n-nPI);
    upper = kin * (n-nPI);
    fEdges = min(max(nEdges1, lower), upper);

    nEdges = nint(fEdges);
};

/*  Default edge-length distribution.
 *
 *  Note that we can't do this in the num_edges frame, because we rely
 *  on nEdges as a parameter, and it could have been modified by the caller.
 */
default_edges = {
    n = -1;
    nEdges = -1;
    delay = -1;

    tot_edgelen = delay<=1 ? nEdges : nint(nEdges * (1 + gauss(0,.5,0,.5)));
    nComponents = nEdges;
    veclen = delay;
    nZeros = 0;
    min_nUnits = nint(max(n, nEdges/2));
    min_nMax = 0;
    /* dp_edges = rand(0.75,1.25) ; */
    dp_edges = gauss(1, .5, 0.75,1.25) ;

    edges = exp_dist(nComponents, tot_edgelen, veclen, 0, min_nUnits, min_nMax, dp_edges);
};


/*  Determine a reasonable max_out.
 */
default_max_out = {
    n = -1;
```

```
    kin = -1;
    nEdges = -1;
    nPI = -1;
    nBot = -1;
    delay = -1;

    /* log-linear version */
    a1 = gauss(-1.07181, 0.5, -2, 1);
    b1 = gauss( 0.8242,  0.5, .7, .9);
    sample1 = exp(a1 + b1 * log(n));

    /* linear function of sqrt version */
    a2 = gauss(2, 1, 1, 5);
    sample = a2 * sqrt(n);

    deflate = rand(1,8);
    upper1 = 512/deflate;
    upper2 = n-nPI;
    lower1 = n-nPI == 0 ? 1 : nEdges / (n-nPI);

    /* have a lower bound based on nPI and possible width */
    nInternal = n - nPI - nBot;
    w3a= delay==5 ? nInternal-kin*nBot-kin*kin*nBot-kin*kin*kin*nBot:0;
    w3b= delay==4 ? nInternal-kin*nBot-kin*kin*nBot : 0;
    w3c= delay==3 ? nInternal-kin*nBot : 0;
    w3d= delay==2 ? nInternal : 0;
    w3e= delay >5 ? (2.5 * nInternal) / delay : 0;
    width_upper = max(w3a, w3b, w3c, w3d, w3e);

    lower2 = width_upper / nPI;

    upper = min(upper1, upper2);
    lower = max(lower1, lower2);

    fmax_out = max(lower, min(upper, sample));
    max_out = nint(fmax_out);
};


/*  Determine a reasonable out-degree sequence
 */
default_outs = {
    n = -1;
    nEdges = -1;
    max_out = -1;
    nZeros = -1;

    dp_outs = rand(0.75,1.25) ;
    outs = exp_dist(n, nEdges, max_out, nZeros, 0, 1, dp_outs);
};


/*  Determine reasonable shapes for POs.
 */
default_IOShape = {
    delay = -1;
    nBot = -1;
    nPO = -1;

    minPOBot = nBot;
    maxPOBot = min(nBot, nPO);
    POBot = nint(rand(minPOBot, maxPOBot));

    POSlope = 1;
    POlen = delay;
    POshape = delay==0 ? bi_linear(nPO, 0, 0, 0, 0, 0)
        : bi_linear(nPO, 0, POBot, POlen, delay, POSlope);
};


/*  Determine reasonable maximum width for the shape profile.
 */
default_width = {
    n = -1;
    kin = -1;
    nTop = -1;
    nBot = -1;
    delay = -1;
```

```
    max_out = -1;
    expand = -1;

    nInternal = n - nTop - nBot;
    upper1 = (delay <= 2) ? nInternal : (nInternal / 2);
    upper2 = expand * expand * expand * expand * nTop;
    upper3 = nint(n / (delay / 2));

    upper4a= delay==5 ? nInternal-nTop-kin*nBot-kin*kin*nBot-kin*kin*kin*nBot:0;
    upper4b= delay==4 ? nInternal-nTop-kin*nBot-kin*kin*nBot : 0;
    upper4c= delay==3 ? nInternal-nTop-kin*nBot : 0;
    upper4d= delay==2 ? nInternal : 0;
    upper4e= delay>5 ? n : 0;        /* no upper4 if have high delay */
    upper4 = max(upper4a, upper4b, upper4c, upper4d, upper4e);

    lower1 = nInternal / (delay - 1);
    lower2 = max(nBot, nTop);

    lower = max(lower1, lower2);
    upper = max(lower, min(upper1, upper2, upper3, upper4));

    mean = (lower + (3 * upper)) / 4;
    fwidth = gauss(mean, 2*sqrt(mean), lower, upper);

    width = (delay <=2) ? max(nTop, nBot) : nint(fwidth);
};


/*  Default shape profile.
 */
default_shape = {
    n = -1;
    kin = -1;
    delay = -1;
    nTop = -1;
    nBot = -1;
    jumps = -1;
    expand = -1;
    width = -1;

    shape = rand_shape(n, kin, delay, nTop, nBot, width, jumps, expand);
};


/*
 *  A combinational circuit.  We have the basic set of parameters,
 *  with additions for fanout, shape, edges and output distributions.
 */
comb_circ = {

    name = "C";
    n = 0;
    kin = 4;
    global_max_out  = -1;        /* passed from fsm.gen or above */

    loc1 = log(n)/log(2);
    loc2 = 2 * sqrt(n)/5;
    loc3 = 2 * exp(log(n)/log(10));

    locality = ceil(max(loc1, loc2));

    /* Circuit is complete comb_circ unless overridden */
    level = 0;
    nLatch = 0;

    /* Choose distribution of PI and PO from defaults */
    IOFrame = (@.default_io) { n=$.n; };
    nPI = IOFrame.nPI;
    nPO = IOFrame.nPO;
    nOUT = nPO + nint(3*log(n));
    nIO = nPI + nOUT;

    /* Number of nodes at the last level of the shape profile */
    nBot = nOUT == 1 ? 1 : nint(gauss(nOUT/2, nOUT/2, 1, nOUT));

    /* Choose number of edges from defaults */
    nEdgesFrame = (@.default_num_edges) {
        n=$.n; kin=$.kin; nPI=$.nPI;
```

```
    };
    nEdges = nEdgesFrame.nEdges;

    /* Choose delay from defaults */
    DelayFrame = (@.default_delay) { n=$.n; kin=$.kin; nBot=$.nBot; };
    delay = DelayFrame.delay;

    /* Choose edge-distribution from defaults */
    edgesFrame = (@.default_edges) {
        n=(($.n) - ($.nPI)); nEdges=$.nEdges; delay=$.delay;
    };
    edges = (delay == 0) ? (0) : edgesFrame.edges;

    /* Choose the maximum out-degree */
    MaxOutFrame = (@.default_max_out) {
        n=$.n; kin=$.kin; nEdges=$.nEdges; nPI=$.nPI;
        nBot=$.nBot; delay=$.delay;
    };
    max_out = global_max_out < 1
        ? MaxOutFrame.max_out
        : min(global_max_out, MaxOutFrame.max_out);

    /* Choose out-degree distribution */
    minzeros = max(0, n - nEdges + max_out);
    nZeros = max(minzeros, nint(rand(nBot, nOUT)));
    outsFrame = (@.default_outs) {
        n=$.n; nEdges=$.nEdges; max_out=$.max_out; nZeros=$.nZeros;
    };
    outs = outsFrame.outs;

    /* Choose the shape of I/O things -- PO */
    IOShapeFrame = (@.default_IOShape) {
        delay=$.delay; nBot=$.nBot; nPO=$.nPO;
    };
    POshape = IOShapeFrame.POshape;

    /* Choose the shape profile */
    jumps = delay<=3 ? 0 : nint(rand(0, delay/3));
    _expand = kin * gauss(3.5, sqrt(3.5), 1, 7);
    expand = max(1, min(_expand, max_out/2));
    widthFrame = (@.default_width) {
        n=$.n; kin=$.kin; nTop=$.nPI; nBot=$.nBot; delay=$.delay;
        max_out=$.max_out; expand=$.expand;
    };
    width = widthFrame.width;
    shapeFrame = (@.default_shape) {
        n=$.n; kin=$.kin; delay=$.delay; nTop=$.nPI; nBot=$.nBot;
        jumps=$.jumps; expand=$.expand; width=$.width;
    };
    shape = shapeFrame.shape;
};
```

3 GEN Sequential Defaults File.

```
/*
 *  GEN default script for sequential circuits.
 */

/*  $Revision: 3.0 $ */


/*
 *  Frames to define a FSM circuit, and utility frames to determine
 *  parameters for fsm_circ.
 */


/*  The number of I/Os to a fsm can be smaller than for comb, because of
 *  all of the ghost I/Os.
 */
default_fsm_IOs = {
    n = -1;
```

```
    combIOFrame = (@.default_io) { n=$.n; };
    nPI1 = combIOFrame.nPI;
    nPO1 = combIOFrame.nPO;

    nPImin = max(2, nint(nPI1/4));
    nPImax = nPI1;
    nPI = nint(rand(nPImin, nPImax));

    nPOmin = max(2, nint(nPO1/4));
    nPOmax = nPO1;
    nPO = nint(rand(nPOmin, nPOmax));
};


default_seq_shape = {
    n = -1;
    nDFF = -1;

    /* Choose number of FFs to have */
    _nDFFmin = max(2, n/50);
    _nDFFmax = n/10;
    _nDFFmean = n/20;
    _nDFF0 = nint(gauss(_nDFFmean, sqrt(_nDFFmean), _nDFFmin, _nDFFmax));
    _nDFF = nDFF>0 ? nDFF : _nDFF0;

    n0 = nint(rand(.6, .85) * n);
    n1 = n - n0 + _nDFF;
};



/*
 *  The definition of a generic finite-state machine.
 *
 *  Things supposed to be parameters:
 * name, n, n0, n1, kin, startlevel, max_out, nGI, nGO, nPI, nPO,
 * nDFF, nDFF0, nBack, locality, delay, avg_in.
 */
fsm_circ = {

    name = "S";
    n = 100;
    kin = 4;
    startlevel = 0;
    levels = 1;
    max_out = -1;

    nGI = 0; nGO = 0;
    nDFF = nint(rand(n/20, n/5)); /* usually overridden */

    locality = nint(6 + exp(log(n)/log(10))/exp(2));

    /* Choose an overall max-delay for the circuit */
    fake_nOUT = nPO + nint(min(nGO, 3*log(n)));
    fake_nBot = fake_nOUT == 1 ? 1
      : nint(gauss(fake_nOUT/2, fake_nOUT/2, 1, fake_nOUT));
    delayFrame = (@.default_delay) { n=3*($.n)/2; kin=$.kin; nBot=$.fake_nBot;};
    delay = delayFrame.delay;

    IOFrame = (@.default_fsm_IOs) { n=$.n; };
    nPI = IOFrame.nPI;
    nPO = IOFrame.nPO;

    shapeFrame = (@.default_seq_shape) { n=$.n; nDFF=$.nDFF; };
    n0 = shapeFrame.n0;
    n1 = shapeFrame.n1;

    avg_in_mean = 2*(kin-2)/5;
    avg_in_sd = (kin-2)/5;
    avg_in = kin==2 ? 2 : 2 + gauss(avg_in_mean, avg_in_sd, .5, kin-2.5);
    nEdges1 = (n-nPI-nDFF)*avg_in + nDFF - nGI;
    lower = 2 * (n-nPI-nDFF) - nGI/2;
    upper = kin * (n-nPI-nDFF) - nGI;
    fEdges = min(max(nEdges1, lower), upper);
    nEdges = nint(fEdges);

    /* Will build 2 circuits.  L0, L1 */
```

```
    /* Changed July 30: nBack = nint(rand(n1/20, n0 - nPI)); */
    nBackMean = n0/10;
    nBackStdDev = sqrt(n0);
    nBackMin1 = n1 < 500 ? n1/10 : 2 * log(n1);
    nBackMin2 = delay <= 3 ? n1/kin : 0;
    nBackMin = max(nBackMin1, nBackMin2);
    nBackMax= min(n0 - nPI, 5*n1);
    nBack = nint(gauss(nBackMean, nBackStdDev, nBackMin, nBackMax));

    /* Divide up the edges */
    n1min = (n1-nDFF)*2;
    n0min = (n0-nPI)*2;
    _usable_edges = nEdges - nBack - n0min - n1min;
    L0edgesmax = (n0-nPI)*kin - nBack;
    L1edgesmax = (n1-nDFF)*kin;
    ratio = L1edgesmax  / (L0edgesmax + L1edgesmax);

    _L1edges = _usable_edges < 1 ? n1min
           : n1min + nint(ratio * _usable_edges);
    L1edges = min(_L1edges, L1edgesmax);

    _L0edges = _usable_edges < 1 ? n0min
    : n0min + (_usable_edges - nint(ratio * _usable_edges));
    L0edges = min(_L0edges, L0edgesmax);

    L0delay = delay;
    L0BotMax = min(nDFF + nPO, (n0-nPI)/(delay-1));
    L0BotMin = min(max(nDFF/5, log(n0)), L0BotMax);
    L0Bot = nint(rand(L0BotMin, L0BotMax));
    L0Zeros = nint(rand(L0Bot, L0BotMax));

    _L1delay_min = max(1, log(log(n1-nDFF)));
    _L1delay_max = max(_L1delay_min, min(delay, log(n1-nDFF)));
    _L1delay_mean = (_L1delay_max + _L1delay_min) / 2;
    _L1delay_sd = sqrt(_L1delay_mean);
    _L1delay = gauss(_L1delay_mean, _L1delay_sd, _L1delay_min, _L1delay_max);
    L1delay = ceil(_L1delay);

    L1BotMin = L1delay <= 3 ? nint((n1-nDFF)/kin) : nint(log(n1));
    L1BotMax =  max(L1BotMin, (n1-nDFF)/(2*L1delay));
    L1BotMean = (L1BotMax + L1BotMin) / 2;
    L1Bot1 = nint(gauss(L1BotMean, sqrt(L1BotMean), L1BotMin, L1BotMax));
    L1Bot = min(L1Bot1, nBack);
    L1Zeros = nint(rand(L1Bot, min(L1BotMax,nBack)));
    back_shape = bi_linear(nBack, 0, L1Bot, L1delay, L0delay, 0);

    L0 = (@.comb_circ) {
        name = ($.name) + "L0";
        level = $.startlevel;
nPI = $.nPI;
nPO = $.nPO;
nLatch = $.nDFF;
delay = $.L0delay;
        n = ($.n0);
        kin = ($.kin);
        nGI = ($.nBack);
        nGO = $.nDFF;
nBot = ($.L0Bot);
nZeros = ($.L0Zeros);
        GIshape = $.back_shape;
locality = $.locality;
global_max_out = $.max_out;
nEdges = $.L0edges;
    };

    L1 = (@.comb_circ) {
        name = $.name + "L1";
        level = $.startlevel + 1;
        kin = ($.kin);
delay = $.L1delay;
        n = $.n1;
        nLatch = 0;
   nPI = $.nDFF;
        nPO = 0;
        nGI = 0;
        nGO = $.nBack;
nBot = $.L1Bot;
nZeros = ($.L1Zeros);
```

```
        GOshape = $.back_shape;
locality = $.locality;
global_max_out = $.max_out;
nEdges = $.L1edges;
    };

    glue = (L0,L1);
};
```

4 GEN Special-Circuit Defaults File.

```
/*
 *  GEN default script for "special" circuits.
 */

/*  $Revision: 3.0 $ */

/*
 *  Special types of circuits:
 */

/*
 *  Generates an empty circuit.  The reason we want this is so that we
 *  can have parameterized gluing.  Gluing a null circuit on to any
 *  other circuit does nothing to it.
 */
null_circuit = (@.comb_circ) {
    name = "null";
    level = 0; delay = 0;
    n = 0; nPI = 0; nPO = 0; nIN = 0; nOUT = 0;
    nEdges = 0; max_out = 0;
    nBot = 0; nZeros=0;
    locality = 0;

    junk = kin == 1 ? 0 : bi_linear(0, 0, 0, 0, 0, 0);
    GIshape = junk;
    GOshape = junk;
    POshape = junk;
    shape = junk;
    edges = junk;
    outs = junk;
};


/*
 *  A circuit with nDFF FFs and no nodes.
 *  Expectation is that the user will define a circuit
 *   X = (@.register_file) {nDFF=16; nGO=32;};
 *  and then glue it to the end of something else.  Note that specifying
 *  nGO==nDFF (the default) results in a deterministic circuit -- nDFF FFs
 *  with one GO each.  If nGO < nDFF, an error will ensue during generation.
 */
register_file = (@.comb_circ) {
    name = "RF1";
    level = 1;
    nDFF = 0; /* must override */
    n   = nDFF;
    nIN = nDFF;
    nGO = nDFF;
    nGI = 0;
    delay = 0;
    nEdges = 0;
    nPI = 0;
    nPO = 0;
    max_out = nint(rand(1, nGO-n));

    junk = bi_linear(0, 0, 0, 0, 0, 0);
    GIshape = junk;
    GOshape = junk;
    POshape = junk;
    shape = junk;
    edges = junk;
```

Appendix B

Abbreviated User's Guide.

1 Overview

This document introduces two tools, CIRC and GEN.

The first tool, CIRC reads an input netlist and performs analysis upon it, outputting either statistical information, or acting as a filter to convert the netlist to an alternative format.

The second tool, GEN, takes a parameterization of a circuit as a program written in the SYMPLE language and creates a netlist which corresponds to the parameterization program.

Though GEN and CIRC are separate tools, their usage is highly related. Many of the most useful products of the research from which they arise is the interaction between the characterization of a circuit and the subsequent generation of a similarly parameterized circuit. Thus, it is more appropriate to document their usage in a single document.

This user's guide is organized as follows. Section 2 discusses the "characteristics" of a circuit. These characteristics then form the basis for the output of CIRC and for the input to GEN. Section 3 describes how to use CIRC to analyze or filter a circuit. Section 4 describes basic usage of GEN to create combinational and sequential circuits. Section 5 discusses more advanced usage of GEN such as the problems involved in modifying existing scripts or scripts from CIRC.

2 Circuit Characteristics

This section defines the terms which will be used throughout the document to describe characteristics and parameters of circuits.

The most basic parameters of a circuit are the following:

name The filename in which the netlist is stored. CIRC will look for name.blif, name.blf, or $MCNCDIR/k/name.blif.

k The lut-size (maximum fanin) of the design.

size The size of a circuit. The size of a circuit is defined as the number of "countable functional nodes" in a graph-theoretic sense, hence it is the sum of the number of (see below) PIs (primary inputs), LUTs (or logic nodes), and DFFs (flip-flops). Primary outputs are not counted, because we consider this to be an attribute rather than a separately named node.

nPI The number of primary inputs designated in the .inputs line of the input or output netlist.

nPO The number of primary inputs designated in the .inputs line of the input or output netlist.

nDFF The number of D-type flip-flops which are defined in the input or output netlist. Currently the only type of sequential logic element which is understood by CIRC and GEN is the DFF with no defined preset or clear.

nEdges The number of edges in the circuit-graph. Equivalently either the sum over all nodes x of fanin(x), or the sum over all nodes of fanout(x).

Currently the tools recognize only a *single* clock. In the case of CIRC this means that all clock-inputs are ignored, and replaced by a single primary-input called "clock," effectively forcing all DFF to use the same clock regardless of the design specification. Similarly, GEN will output circuits of the same form.

nCC The number of connected components: essentially the number of completely separate circuits which are defined in the same file. This value is output by CIRC but GEN will only output circuits which are fully connected (one component).

unusable nodes The number of nodes which do not affect any PO in an input design. These are deleted by CIRC before processing, and should not every be produced by GEN.

unreachable nodes The number of nodes which (recursively) cannot be reached from a PI, and hence will never have a logical value. These are also deleted by CIRC before processing, and should not be produced by GEN.

The basic element in CIRC and GEN processing is the combinational circuit, using the combinationl delay of the circuit as an important point of reference. Thus we have several items defined on the basis of combinational delay.

delay Combinational delay is defined, for all nodes x in a circuit, as follows: delay(x) = 0 if x is either a PI or a DFF. delay(x) = 1 + MAX{delay(y)}, for all fanins y to x; essentially a standard unit-delay model of combinational delay. The delay of a circuit is then the maximum combinational delay over all nodes x in the circuit.

shape The combinational shape of a circuit is defined as the distribution of node combinational delays. It is vector of length delay + 1 (0..delay). In a purely combinational circuit, shape[0] is necessarily nPI, and shape[delay] is no more than nPO (though it need not be nPO, because nodes of earlier delay can be designated as POs). Thus a shape of [4, 8, 3, 2] specifies a circuit with 4 PIs, 8 nodes which have inputs only from the PIs, 3 which have at least fanin from delay level 1, and 2 which have at least one fanin from level 3.

nBot The number of "bottom" nodes in the shape distribution. Though nBot is redundant information in general, it is referred to in various places, and is used as an intermediate calculation in the creation of a default/random shape vector by GEN.

POshape In the same way that shape[] is defined, we can have a vector to represent the distribution of POs in the circuit. This is both reported by CIRC and used as a parameter by GEN.

edges Also given the combinational delay of each node in the circuit-graph, we can define a distribution based on the edges of the graph. The length of an edge (x, y) is defined as delay(y) - delay(x), yielding a vector [0..delay] with sum the number of edges in the graph.

Fanout is also both an important characteristic of a circuit and parameter to generation. We have

max-fanout The maximum fanout (number of edges leaving) any node x in the circit-graph.

outs A vector representing the distribution of fanouts in the circuit. The vector is of length [0..max-fanout], is non-zero in the last element (or max-fanout is incorrect), and outs[0] $\leq$ nPO necessarily.

We also have a number of statistics which are output by CIRC which are calculated from the above metrics. For example, the average fanin/fanout, and the average fanin/fanout of each combinational delay level and associated standard deviations. These are not documented further at this time. However, they appear in the GEN defaults files as intermediate calculations when creating a default out-degree distribution.

Throughout CIRC and GEN, sequential circuits are described as a collection of combinational circuits. Within CIRC, a circuit is processed into *sequential levels* and we define "ghost" edges which cross the boundary between one combinational sub-circuit (sequential level) and another.

level The sequential level of a node x in a sequential circuit is defined as the *minimum* number of DFF on a directed path from any primary input. More formally, level(x) = 0 if x is a PI, level(x) = 1 + level(fanin d) if x is a DFF, and MIN(level(y), over all fanins y to x) otherwise.

back-edge An edge in the circuit which connects x to a node y at a *preceeding, different* sequential level. In other words, a feedback edge.

bottom-node A node which has all fanout-edges as back-edges is at the "bottom" of its combinational sub-circuit. The number of such nodes is relevant in the understanding of sequential circuits and how to generate them.

invisible-node Sometimes, especially when building a clock splitter or similar structures, it is possible to have a set of registers and logic which is self-contained and feed purely from itself (no PIs affect the output) and just outputs values. This is different from being unreachable (see above), because the value *is* affected by the clock. These nodes are not deleted by CIRC, becasuse they are important to the understanding of circuits, but they have to be treated as special cases to our basic model of a circuit because they have no real concept of sequential level.

forward-edge A forward edge is one which follows the normal rules of combinational delay when level is ignored, or which connects to a DFF at the next sequential level. That is, an edge which is not a back edge as previously defined.

This allows us the concept of level-shape and a distribution of back edges between levels (i.e. difference in sequential levels), but this will not be discussed at this time.

Because sequential circuits are generated at the base level as combinational circuits, we need a mechanism to define future back edges and forward edges to a DFF. This is done in terms of ghost input and output edges:

GI, nGI Each node in a hierarchically defined circuit or sequential input design will have its fanin divided into nodes which appear in the same sub-circuit and those which do not, called *ghost inputs (GI)*. The number of ghost inputs to a node (nGI) is defined for each node, and the total number of ghost inputs over all nodes is nGI for the circuit. $nGI(x)$ is always strictly less than kin, as one input to each node must be "real" for it to belong to one sub-circuit.

GO, nGO Similarly, we have ghost outputs, and nGO.

GIshape In the same way that shape[] is defined above, we have the concept of a distribution vector of GIs. Note that, when talking about sequential sub-circuits, we count nDFF in the shape profile, not in the GI shape profile, mainly due to internal details of shape generation beyond the scope of this document.

GOshape Similarly, we can store the combinational delay of each ghost output edge. as the delay of its source. Note, though it is required that any ghost edge has either dst.type == DFF or delay(src) < delay(dst), it is not necessarily true that delay(src) == delay(dst) - 1, because of the MAX relationship in the definition of delay.

Note that for a final circuit nGO(C) == nDFF + nGI(C) necessarily, as each ghost output corresponds to exactly one ghost input, or else eventually feeds one DFF.

It is important to note that PI and PO refer to nodes, whereas GI and GO refer to ports in or out of nodes, more like edges in a graph.

One final characteristic of circuits is the reconvergence number, or **rnum**. This is output by CIRC, but is not used by GEN so will not be discussed further here. Details on reconvergence calculation are contained in the published papers.

3 Using CIRC.

CIRC is a command-line based tool. The calling sequence is as follows:

```
circ in=<name> [k=<kin>] [options] [xnf | verilog | tdf | adl | gen] [out=<file>]
```

The only required parameter is the name of the file to be analyzed. The input format to CIRC is exclusively blif, so all files must be externally converted to blif before processing. CIRC will search in the current directory for the files name, name.blif, or name.blf, then search in the MCNCDIR (environment variable) directory in the 'k' subdirectory (k defaults to 4 if not specified).

The output of circ is to stdout. This can be overridden with the out= option. Note that the xnf,verilog,tdf,gen options automatically set out to 'name' with the appropriate file extension.

3.1 Using CIRC for format conversion.

To use CIRC as a filter to convert test.blif to either xnf, verilog or ahdl (tdf) formats, use the following syntax:

```
circ in=test <format>
```

where format is one of {xnf, xnfROM, verilog, tdf, ahdl} (tdf and ahdl are the same thing).

Note that k will automatically be set to 4, because all formats are output using the 4-LUT primitive. The program will fail if any node exists in test.blif which has fanin>4.

Currently, the ahdl and verilog formats output only NAND gates for LUTs. The xnf option will output ROM-based output if the option is specified as "xnfROM," but input which originated from GEN will still have only NAND gates defined (i.e. will simply be ROMs which define a NAND gate).

3.2 Using CIRC for statistical output.

Currently the "dump" format is the most stable form of output. There are other options available, but they are obsolete. The command:

```
circ in=test dump
```

will output a complete description of the design test. The output format is such that it is easy to use AWK, SED or GREP to extract and build tabular information from the output files of multiple circuits.

We will go through the output for an MCNC circuit, bbrtas:

First, there is some informational output to stderr (which does not appear in the output file). This gives the version and compile date of the software, and warning/error conditions encountered. In this case, bbrtas has a single unusable node "pclock." This is not a problem, CIRC is just noting that it dropped pclock as an unusable input when it replaced all clocks by the global signal 'clock.'

```
CIRC 2.2, compiled Fri May 24 12:04:30 PDT 1996.
Analysis of bbrtas beginning at Mon May 27 14:07:13 1996
Warning:  Deleting PI pclock because it does not drive a primary output
Warning:  (For further such nodes, use verbose option)
Warning:  Circuit has 1 unusable nodes
```

Note the mention of a command-line option "verbose" to see more detailed information, especially about error and warning messages.

Within the output file, we begin with introductory output, listing the options and the actual file name used. The filename is important because we used an MCNC circuit; this shows that we picked up the correct circuit. If we had a bbrtas in the current directory, CIRC would have used that instead.

```
File options:  in=bbrtas out=<stderr> err=<stderr>
Output options:
Displaying:
Reading input from file '/users/mdhutton/mcnc/4/bbrtas.blif' (k=4)
```

The next section of the output file gives basic statistics, as defined in Section 2. Note that there is actually an internally represented "service" (0th) component reported in parenthesis in the component list. This can be ignored.

```
name:   bbrtas
size:   417
edges: 1440
levels: 1
delay: 18
nPI: 4
nPO: 2
nDFF: 7
nLOG: 406
num_unusable: 1
num_unreachable: 0
ncomponents: 1 ( (4) 417 )
```

The degree information comes next. We have the average in-degree of LUTs, average out of LUTs + DFFs + PIs, and each separately; the average and total fanin and fanout by combinational delay level, the max-fanout, nodes with fanout beyond 1 standard deviation of the mean, and larger than 10 in absolute value, and the fanin and fanout vectors as sparse vectors and in full form.

```
avgin_log: 3.55 (0.46)
avgout: 3.47 (10.28)
avgout_dff: 74.43 (12.48)
avgout_pi: 26.00 (6.00)
avgout_log: 2.02 (3.39)
avgin_vec: ( 0.00 3.34 3.58 3.67 3.81 3.22 3.94 3.54 3.00 3.73 3.00
             3.97 3.50 3.00 3.95 3.40 3.80 3.50 4.00 )
avgout_vec: ( 0.00 1.74 1.00 5.33 2.62 3.67 2.83 1.11 6.40 1.73 18.00
              1.24 2.67 12.67 1.10 5.60 1.10 1.00 1.00 )
totin_vec: ( 0 454 283 33 61 58 71 99 15 56 9 115 21 9 83 17 38 14 4 )
totout_vec: ( 625 236 79 48 42 66 51 31 32 26 54 36 16 38 23 28 11 4 1 )
visible_edges: 1449
max_out: 90
high_degree_log: 14
high_degree_pi: 4
high_degree_dff: 7
degree_10plus_log: 17
degree_10plus_pi: 4
degree_10plus_dff: 7
fanin: (0,4) (1,7) (2,34) (3,116) (4,256)
fanout: (1,345) (2,12) (3,6) (4,4) (5,7) (6,9) (8,2) (9,4) (10,1)
        (13,2) (14,5) (15,1) (16,2) (17,3) (20,2) (21,1) (24,1) (29,1)
        (32,2) (53,1) (60,1) (75,1) (76,1) (81,1) (86,1) (90,1)
fanin_vec: ( 4 7 34 116 256 )
fanout_vec: ( 0 345 12 6 4 7 9 0 2 4 1 0 0 2 5 1 2 3 0 0 2 1 0 0 1 0 0 0 0 1
              0 0 2 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 1
```

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0 1 0 0 0 1 )
```

Next we have sequential information, since bbrtas is a sequential circuit. This information would not be displayed were it purely combinational.

We have the number of bottom nodes (over all levels), the number of invisible nodes (total and broken into DFF, LOG and PO), the number of nodes at each sequential level (sequential shape), the number of back-edges (number of GOs from level 1 to level 0) and their shape-like distribution, and the maximum combinational delay of each level.

```
bot_nodes: 89
invis_nodes: 0
invis_DFF: 0
invis_LOG: 0
invis_PO: 0
seq_shape: ( 304 113 )
back_edges: 325
where-back: ( 0 325 )
level_maxdelays: ( 18 2 )
```

Combinational shape vectors follow. These are over the *entire* circuit, summed. This section is of limited value for sequential circuits.

```
shape      417: ( 11 136 79 9 16 18 18 28 5 15 3 29 6 3 21 5 10 4 1 )
POshape      2: ( . 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 )
edge-length: ( 0 1035 179 30 35 39 30 25 6 21 6 24 6 4 0 0 0 0 0 )
forward-edge-length: ( 0 889 89 27 25 25 16 14 3 13 3 4 4 3 0 0 0 0 0 )
back-edge-orig: ( 196 116 13 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 )
back-edge-dest: ( 0 67 159 5 11 15 6 24 0 10 0 24 1 0 3 0 0 0 0 )
back-edge-length: ( 0 146 90 3 10 14 14 11 3 8 3 20 2 1 0 0 0 0 0 )
```

Then the vectors for each individual level are presented. The input design has 1 sequential level (beyond the 0th level, the combinational part).

At level 0, we see the number of nodes, GI, GO, PO and shapes of each, as well as the fanout distribution for L0 alone. Note again that the GI/GO counts will seem high, but they count ports/edges not actual nodes. It is quite common, for example, to have a level 1, such as shown, with 120 nodes and 325 ghost outputs.

```
n0 = 308;
L0shape = (4,43,66,9,16,18,18,28,5,15,3,29,6,3,21,5,10,4,1)
nGI0 = 325;
L0GIshape = (67,159,5,11,15,6,24,0,10,0,24,1,0,3,0,0,0,0,0)
nGO0 = 7;
L0GOshape = (0,0,0,0,1,0,1,0,0,1,0,1,0,0,1,0,1,0,1)
nPO0 = 2;
L0POshape = (0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
nEdges0 = 769;
L0edges = (0,566,66,27,25,25,16,14,3,13,3,4,4,3,0,0,0,0,0)
L0out = (4,250,8,4,3,2,6,1,1,4,1,0,0,2,5,1,2,3,0,0,2,1,0,0,1,
          0,0,0,0,1,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
          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)
```

Similarly for level 1:

```
n1 = 120;
L1shape = (7,93,13,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
nGI1 = 0;
L1GIshape = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
nGO1 = 325;
L1GOshape = (196,116,13,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
nPO1 = 0;
L1POshape = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
nEdges1 = 346;
L1edges = (0,323,23,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
L1out = (88,16,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
          0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,
          0,1,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,
          0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
```

And, finally, we close with a statement of the time and memory used for the analysis.

```
Circuit Analysis complete.  cpu: 7.21 sec, mem: 888K, time: 8 sec
```

3.3 Using CIRC as input to GEN.

CIRC is also able to output a GEN program in the SYMPLE language which can be then run through GEN to make a "clone" circuit of the original input circuit.

For example, the output of

```
circ in=bbrtas gen
```

appears in bbrtas.gen as the following:

```
/* CIRC 2.2, compiled Mon May 27 14:13:55 PDT 1996.
 */
X = {
    name="bbrtasclone";

    L0 = (@.comb_circ) {      exact=1;
        name="L0"; n=304; kin=4; nPI=4; nDFF=0; level=0; delay=18;
        nBot=1;
        shape=(4,43,66,9,16,18,18,28,5,15,3,29,6,3,21,5,10,4,1);
        nGI=325;        GIshape=(67,159,5,11,15,6,24,0,10,0,24,1,0,3,0,0,0,0,0);
        nGO=7;  GOshape=(0,0,0,0,1,0,1,0,0,1,0,1,0,0,1,0,1,0,1);
        nPO=2;  POshape=(0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);
        nEdges=769;
        edges=(0,566,66,27,25,25,16,14,3,13,3,4,4,3,0,0,0,0,0);
        outs=(4,250,8,4,3,2,6,1,1,4,1,0,0,2,5,1,2,3,0,0,2,1,0,0,1,
            0,0,0,0,1,0,0,2);
        max_out=32; nZeros=4;
    };
```

```
    L1 = (@.comb_circ) {      exact=1;
        name="L1"; n=113; kin=4; nPI=0; nDFF=7; level=1; delay=2;
        nBot=13;
        shape=(7, 93, 13);
        nGI=0;  GIshape=(0, 0, 0);
        nGO=325;        GOshape=(196, 116, 13);
        nPO=0;  POshape=(0, 0, 0);
        nEdges=346;
        edges=(0, 323, 23);
        outs=(88,16,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
            0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,
            0,1,0,0,0,0,0,0,0,1,0,1);
        max_out=61; nZeros=88;
    };

    glue=(L0, L1);
};
output(circuit(X));
```

We will discuss the parameterization file further in Section 4 of the document.

4 Using GEN to generate circuits.

GEN has some command-line options, but the vast majority of the parameterization is too complicated to be specified at the command line. GEN is augmented with a rich specification language with which parameter-programs can be written.

4.1 Generating a simple combinational circuit

Thus far we have only talked about how parameters are evaluated in the SYMPLE language. This doesn't actually create a circuit. To have the parameters passed to GEN, and a circuit generated, we use the "output_circuit" command in our script "x.gen" which we invoke with "gen x.gen"

```
    X = comb_circ { n = 500; };
    output(circuit(X));
```

Output from gen is split into two streams. In the stdout stream, the output blif-formatted netlist is printed to x.blif, as specified. The log of information and errors goes to stderr, and is described as follows:

```
  GEN Development Version 2, compiled Thu May 23 09:59:10 PDT 1996.
  Parsing parameters for circuit 'C'
  Random shape with n=500 nTop=9 nBot=2 delay=11 width=182 jumps=5 expand=9.00:

  Generating combinational specs for C
  -- n = 500  nPI=9  nDFF=0  kin = 4  delay = 11  seed=833239250
  --    Shape (  500):   9  60 182 105  57  42  16   7   7   7   6   2
  -- PO Shape (    3):   0   0   0   0   0   0   0   0   0   1   0   2
```

```
-- GO Shape (    0):   0   0   0   0   0   0   0   0   0   0   0   0
-- GI Shape (    0):   0   0   0   0   0   0   0   0   0   0   0   0
-- Edges    ( 1172):   0 950 141  46  29   4   0   2   0   0   0   0
-- Out-degrees (max=18):  3 311 48 44 18 22 16 14 8 2 6 3 0 0 1 1 0 1 2
Building the circuit level-graph
Graph passed steps one and two.  Best was method 3
Splitting nodes to generate the complete circuit-graph.
Degrees fudged: 461, edges fudged 56, edges lost 0 (of 1672 total)
Warning:  Forced to add 1 extra outputs at delay level 11
Warning:  Fixing IO dist'n results in 1 extra nodes, 1 extra outputs.
Graph generated, converting to a circuit.
(Sub)circuit 'C' has been generated.
Circuit generation successful
Elapsed time 3 seconds
```

We have the version and compilation date of the program. Another important parameter is the random number seed (taken from the clock). To get exactly the same circuit again, we should specify "seed=833239250" on the command line.

The defaults.gen file (hence comb.gen) is read for default information, then x.gen is processed. We specified n=500, from which comb_circ specified 9 PIs and the remaining LUTs, with combinational delay 11 and 3 POs). The number of edges was 1172, so the average fanin was about 2.2 (not a particularly dense circuit). Similarly, the combinational delay and distribution of nodes, edges and fanouts are shown. If we run the command line again without specifying the same seed, we will get both a different parameterization and a different circuit. Had we specified the complete parameterization, we would get a different circuit with the same parameterization.

Generating a combinational clone:

To generate a clone of an MCNC (or other) circuit (in xnf), do the following (for example, we use the circuit 5xp1):

```
circ in=5xp1 gen
gen  5xp1.gen
circ in=5xp1clone.blif xnf
```

4.2 Generating a hierarchical or sequential circuit

Sequential circuits are specified in GEN as hierarchical circuits with "glue" ports to combine them together. For example, a finite state machine is viewed as two or more combinational circuits, one of which has primary inputs, and the others of which have DFFs as its primary inputs. GEN will make a sequential circuit in this way by generating the two combinational circuits separately, then gluing them together following a number of rules beyond the current scope of this document.

The user has control over the type of sequential circuit that is generated in the input script. At the simplest level, the user can specify the size of the circuit and the number of I/Os and DFFs and let the rest come from the defaults. For example

```
X = fsm_circ {
```

```
    name = "example5";
    n=500;
};
output(circuit(X));
```

will generate a "fsm-like" circuit with 500 nodes directly from the defaults. On my machine, with seed=834610821, I got a circuit with 6 PIs, 471 nodes, combinational delay 10, 2 POs, 29 DFFs and 145 back-edges (GOs at level 1).

You can also specify the amount of interaction between the levels by giving values for nGI, nDFF and so on. For example

```
X = fsm_circ {
    name = "example2";
    nPI=63; nPO=36;
    nDFF=120;
    n=450+nDFF+nPI;
    kin=4;
    n0=n/2;
    n1=n/2;
    nBack=n/3;
};
output(circuit(X));
```

Above we have specified the number of back-edges in terms of the size of the circuit, and specify the number of DFFs and I/Os exactly. We have asked for 450 LUTs, giving a size 450+nDFF+nPI for the entire circuit.

It is possible to make more difficult hierarchical circuits, but this part of the code is very new, and there will be problems when you try to do it.

For example, see gendir/5-way.gen, which generates 5 separate sequential circuits with a specified number of ghost I/Os, and then glues all 5 together simultaneously.

See also 40K.gen, which generates a large circuit (40000 4-LUTs) from several smaller circuits, with a specified cut-size (for example, to test a partitioner). Here the result is seen as a combination of several state-machines which provide control into a combinational circuit at the next level. By manipulating the parameters it is possible to make a number of different configurations.

Note that the probability of errors increases multiplicatively with the number of circuits in the hierarchy. Whereas there is a 85% or more chance of success at generating a circuit with 5000 nodes, generating 5 such circuits to glue together will only succeed about 44% of the time. This means that multiple runs are often required. However, I have successfully generated circuits with this amount of hierarchy to 150000 4-LUTs within about 10 tries. It is expected that as we refine the parameterization scripts and build more error handling and correction into GEN that this will disappear, and we will be able to generate circuits with a great deal of hierarchy.

Appendix C

Further Examples.

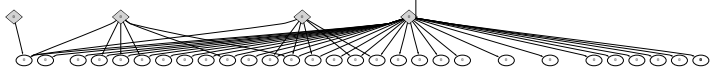
This appendix shows a number of examples of circuits produced by GEN.

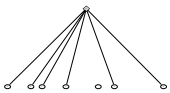
Figures C.1 through C.4 show examples of short GEN scripts and drawings of two circuits produced by each script. The scripts show how the user can choose specific features of a circuit, such as the maximum fanout, the combinational delay, or the relative shape profile in generating circuits. Note that whenever parameters are omitted from the figure caption, the defaults were used by GEN.

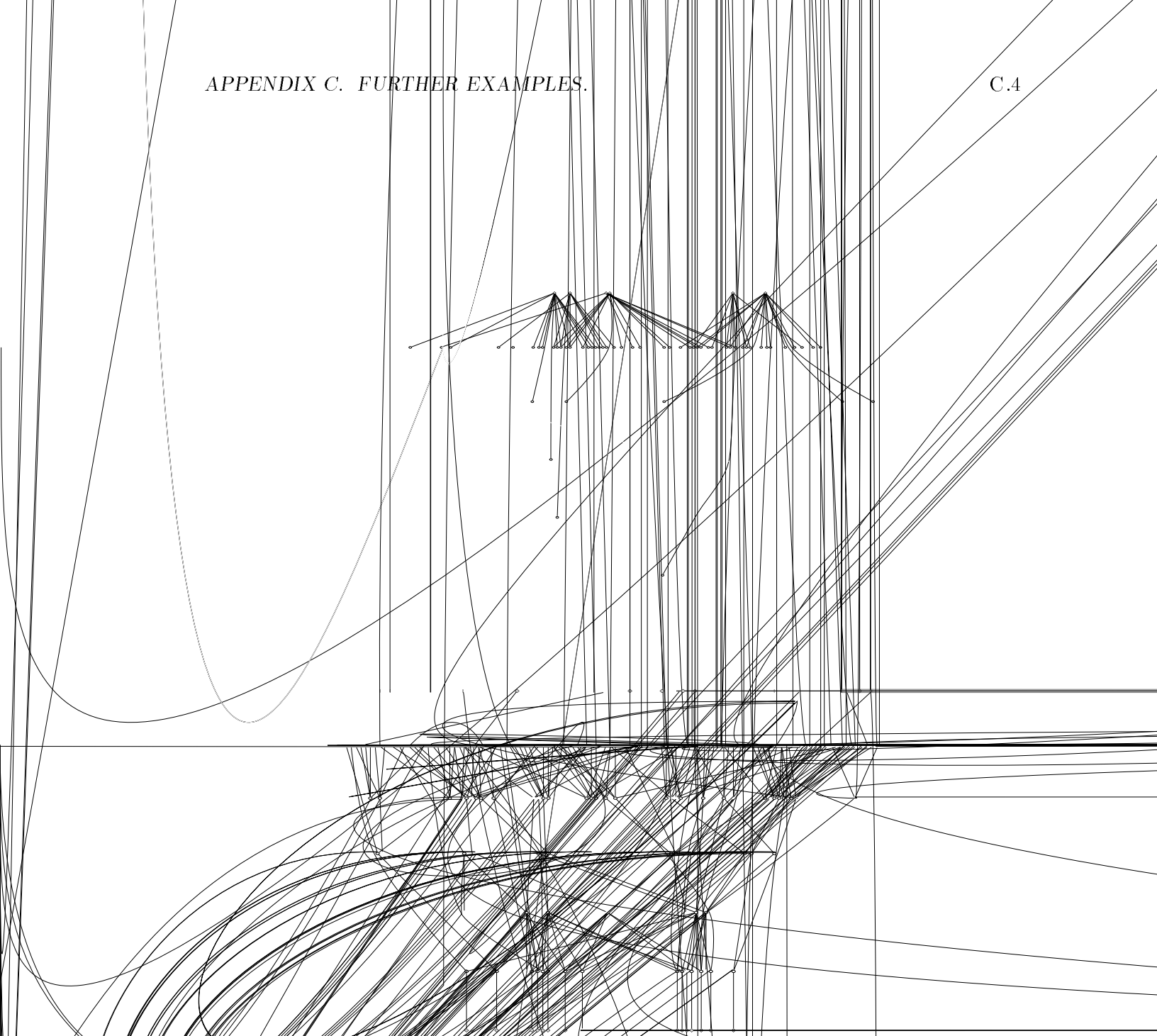
Figures C.4 and C.5 show two different sequential circuits generated from the same clone script, in which the user specifies only high-level information such as the number of flip-flops.

Figures C.6 through C.10 show four different MCNC combinational circuits, and their clones produced by GEN.

Figures C.11 and C.12 show two different MCNC sequential circuits, and their clones produced by GEN.



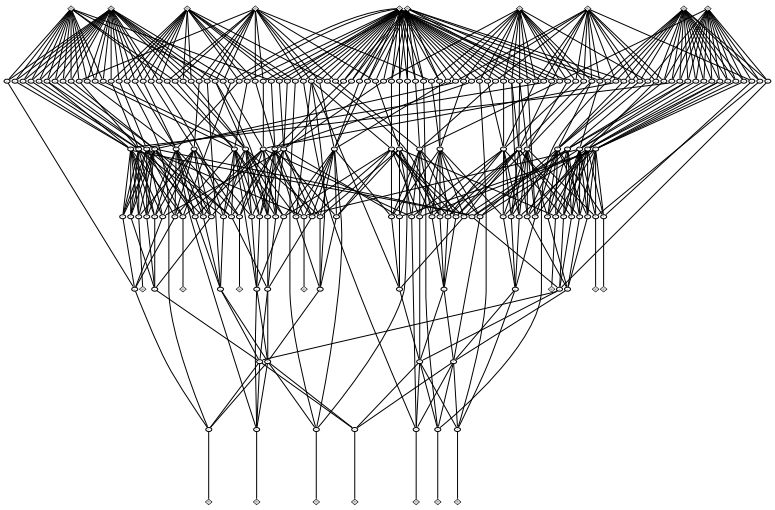




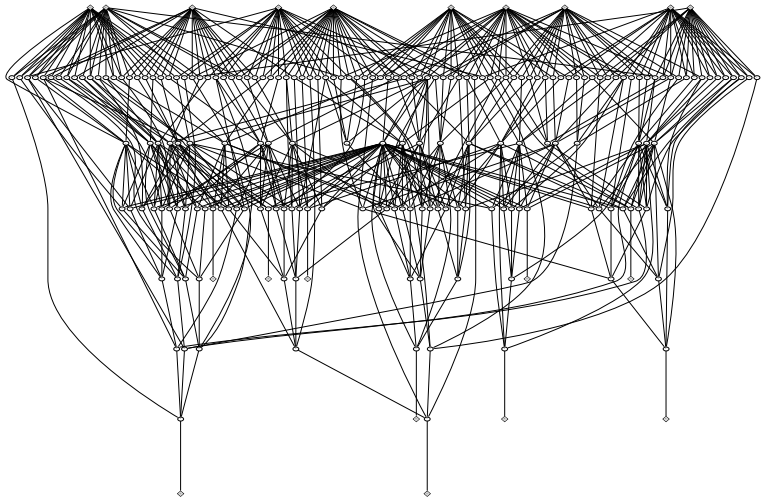
Circuit 2

```
X = comb_circ {name="A"; n = 200; nPI = 30; nPO = 20;
               delay=5; shape=(1,5,4,3,2,1); };
output(circuit(X));
```

Figure C.3: Two combinational GEN circuits with specified shape.



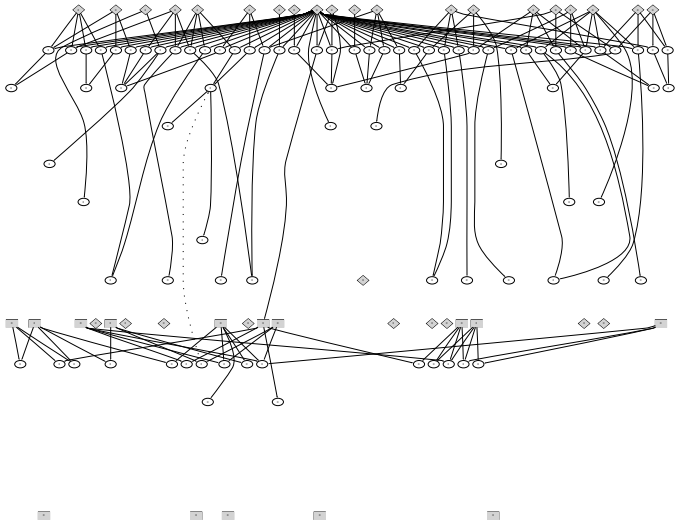
Circuit 1

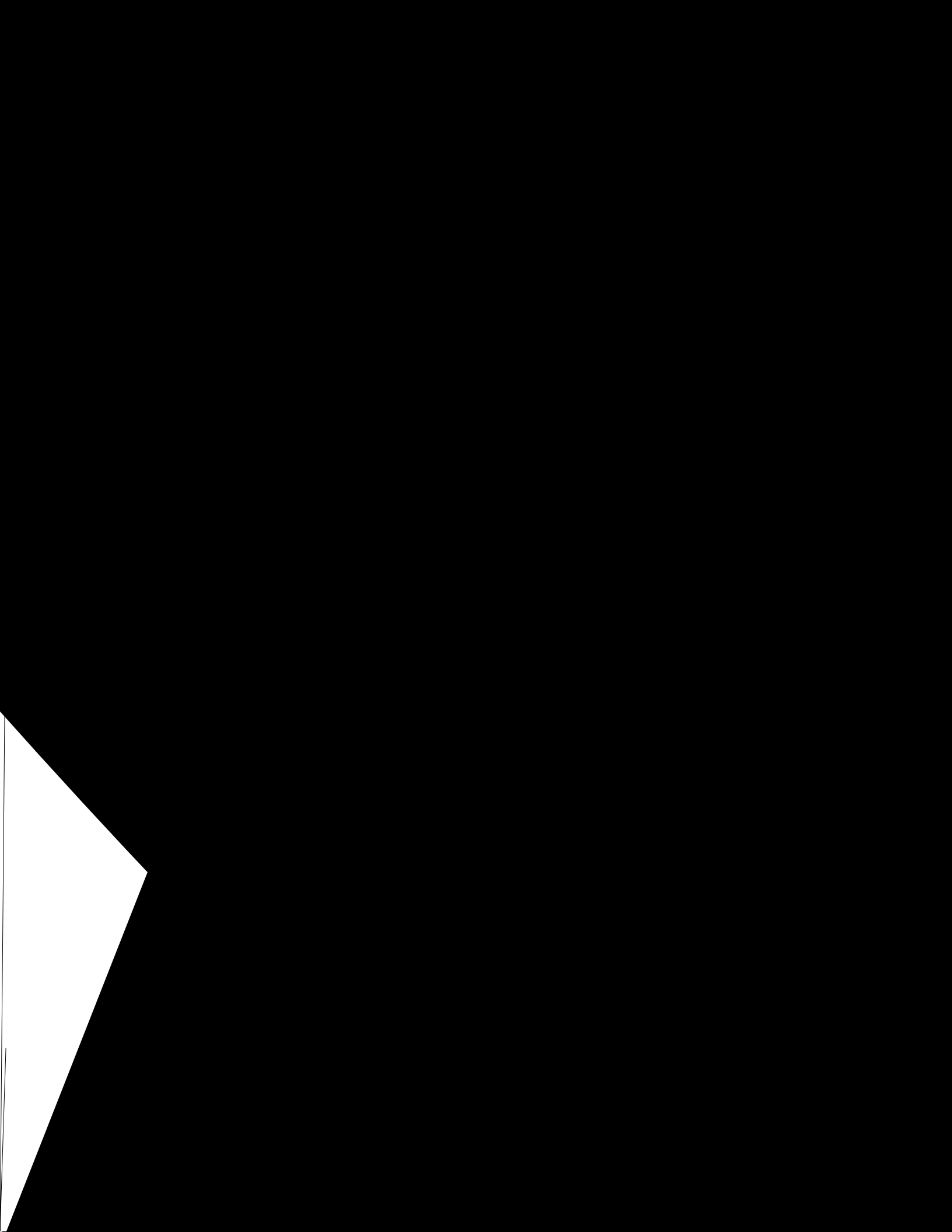


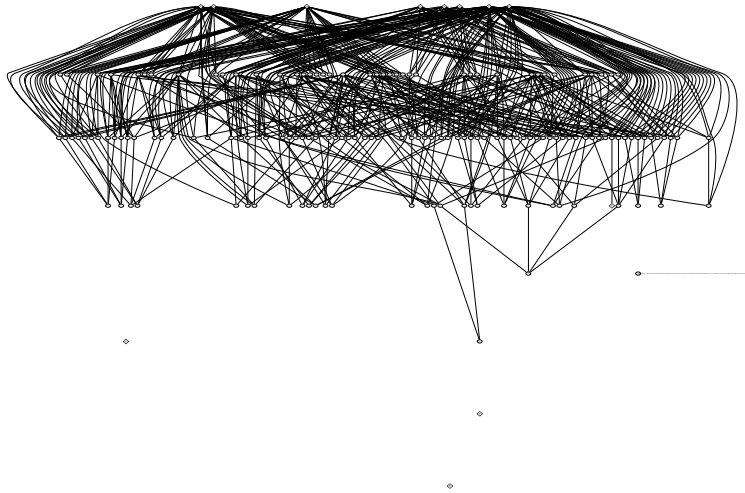
Circuit 2

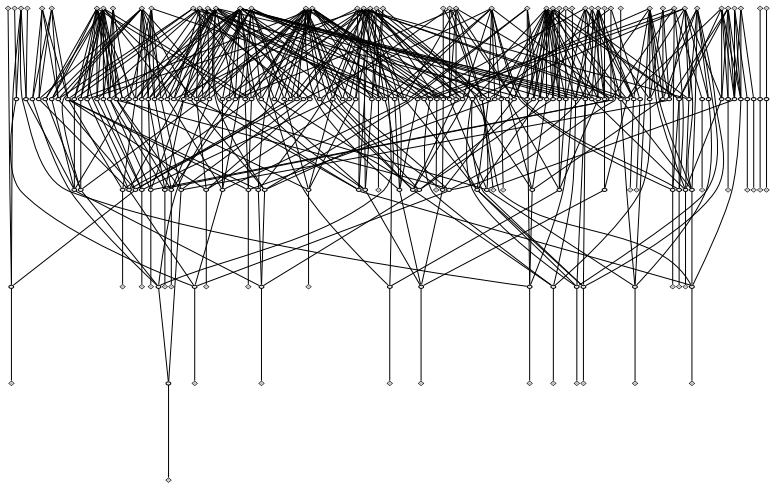
```
X = comb_circ {name="A"; n = 200; nPI = 30; nPO = 20;
               delay=5; shape=(1,5,4,3,2,1); };
output(circuit(X));
```

Figure C.4: Two combinational GEN circuits with specified shape.

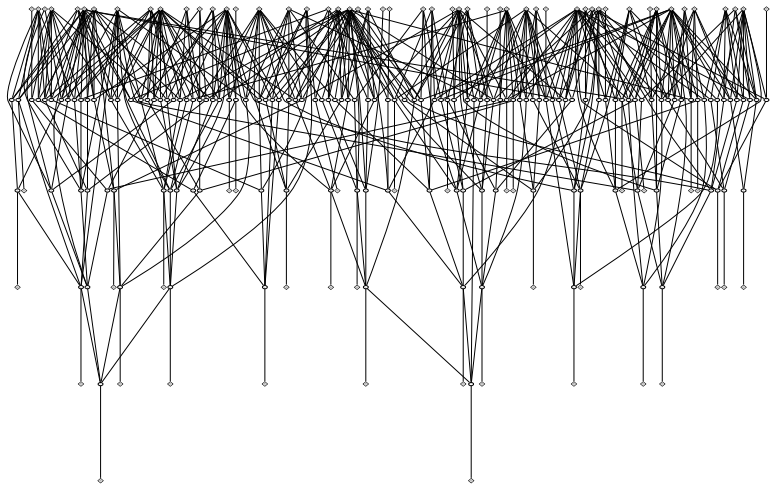






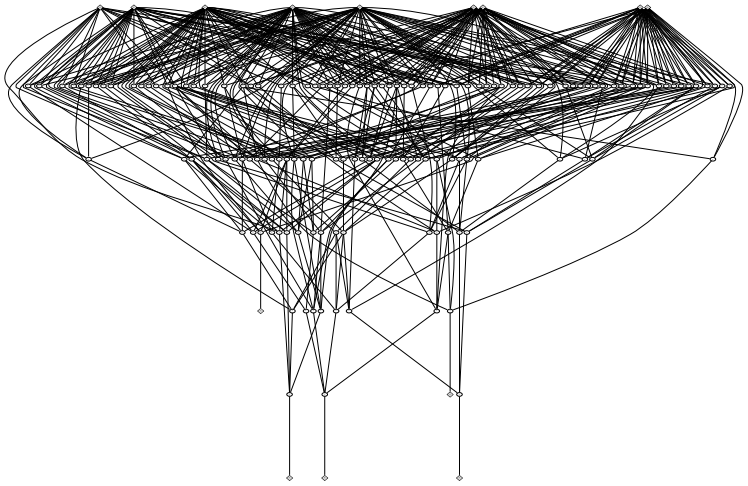


Original circuit

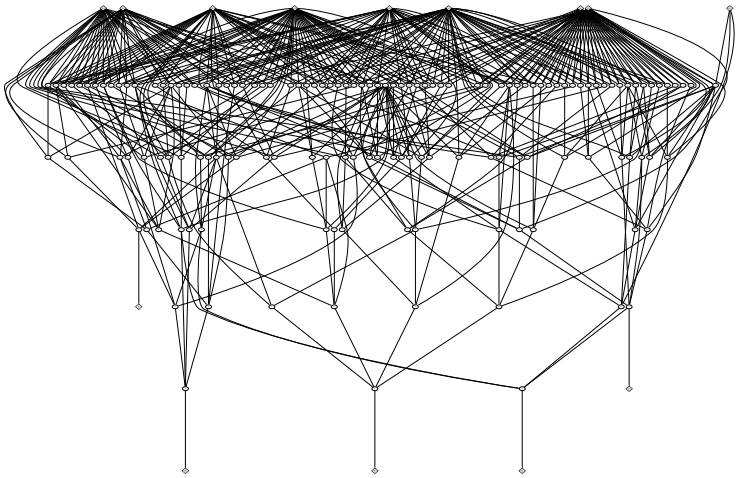


Clone circuit

Figure C.8: MCNC circuit **x1** and a clone by GEN.

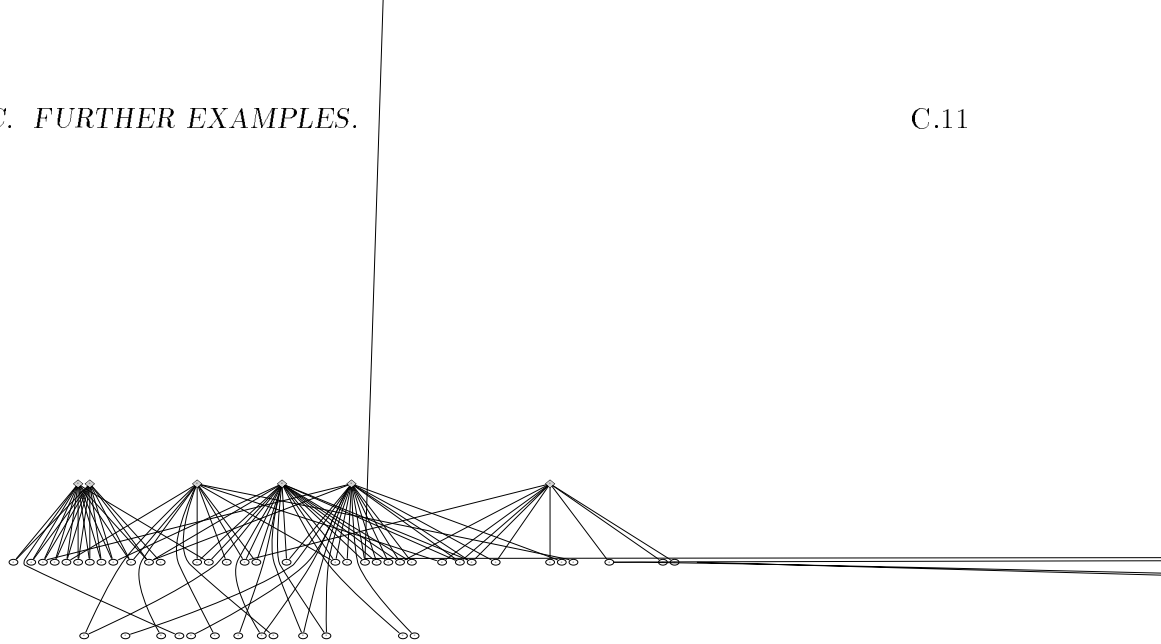


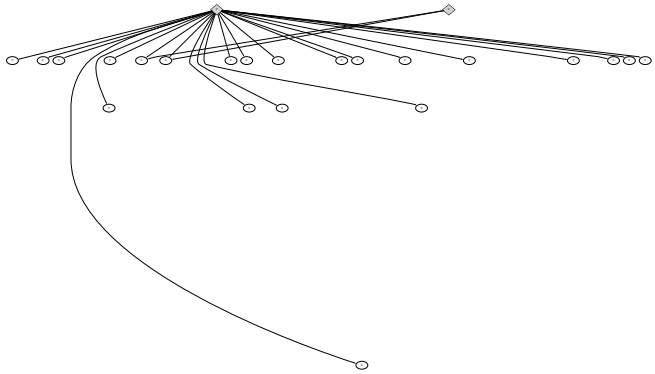
Original circuit

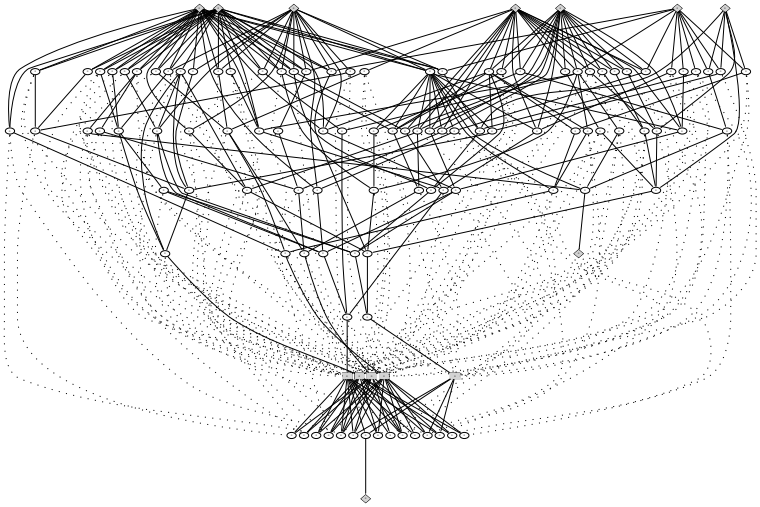


Clone circuit

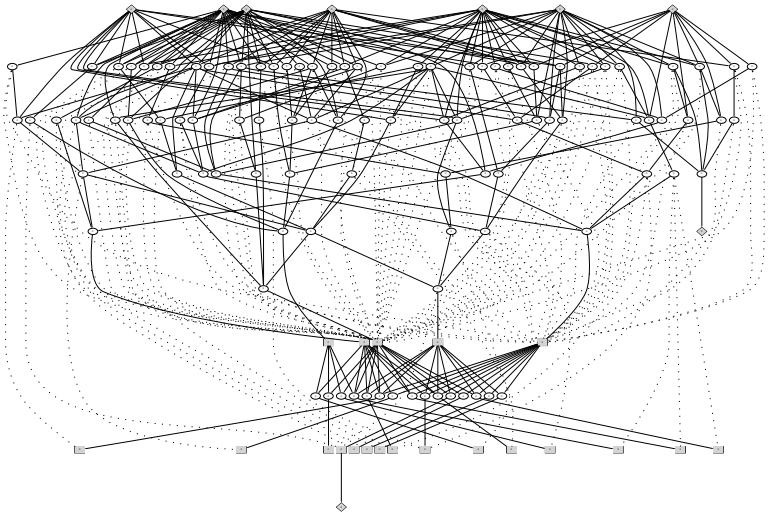
Figure C.9: MCNC circuit **clip** and a clone by GEN.





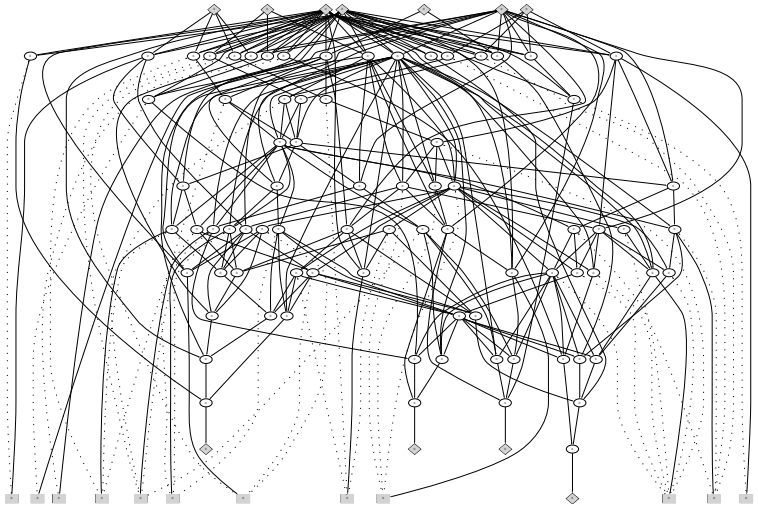


Original circuit

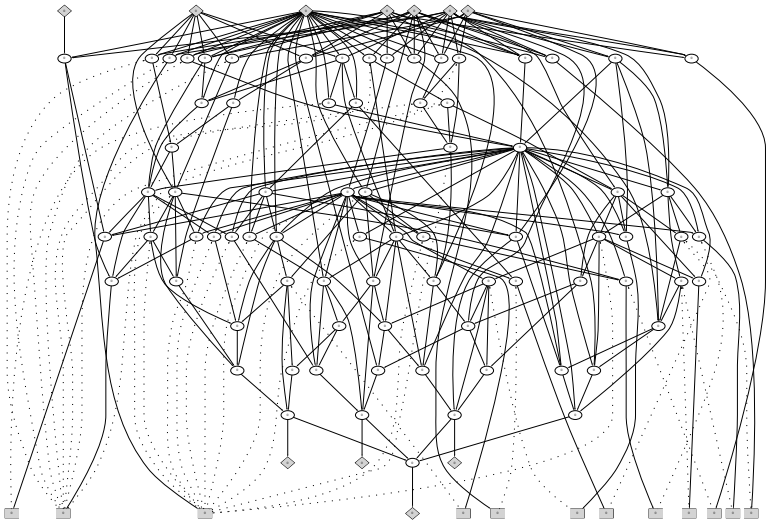


Clone circuit

Figure C.12: MCNC sequential circuit **keyb** and a clone by GEN.



Original circuit



Clone circuit

Figure C.14: MCNC sequential circuit **mm4a** and a clone by GEN.

```
    outs = junk;
};
```