A System-Level Synthetic Circuit Generator for FPGA Architectural Analysis

by

Cindy Mark

B.A.Sc., Queen's University, 2006

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Applied Science

in

The Faculty of Graduate Studies

(Electrical and Computer Engineering)

The University Of British Columbia

(Vancouver)

November, 2008

Abstract

Architectural research for Field-Programmable Gate Arrays (FPGAs) tends to use an experimental approach. The benchmark circuits are used not only to compare different architectures, but also to ensure that the FPGA is sufficiently flexible to implement the desired variety of circuits.

The most common benchmark circuits used for architectural research are circuits from the Microelectronics Center of North Carolina (MCNC). These circuits are small; they occupy less than 3% [5] of the largest available commercial FPGA. Moreover, these circuits are more representative of the glue logic circuits that were targets of early devices. This contrasts with the trend towards implementing Systems on Chip (SoCs) on FPGAs where several functional modules are integrated into a single circuit which is mapped onto one device.

In this thesis, we develop a synthetic system-level circuit generator that connects pre-existing circuits in a realistic manner to build large netlists that share the characteristics of real SoC circuits. This generator is based on a survey of contemporary circuit designs from industrial and academic sources. We demonstrate that these system-level circuits scale well and that their post-routing characteristics match the results of large pre-existing benchmarks better than the results of circuits from previous synthetic generators.

Table of Contents

List of Tables

List of Figures

List of Programs

Acknowledgements

I would like to thank my supervisor Professor Steve Wilton for his patience and guidance these two last years. He is truly dedicated to the development of his students and he was always ready to give help and advice. I am proud to say that I learnt a lot under his supervision.

I would also like to acknowledge the students in the SoC group; you were always there to help and to lend a listening ear through all the tough times. I'd like to single out Scott and Dave who were incredibly helpful during this research. In particular, I would also like to thank those SoC students who started at the same time as I did: Andrew, Paul, Darryl, Darius, and Johnny. We shared the same ups and downs and it was comforting to know that I was not facing the challenges of the two past years alone.

I would like to thank my fellow residents at Green College, notably Tal and Nicolas, who kept me on an even keel. Without you, this thesis would have taken much longer to finish.

Lastly, I would like to dedicate this work to my family for their unflagging support and encouragement.

Chapter 1

Introduction

1.1 Motivation

In the past twenty years, the density of Field-Programmable Gate Arrays (FPGAs) has increased by 200x [59]. This increase in density has led to new architectures; modern FPGAs look significantly different than their early predecessors. The routing architecture has evolved from a homogeneous gridded network to include a variety of elements. The logic architecture now consists of "fracturable" lookup-tables arranged in large tightly-connected clusters. This continued evolution of FPGA architectures is essential to support the increasing computation requirements of digital systems.

The development of these new architectures often employs an experimental methodology. A potential architecture is modelled, benchmark circuits are mapped to the architecture, and detailed models are used to evaluate the density and speed [7], and the power [50] of the implementation. A critical part of this experimental methodology is the set of benchmark circuits. These circuits must be representative of the circuits that will eventually be implemented on the FPGA. However, most benchmark suites used today are more representative of the glue logic circuits that were targets of early devices. Circuits from the Microelectronics Center of North Carolina (MCNC) have become almost ubiquitous in recent publications, yet even the largest of these circuits contains only 7694 logic elements [65], which is approximately 2.3% of the largest available Altera Stratix III [5]. Other benchmark circuits are available (e.g. [13]), however, even these

are significantly smaller than those that will be implemented in future devices. Commercial vendors have large databases of circuits, but also report that obtaining circuits representative of those that will be implemented in next-generation devices is a challenge.

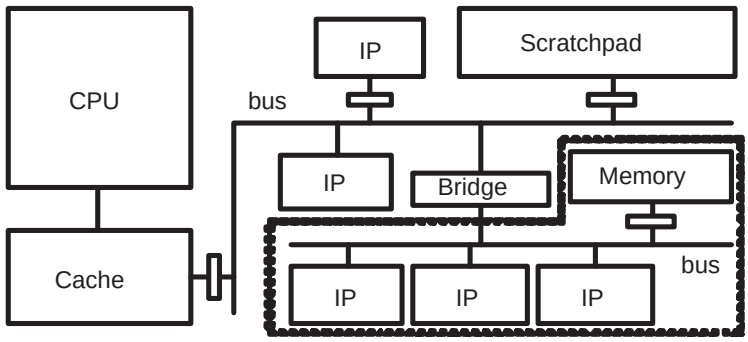


Figure 1.1: Example SoC: System with two hierarchical levels

A potential solution is to use stochastically-generated benchmark circuits [22, 26, 27, 33, 56]. Typically, these circuits are created using a *circuit generator* which randomly creates netlists according to constraints that ensure the netlists share many of the structural characteristics of real circuits. Although these circuits are not "real", this approach has a number of advantages: an FPGA architect can generate as many circuits as desired, the circuits can be of any size, and often the generator can be further tuned to create only circuits with certain properties (e.g. datapath circuits in [33]). This latter advantage is critical during early architectural evaluation, when it is important to understand *what types* of circuits work well and what types do not work well. A generator that can create structures with a particular property can be an invaluable aid as new architectures are uncovered and evaluated.

These stochastically-generated circuits, however, must be realistic. Most existing generators build circuits using a "bottom-up" methodology using predetermined structural parameters and/or Rent's Rule. The resulting circuits realistically model glue-logic control or datapath circuits. However, circuits implemented on today's FPGAs are typically entire systems consisting of processors and intellectual property (IP) modules connected

using buses and on-chip networks. Often, the blocks contain several sub-blocks also connected using buses or networks; an example system is shown in Figure 1.1. It is unclear how well circuits generated using previous techniques reflect systems designed using such hierarchical system-on-chip (SoC) techniques.

1.2 Research Goals

This thesis presents *a circuit generator which creates benchmark circuits that better reflect the system-level circuits that are being implemented on today's FPGAs.* The focus is on the interconnect between the functional modules of the circuit. The generator stitches together smaller circuits using common connection patterns.

More specifically, the generator has the following characteristics:

- It is based on results from a survey of contemporary circuits.
- It creates circuits by combining modules using bus, dataflow, or star connection patterns as described in Chapter 3.
- The generated circuits are hierarchical; the modules can themselves consist of sub-modules connected by buses or networks.
- At the lowest hierarchical level, the modules are obtained from existing benchmark suites (such as MCNC circuits), or from previous circuit generators.
- Key parameters such as the number of hierarchical levels and the number of modules in the circuit can be given as constraints to the generator, allowing the user to generate circuits that reflect current and future system-on-chip circuits.
- If these parameters are not specified by the user, the generator is able to stochastically choose reasonable values. If partial information is provided, the parameters will be determined using the other data.

1.3 Research Approach

There were three stages to this research. First, we performed a careful structural analysis of existing circuits described in recent academic publications and industrial datasheets. We analysed 66 circuits and recorded quantities such as the number of hierarchical levels, the number of sub-modules, and the interconnection pattern between these sub-modules in each level and the types of the sub-modules. Where possible, we measured the correlation between the various quantities.

Second, we developed a circuit generator that stitches together smaller circuits in a hierarchical manner using common connection patterns. The generator was calibrated using the results of the circuit analysis described above. The generation algorithm was developed to ensure that the modules are connected as realistically as possible. The algorithm also tries to ensure that there is enough variety between circuits so that the resulting suite of circuits is suitable for FPGA architectural experimentation.

Third, we characterised the circuits created by the generator. Using academic CAD tools, we mapped our circuits to a model FPGA, and measured key quantities such as wirelength and channel width. We repeated the experiment for circuits generated by previous circuit generators, as well as "real" benchmark circuits obtained from a structured ASIC company. In doing so, we show that our circuits are more similar to these "real" circuits than circuits built by previous generators.

1.4 Organization

This paper is organized as follows. Chapter 2 covers background relevant to the thesis, and describes previous synthetic circuit generators. A new model to encode the structure of SoC circuits is described in Chapter 3 and it is applied in a survey of contemporary SoC circuits. Chapter 4 uses the data found during the survey to design and calibrate

a synthetic circuit generator. Characterisation and validation of these synthetic system-level circuits are shown in Chapter 5. Finally, Chapter 6 concludes. Much of this work was published in [41]

Chapter 2

Background

Unlike application specific integrated circuits (ASICs) which are designed to implement a single circuit, FPGAs are generic platforms that can implement many digital circuits. By using configurable look-up tables to implement logic functions, and memory cells to control signal routing, the FPGA can implement a variety of functions. More flexibility increases the range of circuits that can be implemented; however, this flexibility comes at a cost since these configurable elements consume power and area. A good architecture will have as little flexibility as possible while still allowing circuits to be successfully implemented. The FPGA architect has the challenge of balancing flexibility and overhead.

To determine whether a circuit can be efficiently implemented on a particular FPGA architecture, the circuit must be mapped onto it. While analytical tools are useful to predict the characteristics of the mapped circuit, researchers validate architectures experimentally. Benchmark circuits are particularly important for FPGA evaluation. They not only form a set of common reference points used to compare different architectures, but they also test the flexibility of the device. A good benchmark suite should be a representative sample of the structures that should be successfully implemented on the FPGA.

This chapter first reviews modern FPGA architectures in Section 2.1 and a typical synthesis flow in Section 2.2. It then discusses FPGA architecture experimentation in Section 2.3, the ensuing benchmark requirements in Section 2.4.1, and lastly, existing benchmark sources in Sections 2.4.2 to 2.4.4.

2.1 FPGA Architecture

An FPGA is a configurable integrated circuit. It is composed of programmable logic elements (LEs) that implement boolean functions, and routing elements that connect the parts together [8]. A recent trend is the addition of hard blocks that implement complex logic functions [3]. The programmability of the logic elements and the routing allow an FPGA to represent a wide variety of logic designs. There are several types of FPGA architectures. The most common commercially used architecture today is the island-based architecture which is described in this section. Other possible architectures are row-based and hierarchical [7, 8].

A LE is the basic building block of the FPGA. It contains a lookup table (LUT) and a flipflop (FF). Figure 2.1 shows several LEs connected into a cluster [7]. By changing the values held in the static random access memory (SRAM) cells of the lookup table, a k-input lookup table can be programmed to implement any k-input logic function. The output of the lookup table can be stored in a flipflop to implement sequential logic.

Small groups of LEs are connected using an interconnect matrix into clusters [7]. The clusters are tiled across the FPGA, and a second level of programmable routing connects all the clusters. The intracluster routing is faster than the intercluster routing because the wires are shorter and have less parasitic capacitance. Although increasing the size of the cluster leads to a reduced number of signals being routed using intercluster wires, this benefit is counterbalanced by the rapid increase in area used by the intracluster routing as the cluster size grows.

First generation FPGAs consisted of a homogeneous array of clusters. Today, most FPGAs have a heterogeneous fabric that includes hard blocks such as multipliers, carry chains, memories, and possibly entire processors (CPU) [1, 38, 64]. While including these resources adds complexity to the synthesis process, an implementation using hard blocks is faster and uses less area than an implementation using lookup tables. The result is an

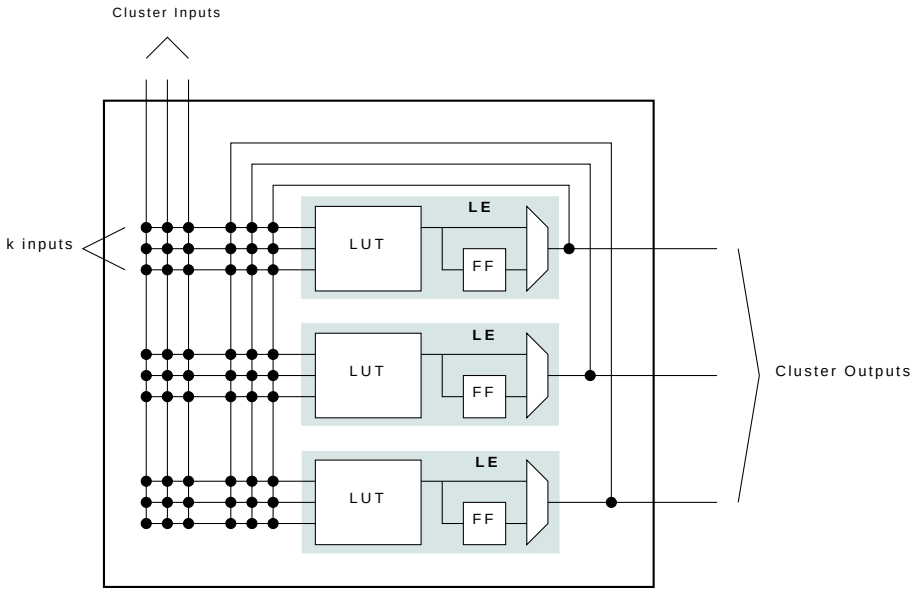


Figure 2.1: FPGA Cluster

increase in the efficiency of the FPGA. These benefits are only realized, however, if the hard blocks are used; otherwise, they occupy space that could be used for more general-purpose configurable logic [28]. To justify adding a hard block, the possible gains must offset the proportion of circuits that would not make use of this resource.

The intercluster routing fabric in an FPGA consists of wire segments, and the switches necessary to connect the wires both to each other, and to the clusters. The wire segments can be of different lengths, ranging from the width of one cluster to the full width of the chip [8]. Wire segments lie in tracks. Several tracks lie in parallel in each channel, and the channels cross the chip in the horizontal and vertical directions between the clusters as seen in Figure 2.2.

Programmable switches are used to connect wires together to make a path between a source and its sinks. A switch can be implemented using pass transistors or tristate buffers. Each switch can connect a pair of wires and the settings for these switches are stored in memory, so the connections are programmable [7].

These switches can be found in the connection block and the switch block. The output

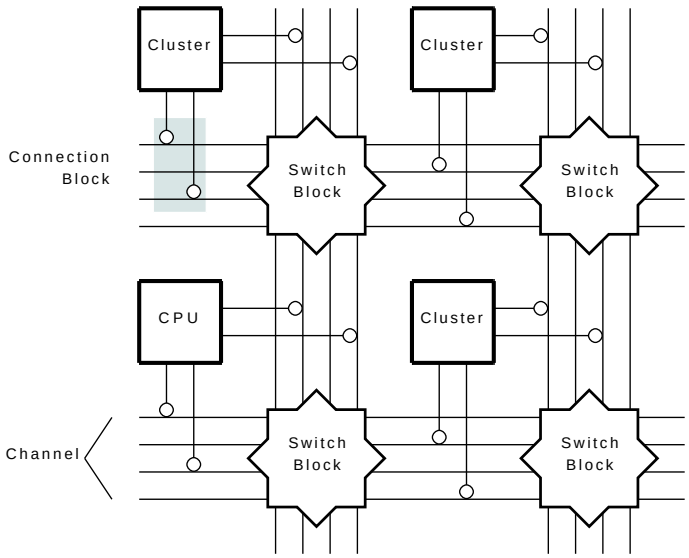


Figure 2.2: FPGA Architecture

from a logic element is connected to the output of the cluster. Each cluster connects to the wires in a channel through the connection block. The connection block contains a number of switches that can each connect a cluster pin to a particular track [37]. The switch block is a set of programmable switches found at the intersection of the channels that allows a signal to change tracks. This allows the signal to turn corners or to continue straight in order to reach its destination.

The connection block and the switch block both have switches for only a subset of the possible connections. Although this limits the flexibility of the circuit, it reduces the area occupied by the routing. Several studies [9, 37, 43] have investigated the effect of different switch patterns on the routability of circuits.

The area occupied by the routing channels forms the majority of the area on an FPGA. A reduction in the width of the channel leads to significant area savings but the FPGA area may actually be used more effectively if it has more routing elements [37]. A patch of dense connectivity in the netlist might require that these LEs be placed over more area so that more routing elements are available [67, 58]. In the worst case, if the channel is too narrow, the circuit cannot be routed at all.

2.2 Circuit Synthesis for FPGAs

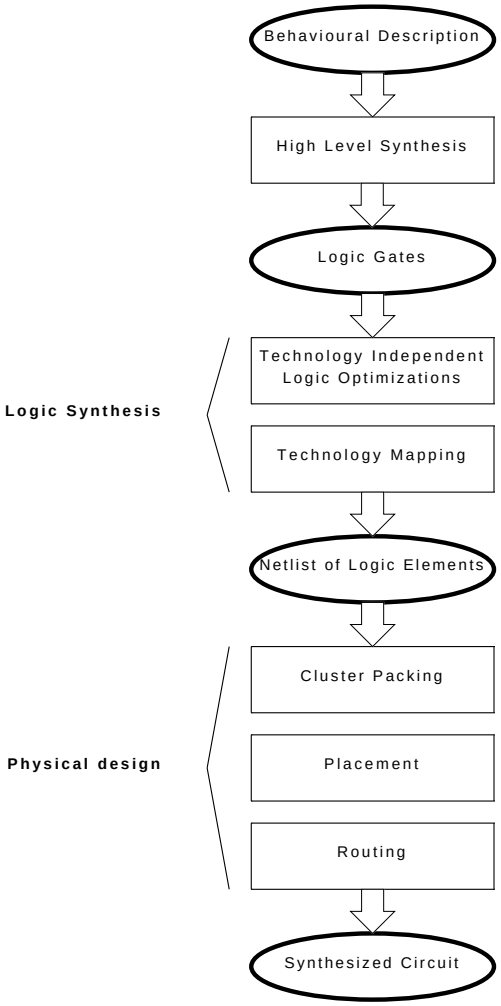


Figure 2.3: Circuit Synthesis Process for FPGA

Figure 2.3 summarises the steps used to translate a circuit from a high level description to an FPGA implementation. Each of these steps is computationally complex, so heuristic algorithms are used. The implementation obtained, however, is usually sub-optimal [7]. Measuring the effectiveness of a particular architecture is inextricably linked to the effectiveness of the synthesis algorithms.

A designer creates a behavioural description of the circuit, usually in a hardware description language (HDL). The high level synthesis algorithms analyse the code and

decompose the circuit into simple register transfer level (RTL) gates [16].The first step in the process is to translate the behavioural code into simple operations. Some optimisations such as loop unrolling are performed. Then, the operations are scheduled to specific cycles and allocated to resource units. Often however, designers will work directly at the RTL level, rather than use high-level synthesis.

Next, during the logic synthesis process, technology independent logic optimisation is performed to simplify the logic expressions. The operations are then mapped to lookup tables. Nodes are grouped together such that the group's inputs and outputs do not exceed the maximum number allowed. Mapping often uses cut-based algorithms. This process can be optimised for the depth of the logic (the potential critical path) [12], or for the logic area by packing as many operations as possible into one lookup table [45]. At this point, the circuit has been transformed to a netlist of blocks and nets.

During physical design, the logic elements are first packed into clusters. Ideally, related LEs are grouped inside the same cluster thus maximising the use of the faster intracluster routing [37].

These clusters are then assigned a location in the FPGA fabric such that factors like the distance between connected clusters, and the density of the nets across the FPGA are optimised. The closer the connected clusters, the shorter the possible net length and the lower the expected delay.

Two common techniques used to perform FPGA placement are simulated annealing algorithms and analytic methods. Simulated annealing [51] first places the clusters randomly onto the FPGA. It uses a cost function to represent the optimality of the current solution. The cost is proportional to the distance between connected clusters. Pairs of clusters are randomly selected and swapped. If the move results in a lower cost, the swap is accepted. If the move results in a higher cost, the move may still be accepted with a certain probability. The probability of accepting a poor move decreases in later

iterations so that the solution converges. In contrast, analytical methods first derive a series of quadratic equations that describe the sum of the net lengths attached to each cluster. Variables are used to describe the cluster locations. Since the solution allows clusters to be placed on a continuous axis, the challenge is to maintain a good solution while legalising the solution so that the clusters are at discrete positions [30].

The last step in physical design is to route the nets. Routing can be done using a two-step or a one-step process. The two-step process consists of global routing followed by detailed routing. In global routing, the nets are assigned to specific channels. In detailed routing, specific tracks and routing elements are allocated to each net. While this method is frequently used in ASIC design, the limited routability in FPGAs is more easily dealt with using one-step routing. A common algorithm used to perform single step routing is negotiated congestion [18]. In this method, each routing element is assigned a cost and nets are routed using the minimum cost path. If there is congestion on a particular routing element, the cost of that element increases. All the paths are removed, and every net is rerouted using the new cost until all the paths are successfully routed.

Once this sequence is finished, the resulting circuit has been converted to a bit stream that can be used to program an FPGA.

T-VPACK/VPR is a commonly used academic suite for physical design. VPR has been recently updated to use uni-directional wires and to allow heterogeneous blocks [34] which results in a more realistic representation of a modern FPGA. This updated suite is used in this research. T-VPACK packs the LEs into clusters such that the number of intercluster nets on the critical path is minimized [42]. VPR uses simulated annealing to perform placement and negotiated congestion to perform routing [7].

2.3 FPGA Architectural Experimentation

An FPGA architecture research project starts with the researchers devising a new architecture. To take advantage of the architecture's new features, the synthesis software often needs to be modified. The benchmark circuits are synthesised to the FPGA, and then the area, timing and power characteristics of the circuits are measured and evaluated. If necessary, modifications are made to the architecture, and the cycle restarts. This process is shown in Figure 2.4.

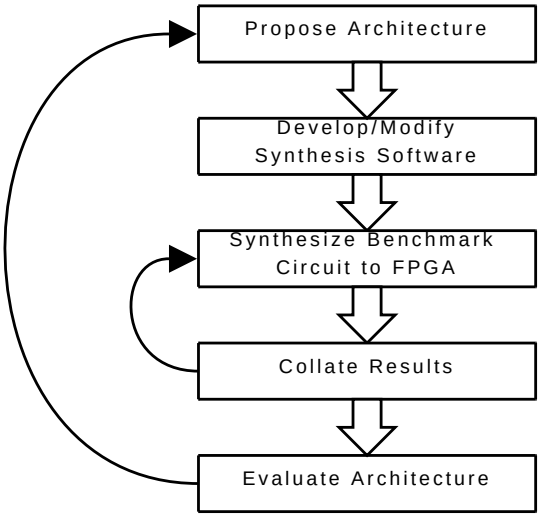


Figure 2.4: Architecture Design Cycle: The inner loop iterates over the circuits in the benchmark suite. The outer loop iterates until the researcher is satisfied with the results.

The primary metrics for evaluating the architecture of an FPGA are speed, area and power. Speed can only be measured after the circuit has been implemented on the actual device, but estimates can be obtained after the routing stage when the critical path is known. The logic depth of the circuit can be used to estimate the critical path, but since interconnect delay is generally the most significant component in circuit speed, these estimates are best made after routing.

Area can be measured after routing. At this stage, the circuit has been fully synthesised, and a list of all the elements necessary to implement the design has been compiled.

Area is highly sensitive to the high level algorithms used to convert the behavioural code into logic elements. Poor algorithms will result in unnecessary logic elements being generated. As synthesis progresses, there is generally less opportunity to reduce the area.

Measuring the power requires a chip to be built which is costly. As a result, most architectural experiments place and route the benchmark circuits and estimate the power consumption. Although transistord-level simulation of a configured FPGA device is the most accurate method, few researchers use this method because of the time necessary to simulate a circuit in that detail. Most FPGA researchers use gate-level power estimation. The majority of the power in an FPGA is dissipated in dynamic switching [15], so it is important to estimate the activity of the signals. The estimations can either be based on simulation or probabilitic methods [46]. Both of these methods depend upon the logic functionality of the circuit.

In addition to speed, area, and power, an FPGA architecture is also judged by its flexibility. The great advantage of FPGAs is its ability to effectively implement a wide range of circuits; therefore, an architecture that fails on a pathological circuit structure is much less useful. Consequently, an FPGA architecture must be tested against a wide variety of circuits. The broad variation in circuit structures makes it extremely challenging to analytically characterise the area, speed, and power results for a general circuit. New analytical tools are now available that can help predict the characteristics of the architecture [35, 19]; however, new architectures are usually still validated experimentally.

2.4 Benchmark Circuits

2.4.1 Benchmark Suite Requirements

As discussed in the overview, the suite of benchmark circuits represents the range of circuit structures that are expected to be implemented on the device. The ideal circuit suite

should have a wide range of circuit structures in order to test the proposed architecture over as many different cases as possible.

The circuits need to show realistic properties after synthesis to an FPGA. In order measure the area and routing, only the structural properties of the circuits need to be realistic. To estimate power using activity estimation, the circuit also needs to be functional so that the logic values can propagate correctly through the circuit.

In architectural experiments that make fine-grained changes to the FPGA fabric, each change is replicated many times over the FPGA. A medium size circuit might be large enough to cover several of these instances when implemented on the device. This circuit, when synthesised, would then provide several different test cases. A coarse-grained change to the FPGA might be replicated only a few times and a synthesised circuit might only be placed over one of these instances. Each benchmark then would represent only one test case.

Larger circuits would generate more test cases per synthesised circuit. In some experiments, it would also allow trends to be evaluated over a greater range of sizes. Circuits with specific structural properties would have a limited variety of test cases, allowing a narrow set of conditions to be investigated more deeply. This additional information would help develop better insight into the properties of the architecture.

These circuits should be in a common format. The Berkeley Logic Interchange Format (BLIF) is commonly used to represent the netlist of a circuit. It is supported by many academic tools, but it does not include all modern circuit features. HDL formats such as Verilog or VHSIC Hardware Description Language (VHDL) are also popular, but these formats are designed for detailed description of circuits and so may contain complex structures that may not be supported by academic tools.

2.4.2 Circuit Suites

A common source for benchmark circuits are circuit suites, notably the suite from MCNC. These circuits were collected in the 1980's. This suite comprises 205 circuits ranging in size from 24 LEs to 7694 LEs, and covers several applications, but the majority of the circuits, at least 85 of the 205 circuits, are state machines or arithmetic logic [65]. Only the 77 multilevel benchmarks were originally available in BLIF formats, but some of the others have also since been converted into BLIF files. Although these circuits are small, their ease of use and wide acceptance make them a popular choice for comparison.

The Quartus University Interface Program (QUIP) benchmark suite [13] is more recent. It currently contains 45 real circuits. This suite was released by Pistorius et al. in [48] where they proposed the addition of a "black box" directive to the BLIF format which would represent hard blocks. Although these circuits are much larger (134,341 LEs) than the MCNC circuits (7694 LEs), only 7 of them can be synthesised to standard BLIF files, and of those the largest is 9,867 LEs.

Adoption of the "black box" directive would allow researchers to take advantage of circuit suites that are released by the ASIC community such as the International Symposium on Physical Design (ISPD) 2007's placement and routing benchmarks and the International Conference on Computer Aided Design (ICCAD) 2004's mixed-size placement benchmarks.

The eASIC suite is a recently released suite that provides a series of 5 ASIC netlists [17]. Their large size, from 125k to 1.01M elements, attraceive for FPGA research. However, they are composed of logic cells that do not translate well to FPGA structures.

These ASIC suites are regularly updated, although they only include a few circuits with each release, and the circuits require conversion to BLIF files to be used for FPGA synthesis.

2.4.3 Open Source Circuit Repositories

OpenCores [47] is an online repository of open source circuits, often written in an HDL language. It contains a wide range of circuits and is frequently updated, however, converting circuits into BLIF files is tedious. The circuits in the repository tend to be single function logic cores which are intended to be stitched together by the user to form more complete circuits. As a result, the circuits found in the repository are small; one of the processors contains 5,166 LEs.

2.4.4 Synthetic Circuits

There have been several previous attempts to generate synthetic circuits. These generators were constructed assuming a homogeneous circuit structure; as described in the introduction, this may not match the structure of large system-level circuits. There is little information about how well these circuits scale, since most of the circuits are validated against the MCNC benchmarks. An advantage of synthetic benchmark circuits is that they can be "made to order" while other circuit sources are inflexible. The approach used by previous generators can be divided into two different approaches, bottom up and mutation.

Rent Parameter Many of these generators attempt to ensure that the Rent parameter of each output circuit is reasonable. The Rent parameter is used in Rent's rule and the parameter is an indicator of the interconnect complexity of a circuit. Rent's rule is an experimentally observed relationship between the size of subcircuits returned after partitioning and the number of I/O pins in the subcircuits [36]. The relationship is described by the following equation

$$T = tg^p$$

where T is the number of pins, t is a constant scalar, g is the size of the subcircuit, and p is the Rent parameter.

The Rent parameter is often measured using recursive partitioning. According to Rent's rule, the value of the parameter should be constant as partitioning proceeds. As shown in [36], however, when the subcircuits are large, the measured p tends to be lower than when the subcircuits are smaller. This observation is attributed to the efforts of circuit designers to restrict the number of I/O in the circuit to what can fit on the perimeter of a minimum-size chip. In [53], it was shown that when partitioning very small subcircuits p is higher. This is attributed to the lack of hierarchy in the logic of these small subcircuits.

In all cases, measurement of the Rent parameter is highly dependent on the method used to partition the circuits. A poor partitioner can return partitions with high pin counts thus skewing the measured Rent parameter. The work in [23] therefore proposes the concept of an intrinsic Rent parameter $p*$ which is the minimum possible Rent parameter over any partitioning of that circuit.

Bottom Up

GEN Hutton [27, 26] developed a generator (GEN) using an empirical approach that was later extended by Kundarewich [33]. The generator is based on a number of structural circuit characteristics. The characteristics used are the circuit size (nodes and edges), the number of external inputs and outputs (I/Os), the node shape (the ratio of nodes at each delay level), the edge shape (the relative distribution of edges between delay levels), the fanout distribution, and the reconvergence (a function of the nodes' output cones). A complementary program, CIRC, was developed to measure the value of these characteristics for an existing circuit.

Typically, a circuit is first characterised using CIRC to obtain a specification file that

describes that circuit's properties. This file becomes the input to GEN which builds a circuit "clone" matching the described properties. The researcher can manipulate the values in the specification file to tune the properties of the generated clone. The generator can also assign reasonable values for parameters that the user has left blank.

The generation process has two stages. First, the values in the specification file are defined if the user left them blank; then, the circuit is generated from this description. The distributions used to generate values for the blank parameters are based on the structure of MCNC circuits. The Rent parameter is used to determine the number of I/Os, but the other parameters depend on the circuit size.

The algorithm next allocates the nodes and edges to a logic delay level according to the node shape and the edge shape respectively. The values for all four of these variables are defined in the specification file. Next, the edges are assigned a length. The length of an edge is defined as the number of logic levels the edge crosses.

The generator then connects these elements into a complete circuit in a multistep process. The edges are first assigned to the different levels. Each level is assigned a number of fanout degrees, one degree for each edge in this level. Next, the nodes within each level are assigned an index. The next step pairs each node with an edge and a fanout degree. Lastly, a sink is assigned to each edge. The output pins are generated in a final step. The outputs of the nodes on the bottommost delay level are assigned to external pins, and any remaining unconnected external pins are driven by random nodes.

Locality is used to help constrain the generation so that the connections are not completely random. Note that the nodes on each level are ordered. This affects the generation in two ways. First, the allocation of fanouts to each individual node is done such that the high fanout values tend to be given to the central nodes in each level. Second, the locality affects the allocation of sinks. The nodes on the source level and the nodes on the sink level are divided into a number of intervals equal to the locality

parameter. For each edge, the sink node in the interval closest to the source node is selected.

To validate this work, the post-routing results of the cloned circuit were compared to the results of the original circuits using the process shown in Figure 2.5. The difference between the results for these two circuits averaged between 17% for the wirelength, and 12% for the critical path delay using the hierarchical extension. There is no investigation into the effects of increasing circuit size.

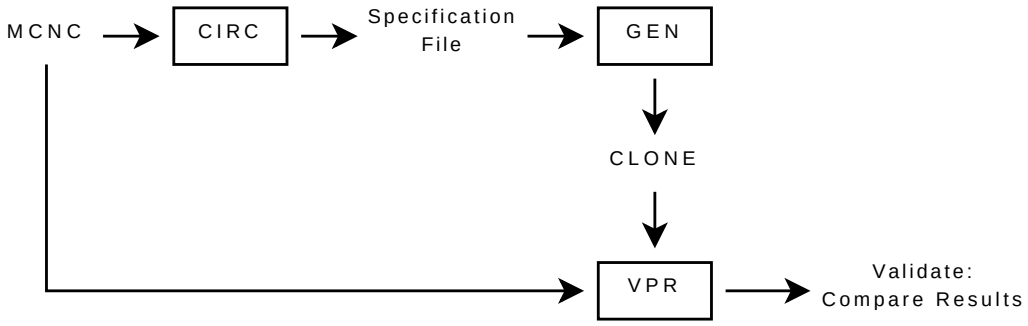


Figure 2.5: GEN Validation Process

The post-routing results of the generated clones were also compared to the post-routing results of random netlists. They found that the structural characteristics of the random circuits did not match patterns observed in the MCNC circuits, and that these circuits were universally harder to place and route than MCNC circuits of equivalent size.

In the first extension [26], the generator was modified to produce both combinational circuits and sequential pipeline-style circuits. To generate sequential circuits, a number of combinational subcircuits are defined and generated. Registers are placed between the subcircuits so that the result is a pipelined, sequential circuit.

With the extension developed by Kundarewich [33], GEN can also use partitioning information to develop hierarchical circuits, although the ability to automatically choose circuit characteristics was lost. CIRC extracts additional information related to the clustering of the circuit, such as the number of intercluster edges, and the probability

that the outputs of a cluster connects to flipflops or combinational nodes of the subsequent cluster. This information is used to connect the individually generated clusters together into one macro-circuit.

Gnl Stroobandt's generator Gnl [54] uses an analytical approach. The circuit model is based on Rent's rule and on the ratio of multiterminal nets in a circuit. The nodes are recursively clustered in pairs while maintaining the desired Rent parameter. These circuits were developed to test partitioning algorithms so the primary validation metric is the degree to which the synthetic circuits' Rent parameter matches the desired values.

Since the Rent parameter is different at various levels of the circuit [36, 53], Gnl allows the user to specify the Rent parameter for specific levels of the circuit. Stroobandt obtains the ratio of multiterminal blocks from [32] which states that 75% of the nets in ASIC designs are 2 and 3 terminal nets and from further research by [55] which proves that given Rent's rule, the distribution of net degrees follows a power law decay. Stroobandt uses the same distribution for the block terminals, with 75% of the blocks having 2 or 3 terminals and a power law decay distribution for blocks with an increasing number of terminals.

In Gnl, the user provides a specification file describing the library, the population of each library element in the circuit, and a step function describing the generation constraint that will be active when the circuit reaches particular sizes. Each library element is described by the number of its input and output terminals. The constraints that can be specified by the user are the Rent parameter, the number of I/O pins, and the ratio of input to output pins.

The nodes are clustered recursively in pairs. First, the number of external terminals for the resulting cluster is calculated using the size of the component clusters (sub-clusters) and the Rent parameter. Again, using the Rent parameter, limits are derived for the number of connections allowed between the sub-clusters. Limits for nets that stay

purely inside the cluster, nets that connect outside of the cluster, and nets that connect both within the cluster and outside the cluster are considered. Using a predefined ratio of external to internal connections, the connections are distributed such that the limits are obeyed.

The limits imposed on the number of connections per block ensure that the resulting circuit adheres to Rent's rule. However, problems can arise if the number of I/Os specified by the user is low. The number of inputs is easily reduced by merging pins together, but outputs can only be removed by connecting them to inputs. In order to obey the ratio of input pins to output pins, the circuit will connect the output pin of a sub-cluster to the input pin of the same sub-cluster thus generating combinational loops. This problem becomes more frequent as the circuit grows larger.

Extensions were added to improve the structural characteristics of the circuits. In order to make the work more suitable for timing-aware experimentation, [62] adds flipflops when necessary to control the logic depth. Stroobandt compares the resulting average wirelength both before and after placement with the ISPD98 benchmarks, but there is no comparison of the maximum path length. In another extension, some randomness is added to the number of terminals of each cluster so that the Rent parameter does not adhere so closely to the specified value, but rather shows some variation as might be expected in a real circuit [62]. This extension also allows the user to specify subcircuits as part of the library thus allowing heterogeneity to be explicitly defined in the circuits.

PartGen Pistorius's work in [49] is the most similar to the work described in this thesis. Pistorius assumes that circuits are composed of several different kinds of logic: regular and combinational, irregular and combinational, memory, controller, and interconnection, each of which has its own properties. By varying the proportion of these various types of logic, this generator is able to mimic different kinds of circuits in a realistic and controllable manner. The tool joins together sections of logic on a per bit basis into

primary circuits.

The tool is based on the composition of five industrial netlists. This work was developed to test the partitioning of circuits onto multi-FPGA systems; therefore, the circuits are validated by comparing the number of FPGAs necessary to implement a circuit built by GEN versus the equivalent circuit built by Partgen. The regular combinational logic is created using a multiplier generator. GEN is used to generate the irregular combinational logic, while the memory and the controller logic are represented using predefined functional elements. The interconnection is generated as necessary to connect the modules but the properties and the realism of this logic is never explicitly investigated.

Tom Method Another synthetic circuit generator was described in [57]. This generator was developed in order to build circuits to help the development of a new logic block clustering algorithm. This generator builds large circuits by randomly connecting BLIF subcircuits. These circuits were built by connecting the I/Os either in pipeline-style where the outputs of one block are driven to the inputs of a successor block, clique-style where the outputs of each block are distributed to the inputs of the other blocks, or independent-style, where no connections are made between the blocks. These algorithms result in circuits that contain a large number of I/O pins. The generator was never explicitely evaluated in [57].

Mutation

Another method to generate circuits is the mutation approach. The most recent such work is Perturber [22], which modifies only localised portions of the logic so structural characteristics of the circuit such as fanout, and I/O are kept the same. Other works include Ghosh [20], and Harlow [24] which are signature-based. This approach characterises circuits using an algorithm to compress a feature of a circuit so that it can be expressed by a smaller set of values, the signature. All circuits that share the same sig-

nature are expected to have similar values for the specified feature and are assumed to be comparable in the rest of their properties.

The mutation method is effective at generating a family of circuits similar to an existing circuit, but it lacks the ability to generate new circuits of different size or of different structure.

Perturber Perturber was developed to test incremental place and route algorithms [22]. These circuits try to mimic small changes that may be made by a designer as he or she iteratively develops a design. The user selects a region of the circuit to perturb. Random pairs of edges that have identical source and sink levels and that are not outputs of either an input node or a flipflop swap sinks until the desired percentage of the edges have been modified. In order to preserve the locality of the logic, the algorithm further specifies that the edge pairs must share a common ancestor within x levels, where x is a user defined value. This additional constraint improves the post-routing characteristics of the circuit. With ancestor control, the most significant difference found between the mutated circuit and the original MCNC circuit was in the average wirelength which differed by 17%.

Ghosh Ghosh generates circuits that have equivalent wiring signatures. Similar to the work by Grant [22], this method preserves the nodes' logic depth, and only changes the edges; however, the changes are not localised to a specific region, and the fanout degrees of the circuit nodes can also be changed. The signature generation algorithm in [20] accepts only circuits that are acyclic, and that are composed of circuit elements with only one or two inputs. This method first organizes the circuit nodes by their maximum slack. The algorithm then characterises the distribution of node types (input, output, etc) and the fanout bounds for each of the node types at each delay level. The user decides what percentage of the edges will be ripped up. The algorithm replaces edges

randomly such that the final circuit respects the signature described earlier.

The resulting circuits are validated by comparing their graph properties against the properties of randomly generated circuits. The signature of one benchmark circuit was used to generate 8 sets of circuits each with a different percentage of modified edges. One hundred circuits are in each set. These 8 sets were placed and routed using OASIS [31], a standard cell VLSI synthesis suite. For the layout area, there was a maximum difference of 1.5% within the 100 circuits in each set, and a maximum difference of 33% between the set of circuits with unmodified edges and the mutant sets. The total wirelength had a similar difference between the circuits within each set, but a maximum difference of 63% between the original and mutant sets. The amount of redundant logic varied by a maximum of 6% between the original and the mutant sets.

Harlow Harlow's method [24] is designed to generate a set of equivalent binary decision diagrams (BDDs). A BDD is a tree structure which can represent a combinational circuit, but this work does not directly address the properties of the equivalent circuits. The generation process first calculates the output values of the original circuit for each set of possible inputs. Next, the information entropy for the set of output values is calculated. If there are multiple outputs, the entropy is calculated for each output pin. The entropy is used as the signature for equivalent circuits. This work simply swaps the order of the output values in the set; thus generating circuits with the same entropy, but with different logic functions. The study found that the minimum size BDD representation for the mutated circuits varied showing that the mutated circuits were structurally different than the original. The difference found was proportional to the number of output values that were swapped.

2.5 Focus and Contribution of Thesis

Instead of generating circuits at the bit level, this work generates circuits at the word level. Existing circuits are connected in a realistic manner into larger circuits. This approach reflects the modular philosophy used to design the majority of modern circuits.

The algorithm described in this thesis can stochastically generate high level descriptions of SoC-style circuits with little required user input. SoC block diagrams were examined to determine the distributions of the functional composition and structure of current circuits. The generator uses these distributions to automatically create the specified circuits in a top-down manner, first generating logical connections between the modules in the circuit, then moving to the bit level to make the connections between the module pins.

By sorting the functional modules into four categories, and by providing three common connection styles, this method can flexibly generate different types of circuits. The modularity of the circuits leads to naturally heterogeneous structural properties. This work examines the ability of the generator to build realistic circuits. It also examines the generator's scalability.

The contributions of this thesis are:

1. Results from a characterisation of the composition and structure of contemporary SoC circuits.

2. A top-down synthetic circuit generator that integrates existing circuits into a larger SoC-style structure based on the above results.

3. An understanding of how various circuit generation methods respond to scaling.

Chapter 3

Circuit Analysis

3.1 Overview

To generate realistic circuits, it is first necessary to understand the characteristics of modern SoCs. The data described in this chapter lays the foundation for the circuit generation algorithms and also for future research based on SoC circuits. We know of no previous work that analyses relationships between SoC characteristics despite the prevalence of such designs.

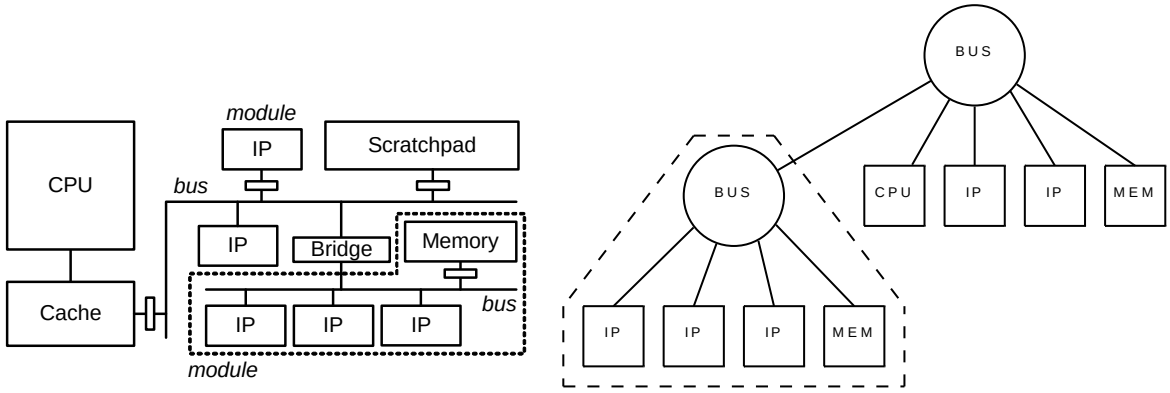
In this chapter, we first develop a model that abstracts a real circuit into a standard form which is more easily characterised. The circuit model described in Section 3.2.1 reflects the modularity and connection hierarchy found in many SoCs. It describes the word-level connections found between modules; single-bit nets are added to the model in Section 3.2.2 to represent system-level control signals.

In Section 3.3, the results from the circuit survey are presented. The information gathered from the survey will be used in Chapter 4 to calibrate the generation algorithms, and to define default parameters for the stochastic construction of the circuits.

3.2 Circuit Model

3.2.1 Definition

Our model of a SoC circuit is as follows. A circuit is a *tree* of *modules*. Each module can contain a number of lower-level sub-modules. Within a module, the sub-modules are connected using a common pattern such as a bus, a network-on-chip (NoC), or a datapath configuration. The interconnect structure within a module will be referred to as that module's *network*. The network includes all the logic and circuitry necessary to connect the functional modules. The lowest-level modules will be referred to as *leaf modules*. These can be processors, memories, or other sequential or combinational logic circuitry. This structure reflects the SoC design philosophy where IP modules are hierarchically combined to form a larger circuit.



(a) Original SoC circuit (b) Model Representation

Figure 3.1: Application of the Circuit Model to an Example SoC Circuit

Applying this definition to the circuit in Figure 3.1(a) results in the representation shown in Figure 3.1(b). The top-level module consists of five lower-level sub-modules: a CPU (central processing unit)/cache, a scratchpad memory, two IP modules, and one sub-module containing a child network, all connected using an on-chip bus (all but the sub-module containing the child network are leaf modules). The lower-level module

contains three IP sub-modules and one memory sub-module connected using a bus. All four of these sub-modules are leaves. The cache memory is connected exclusively to the CPU, so these two blocks are counted as one module. The bus bridge and the bus interface logic are considered to be part of the network implementation and thus are not counted. Our representation is simple, but it still retains the major outline of the circuit.

The International Test Conference (ITC) 2002 benchmark suite [40] for ASIC modular circuit verification uses a similar circuit model. It also assumes a tree-like hierarchy of modules. For each module, the benchmark provides its location in the tree, the number of I/O pins, and some verification information. However, descriptions of the module contents are not provided, nor is information about the style of connection between modules.

3.2.2 Fine-Grained Connections

The model describes word-level connections between modules but it does not describe bit-level connections. These single bit connections tend to be communication flags or similar signals. They relay information between modules, within networks or even across the whole chip.

These signals were not visible on the block diagrams that were surveyed to parameterise the generator. We assume the existence of reset, interrupt, and clock nets in each circuit. These are typical cross-chip signals that are used to coordinate activities across the SoC. Single bit nets between modules or within networks, for example flags that coordinate transfers between a bus interface and its IP block, are assumed to be included within the network connection, and so are not explicitly added to the model.

3.3 Survey Process

To calibrate the circuit generator, 66 different SoCs were surveyed. Of these, 42 were academic designs gathered from conference proceedings that appeared between 2004 to 2008. The remaining 24 were from industrial sources. The chips spanned a wide range of applications including network communications, multicore processors, and multimedia and were designed for various platforms, including FPGAs, and ASICs.

The survey characterises these published block diagrams using the circuit model in Section 3.2. Interpretation was sometimes needed to apply the model to a circuit. For example, connections between modules were broken so that the resulting structure was a tree in some cases. Information was extracted from both the block diagrams and the accompanying text in order to help make these decisions. Ultimately, however, the analysis was limited to what is provided in the documentation.

The structure and the content of each circuit were studied. The characteristics of the hierarchy were analysed and the results are reported in Section 3.3.1. Then, to characterise the contents of the circuit, the networks and the leaf modules were categorised into different types and their frequency is described in Section 3.3.2 and Section 3.3.3 respectively. Lastly, the distribution of the modules per network was analysed and the results are reported in Section 3.3.4.

Though one of the main goals of this research is to generate realistic, large circuits, the relationship between the size of the surveyed circuits and circuit model parameters was not tracked. Since our model treats modules as black boxes, we would need to to correlate the size of the circuit with the number of elements in our SoC circuit model. The accompanying documentation usually only provides the size for the circuit as a whole, and not for each of its components, yet, the content of a module in a block diagram can vary widely depending on the level of abstraction. As a result, the model cannot explicitly describe the size of the circuit.

3.3.1 Circuit Hierarchy

The circuit hierarchy was identified in the block diagrams. In some circuits, there were two unconnected networks. Multimedia circuits, for example, often have separate datapaths for audio and video. These circuits were broken into two designs and analysed separately. By this definition, the analysis includes 79 different designs. As described earlier, some connections do not follow a tree structure; these connections were ignored for this part of the analysis.

Figure 3.2 shows the frequency of hierarchy depths observed in the survey. The number of circuits seen with a particular hierarchy depth is represented by the vertical bars. Note that the solid line in Figure 3.2 (and subsequent graphs) represents the probability distribution that is used in synthetic circuit generation. It will be discussed in the next chapter. The hierarchy depth is indexed starting from 0. The 0th level, by definition, has a single network connecting a collection of sub-modules. The maximum depth seen in the survey was 3; however, this does not include any hierarchy which may be present inside the leaf modules (and hence is not shown on the block diagrams). Thus, the true hierarchy depth may be more than what is shown in Figure 3.2.

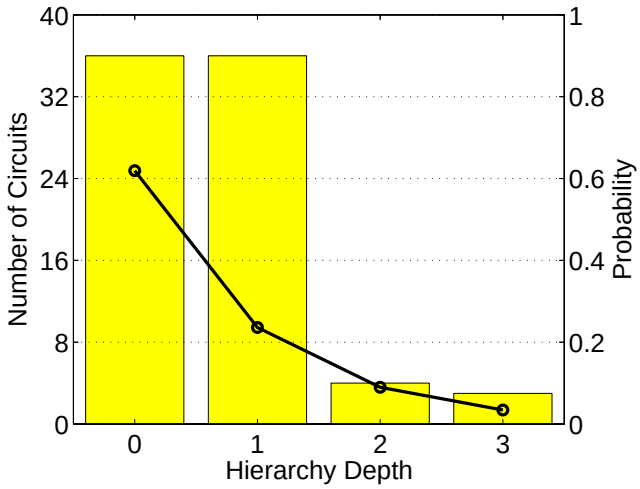


Figure 3.2: Hierarchy Depth Distribution

Table 3.1 shows the average number of networks at each hierarchical level. In a full tree, the number of networks per level would grow exponentially. The results in Table 3.1 show that this is not the case in these circuits; the number of networks for levels one and higher remains roughly constant. Many of the circuits have only a few modules that contain sub-modules, making the hierarchy far from a complete tree.

Table 3.1: Average Number of Networks per Level

Max Depth	Average
0	1
1	1.81
2	1.75
3	2.16

Figure 3.3 shows the distribution of the number of networks seen at a hierarchy depth of 1. Characterisation for other levels in the circuits was attempted, but there were prohibitively few examples.

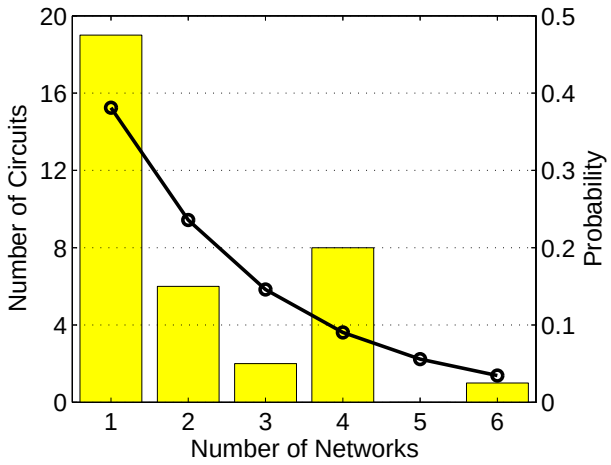


Figure 3.3: Number of Networks at Hierarchy Depth 1

3.3.2 Network Types

Networks connect modules together within a hierarchical level. We define three common connection patterns which will be referred to as *bus*, *dataflow*, and *star*, as shown in Figure 3.4.

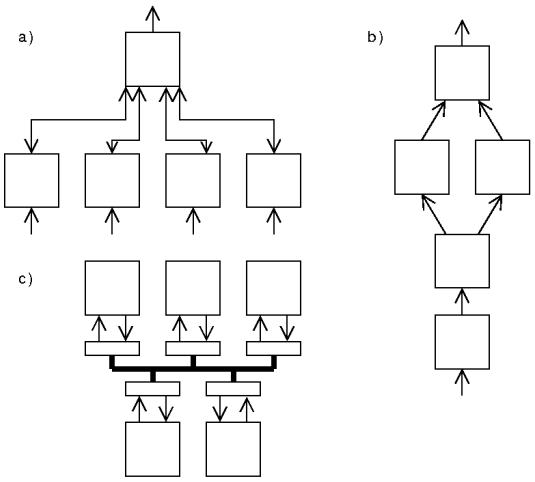


Figure 3.4: Example Network Block Diagrams a) Star b) Dataflow c) Bus

Figure 3.4(a) shows an example of a star connection pattern which is characterised by a bidirectional flow of data between the head and the tail sub-modules. This sort of pattern was commonly found in simple embedded controller chips, where the head sub-module is a CPU and the tail sub-modules are interface modules to off-chip resources. Figure 3.4(b) shows a dataflow pattern. This is defined by a predominantly unidirectional data flow between sub-modules. These circuits may or may not have feedback. The dataflow pattern is commonly seen in multimedia chips, and reflects the sequential nature of these applications. The sub-modules at the ends of the network tend to be interfaces for either off-chip communication or for communication with other functional units in the circuit. Figure 3.4(c) shows a bus interconnect structure, in which all sub-modules are connected to a common bus using an (often industry-standard) access protocol. Some of the protocols seen included Advanced Microcontroller Bus Architecture (AMBA) style

Advanced High-Performance (AHB) or Advanced Peripheral (APB) buses [6], and STMicroelectronic's SuperHyway [52]. Other network types (such as packet-based gridded networks) are possible, however, they rarely occurred in the circuits that were examined, so we do not present data on them. The ratio of the network types found in the survey is shown in Table 3.2.

Table 3.2: Network Type Distribution

Type	Ratio
Bus	53%
Dataflow	33%
Star	7%
Other	7%

Network characteristics for the bus and dataflow were analysed as well. The bus widths observed ranged from 16 bits to 256 bits. In order to characterise the dataflow network, each sub-module was first assigned to a stage. A sub-module is assigned to the stage subsequent to the the maximum of its predecessors, where the order of stages represents the number of modules passed in a forward walk along the data connections between sub-modules from the inputs not allowing feedback. The length of the dataflow is the number of stages while the width is the maximum number of modules in one stage. We found that 80% of the dataflow networks are longer than they are wide. Table 3.3 shows the distribution of dataflow lengths and widths found. The actual number of modules in the dataflow is generally much less than the maximum obtained by the product of the length and width.

3.3.3 Leaf Modules

For each circuit, the leaf modules were identified and sorted into five categories according to their function in the circuit: processors, interfaces, controllers, memory, and miscellaneous. Logic modules that are necessary for the network implementation are not counted

Table 3.3: Dataflow Dimensions

Length	# Circuits
1	2
2	13
3	7
4	9
5	8
6	1
7	1
8	2
8+	2

Width	# Circuits
1	18
2	16
3	4
4	5
5	1
6	1
7	0
8	0
8+	0

as leaf modules. For example, bus interface modules, bus arbiters, and buffers between pipelined stages are not counted.

Table 3.4 shows the frequency of each type. *Processor* circuits can be either CPUs or GPUs. *Interface* modules, for example UART interfaces, are used for simple data transfers between external devices. *Controllers*, such as memory controllers, are often built as finite-state machines (FSMs). Some modules that manage more complex data transfers such as USB controllers also fall into this category. *Memory* modules are stand-alone memory blocks such as scratch pad memories. The remaining modules are classified as *miscellaneous* and range from image processing circuits to custom purpose circuits.

Table 3.4: Module Type Distribution

Type	Overall	w/o Memory
Processor	10%	12%
Interface	34%	38%
Controller	22%	24%
Miscellaneous	23%	26%
Memory	10%	-

As will be discussed in the following chapter, our synthetic circuit model does not generate circuits containing memory blocks. The second column of the table shows the equivalent percentages scaled for the remaining four types. Table 3.4 shows that a chip

will commonly contain many interface modules for communication with other devices.

The modules themselves often contain memory. Many modules are connected to an private buffer or cache. These memories were not counted as independent modules because they are accessible by only a single functional unit. As seen in Table 3.5, processors frequently contain memory; often these memories are configured as one or two levels of cache. In many cases, the function of the modules implies that there would be internal memory, but the memory is not labelled on the diagram. Interfaces can include memory in order to buffer the incoming data, but their memory requirements are low, so often it is not explicitly shown. The actual percentage of modules that have internal memory is likely much higher than in Table 3.5.

Table 3.5: Modules with Internal Memory

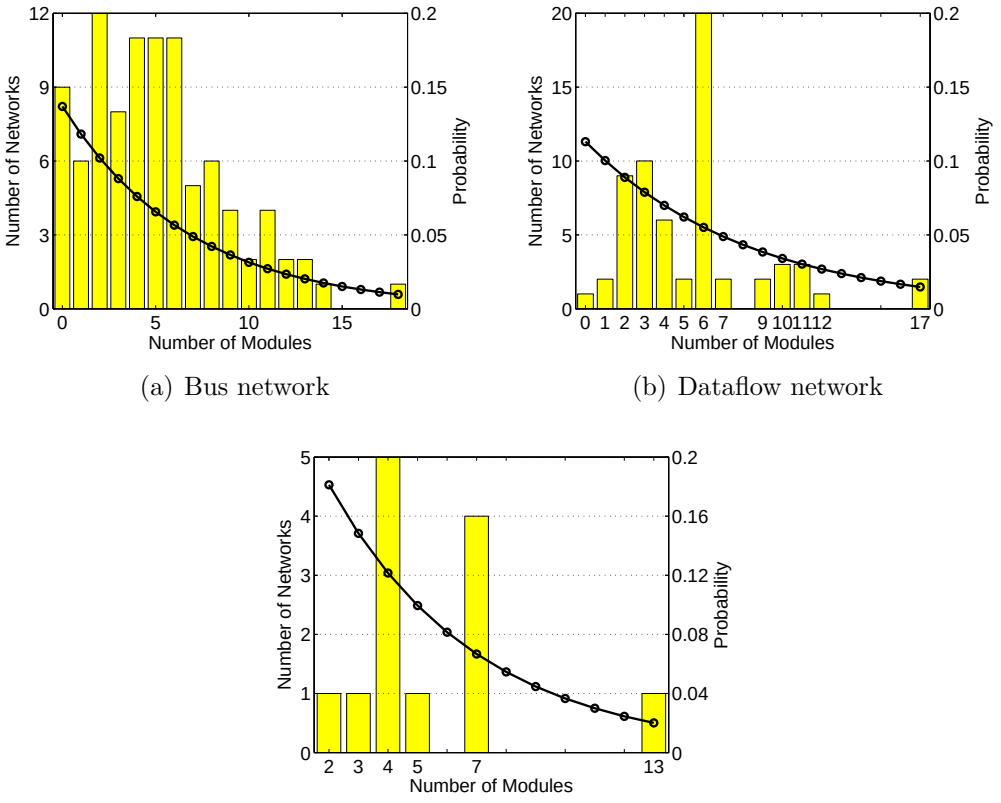
Type	Percentage (%)
Processor	53%
Interface	3%
Controller	5%
Miscellaneous	6%

3.3.4 Network Composition

Finally, the relationship between the networks and the leaf modules was examined. Figure 3.5 shows the distribution of the number of leaf modules in a network for each of the three network types.

Buses are generally large since the complexity of implementation remains relatively constant as the number of sub-modules grows. A dataflow typically has several stages; short pipelines are relatively rare. Star networks often connect a limited number of sub-modules because of the communication limits of this topology.

The distribution of leaf modules across the different hierarchy depths was also examined. These results are shown in Table 3.6. Circuits with only one level of hierarchy and



(a) Bus network

(b) Dataflow network

(c) Star network

Figure 3.5: Distribution of the Number of Leaf Modules for each Network Type

circuits with multiple levels of hierarchy are separated to isolate the effect of increasing depth. Only the first two levels are shown because there were only a few circuits with depths greater than 1. There are two clear trends: CPUs are found more often on the uppermost levels, and controllers tend to be at the lower levels. Leaf modules are commonly arranged such that the main computing elements are connected using a high speed network, while the peripherals are connected using a lower level, lower speed network.

3.4 Summary

In this chapter, we first presented the circuit model developed to represent SoC style circuits. The circuit is modelled as a collection of leaf modules, which represent the

Table 3.6: Composition of Leaf Module Types per Hierarchy Level

		Multiple Hierarchical Levels	
Type	Single Level	Level 0	Level 1
CPU	26%	17%	7%
Interface	8%	14%	17%
Controller	31%	24%	44%
Miscellaneous	36%	45%	32%

component functional subcircuits. These leaf modules are connected into a tree structure using networks, which represent the communication patterns used to join the subcircuits. Fine grained nets are represented in this model by adding reset, interrupt and clock nets to the system.

The model was then applied to existing circuits in order to characterise SoC designs. Each circuit was first characterised in terms of its hierarchy and then in terms of its contents. The networks were categorised into bus, dataflow, and star while the leaf modules were categorised into processor, controller, interface, memory and miscellaneous. The distribution of these elements was then measured with respect to different variables. The next chapter will describe how these trends are used to generate synthetic circuits.

Chapter 4

Circuit Generation

This chapter describes the synthetic circuit generation algorithm that was developed as part of this research. The generator was developed based on observations from the circuit survey described in the previous chapter. The circuit survey was also used to calibrate various probability functions used in the generator. A C language implementation of the algorithms has been made available to the research community.

The generator is required to produce circuits suitable for FPGA architectural experimentation. Such circuits must be realistic, and there must be enough variety in the generated circuits to adequately test an architectural feature under evaluation. We provide the ability to constrain the generation, allowing the user to create a "family" of benchmark circuits that have a specific number of modules or that have a specific property.

Since we intend to use these circuits to evaluate physical design CAD tools as well as FPGA interconnect and logic architecture, we are primarily concerned with the *structural* characteristics of the circuits. Of primary importance is the interconnect patterns within the circuits. Conversely, since we do not intend to use these circuits to evaluate logic synthesis algorithms, the functionality of our generated circuits is not important. In all cases, our circuits do not actually "do anything", but for the types of studies we intend to perform, this is not a requirement.

This chapter is organized as follows. We first describe an overview of the generator in Section 4.1. The algorithm is then detailed in Sections 4.2 to 4.4. Finally, Sections 4.5 describes the mechanics of our implementation.

4.1 Generator Overview

The structure of our generator is summarised in Figure 4.1. The user supplies parameters in a *constraints file* as well as a library of logic circuits that will be used for the leaf modules in the circuit. The library is divided into different directories according to the circuit category (processors, controllers, interfaces, and miscellaneous) as described in Section 3.3.3. In this research, circuits from the MCNC benchmarks were separated according to their function and used in the library but these circuits can also come from other existing benchmark sources [47, 65] or even from other synthetic circuit generators [22, 26, 27, 33, 56].

The generator uses the constraints file and the library of circuits to construct the synthetic circuit. The result is written in a Berkeley Logic Interchange Format (BLIF) file, which is the format needed for T-VPACK/VPR synthesis.

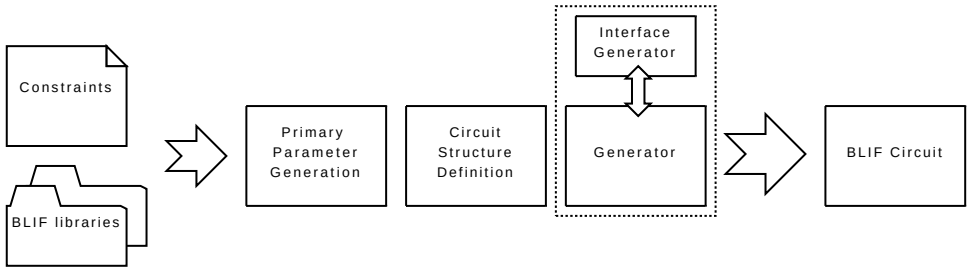


Figure 4.1: Circuit Generation Flow

Within the generator, construction consists of three stages. First, the data distributions seen in the circuit analysis are used to select values for four *primary parameters*. These primary parameters describe the overall size and "shape" of the circuit. The selection of primary parameters is described in Section 4.2. The second stage uses these primary parameters to construct the overall hierarchy of the circuit, in terms of the size and interconnect pattern of the networks at each level of the hierarchy. This second stage is described in Section 4.3. The third stage, which is described in Section 4.4, con-

nects the leaf module pins at the bit-level to form the networks described by the circuit structure, and resolves mismatched pin counts between the sub-modules.

4.2 Primary Parameter Generation

The first phase of the generation is to determine values for the *primary parameters*. These parameters, which are summarised in Table 4.1, directly describe the high-level characteristics of the circuit: the hierarchy depth, the total number of networks, the number of leaf modules and the bus width. The number of leaf modules and networks set the contents of the circuit while the hierarchy depth roughly constrains the shape of the network tree. We assume that all buses within a single circuit are of the same width. This is realistic for many signal processing-style circuits which operate on fixed size words; the bus width parameter indicates how wide these words are.

Table 4.1: Primary Parameters

Bus Width
Hierarchy Depth
Total Number of Networks
Total Number of Leaf Modules

The user can set values for these primary parameters in the constraints file. Any combination of these primary parameters can be specified and the unspecified values will be determined by the algorithm. All other parameters and constraints must be specified. Default values for these other parameters are described in the following sections. While the defaults provide a good starting point to generate new circuits, these values should be tuned to match the characteristics of the circuits in the library or the desired range of circuit structures.

The following subsections describe how we choose values for any unspecified primary parameters. We first outline how we choose values for the parameters if *none* of the

parameters are specified by the user, if *all* of the parameters are specified, and if *one or more* parameters are specified.

4.2.1 No Primary Parameters Specified

If no parameters are specified by the user, the algorithm must choose values for all four parameters.

All parameters except for the bus width are dependent on each other. The value chosen for the hierarchy depth, for example, affects the reasonable range of choices for the total number of networks and the total number of leaf modules. Our approach selects values in this order: the number of hierarchical levels, the total number of networks, and finally the total number of leaf modules. For each of the latter two parameters, constraints based on the value of the previous parameter are used to guide the selection. In addition, the user can specify additional constraints for each of the parameters.

The following subsections describe each parameter; the constraints used for each parameter are summarised in the second column of Table 4.2. With the exception of the constraint defining the minimum hierarchy depth, the remaining constraints relate adjacent primary parameters. For example, the constraint for the number of leaf modules relates this parameter to the number of networks in the circuit.

Table 4.2: Primary Parameters and Primary Parameter Constraints

Primary Parameter	Primary Parameter Constraints
Bus Width	Minimum bus width exponent, maximum bus width exponent
Hierarchy Depth	Minimum hierarchy depth
Number of Networks	Maximum average number of networks per level
Number of Leaf Modules	Minimum average number of leaf modules per network, maximum average number of leaf modules per network

Bus Width

The bus width is independent of all other parameters. In this research, the probability distribution used to stochastially select values for parameters where the range is finite is chosen to match the observed data gathered during the suvey wherever possible. If there is insufficient data, the distribution is assumed to be uniform. For this parameter, we choose a power-of-two between 16 and 256, with all powers-of-two being equally likely. This range matches the range we observed during our circuit analysis, however the user can override the upper and lower bound in the constraints file.

Hierarchy Depth

The value for the hierarchy depth is chosen from the exponential probability distribution $p(x) = \lambda e^{-\lambda x}$ where $p(x)$ is the probability of choosing a hierarchy depth x and $\lambda = 0.97$. The value for λ was obtained by fitting the data in Figure 3.2 to the exponential distribution using the regression tools in Microsoft Excel. In this research, an exponential distribution was used to stochastically select values for variables where the range is unbounded. The limited data found in the survey makes it challenging to confidently select a representative distribution for such variables. The exponential distribution has the benefit of being simple to generate and to control. The user can override the default value of λ in the constraints file, and in the case of primary parameters such as the hierarchy depth, the user can also set the value of the variable manually if desired.

The value chosen for the hierarchy depth is further constrained to be larger or equal to the *minimum hierarchy depth* constraint as shown in Equation 4.1. The value for the minimum hierarchy depth constraint is specified in the user file. Note that the number of hierarchy levels is one higher than the maximum hierarchy depth (the hierarchy depth is indexed from 0).

In this generator, an exponential probability distribution is used to stochastically

generate values for parameters where the range is unbounded such as the number of networks found in a circuit.

$$\text{hierarchy depth} \geq \text{minimum hierarchy depth} \tag{4.1}$$

Total Number of Networks

The selection of a value for the total number of networks is based on the data from Table 3.1 and Figure 3.3. Figure 3.3 shows the observed distribution of the number of networks on Level 1. There was insufficient data for the higher levels to observe any clear trend. However, as shown in Table 3.1, the average number of networks on each level above 0 stays relatively constant; therefore, it is assumed that the distribution in the number of networks seen on Level 1 is also true for higher levels.

Rather than choosing a single value for the total number of networks directly, we compute the sum of n independently generated values, where n is the number of hierarchical levels (determined previously). Each of the n values, n_i, corresponds to the number of networks on hierarchical level i. The value for each n_i is generated using an exponential distribution that is fitted to the data in Figure 3.3 according to the equation $p(n_i) = \lambda e^{-\lambda(n_i - 1)}$ where $n_i \geq 1$. From the data in Figure 3.3, we estimate $\lambda = 0.480$, although the user can override this value in the constraints file.

The *total* number of networks is then obtained by summing the number of networks per level over all levels:

$$\text{total number of networks} = \sum_{i=0}^{n} n_i \tag{4.2}$$

where n is the number of hierarchical levels. The total number of networks is then compared to the upper and lower user defined bounds for this parameter according to

the following equations:

$$\text{total number of networks} \leq \text{max average number of networks per level} * (\text{hierarchy depth} + 1) \quad (4.3)$$

$$\text{total number of networks} \geq \max(1, \text{hierarchy depth}) \quad (4.4)$$

where the maximum average number of networks per level is a constraint that is specified by the user in the constraint file. If either of Equations 4.3 or 4.4 do not hold, a new set of n values is generated as above.

Note that we have chosen not to include a constraint for the minimum number of networks per hierarchical level. Such a constraint would help ensure that the number of networks grows with the hierarchy depth, but as seen in Table 3.1, the average number of networks for levels above 0 is approximately 2. In order to allow sufficient variation, the minimum average number of networks per level would have to be set very low. Instead, as described above, we constrain the total number of networks to be at least as large as the number of hierarchical depths, which is equivalent to a minimum average number of networks of 1.

Total number of leaf modules

As seen in Figure 3.5, the network type (bus, dataflow, or star) affects the distribution of the number of leaf modules in the network. Therefore, to calculate the total number of leaf modules, it is necessary to know how many networks of each type are present in the circuit.

Although it would be possible to select the number of each type in the circuit directly from the results in Table 3.2, doing so would result in the frequency of each network type in a large circuit approaching the mathematically expected value. To ensure that our

generator produces a diverse family of benchmarks even as the size of the circuit grows, we do the following.

We first determine the number of network types that will be present in the circuit. This is a value between 1 and 3; a value of 1 means that only one type of network is represented in the circuit, and a value of 3 means that each network type is represented at least once in the circuit. We choose this quantity randomly, assuming that probability of having 1, 2, or 3 types is 40%, 50%, and 10% respectively. The user can override these values in the constraints file. The number of network types is also constrained by

$$\text{number of network types} \leq \text{total number of networks} \tag{4.5}$$

If the above does not hold, a new value is chosen.

Once the number of network types is chosen, we choose the number of networks of each type using the data in Table 3.2. In the following, we denote the number of bus networks as m_{bus}, the number of dataflow networks as m_{dataflow}, and the number of star networks as m_{star}.

Next, the total number of leaf modules in the circuit is generated. For any given network, the number of leaf modules was assumed to follow an exponential distribution $p(x) = \lambda e^{\lambda x}$ where $p(x)$ is the probability of choosing x leaf modules and λ depends on the type of the network, and was obtained from the data obtained during our circuit analysis. From this analysis, we estimate that λ for a bus network is 0.147, λ for a dataflow network is 0.120, and λ for a star network is 0.200 (the user can override these values in the constraints file). We compute the number of leaf modules on each network independently; in the following we denote the number of leaves on bus network i as $\text{leaves}_{\text{bus},i}$ where $(1 \leq i < m_{\text{bus}})$, the number of leaves on dataflow network i as $\text{leaves}_{\text{dataflow},i}$ where $(1 \leq i < m_{\text{dataflow}})$, and the number of leaves on star network i as $\text{leaves}_{\text{star},i}$ where $(1 \leq i < m_{\text{star}})$.

We then sum the results as follows:

$$\text{number of leaves} = \sum_{i=0}^{m_{\text{bus}}} \text{leaves}_{\text{bus},i} + \sum_{i=0}^{m_{\text{dataflow}}} \text{leaves}_{\text{dataflow},i} + \sum_{i=0}^{m_{\text{star}}} \text{leaves}_{\text{star},i} \quad (4.6)$$

Finally, the resulting number of leaves is compared to the following constraint:

$$\text{total number of leaf modules} \leq$$
$$\text{maximum average number of leaf modules per network} * \text{number of networks} \quad (4.7)$$

$$\text{total number of leaf modules} \geq$$
$$\text{minimum average number of leaf modules per network} * \text{number of networks} \quad (4.8)$$

where the maximum average number of leaf modules per network is specified in the constraint file. If the above constraints are not satisfied, new values for the number of leaves on each network are selected, and the process repeats.

4.2.2 All Primary Parameters Defined

If values for all of the primary parameters have been defined by the user, the number of networks of each type in this circuit is calculated as described in Section 4.2.1. Generation then progresses to the next stage.

4.2.3 Some Primary Parameters Specified

Section 4.2.1 assumes that the algorithm has the freedom to choose values for all primary parameters. However, if the user specifies values for some of the parameters, additional constraints are imposed on the generation of values for the remaining parameters. As an

example, if the user specifies that there are 100 leaf modules, then this should influence the selection of the number of networks. In this section, we describe how these additional constraints are determined.

There are six possible ways in which the user can partially specify the primary parameters, as shown in Table 4.3. In the table, each column represents one possible scenario; for example, in Scenario A, the user has specified the hierarchy depth and the number of networks, but not the number of leaf modules. The bus width is not shown, since it is independent of all other parameters.

Scenarios A and B are straightforward extensions to what was described in the previous section, and no new constraints are imposed. Scenarios C to F, however, are more complex and are described below.

Table 4.3: Possible Combinations of User Input

	A	B	C	D	E	F
Hierarchy Depth	x	x	x			
Number of Networks	x			x	x	
Number of Leaf Modules			x	x		x

x represents a user defined value

Generation for Scenario C

In Scenario C, the number of networks must be chosen. A value is chosen as in Section 4.2.1 with the additional constraints:

$$\text{number of networks} \geq \frac{\text{number of leaf modules}}{\text{maximum average number of leaf modules per network}} \tag{4.9}$$

$$\text{number of networks} \leq \frac{\text{number of leaf modules}}{\text{minimum average number of leaf modules per network}} \tag{4.10}$$

where the maximum and minimum average number of leaf modules per network are specified by the user in the constraints file.

Generation for Scenarios D and E

In Scenarios D and E, the hierarchy depth must be chosen, and should be constrained by the specified total number of networks. This is done by adding the following two constraints to the generation of the hierarchy depth:

$$\text{hierarchy depth} \leq \text{number of networks} \tag{4.11}$$

$$\text{hierarchy depth} \geq \frac{\text{number of networks}}{\text{maximum average number of networks per level}} - 1 \tag{4.12}$$

For Scenario E, the value for the number of leaf modules is then computed as described in Section 4.2.1.

Generation for Scenario F

In Scenario F, there are two unspecified parameters. In this case, the constraints in Equations 4.9 and 4.10 still apply. Substituting these into Equations 4.11 and 4.12 gives the following additional constraints on the number of hierarchy levels:

$$\begin{aligned} \text{hierarchy depth} &\leq \frac{\text{number of leaf modules}}{\text{minimum average number of leaf modules per network}} \\ \text{hierarchy depth} &\geq \frac{\text{number of leaf modules}}{\text{maximum average number of leaf modules per network}} * \\ &\quad \frac{1}{\text{maximum average number of networks per level}} - 1 \end{aligned} \tag{4.13}$$

4.3 Detailed Circuit Structure Generation

In this stage, the circuit elements are arranged into a tree structure as described by the model in Section 3.2. Figure 4.2 illustrates the terminology used in the following sections. This section describes the algorithm employed in this stage.

The inputs to this stage are the the primary parameters and the number of networks of each type from the previous stage. Intermediate values such as the number of networks for each hierarchy depth that may have been generated in the process of determining values for primary parameters are discarded. In this manner, the generation of the detailed circuit structure is independent of the method used to assign values to the primary parameters.

First, the network tree structure is constructed. Each hierarchy level is assigned a single network. The algorithm then assigns the remaining networks to a random hierarchy level. All levels between 1 and the maximum hierarchy depth are equally likely, so that the expected number of networks on each level is equal. Level 0 is not considered, since level 0 always contains exactly one network. To connect the networks into a tree, each network is randomly assigned to a parent network, chosen from the networks on the previous level. At the end of this step, the modules have been arranged into a tree.

Second, the algorithm labels each network with a type (bus, dataflow, or star). The number of labels for each network type was determined in the previous stage. As each network is labelled, the number of unassigned labels for the respective type is decremented. The probability of selecting a type is directly proportional to the number of unassigned labels of that type.

Program 4.1 Pseudocode for Network Type Allocation

```
/*Assign network types to networks given the number of each network type
in the circuit*/
for i = 1 to number_networks{
    total_unassigned_types = unassigned[bus] + unassigned[dataflow] +
                                  unassigned[star]
    network[i]->type = weighted_probability(
                           unassigned[bus]/total_unassigned_types,
                           unassigned[dataflow]/total_unassigned_types,
                           unassigned[star]/total_unassigned_types)
    unassigned[network[i]->type] --
}
```

Third, the leaf modules are assigned to the networks. Program 4.2 summarises the algorithm used. Each network module that has not already been assigned a sub-module (i.e. if it is not already attached to a child network) is assigned one leaf module. A desired number of additional leaf modules is generated for each network using the appropriate exponential distribution function in Figure 3.5. The total number of desired leaf modules over all the networks is scaled to equal the remaining number of leaf modules. Each network is allocated a number of leaf modules equal to its desired number of leaf modules multiplied by the same scaling factor.

Program 4.2 Pseudocode for Allocation of Leaf Modules

```
remaining_leaves = number_of_leaves
for i = 1 to number_networks{
    if network[i]->modules < 1{
       network[i]->modules ++
       remaining_leaves --
    }
}
for i = 1 to number_networks{
    desired_leaves[i] = exponential_probability(network[i]->type)
}
total_desired_leaves = sum(desired_leaves)
for i = 1 to number_networks{
    network[i]->leaves = desired_leaves[i]/total_desired_leaves*
                              remaining_leaves;
}
```

At the end of these three stages, the circuit structure has been specified. Modules containing networks have been connected into a tree, and have been allocated a certain number of leaf modules. The networks have been assigned a type. Next, the remaining details of the circuit structure are defined. An order is assigned to the sub-modules of dataflow and star modules, and the leaf modules are assigned a type and a library circuit.

The dataflow and the star modules have an implied ordering among their sub-modules. The algorithm first determines the "shape" of the dataflow module by randomly selecting

a *width* and *length* for the module. As explained in Section 3.2.1, the length of a dataflow module is the number of stages found in the network while the width is the maximum number of sub-modules in one stage. The length is chosen with uniform probability from 1 to the number of sub-modules connected by the network. The width is then chosen with uniform probability between the maximum and minimum possible values shown in Equations 4.14 and 4.15.

$$\text{width} \leq \text{number of sub-modules} - \text{length} + 1 \tag{4.14}$$

$$\text{width} \geq \left\lceil \frac{\text{number of sub-modules}}{\text{length}} \right\rceil \tag{4.15}$$

One sub-module is assigned to each stage, and one stage is chosen to contain the maximum number of sub-modules allowed by the width. This ensures that the dataflow will obey the selected length and width. The remaining sub-modules within the dataflow module are then randomly assigned a stage within the module. If the selected location for the sub-module will cause the dataflow to exceed the limits set by the length and width, a new location is selected.

For the star network, the first sub-module in the star module's data structure is chosen to be the head; the rest of the sub-modules are automatically the tails.

Next, a module type is assigned to each leaf module: processor, core, controller, or interface. The overall probability for each leaf module type is the same as the observed percentage of leaf modules of that type seen in the survey as shown in Table 3.4. In order to mimic the changing leaf module composition seen between the different levels of the hierarchy, as shown in Table 3.6, a user controlled parameter modifies the relative probability of each type depending whether the module is found on the upper or the lower half of the circuit structure tree. For example, the user can decide to double or triple a type's probability for leaf modules on the top half of the circuit hierarchy. The probability

of that type will concomitantly be cut in half or a third respectively for the bottom half of the hierarchy. Program 4.3 illustrates how the final probability is determined. Note that if the user decides to change every type identically, the probabilities of each type remain unchanged.

Program 4.3 Pseudocode for Hierarchy Depth Dependent Leaf Module Type Probability

```
/*given the level of the module*/
for i = 1 to number_of_leaf_types {
   depth_dep_prob[i] = user_def_weight[i] * original_prob[i]
}
scale_to_sum_1(depth_dep_prob[i])
```

In addition to adjusting the probability of the module types for the top or bottom halves of the circuit tree structure, the user can further adjust each module type's probability for the middle section and the end sections of a dataflow via a user parameter. This user parameter also controls the probability of the leaf module types for the head sub-module and for the tail sub-modules in a star network. The head sub-module will have the same probability distribution as the sub-modules in the middle of the dataflow network and the tail sub-modules will have the same probability as the sub-modules at the ends of the dataflow network. It was observed that the sub-modules at the ends of a dataflow network and the tail sub-modules in star networks were more likely to be controllers or interfaces as explained in Section 3.3.1.

Calculation of the probability distribution for the location of the sub-module within star and dataflow networks is done in a similar manner to the calculation shown above for the hierarchy depth of the sub-modules. The additional weight factor resulting from the sub-module location is multiplied with the depth dependent probabilities described above and the resulting values are scaled to 1.

Once the probability for the different leaf module types has been established, the program loops through all of the leaf modules and stochastically selects a type using a

probability distribution based on the position of the leaf module in the circuit hierarchy and its position within the network if applicable.

The program then selects the content for each leaf module from the circuits found in the library. As noted before, the circuit library is separated into four directories, one for each module type. For each leaf module, the algorithm randomly chooses a circuit from the matching library directory. The selection of a circuit within the directory is done with equal probability. The researcher can manipulate the probability of certain circuits being chosen for each type by changing the mix of circuits found within the directory.

The reset and interrupt modules are selected with uniform probability among the leaf modules of the circuit. The bus master for a bus network is the node that connects to the parent network; however, if the top-level network is a bus, a bus master needs to be selected. In this situation, one of the sub-modules connected by the top-level network is chosen at random to be a bus master.

At the end of this stage, the circuit structure has been fully specified.

4.4 Circuit Construction

During construction, the circuit structure's modules are glued together according to the characteristics of the network types described in the previous section. Section 4.4.1 describes how the pins are glued into the appropriate network connection pattern, and then Section 4.4.2 shows how the bit-level nets for the reset, interrupt and clock are connected. Figure 4.2 illustrates the terminology used in this section.

The primary challenge when constructing the circuit is gluing output pins of one sub-module to input pins of another sub-module in a realistic way. Since the contents of the modules are treated as a black box, we have no information about the meaning of individual pins. Thus, during construction, we treat all pins equivalently. The only exception is the clock pin which can be identified by searching for flipflops in the circuit

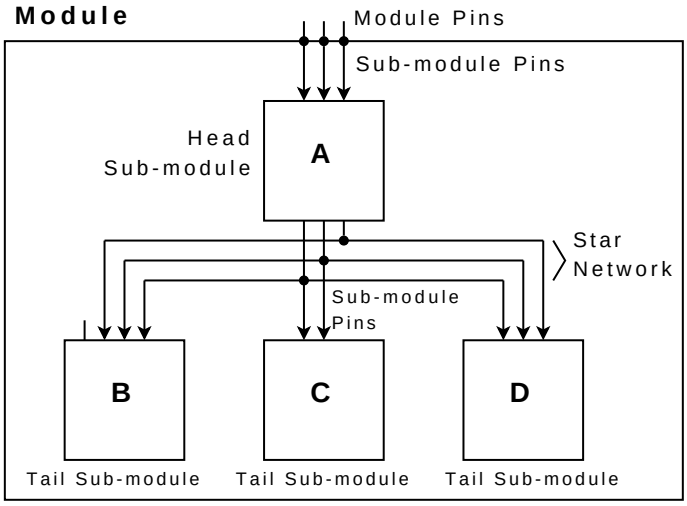


Figure 4.2: Example Sub-Module Connection

and then tracing the clock signal to a module pin. The advantages of this approach are that it simplifies the model and reduces the amount of information that must be supplied by the user. The disadvantage is a reduction in the "realism" of the generated circuits. Typically, for a leaf module, some pins will represent data-type connections and some will represent control-type connections. Since we do not have information about which pins are which, it is possible that we connect data pins to control pins or vice versa. However, we suspect that the inaccuracies in connecting pins in this way will not significantly affect the architectural conclusions that could be drawn using our circuits.

Since the library circuit files for the leaf modules are chosen randomly, the number of pins on the sub-modules may not be appropriate for the connection that is being built. For example, if a module is to consist of a number of sub-modules connected using a 16-bit bus, it is unlikely that all of the selected sub-modules will have exactly 16 bits. As another example, if we are connecting a star network, it is unlikely that the sub-module selected as the "head" will contain exactly the same number of pins as the total pin count for all the tail sub-modules. This problem is illustrated in Figure 4.2. The sub-modules need to be connected by a star network, but the head sub-module only has 3 outputs,

while, in total, the tail sub-modules have 9 outputs. A major challenge in this section was connecting sub-modules with very different numbers of pins.

4.4.1 Network Construction

The heart of the circuit construction algorithm is the manner in which the sub-modules are glued together. The circuit is constructed recursively from the bottom of the hierarchy from the circuit structure description using the algorithm in Program 4.4. The challenge is to connect modules with mismatched pin counts in a realistic yet robust manner. Each network type is constructed differently and is described separately below.

Bus

The bus network is inspired by the AMBA AHB/APB single master specification [6] which was the most common type of bus represented in the survey from Section 3.3. However, some changes were made to simplify the implementation.

A customised bus interface is generated for each sub-module. The bus interface and the sub-module are glued together to form a slave or master node. These nodes are, in turn, connected to form the bus structure. The slaves drive their outputs to a multiplexer which feeds the bus master. The multiplexer selects the output source using the upper bits of the address bus. The bus master, in turn, drives the inputs to all of the slaves, but instead of using a separate arbiter module to activate the slaves as in the AMBA specification, each slave interface performs its own comparison. This is similar to the structure used in the Avalon [4] bus topology which is developed for FPGAs. Flags are present to coordinate the transmission of data between the sub-module and the interface. For simplicity, we assume that the width of the data bus and the address bus are equal to the bus width primary parameter. Appendix B describes the pins found on the interface.

In order to accommodate sub-modules with pin counts that do not match the bus

Program 4.4 Pseudocode for Circuit Construction

```
Generate_circuit {
   for each module {    /* Recurse */
        if this module is a leaf
            glue selected subcircuit from library
        else
            generate_circuit(module)
   }
   /* Generate connection between sub-modules */
   case (network type) {
      BUS:
         randomly select one sub-module to be master
         for each sub-module in the module {
              generate the bus interface and connect it to the sub-module
         }
         wire address and data buses to all nodes
      DATAFLOW:
         for each sub-module in the module
            Connect the outputs to the inputs in the adjacent stage
         for each sub-module in the module
            Connect empty outputs to empty inputs from any later stage
         for each sub-module in the module
            Connect empty outputs to empty inputs from any earlier stage
      STAR:
         if the number of sources from the head sub-module exceeds the
         total number of sinks from the tail sub-modules
            distribute the pins among the tail sub-modules
         else
            wire each pin to every tail sub-module while they have empty
            sinks
         if the number of sinks in the head sub-module exceeds the total
         number of sources from the tail sub-modules
            connect the tail sub-modules to the head sub-module
         else
            connect an equal number of inputs from every tail sub-module
            to the head sub-module, while the tail sub-module has
            remaining inputs
    }
    connect any empty sources or sinks to module pins
}
connect the interrupt nets
connect the reset net
connect the clock net
```

width, two strategies were employed. If there are too few output pins on a sub-module, the data is replicated onto the other bus interface pins as described in [6]. If there are too many output pins on the sub-modules, the excess pins are assumed to be module pins. The cases for the input pins are the same as in [6]. In the rare occurrence that all of the sub-modules happen to have a pin count that exactly matches the bus width, one randomly selected sub-module duplicates its outputs to the module pins, and some of the inputs of another randomly selected sub-module are disconnected and diverted to the module pins.

Dataflow

The dataflow connection pattern not only allows connections between adjacent sub-modules, but also allows connections to skip stages and to feed back to previous stages. The input pins of the sub-modules in the first stage of the network, and the output pins of the sub-modules in the last stage of the network are always connected to the module pins to ensure that the module always has inputs and outputs. This algorithm is summarised in Program 4.4.

Figure 4.3 shows an example of a connected dataflow network. First as many connections as possible are made between adjacent stages. If there are many sub-modules in adjacent stages, the algorithm connects the outputs from one source sub-module to the inputs on a sink sub-module until either the outputs of the source are exhausted, or the inputs on the sink are exhausted. If there are still free outputs on the source, the outputs are connected to the pins on the the next sink sub-module. If the sink has no more free inputs, the outputs of the source are then connected to the inputs on the next sink sub-module.

This algorithm keeps the outputs from a source sub-module grouped together. A dataflow is expected to have mostly parallel word-level communication between the mod-

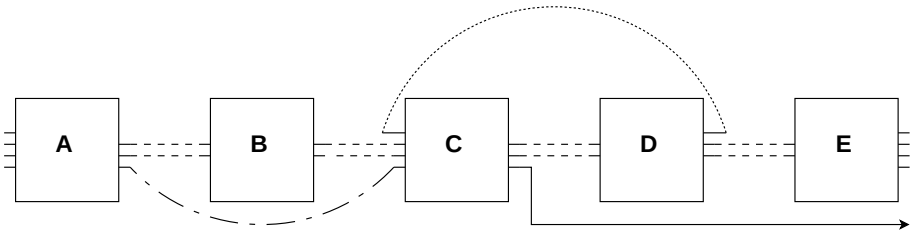


Figure 4.3: Example Dataflow Network Construction: Dashed lines are made between adjacent stages, Dash-dot lines cross multiple stages. Dotted lines represent feedback loops; the intermediary flipflop is not shown. The arrow from sub-module C shows a connection to the module outputs

ules. This decision helps improve the locality of the circuit connections. This algorithm may result, however, in some sub-modules being unconnected to sub-modules in the previous stage; therefore, these sub-modules would not actually belong to their assigned stage. This should occur rarely though, since the number and pin count of the sub-modules on each stage is expected to be distributed uniformly given a library where the circuits' average number of inputs and outputs are equal.

Connections that skip stages such as the dash-dot line from sub-module A to C in Figure 4.3 are then added. Any remaining outputs from a stage connect to the inputs in the closest subsequent stage. Next, feedback loops such as from sub-module D to C are added. These loops contain intermediary flipflops to prevent combinational loops. The feedback connections mimic communication flags between sub-modules. Lastly, any unconnected pins are connected to the module pins. These steps help minimise the number of unmatched pins that are driven to the module pins.

After these three steps, a sub-module still may not be connected to the rest of the dataflow network but this dataflow pattern with several parallel streams was seen in the survey. For example, three colour processing pipelines may operate in parallel. These examples cannot be broken into multiple designs because they are child networks; the parent network connects them into one circuit.

This dataflow connection pattern assumes simple unidirectional data movement be-

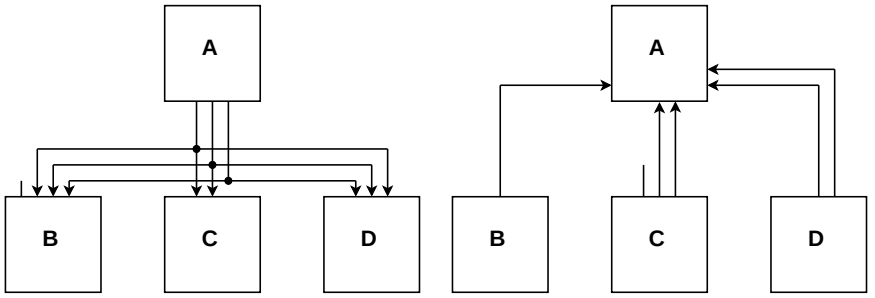
tween modules. More complex dataflow patterns might have stages with bidirectional data movement. These connections could be mimicked in future versions of this tool by changing the order in which connections are generated, for example, by generating feedback loops before connections that skip stages.

Star

The star network connects the outputs of the head sub-module to the inputs of the tail sub-modules, and the outputs of the tail sub-modules to the inputs of the head sub-module. The remaining pins become module pins. Mismatches can easily arise between the pin count of the head sub-module and the pin count for the sum of all the tail sub-modules. The algorithm used to resolve these issues is described in Program 4.4, and an example is shown in Figure 4.4.

If there are too many outputs on the head sub-module, the outputs are driven to the module pins. If there are too few, the output pins are assumed to represent a word. This word is then connected to each tail sub-module, driving as many of the inputs as possible. This situation is shown in Figure 4.4(a). Sub-module A has only three outputs, while the sum of the inputs on the tail sub-modules is nine. The two input pins on sub-module C are driven by the first two pins of A. The first three pins of sub-module B are driven by A, and the last pin will be driven by a module pin (not shown).

Similarly, if there are too many inputs on the head sub-module, they are connected to the module pins, and if there are too few, the excess sub-module pins are diverted to module pins. We ensure that all of the tail sub-modules are connected to some of the inputs on the head sub-module. Figure 4.4(b) demonstrates how these pins are connected if there are too few inputs on the head sub-module. Sub-module A in has only 5 inputs, but the total number of tail sub-module output pins is 6. One pin from each tail sub-module is connected in turn while the head sub-module still has free input pins.



(a) Insufficient outputs from the head sub-module

(b) Insufficient inputs on the head sub-module

Figure 4.4: Example Star Network Construction

Sub-module B's only input pin is connected to sub-module A and sub-module C has one extra pin left. This pin will be driven by one of the module pins.

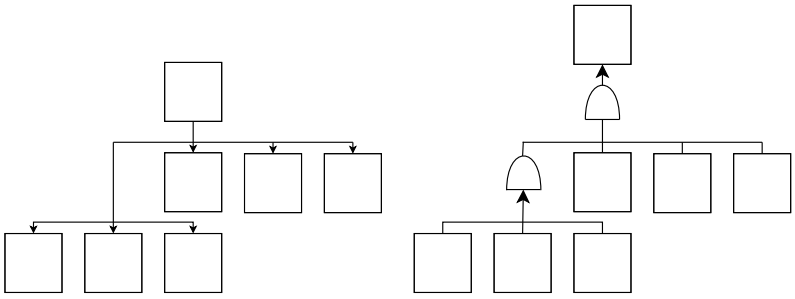
We expect the communication in a star network to be word-style, but it is necessary to distribute the pins from the head sub-module equally to all the tail sub-modules in order to ensure that all the tails are connected to the head. Although unconnected modules were allowed in the dataflow network, we expected that adjacent stages would usually have compatible pin counts, and would rarely result in unconnected modules. In a star network on the other hand, we expect that the total number of pins on the tail sub-modules will usually exceed the number of pins on the head sub-module due to the large number of tail sub-modules in a star network.

4.4.2 Single Bit Net Construction

Lastly, the reset, interrupt, and clock nets are constructed. The clock connects to all the flipflops in the circuit. The model assumes that there is only a single clock in each circuit which is driven by an external source.

The reset and the interrupt structures can be seen in Figure 4.5. The reset signal is modelled by a single bit signal which is driven by one randomly selected leaf module to all the other modules in the circuit. Similarly, one module in each circuit is assumed

to be the interrupt sink, and all the other modules reserve one pin to be an interrupt flag. The interrupt signals of the modules in each network are combined together with an OR gate so that the resulting interrupt sink is only 1 bit wide. Interrupt signals are not added to the bottom hierarchy level of the circuit.



(a) Reset (b) Interrupt

Figure 4.5: Reset and Interrupt Structures

These structures are a simplified representation of actual reset and interrupt signals. All but the most simple circuits implement these signals using intermediary interrupt and reset controllers. These controllers store some extra information in buffers to help process the signal and to help distribute the signal. Although the implementation of these signals is simplified, they still retain some important characteristics of real resets and interrupts, such as significant fanout which adds extra complexity to the circuit.

4.5 Generation Mechanics

This generator was written in C. It contains 22.5k lines of code, and it can be run on either Linux or Windows. Library circuits must be BLIF files, but the output can be in either VHDL or BLIF output. Generating a circuit takes 70 seconds for a circuit with 25 modules and 41k logic elements. Most of the generation time is used to read in and verify the circuits for the leaf modules. Optimally, generation could be done in linear time, but in practice, the complexity is higher as the circuit needs to iterate

over the data in order to ensure validity of the circuit during the construction process. Circuits up to 150000 LEs in size have been built. This program can be downloaded from `http://www.ece.ubc.ca/~stevew/circuit/circuit_agreement.html`.

4.6 Summary

In this chapter, the algorithms used to generate synthetic circuits were described. The generation is separated into three main stages: definition of the primary parameters, generation of a detailed circuit structure, and construction of the circuit. The first stage sets values for any primary parameters left empty by the user. The second stage assigns the elements to the circuit structure tree. The last stage constructs the bit-level nets that form the networks.

With the exception of the primary parameters, the rest of the generation can only be controlled indirectly via constraints and probability distribution parameters found in the constraints file. The probability distributions used in this process are modelled against the data presented in the previous chapter.

The construction of the circuit is straightforward once the circuit structure has been defined, but the mismatched pin counts of the sub-modules can lead to excess numbers of module pins. This chapter presented strategies to reduce the number of module pins for each network type. These strategies attempted to mimic real circuit structure as much as possible while still ensuring robustness. In the next chapter, the resulting synthetic circuits are validated.

Chapter 5

Validation and Characterisation

In this chapter, the SoC-style synthetic circuits generated by this work are validated and characterised. In Section 5.2, the properties of the circuits generated by this work are compared against the properties of circuits built by previous homogeneous synthetic circuit generators, namely GEN and Gnl, in order to identify differences in the manner in which the circuits scale. Then, the results are compared to the post-routing properties of eASIC benchmark circuits in Section 5.3. This section validates the behaviour of these generators against the properties of real circuits and demonstrates that the synthetic circuits described by this work lead to realistic architectural conclusions.

In Section 5.4, the properties of the added network structure are investigated. The effectiveness of the strategies used to resolve mismatched sub-module I/O pins is evaluated and the post-routing effects of the different network types are explored. Lastly, we discuss measurement of Rent parameter for heterogeneous circuits.

5.1 Overview of Experimentation Methodology

The circuit library for these experiments contained 61 circuits from the MCNC benchmark suite. These circuits contained between 28 and 7694 logic elements. The circuits were separated into the four different module categories according to their circuit function as described in [65]. They were supplemented with two circuits from OpenCores [39, 21], in order to add more diversity to the processor and the interface categories.

The primary parameters were left unspecified unless otherwise noted. Since the

MCNC circuits are small, some of the values in the constraints file were modified in order to make it more likely to produce larger circuits containing more modules. The bus width was allowed to vary from 8 bits to 64 bits in order to match the size of the MCNC circuits.

Synthetic circuits can be validated using either *direct* methods or *indirect* methods [60]. Direct methods compare the graph properties of synthetic circuits versus "real" examples. Some examples of these properties are the size, the fanin or fanout of the LUTs, the amount of redundancy in the logic, etc. This is the primary validation method for most of the previous efforts in synthetic circuit generation. Nevertheless, this type of analysis may be less interesting for these new circuits because the majority of each circuit is composed of the leaf module logic; very little new logic is added. Though this method does ensure that the synthetic circuits look like real circuits, it is not clear whether the set of validated properties are sufficient to ensure that the synthetic circuits are comparable to real circuits for architectural research.

A more interesting mode of validation for FPGA experimentation is the indirect method. This method compares results from processing the circuits using various tools. If two circuits are comparable, then a tool should return similar results for both circuits. Since these new circuits are intended to be used as benchmarks for developing future architectures, it is critical to ensure that these circuits return FPGA characteristics similar to those returned by real circuits. For this research, the validation emphasis is on the post-routing characteristics of our circuits after synthesis, using the T-VPACK/VPR synthesis suite.

This research compares the Rent parameter, the number of post-clustering nets, the average post-routing net length, and the minimum channel width with which the circuit can be routed. Though the circuits are not functional, the critical path was measured as well in order to gain an understanding of the overall qualities of the circuit's netlist.

5.2 Comparison to Previous Circuit Generators

In this section, the synthetic circuit generator is compared against previous work. The circuits generated using our program are compared against those generated using two homogeneous circuit generators, GEN and Gnl and differences are identified in the structural properties and the post-routing characteristics.

5.2.1 Experimental Methodology

Twenty-one circuits of varying sizes were synthesised using our circuit generator. Table 5.1 lists statistics regarding these circuits. For each of these synthetic circuits, circuits were generated using GEN [25] and Gnl [61] where the number of logic elements for each GEN and Gnl circuit was constrained to be the same as the corresponding synthetic circuit built by our tool. The size of the circuits in this suite ranges from 5687 to 56296 logic elements which well exceeds the validated range of both GEN and Gnl [26, 61].

Additional parameters for the two previous generators were specified. Based on experience, Gnl was directed to generate circuits with a Rent parameter of 0.7 for the first 3000 nodes, and 0.66 for the rest of the circuit. In addition, a ratio of 2:1:1 for 2, 3, and 4 terminal logic elements respectively was specified. This ratio differs from the default Gnl values which defines a large proportion of the logic elements to be single input elements, which are not necessary in an FPGA setting. A single input logic element is either an inverter or a buffer. Since a LE represents a k input logic function, the logic function can easily be modified to include the inverted input value thus rendering the additional LE unnecessary. The number of flipflops in the Gnl circuits was constrained to be the same as in the circuits built by our tool. This version of Gnl also inserts additional flipflops when necessary to control the logic depth [61]. The number of output and input pins was constrained so that the final circuit was not pin-limited. The default input to output pin ratio was used.

The GEN circuits were strictly combinational; the extensions that allowed sequential circuits were found to be unstable as the circuits scaled. Only the size was specified, so all the other parameters were generated using their default distributions.

Each circuit was then mapped to a minimum-sized FPGA with minimum channel width using T-VPACK and VPR 5.0 [34]. The clustering, placement, and routing were timing-driven. A clustered architecture was used in which each cluster contains four 4-LUTs and has 10 inputs and 4 outputs. Each I/O block was assumed to have 3 pins. A uni-directional routing architecture with single-length segments was assumed.

5.2.2 Experimental Results

Rent Parameter

The Rent parameter is the exponent factor in Rent's rule. Rent's rule is an experimentally observed relationship between the size of subcircuits returned after partitioning and their I/O pins. This parameter is an indicator of the interconnect complexity of a circuit. Experience has shown that Rent parameters of 0.6 to 0.7 are reasonable. Using this relationship between the number of elements and the number of pins, researchers have developed equations that can predict various post-synthesis circuit properties such as wirelength, channel width, etc. Since this value is linked to so many properties, it is important to model it well.

For all circuits, the Rent parameters shown in Table 5.1 were computed using a recursive Fiduccia-Mattheyses partitioning algorithm. The Rent parameter for each GEN and Gnl circuit is shown in Columns 4 and 5 of Table 5.1. Though a Rent parameter of 0.66 was specified for the Gnl circuits, the average Rent parameter in the resulting circuits is 0.82. The effect of the I/O pin constraints on the Rent value was investigated and it was found that the two variables were unrelated. The deviation from the specified Rent parameter is likely caused by the ratio of multiterminal logic elements in the Gnl

Table 5.1: Comparison between Generators

No	Number of LEs	Rent Parameter			Number of Nets			Avg. Net Length			Channel Width			Crit. Path (ns)		
		New	GEN	Gnl	New	GEN	Gnl	New	GEN	Gnl	New	GEN	Gnl	New	GEN	Gnl
1	5687	0.71	0.81	0.80	3893	4128	5901	13	30	18	28	90	46	16	25	44
2	6436	0.69	*0.80	0.79	4399	*4885	6725	16	*28	18	40	*74	52	21	*28	46
3	7425	*0.71	*0.82	0.80	*6077	*6257	7855	*19	*27	19	*32	*110	54	*41	*24	41
4	8001	*0.71	0.80	0.80	*6171	5843	8426	*14	30	19	*36	82	56	*40	38	49
5	8468	0.71	0.80	0.81	6673	6553	9009	13	26	19	38	80	54	40	30	53
6	8967	0.72	0.84	0.81	6072	7678	9465	15	39	20	36	124	60	19	38	48
7	10937	0.71	0.67	0.82	7116	5660	11536	18	24	21	40	44	64	32	28	61
8	11643	0.73	0.82	0.82	8509	8917	12732	15	47	24	38	90	70	39	57	60
9	13012	0.71	0.81	0.82	10619	9003	14004	21	39	22	36	84	64	42	38	60
10	13790	0.73	0.83	0.82	10185	10621	14353	15	39	24	38	122	70	38	38	65
11	14099	0.74	*0.82	0.82	9893	*10430	15078	15	*46	24	38	*86	74	39	*49	66
12	14538	0.73	0.82	0.82	10306	12994	15512	14	38	23	38	126	70	40	54	65
13	15778	0.75	0.82	0.82	11474	11900	16176	18	51	24	50	106	74	37	44	68
14	18550	0.75	0.81	0.83	13127	14622	19507	14	34	25	38	128	76	39	49	72
15	19915	*0.74	0.81	0.83	*14877	13216	21196	*17	39	27	*40	86	80	*39	74	83
16	27577	*0.76	0.75	0.84	*20311	16701	28686	*19	36	29	*42	72	90	*40	81	101
17	37466	*0.78	0.65	0.85	*27902	17021	38452	*23	-	-	*44	-	-	*47	-	-
18	42261	*0.77	0.81	0.85	*30496	26900	51220	*18	-	-	*52	-	-	*46	-	-
19	46255	*0.77	0.81	0.84	*31913	26219	43898	*18	-	-	*42	-	-	*56	-	-
20	50275	0.77	0.78	0.85	37494	27600	49034	28	-	-	72	-	-	55	-	-
21	56296	*0.78	0.81	0.82	*41366	33108	57989	*20	-	-	*62	-	-	*43	-	-
Avg first 16		0.73	0.80	0.82	9356	9338	13510	16	36	23	38	94	66	35	43	61

Starred entries are pin-limited. Empty entries did not complete after 72 hours of place and route.

circuits. Since the tool was calibrated with a large number of single-input logic elements, the removal of these elements forces the output of each module to connect to more inputs, thus increasing the Rent parameter.

Rent parameters in the GEN circuits tend to be higher than average circuits, most likely because GEN was designed and calibrated assuming smaller benchmark circuits than the ones generated here. The Rent parameters measured in the new circuits built by our tool ranged from 0.69 to 0.75 (Column 3). More discussion regarding Rent parameter calculations for heterogeneous circuits can be found in Section 5.4.2.

Logic Locality

The observed differences between the post-routing characteristics of the different circuit generators are strongly related to the differences in the locality of the logic. The less locality, the more interconnected the logic becomes. The circuits from the generator in this work display stronger locality than circuits built by the previous generators. Figure 5.1 shows a sample synthetic circuit made by this generator after clustering and placement while Figures 5.2 and 5.3 show sample circuits from GEN and Gnl respectively. The nets connecting the clusters are shown. The sample circuit generated by our tool contains two copies of the MCNC benchmark cpu8080, and three smaller interface modules all connected by a bus. It is easy to observe that the majority of the nets are localised, leading to distinct groupings of nets. Most of the nets connect only within a module; only a few nets belong to the network that communicates between modules. Appendix D shows the clusters used by the logic of each module and its bus interface.

In contrast, the nets for equivalently sized circuits from GEN and Gnl seen in Figures 5.3 and 5.2 respectively are distributed much more homogeneously; thus as will be explained in subsequent sections, the net lengths are longer, the channel width is higher, and the critical path is longer.

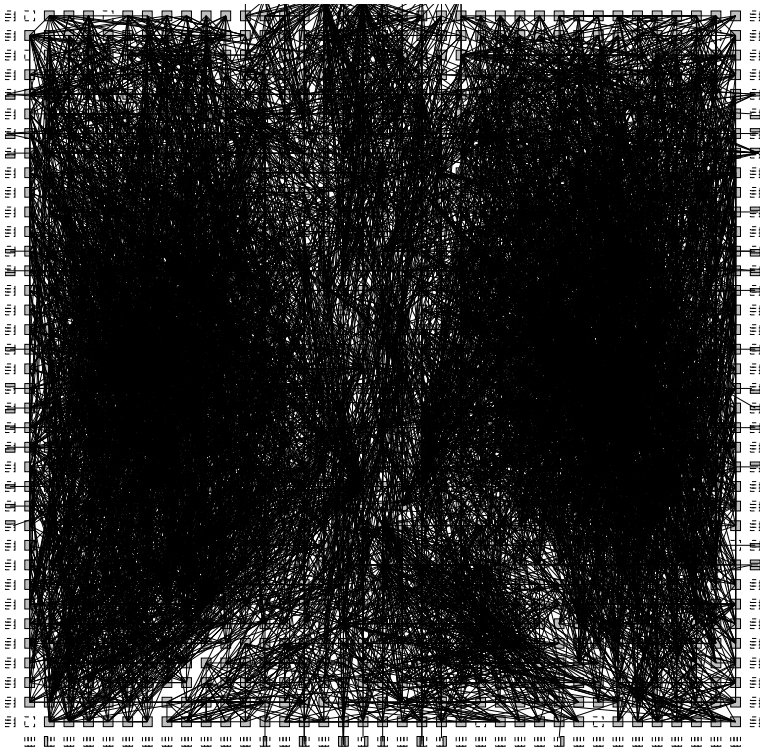


Figure 5.1: Sample Synthetic Circuit from our Tool after Placement

Number of Nets

Columns 6 to 8 of Table 5.1 show the number of nets in the clustered circuits. This is the number of connections that need to be routed in the channels which is a major factor in the required amount of routing resources. This number is also related to how well the clustering tool can share inputs between LEs in a cluster, that is, how many nets can be absorbed into clusters during clustering.

As seen in the table, the number of nets in the Gnl circuits is on average 45% higher than in the other circuits. One reason could be that there are too many multiterminal nets which are more challenging to completely absorb within one cluster. A high number of multiterminal nets could be linked to the change in the composition of multiterminal logic elements used in the circuits. As explained before, the single input terminal elements were removed; thus increasing the ratio of multi-input logic elements. Only 20% of the MCNC circuits contain such single input LEs, yet, as seen in Table 5.1, the circuits built

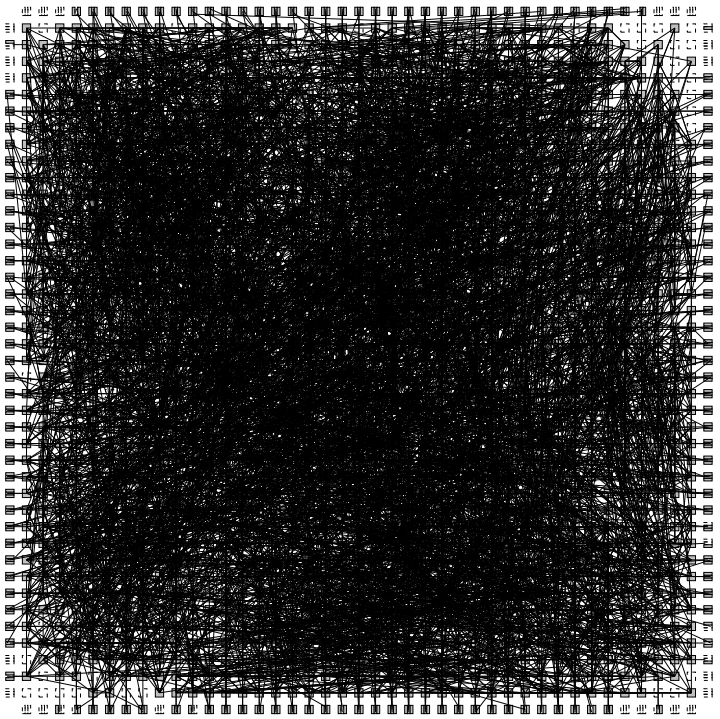


Figure 5.2: Sample Gnl Circuit after Placement

by our generator using the MCNC circuits do not have such high numbers of nets. This suggests that the Gnl nets do not display the locality that is seen in real circuits and are therefore too random to be effectively absorbed into clusters.

The number of nets in the GEN circuits and the circuits built by this work is approximately the same. The locality parameter in GEN is sufficient to generate enough locality for effective clustering.

Average Net Length

Columns 9 to 11 of Table 5.1 show the average post-routing net length of each implementation. The average net length is measured in terms of the number of clusters that the net spans. This characteristic affects how many routing resources are used by a net. Also, since the intercluster routing is slower than intracluster routing, long nets will add significantly to the signal delay.

As the table shows, the average net length is significantly higher in both the Gnl

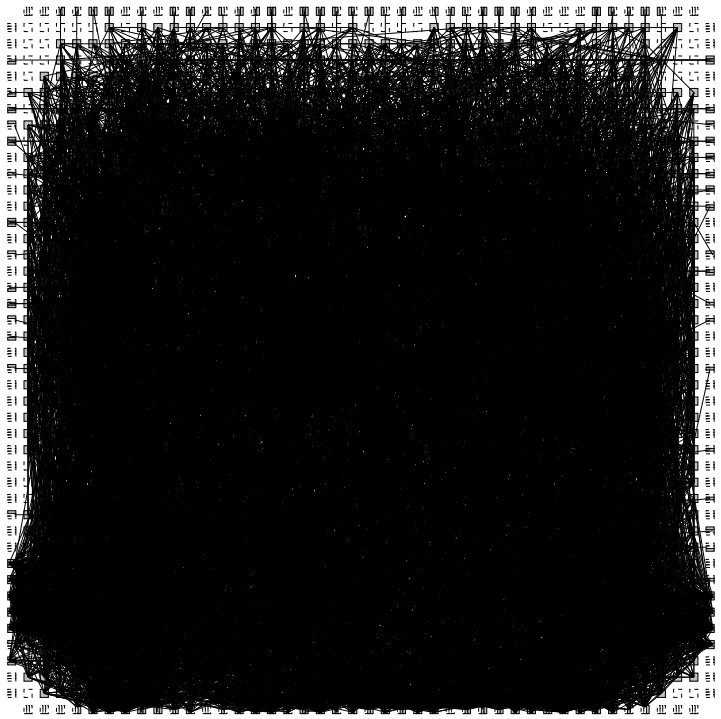


Figure 5.3: Sample GEN Circuit after Placement

and the GEN circuits than in our new circuits. Experience has shown that average net lengths for multi-terminal nets in the range of 11 to 15 are reasonable [14]. The longer net length in the Gnl circuits is likely due to the increased number of nets. This leads to more congestion in the circuit and in turn, longer nets when a minimum channel width is used.

The GEN circuits show more than twice the average net length seen in our circuits. One factor is the higher Rent parameter which leads to a longer net length as explained in [11], but as seen in Table 5.1 the results vary much more than would be suggested by the Rent parameter alone. The Gnl circuits have approximately the same Rent parameter yet when compared to the steady growth in the average net length for Gnl's circuits, the results for the GEN circuits vary quite dramatically.

One reason for this difference could be the manner in which Gnl builds the circuits. It builds circuits by connecting pairs of subcircuits recursively. This allows the connections to be more localised at higher structural levels. GEN, on the other hand, considers

the whole circuit at once, and progressively adds more detail to the structure. While a locality parameter is used to control the generation of nets, this appears to be insufficient to ensure locality at the higher levels of the structure. We suspect that this was addressed in later extensions, but they were not used in this experiment.

Minimum Channel Width

The minimum channel width is a measure of the routing resources required to implement a circuit. This is an important metric as it directly impacts the area of the FPGA. To model the FPGA area accurately, the benchmark circuits should lead to channel width requirements very similar to those of real circuits.

Columns 12 to 14 of Table 5.1 show the minimum channel width required to route each circuit. As expected, the larger number of nets and the higher average net length for the Gnl and GEN circuits lead to more demand for the routing resources in the circuit, thus resulting in a higher channel width. An equation describing this relationship is presented in [19]. Gnl's average channel width is 1.7x higher than the circuits from this work, and the GEN circuits are 2.5x our channel width. The variations in the GEN circuits' average net length are reflected in the increased channel width. The significant increase in the channel width for the GEN circuits is likely exacerbated by the lack of locality in the nets; this results in more congestion.

Critical Path Delay

Finally, Columns 15 to 18 of Table 5.1 show the critical path delay of each mapped circuit. The critical path delay determines the maximum speed at which the circuit can run. Though these circuits are not functional, the critical path of the benchmark circuits should still give an idea of the relative performance of different architectures. The differences in the critical path between the generated circuits are dominated by the

major differences in average net length and the number of nets.

Since the GEN circuits are purely combinational, a long critical path is expected. For such large circuits, there would normally be some flipflops that would break up the critical path. The Gnl circuits also have long critical paths; in fact, the delay exceeds that of GEN. As described earlier, however, these circuits have a restricted logic depth. This makes the length of the critical path rather unexpected. The cause is related to the results for the previous post-routing parameters. A larger proportion of nets are routed using slower inter-cluster nets, and the average net length of these circuits is higher than average. These two factors negate any advantage that was gained by the constrained logic depth.

5.2.3 Summary

This section investigated the number of nets, the average net length, and the channel width of the circuits built by our generator, and by Gen and Gnl. This allows us to evaluate the resource demands of these circuits. The circuits' critical path reflects the variations in these different parameters. The results demonstrate that the connections in our circuits result in simpler circuits that are easier to cluster, place, and route and require less FPGA resources to implement. Moreover, the place and route algorithms took several times less computing time to complete for our circuits. These properties enable the circuits built by this work to scale to larger sizes more readily as will be demonstrated in the next section.

5.3 Validation against eASIC circuits

This section will compare the results gathered in the previous section against the post-routing results of a real benchmark suite. This will tell us whether the properties of

the synthetic circuits mimic those of real circuits. The analysis evaluates the possibility of using these synthetic benchmark generators for structural FPGA architecture experiments.

5.3.1 Experimental Methodology

eASIC Benchmarks

The eASIC benchmark suite is composed of 5 ASIC netlists [17]. These circuits are from industrial sources, and were originally released to help test placement algorithms. An eASIC circuit is composed of flipflops, memory blocks, and eCells. Each eCell represents a small cluster of logic elements. The logic function implemented by each eCell is not described to protect the IP; therefore, they cannot be used for FPGA power experiments. We suspect that other modifications to the netlist were also made to further address IP concerns.

Each eCell has a maximum of 7 inputs and 2 outputs. In order to make the circuits suitable for FPGA synthesis, several alterations to the netlist were necessary. Table 5.2 shows the composition of each circuit before and after conversion. The modifications needed to convert these circuits into a BLIF format with 4-input logic elements are listed here.

Table 5.2: eASIC Benchmark Suite

	Original				Converted 4-input BLIF		
	I/O	eCells	Flipflop	Memory	I/O	LUTs	Flipflops
1	446	832824	87052	282	462	946946	87052
2	102	812200	45478	861	111	875730	45478
3	421	961063	52780	192	422	970204	52780
4	414	102038	23300	44	415	125637	23300
5	154	913853	84505	407	175	1008121	84505

Multiple outputs per eCell Each output becomes its own LUT. The inputs of each

LUT are still connected to all of the inputs of the original eCell.

No logic function Each LUT is assigned the AND logic function.

Multi-input flipflops These additional pins were likely set/reset logic signals. The flipflop is replaced as follows. The input signals to the eCell are now fed to a LUT which in turn drives the flipflop.

Block RAMs The memory block is removed and replaced with logic. Figure 5.4 demonstrates a memory block conversion if there are more inputs than outputs. Each output is replaced by a LUT. Each LUT received an equal portion of the inputs to the original memory block. If one of the LUTs exceeded seven inputs, it was further decomposed as follows. The inputs were redirected to an array of LUTs each with up to 7 inputs. This array of LUTS then drives a LUT which in turn drives the original memory output. The tree never exceeded a depth of two elements. This process is illustrated in Figure 5.4.

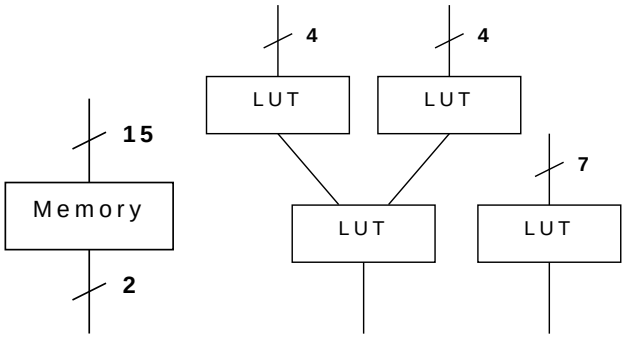
If there are more outputs than inputs, each output is still assigned to a LUT. However, each input signal now fans out to an equal portion of the LUTs.

While this strategy does change the structure of the circuit, the memory blocks form less than 1% of the logic. The highest number of outputs found on a memory block was 32 and the highest number of inputs found was 45, hence the circuit size is increased by less than 2%.

eCells with no outputs These signals are assumed to have one pin. This pin drives a new external output.

Multiple clocks All the clock pins are now driven by one external input pin.

Clocked constant signals The clock was removed from these blocks.



(a) Original memory block (b) Transformed memory block

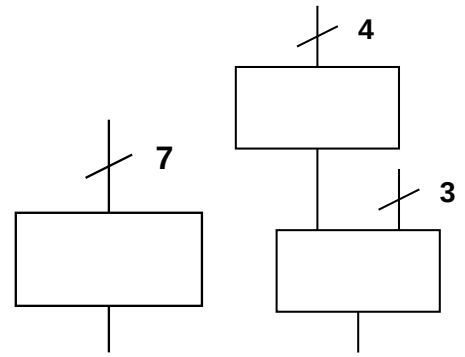
Figure 5.4: eASIC Memory Block Transformation

To convert these files into a format more standard for FPGA research, the files were further modified as follows.

Seven input LEs Each LE with more than seven inputs was replaced with a collection of 4-input LEs. For each such LE, the excess pins were moved to another LEs which then fed the original LE. This process is shown in Figure 5.5. As shown in [2], the area delay product for LEs with 4 inputs is marginally better than for LEs with 7 inputs. More importantly, conversion to 4 input LEs makes it easier to compare the results with the previous experiment.

Combinational logic cycles This problem results from the block RAM conversion. These circuits were placed and routed using non-timing-driven methods. The table of post-routing results in Appendix E identifies these circuits.

Intractably large These circuits would require more than 12GB of RAM if they were synthesised as is [10] using VPR. They were partitioned into smaller circuits to make these circuits more manageable. These circuits would likewise need to be partitioned in order to be implemented even on the largest current FPGA devices [64]. The partitioning of these circuits was performed using kMETIS [29] which parti-



(a) Original 7-input LE (b) Transformed 7-input LE

Figure 5.5: 7-input LE Transformation

tions graphs into k subcircuits. The five eASIC circuits were partitioned into ten different circuits. The size of the subcircuits was allowed to deviate no more than 10% from the size of a strictly equal partitioning. The partition returned was the best of 10 runs where the algorithm tried to minimise the number of nets cut.

These circuits were clustered using T-VPACK and then placed and routed using VPR. The smallest eASIC circuit is already of an appropriate size for VPR synthesis, so the non-partitioned circuit was also synthesised. The FPGA architecture used was identical to the one in Section 5.2, but the number of I/O pins allowed per I/O pad was increased to four instead of three.

Additional Synthetic Circuits

Even after partitioning, the eASIC benchmarks are still much larger than the synthetic circuits used in Section 5.2. Thus, additional synthetic circuits were generated. The results from these additional circuits are described in Table 5.3.

Table 5.3: Additional Larger Synthetic Circuits

No.	LEs	Rent	Nets	Avg. Net Length	Channel Width	Crit. Path (ns)
22	59725	0.79	39015	24	72	32
23	63461	0.79	43519	20	58	80
24	72625	0.80	48133	21	58	52

5.3.2 Experimental Results

This section compares results obtained from the eASIC circuits to those obtained using our generator as well as GEN and Gnl. The raw data collected from the eASIC circuits are presented in Appendix E.

Number of Nets after Clustering

Figure 5.6 compares the number of nets after clustering for each of our circuits. Each point corresponds to one circuit. The x-coordinate of the point indicates the number of logic elements in the circuit, while the y-coordinate indicates the number of nets after clustering for that circuit. The points corresponding to circuits from our generator are connected using a dashed line ("NEW"), while the points corresponding to GEN and Gnl are connected using solid lines and are labelled appropriately. The results from the eASIC circuits are plotted as points and not connected using lines. As described earlier, four of the five eASIC circuits were partitioned, each creating 10 subcircuits. The data for the set of subcircuits from eASIC circuit 1 are plotted as hollow circles, the subcircuits from eASIC circuit 2 are plotted as x's, etc. The fourth eASIC circuit was not partitioned, so it corresponds to one point on the graph.

Extrapolating the GNL, Gen and "NEW" lines suggests that our circuits better match the trends observed in the eASIC circuits. Assuming the number of nets in the Gnl circuits continue to grow as a function of circuit size, we anticipate that if Gnl was used to create circuits that were of the size of the eASIC circuits, the number of nets would be considerably larger than in the eASIC circuits (we were not able to perform this

experiment because of excessive run-times compiling the Gnl circuits).

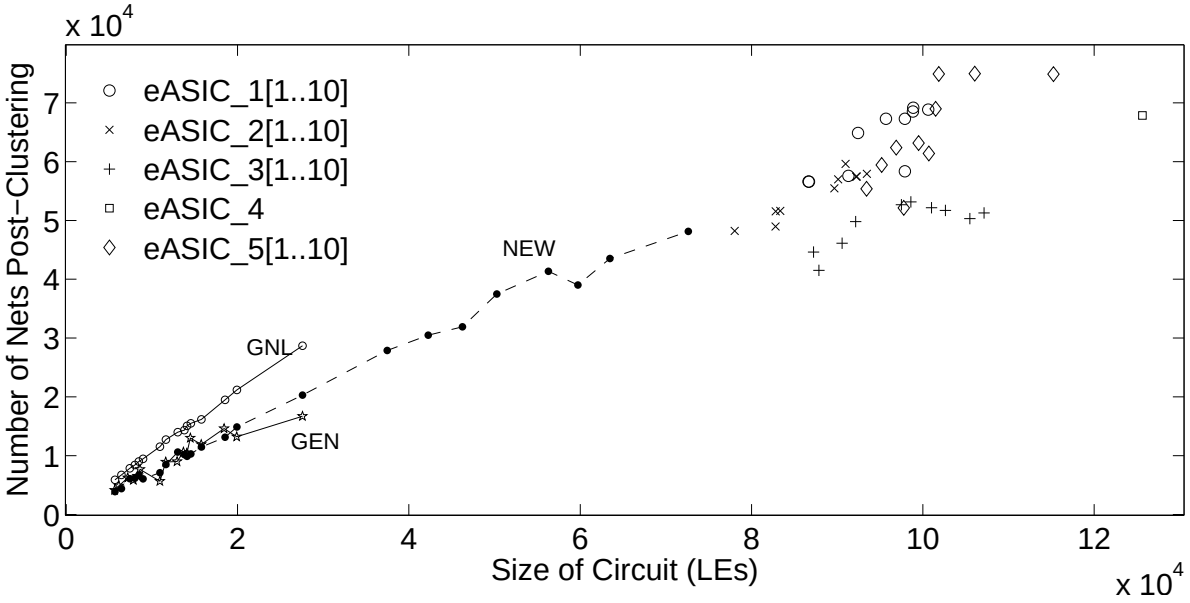


Figure 5.6: Number of Nets Post-Clustering Comparison

Average Net Length

Figure 5.7 compares the average net length after routing for each of our circuits. Again, the horizontal axis represents circuit size and each point in the graph corresponds to one circuit. The points corresponding to our generator, Gnl and GEN are connected using lines and labelled as before. Again, the partitioned eASIC circuits are represented as points.

The results in Figure 5.7 suggest that our circuit generator results in more realistic net lengths as the circuit size becomes large, compared to GEN and Gnl. Compared to Figure 5.6, there is significantly more variation in the measured net length as a function of circuit size. We suspect that this is due to the generation of pin-limited designs, as described in Section 5.4.

Note that in this experiment (and subsequent experiments), we partitioned eASIC circuit 4 and show the smaller subcircuits that result from this partitioning, as well as

the original fourth eASIC circuit. In Figures 5.7 to 5.9, we can see that the relationship between the original eASIC_4 circuit and its subcircuits has approximately the same slope as the trends exhibited by the circuits built by this tool, again suggesting that the results built by our tool are realistic.

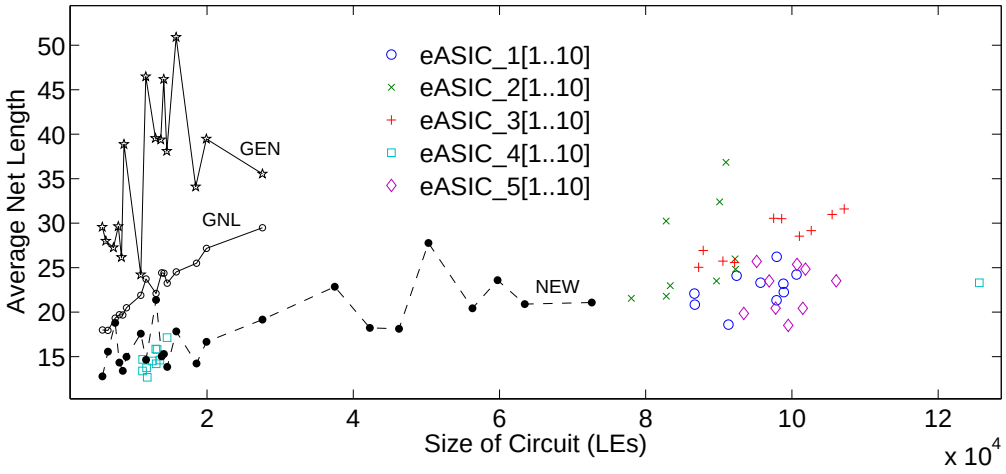


Figure 5.7: Average Net Length Comparison

Minimum Channel Width

Figure 5.8 shows the minimum required channel width required to route the circuits. It is difficult to extrapolate a trend from the results of the GEN circuits; however, the results for these circuits significantly exceed the results of the partitioned eASIC_4 circuits, and already match the results for the larger eASIC subcircuits despite being approximately an order of magnitude smaller in size. The trend shown by the Gnl line suggests that if Gnl was used to create circuits as large as the eASIC circuits, the channel width would become unreasonable (again, we could not perform this experiment because of the excessive place and route times for the Gnl circuits). Extrapolating the results from our generator, however, suggests that the channel width required by our circuits is more realistic.

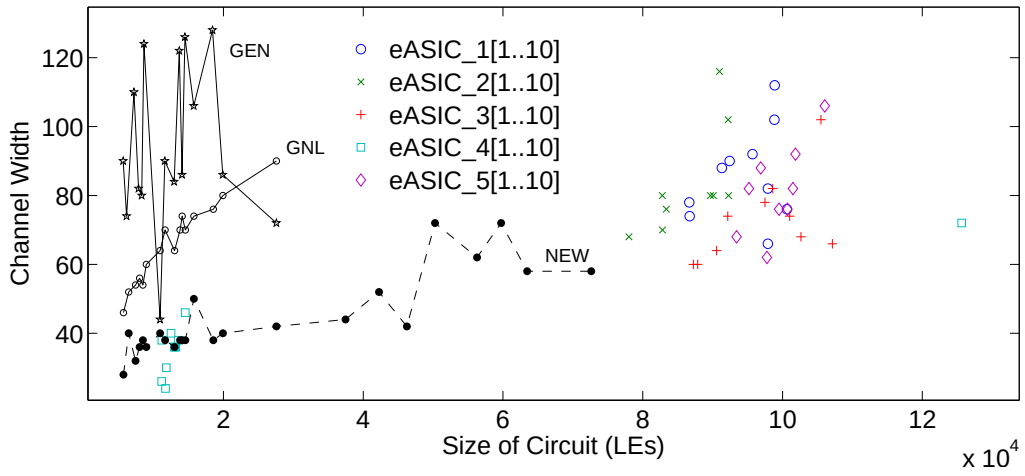


Figure 5.8: Channel Width Comparison

Critical Path

The critical path results are shown in Figure 5.9. In this experiment, there is a lot of spread in the results for the partitioned subcircuits. As explained earlier, the critical path is highly dependent on the quality of the synthesis tools. Nonetheless, the results suggest that our critical paths are reasonable for very large circuits.

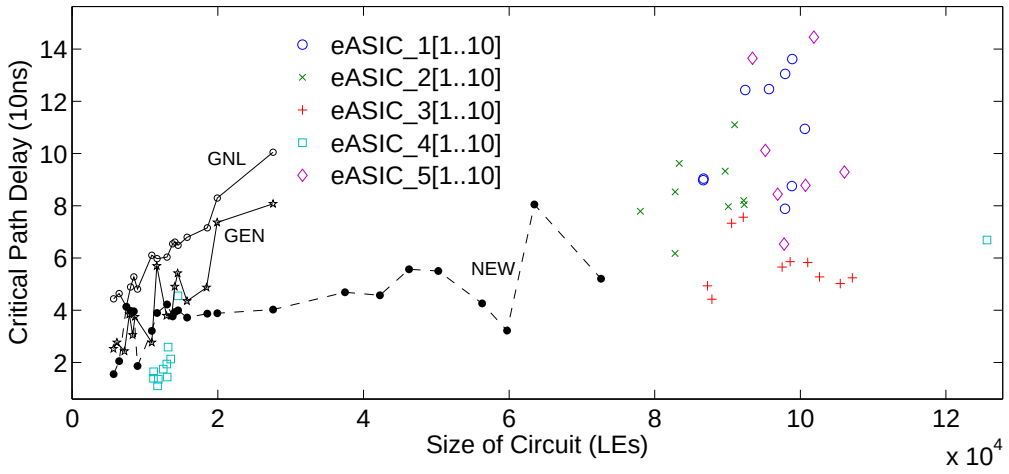


Figure 5.9: Critical Path Comparison

5.3.3 Summary

This section demonstrated that circuits built by this work return similar FPGA characteristics as those from real benchmark circuits. If the trends observed for the circuits from this generator continue for large circuit sizes, the results would be within the range seen for the partitioned eASIC circuits. This trend holds for the number of post-clustering nets, the average net length, the channel width and the critical path. This demonstrates that the circuits from our generator are suitable for FPGA architecture experimentation.

5.4 Characterisation

In this section, the effect of the added network logic on the characteristics of the resulting SoC circuit is examined. The ability of our generator to realistically connect modules with mismatched I/Os is examined in Section 5.4.1 and the effect of the network type is investigated in Section 5.4.2.

5.4.1 Mismatched Pins

In this experiment, the relationship between the circuit size and the number of I/Os was examined in order to investigate the ability of our generator to realistically connect modules with mismatched I/O pin counts.

Experimental Methodology

Ninety-eight circuits of varying sizes were generated using the same library as described in the earlier experiments. To quickly generate circuits with large numbers of modules, the λ values used to stochastically generate values for the hierarchy depth, number of networks and number of modules were modified.

Experimental Results

This new set of circuits and the circuits generated by this tool in Section 5.2 are plotted on a log-log graph as shown in Figure 5.10. The circuit size in LEs is the x-axis and the number of I/O found in the circuit is the y-axis. The best-fit line was calculated using a Matlab package [44].

The relationship between the number of I/O pins and the circuit size was best fitted using a linear equation of slope 0.059. This result suggests that the number of pins found on these circuits is proportional to the number of elements in the circuit. However, the number of pins should be proportional to the square root of the size of the circuit, if the circuits are not to be pin-limited. This signals that circuits will more and more likely be pin-limited as the size of the circuits increases. This trend has already been seen in previous circuit sets. This behaviour indicates that realistic circuits can be built only up to a certain size using our generator.

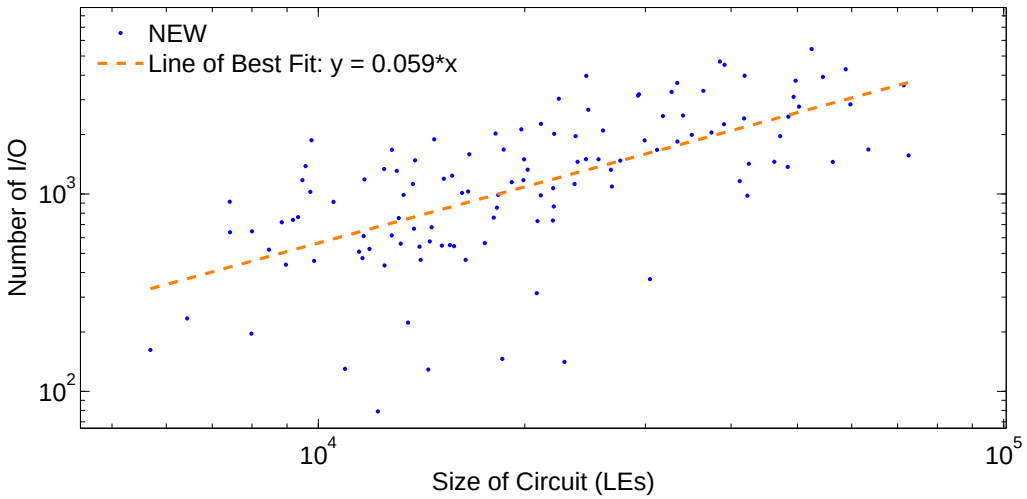


Figure 5.10: Circuit Size versus Circuit I/O

The number of pins found on an SoC-style circuit built by our generator depends on both the degree of the pin count mismatch between the modules and the number of modules in the circuit (i.e. the opportunities for mismatch). This suggests that if

realistic circuits with many modules are to be built, the library of circuits will have to be chosen such that the circuits' I/O pins match well, but if circuits with fewer modules are desired, the contents of the library do not have to be chosen as carefully.

5.4.2 Network Type

Experimental Methodology

To investigate the sensitivity of these statistics to the type of network used in the circuit, 11 sets of benchmarks were created. Each set contains three circuits, all three circuits within a set have the same hierarchy and leaf modules; they differ only in the type of network used (bus, dataflow, or star). Physical synthesis was performed using VPR. The Rent parameter was calculated by partitioning the circuits using a recursive Fiduccia-Mattheyses algorithm.

Experimental Results

Table 5.4 presents statistics on the circuits and the implementation of these circuits on a minimum-sized FPGA. For each statistic and each network type, the table presents the average for all 11 sets for each type of network. The data for each circuit can be found in Appendix C.

The results show that the bus-based circuits are 25% larger than the dataflow and the star network. This additional logic implements the interface circuitry required to coordinate transfers on the bus. Some of the dataflow circuits contain additional flipflops which are added when feedback loops are introduced. The star-based circuits additionally contain only the logic required to implement the single bit nets. This added logic is negligible relative to the average size of the circuits.

The results also show that the dataflow and star-based circuits have significantly more I/O than the bus-based circuits. This points to an important limitation of this generator.

Table 5.4: Comparison between Network Types

	Average		
	Bus	Dataflow	Star
Number LEs	18019	14839	14839
Number I/Os	263	631	765
Rent Parameter	0.737	0.723	0.726.008
Number of Clusters	4559	3667	3656
Number Nets	12758	10712	10724
FPGA size	63x63	71x71	76x76
Net Length	14.5	16.9	17.9
Channel Width	38.5	40.0	40.0
Critical Path(ns)	34.1	33.6	31.6

When connecting circuits in a dataflow pattern, pins are created for the inputs of those sub-modules in the first stage of the dataflow at the top level, and for the outputs of those sub-modules in the last stage of the dataflow at the top level. Depending on the sizes of these sub-modules, this could result in a large number of pins. Similarly for the star-connected circuits, pins are created for nets not involved in the star pattern. For the bus circuits, pins are created for all of a leaf module's inputs or outputs beyond the bus width. The FPGA size for the dataflow and star circuits is larger than the bus circuits to accommodate the extra I/O pins.

The average net length of the bus-based circuits is less than that of the other circuits likely due to the smaller average FPGA size. Nonetheless, bus wires are expected to be long since they connect different modules. This is counteracted by the locality of the logic which allows the bus logic to be placed close together as the figures in Appendix D show.

The other parameters are not strongly affected by the network type.

5.4.3 Rent Parameter for Heterogeneous Circuits

All circuits used in this work were partitioned to find the Rent parameter. When partitioning heterogeneous circuits, the number of external pins on the subcircuits observed is expected to show more variation than for homogeneous circuits. The logic contained in each partition may originate from different sub-modules with different Rent parameters; thus the number of external pins would vary.

This intuition was formalised in [66] which derives an equation for the Rent parameter of heterogeneous circuits given the subcircuits' Rent parameters and sizes. According to their work, the Rent parameter for heterogeneous circuits should be the weighted average of the component modules where the weights are proportional to each module's size. This work assumes that when the number of partitions is large, each subcircuit only contains logic from one sub-module.

Ignoring any added logic for the network implementation, and using the same algorithm to calculate the Rent parameters for the sub-modules, the technique in [66] produced Rent parameters for the circuits built using this generator that are consistently lower than the values measured using our partitioning method. The average Rent parameter measured using the equation in [66] for the 11 benchmark sets in Section 5.4.2 is 0.665 while the average values measured using partitioning is 0.729.

Assuming a Rent parameter of 1 for all the additional bus interface logic, the average Rent parameter for the bus network circuits would then be 0.747, which is slightly higher than the value measured using partitioning. This suggests that the Rent parameter for the added bus interface logic is quite high. Though the bus network contains logic to implement the bus interfaces, the dataflow and star networks contains only a negligible amount of additional logic. The difference between the measured value using partitioning and the value predicted using the equation in [66] is likely a result of the complexity of the nets connecting the different modules.

This suggests that measuring the Rent parameter using solely the Rent parameters of the sub-modules is insufficient. Regardless, the Rent parameter calculated using either partitioning or the equation described in [66] reflects the *overall* Rent parameter. Care must be taken when trying to predict circuit characteristics for heterogeneous circuits using methods derived using Rent's parameter such as as [11]. If the derivation assumes a homogeneous Rent parameter to model the change in nets or pins while integrating over an area, it may not be able to take into account the effect of local variation.

5.5 Summary

This chapter compared the properties of SoC-style circuits built by this generator against the properties of circuits built by the previous synthetic generators, GEN and Gnl. These results were also compared to those obtained the eASIC benchmark suite. It was found that the post-routing results from this circuit generator scale well with respect to size. Assuming this trend holds as the circuits become larger, the estimated results for these circuits would be within the observed set of results for the eASIC circuits. The results from GEN and Gnl were found to quickly increase in complexity with respect to the size of the circuit. If these post-routing results were projected to the size of the eASIC benchmarks, the results from these circuits would be larger than the results from the eASIC benchmarks.

The growth in the number of I/O pins is linear relative to the size of the circuit. This shows that the mismatch in the pins of the sub-modules limits the range of realistic circuits that can be built using this generator. The effect of the network logic was isolated and evaluated. Bus-style networks add on average 25% to the circuit size when MCNC circuits are used in the library. We found that circuits constructed using the dataflow and star modules have particularly high I/O pins requirements; however, the remaining circuit characteristics are relatively unaffected by the network type.

Chapter 6

Conclusions

6.1 Summary

The first part of this thesis described the findings of a survey of 66 academic and industrial SoC circuits. We viewed each circuit as a tree circuit structure where the IP modules are leaves. Both the structure and the composition of the circuits were analysed. Three common connection patterns were identified during this analysis: bus, dataflow, and star.

The second part of this thesis described the algorithms used in the stochastic SoC circuit generator. The probability distributions used in this process were derived using data from the survey done in the first part of this work. A method was developed to automatically generate a reasonable set of values for the primary circuit parameters given that the set may be partially specified by the user. An algorithm was then described that connects circuit elements into a legal circuit structure. Lastly, bit-level strategies to connect sub-modules with mismatched pin counts were developed. This generator has been released to the research community.

In the third part of this thesis, the circuit generator implemented in this work was evaluated. Circuits from our generator were compared to circuits from the previous circuit generators GEN and Gnl. These SoC-style circuits show lower FPGA resource requirements than the homogeneous circuits likely due to the locality of our circuits. When compared to the eASIC benchmark circuits, the post-routing results for our SoC-style circuits matched the range of values found for the benchmark results. The homogeneous

circuits, on the other hand, grew quickly in complexity and would require far more resources than the eASIC circuits for the same circuit size. The similarity of the resource requirements between the SoC-style synthetic circuit generator developed in this work and the eASIC benchmark circuits suggests our synthetic circuits are suitable for FPGA architecture experimentation.

The number of I/Os on the circuits grows linearly relative to the size of the circuits. This indicates that the range of realistic circuits that can be built by this generator is limited by the mismatch in the sub-module pins. The network type does not strongly affect the circuit characteristics, although bus-style networks add approximately 25% more logic on average; and dataflow and star networks generally result in more module I/O.

These stochastic circuits are an important tool in the arsenal of any FPGA architect. Suitable "real" benchmarks that are large enough to investigate future FPGA architectures are difficult to obtain. Even if they could be obtained, there is value in being able to generate a "family" of benchmark circuits that have a specific size or that have a specific property. This allows FPGA architects to study exactly what types of circuit structures are well supported by a proposed device, and what sort of structures are not supported well. Synthetic circuits will not replace "real" benchmark circuits entirely, but used correctly, these synthetic circuits will enable FPGA architecture research that would otherwise be very difficult to perform.

6.2 Summary of Contributions

This work has made the following contributions:

- Results from a study of contemporary SoC circuit structure and composition were presented.

- Algorithms for a top-down synthetic SoC circuit generator were developed using the observations gathered during the survey. These algorithms were implemented in a software program that has been released to the research community.

- An investigation into the scalability of different synthetic circuit generators was performed. The post-routing results from this generator closely match the results generated by the eASIC benchmark circuits, while the results from previous circuit generators would exceed the eASIC resource requirements. This leads us to believe that the circuits built by our generator are suitable for use in FPGA architecture experimentation.

6.3 Future Work

There are a number of areas in this research where further work can be done in order to make the generator more realistic and more scalable. These improvements can be made to both the behaviour of the algorithm and also to the contents of the modules.

One area of potential improvement is to modify the behaviour of the generator with respect to the I/O pins. As described earlier, a large number of pins are created when the top level contains a dataflow or star connection pattern. The circuit generator is meant to be used in FPGA architectural experiments, and in most of these experiments, an excess of I/O pins will not be a problem. However, such a circuit is more likely to become pad-limited, relaxing the importance of effective packing and making routing easier. Thus, when using these new circuits in FPGA architectural experiments, it is important to take note of whether a circuit is pad-limited, and if so, interpret the results appropriately.

Embedded memory blocks are an important part of modern FPGAs as seen in Tables 3.4 and 3.5. For FPGA architecture experiments involving the logic fabric only, this will

not be a problem; however, to fully exercise all parts of an FPGA, memory should be included. It would be straightforward to add memory blocks as leaf modules; a more careful extension would consider common memory connection patterns such as those described in [63] and integrate these patterns into the circuit generator. These memory blocks would also help control the number of excess pins, since some pins on large blocks such as CPUs would normally be expected to connect to an exclusive memory cache.

Another area of future research is to add more elaborate network-on-chip structures that may be found on future SoC's. Packet-based gridded networks are emerging as the number of elements on a SoC grows, and these structures could be included in this generator. It is anticipated that these structures would contain a large number of point-to-point connections; the results from Section 5.4.2 suggest that this will influence the average net length and channel width of the resulting implementation.

Support

This work was funded by Altera and the Natural Sciences and Engineering Research Council of Canada.

Bibliography

[1] Actel. *Global Resources in Actel Low-Power Flash Devices*, 2008.

[2] E. Ahmed and J. Rose. The effect of LUT and cluster size on deep-submicron FPGA performance and density. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(3):288–298, 2004.

[3] Altera. *FLEX 8000 Programmable Logic Device Family Datasheet*, 11.1 edition, 2003.

[4] Altera. *Avalon Interface Specifications*, 1.1 edition, 2008.

[5] Altera. *Stratix III FPGA Device Family Overview*, 2008.

[6] ARM. *AMBA Specification*, 2nd edition, 2001.

[7] V. Betz, J. Rose, and A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[8] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic. *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, 1992.

[9] Y. W. Chang, D. Wong, and C. Wong. Universal switch modules for FPGA design. *ACM Transactions on Design Automation of Electronic Systems*, pages 330–331, 1999.

[10] S. Y. L. Chin and S. J. E. Wilton. Memory footprint reduction for FPGA routing algorithms. In *International Conference on Field-Programmable Technology (ICFPT)*, pages 1–8, 2007.

[11] P. Christie and D. Stroobandt. On the interpretation and application of Rent's rule. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(6):639–648, 2000.

[12] J. Cong and Y. Ding. Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and systems*, 13(1):1–12, January 1994.

[13] Altera Corp. Quartus II University Interface Program (QUIP). http://university.altera.com/research/unv-quip.html.

[14] J. Das. Personal Communication.

[15] V. Degalahal and T. Tuan. Methodology for high level estimation of FPGA power consumption. In *Asia and South Pacific: Design Automation Conference*, January 2005.

[16] G. DeMicheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[17] eASIC. ePrize1, 2008. http://code.google.com/p/eprize1/.

[18] C. Ebling, L. McMurchie, S. A. Hauck, and S. Burns. Placement and routing tools for the Triptych FPGA. *IEEE Transactions on VLSI*, 3(4):483–482, December 1999.

[19] W. M. Fang and J. Rose. Modeling routing demand for early-stage FPGA architecture development. In *International Symposium on Field Programmable Gate Arrays*, pages 139–148, February 2008.

[20] D. Ghosh, N. Kapur, J. Harlow, and F. Brglez. Synthesis of wiring signature-invariant equivalence class circuit mutants and applications to benchmarking. In *Design Automation and Test in Europe*, pages 656–663, February 1998.

[21] S. Gladston, B. Hoffman, and N. Gregoire. *SXP (Simple eXtensible Pipeline) Processor*. Opencores, 2001.

[22] D. Grant and G. Lemieux. Perturb+mutate: Semi-synthetic circuit generation for incremental placement and routing. *ACM Transactions on Reconfigurable Technology and Systems*, 1(3):1–24, 2009.

[23] L. Hagen, A. B. Kahng, F. J. Kurdahi, and C. Ramachandran. On the intrinsic Rent parameter and spectra-based partitioning methodologies. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(1):27–37, 1994.

[24] J. Harlow and F. Brglez. Synthesis of ESI equivalence class combinational circuit mutants. Technical Report 1997-TR@CBL-07-Harlow, North Carolina State University, October 1997. Also available at http://www.cbl.ncsu.edu/publications.

[25] M. Hutton. The Circuit Characterization and Generation Project at the University of Toronto. `http://www.eecg.toronto.edu/~mdhutton/gen/index.html`.

[26] M. Hutton, J. Rose, and D. Corneil. Automatic generation of synthetic sequential benchmark circuits. *IEEE Transactions on Computer-Aided Design*, 21(8):928–940, 2002.

[27] M. Hutton, J. Rose, J. Grossman, and D. Corneil. Characterization and parameterized generation of synthetic combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):985–996, 1998.

[28] P. Jamieson and J. Rose. Enhancing the area-efficiency of FPGAs with hard circuits using shadow clusters. In *An Analytical Model Describing the Relationships between Logic Architecture and FPGA Density*, pages 1–8, December 2006.

[29] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. Technical Report 98-036, University of Minnesota, 1999.

[30] J. Kleinhans, G. Sigl, F. Johannes, and K. Antreich. Gordian: VLSI placement by quadratic programming and slicing optimization. *IEEE Transactions on CAD*, 10(3):356–365, March 1991.

[31] K. Kozminski and C. Stuart. Version 2.0 of the OASIS system available for distribution. SIGDA Newsletter. vol 22, number 3.

[32] E. S. Kuh and T. Ohtsuki. Recent advances in VLSI layout. *Proceedings of the IEEE*, 78(11):237–263, February 1990.

[33] P. Kundarewich and J. Rose. Synthetic circuit generation using clustering and iteration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(6):869–887, 2004.

[34] I. Kuon and J. Rose. Area and delay trade-offs in the circuit and architecture design of FPGAs. In *International Symposium on Field Programmable Gate Arrays*, pages 149–158, February 2008.

[35] A. Lam, S.J.E. Wilton, P. Leong, and W. Luk. An analytical model describing the relationships between logic architecture and FPGA density. In *International Conference on Field-Programmable Logic and Applications*, 1998. forthcoming.

[36] B. S. Landman and R. L. Russo. On a pin versus block relationship for partitions of logic graphs. *IEEE Transactions on Computers*, C-20(12):1469–1479, December 1971.

[37] G. Lemieux and D. Lewis. *Design of Interconnection Networks for Programmable Logic*. Springer, 2004.

[38] D. Lewis, E. Ahmed, G. Baeckler, V. Betz, M. Bourgeault, D. Cashman, D. Galloway, M. Hutton, C. Lane, A. Lee, P. Leventis, S. Marquardt, C. McClintock, K. Padalia, B. Pedersen, G. Powell, B. Ratchev, S. Reddy, J. Schleicher, K. Stevens, R. Yuan, R. Cliff, and J. Rose. The Stratix II logic and routing architecture. In *Field Programmable Gate Arrays (FPGA)*, pages 14–20, 2005.

[39] O. Lupas. *Serial UART*. Opencores, 2004.

[40] E. J. Marinissen, V. Iyengar, and K. Chakrabarty. ITC'02 SOC test benchmarks. http://www.hitech-projects.com/itc02socbenchm/, October 2007.

[41] C. Mark, A. Shui, and S. Wilton. A system-level stochastic circuit generator for FPGA architecture evaluation. In *International Conference on Field Programmable Technologies*, December 2008. forthcoming.

[42] A. Marquardt, V. Betz, and J. Rose. Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 37–46, 1999.

[43] I. Masud and S. J. E. Wilton. A new switch block for segmented FPGAs. In *International Conference on Field Programmable Logic and its Applications*, pages 274 – 281, 1999.

[44] F. Moisy. Ezyfit: a free curve fitting toolbox for Matlab. U. Paris Sud. Version 2.2.

[45] R. Murgai, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli. Performance directed synthesis for table look up programmable gate arrays. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 572–575, November 1991.

[46] F. N. Najm. A survey of power estimation techniques in VLSI circuits. *IEEE Transactions on VLSI Systems*, 2:446–455, 1994.

[47] OpenCores. http://www.opencores.org.

[48] J. Pistorius, M. Hutton, A. Mischenko, and R. Brayton. Benchmarking method and designs targeting logic synthesis for FPGAs. In *International Workshop on Logic Synthesis (IWLS)*, pages 230–237, May 2007.

[49] J. Pistorius, E. Legai, and M. Minoux. Partgen: A generator of very large circuits to benchmark the partitioning of FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(11):1314–1321, 2000.

[50] K. K. W. Poon, S. J. E. Wilton, and A. Yan. A detailed power model for field-programmable gate arrays. *ACM Transactions on Design Automation of Electronic Systems*, 10(2):279–302, 2005.

[51] C. Sechen and A. Sangiovanni-Vincentelli. The Timberwolf placement and routing package. *JSSC*, 20(2):510–522, 1985.

[52] STMicroelectronics. *UM0339: User manual*, 2007.

[53] D. Stroobandt. On an efficient method for estimating the interconnection complexity of designs and on the existence of region III in Rent's rule. In *Great Lakes Symposium on VLSI*, pages 330–331, 1999.

[54] D. Stroobandt, J. Depreitre, and J. Van Campenhout. Generating new benchmark designs using a multi-terminal net model. *INTEGRATION: the VLSI journal*, 27(2):113–129, 1999.

[55] D. Stroobandt and F. J. Kurdahi. On the characterization of multi-point nets in electronic designs. In *Proceedings of the 8th Great Lakes Symposium on VLSI*, pages 344–350, feb 1998.

[56] D. Stroobandt, P. Verplaetse, and J. Van Campenhout. Generating synthetic benchmark circuits for evaluating CAD tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(9):1011–1022, 2000.

[57] M. Tom, , and G. Lemieux. Logic block clustering of large designs for channel-width constrained FPGAs. In *Design Automation Conference*, 2005.

[58] M. Tom and G. Lemieux. Logic block clustering of large designs for channel-width constrained FPGAs. In *Field Programmable Gate Arrays (FPGA)*, pages 726–231, 2005.

[59] S. Trimberger. Keynote talk: Redefining the FPGA. In *International Conference on Field-Programmable Logic and Applications*, August 2007.

[60] P. Verplaetse, J. V. Campenhout, and D. Stroobandt. On synthetic benchmark generation methods. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 213–216, 2000.

[61] P. Verplaetse and D. Stroobandt. Gnl: Generate netlist. `http://trappist.elis.ugent.be/~dstrooba/gnl/`.

[62] P. Verplaetse, D. Stroobandt, and J. Van Campenhout. Synthetic benchmark circuits for timing-driven physical design applications. In *International Conference on VLSI*, pages 31–37, 2002.

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

[64] Xilinx. Virtex-5 user guide, May 2006.

[65] S. Yang. *Logic Synthesis and Optimization Benchmarks User Guide*. MCNC, 3 edition, 1991.

[66] P. Zarkesh-Ha, J. A. Davis, W. Loh, and J. D. Meindl. On a pin versus gate relationship for heterogeneous systems: Heterogeneous Rent's rule. In *Custom Integrated Circuits Conference (CICC)*, pages 93–96, 1998.

[67] Y. Zhuo, H. Li, and S. P. Mohanty. A congestion driven placement algorithm for FPGA synthesis. In *Field Programmable Logic and Applications (FPL)*, pages 1–4, 2006.

Appendix A

Surveyed Circuits

The author and either the title of the article/manual or the chip name are listed.

A.1 Microprocessor

Bright, A., Ellavsky, M., et al.	Creating the BlueGene/L Supercomputer from Low-Power SoC ASICs
Shiota, T., Kawasaki, K., et al.	A 51.2 GOPS 1.0GB/s-DMA Single Chip Multi-Processor Integrating Quadruple 8-Way VLIW Processors
Luftner, T., Berthold, J., et al.	A 90nm CMOS Low-Power GSM/EDGE Multimedia Enhanced Baseband Processor with 380MHz ARM9 and Mixed-Signal Extensions
Torrii, S., Suzuki, H., et al.	A 600MIPS 120mW 70uA Leakage Triple-CPU Mobile Application Processor Chip
Analog Devices	ADSP-BF54x Blackfin Processor: Hardware Referencee
Renesas	H83644
Renesas	SH7721
Renesas	SH7785
Infineon	TC1775 32-Bit Single-Chip Microprocessor
Cirrus Logic	EP7311

Hattori, T., Irita, T., et al.	A Power Management Scheme Controlling 20 Power Domains for a Single-Chip Mobile Processor
Cappelli, A., Lodi, A.,et al.	XiSystem: a XiRisc-Based SoC with a Reconfigurable IO Module
Yohida, Y., Kamei, T. et al.	A 4320MIPS Four-Processor Core SMP/AMP with Individually Managed Clock Frequency for Low Power Consumption
Chen, Z., Ananthanarayanan, P., et al.	A 25W SoC with Dual 2GHz Power Cores and Integrated Memory and I/O Subsystems
Bae, Y. D., Park, I. C.	A 4.75GOPS Single-chip Programmable Processor Array Consisting of a Multithreaded Processor and Multiple SIMD and IO Processors
Khan, A., Ruparel, K., et al.	Design and Development of 130-nanometer ICs for a Multi-Gigabit Switching Network System
Cali, L., Lertora, F., et al.	Platform IC with Embedded Via Programmable Logic for Fast Customization
Bocchi, M., De Bartolomeis, C., et al.	A XiiRisc-based SoC for Embedded DSP Applications
Pham, D., Behnen, E., et al.	The Design Methodology and Implementation of a First-Generation CELL Processor: A Multi-Core SoC
Jung, Y., Kim, J., et al.	A Digital 120Mb/s MIMO-OFDM Baseband Processor for High Speed Wireless LANs
Khan, A., Watson, P., et al.	A 90nm Power Optimization Methodology and its Application to the ARM 1136JF-S Microprocessor
AMCC	PPC405EP

A.2 Networking

Hilton, C., Nelson, B.	PNoC: a flexible circuit-switched NoC for FPGA-based systems
Lee, K., Lee, S.J., Yoo, H.J.	Low-Power Network-on-chip for High-Performance SoC Design
Murali, S.; De Micheli, G.	Bandwidth-Constrained Mapping of Cores onto NoC Architectures
Axis	FS Designers Reference Axis Communications AB
Broadcom	Univeral Advanced Docsic 2.0 Downstream
Broadcom	Dual Universal Advanced TDMA/SCDMA PHY-Layer Burst Receive
Broadcom	BCM5758: Product Brief - Application Processor for Network Management Applications
Broadcom	BCM4506 - Product Brief - Dual Advanced Modulation Satellite Receiver
Nathawad, L., Weber, D., et al.	An IEEE 802.11a/b/g SoC for Embedded WLAN Applications
Bonnaud, P.-H., Hammes, M., et al.	A Fully Integrated SoC for GSM/GPRA in 0.13um CMOS
Mehta, S., Si, W. W., et al.	A 1.9 GHz Single-Chip CMOS PHS Cellphone
Broadcom	BCM6358

A.3 Multimedia

Broadcom	BCM1101: Product Brief - Enterprise IP Phone Chip
Yamauchi, H., Okada, S., et al.	An 81MHz, 1280x720pixelsx30 frame/s MPEG-4 Video/Audio Codec Processor
Fujiyoshi, T., Shiratake, S., et al.	An H.264/MPEG-4 Audio/Visual Codec LSI with Module-Wise Dynamic Voltage/Frequency Scaling
Kim, D., Chung, K., et al.	An SoC with 1.3Gtexels/s 3D Graphics Full Pipeline Engine for Consumer Applications
Sohn, J.-H., Woo, J.-H., et al.	A 50Mvertices/s Graphics Processor with Fixed-Point Programmable Vertex Shader for Mobile Applications
Broadcom	BCM3551
Broadcom	BCM7038
Broadcom	BCM7440
Broadcom	BCM7021
Cirrus	CS49500/10/20
Cirrus	CS4961xx
Corelogic	CLI5000/5001/5002
Corelogic	CLH31X
Bathaee, M., Ghezelayagh, H., et al.	A 0.13um CMOS SoC for All Format Blue and Red Laser DVD Front-end Digital Signal Processor
Pan, J.-S., Hsu, T.-H., et al.	Fully Integrated CMOS SoC for 56/18/16 CD/DVD-dual/RAM Applications with On-Chip 4-LVDS Channel WSG and 1.5Gb/s SATA PHY

Chang, Y.-W., Fang, H.-C., et al.	124MS/s Pixel-Pipelined Motion-JPEG 2000 Codec without Tile Memory
Lin, C. C., Guo, J. I., et al.	A 160kGate 4.5kB SRAM H.264 Video Decoder for HDTV Applications
Ueda, Y., Yamauchi, H., et al.	6.33mW MPEG Audio Decoding on a Multimedia Processor
Huang, Y. W., Chen, T. C., et al.	A 1.3TOPS H.264/AVC Single-Chip Encoder for HDTV Applications
Vangal, S., Howard, J., et al.	An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS
Abbo, A., Kleihorst, R., et al.	XETAL-II: A 107 GOPS, 600mW Massively-Parallel Processor for Video Scene Analysis
Nam, B.-G., Lee, J., et al.	A 52.4mW 3K Graphics Processor with 141Mvertices/s Vertex Shader and 3 Power Domains of Dynamic Voltage and Frequency Scaling
Khailany, B., Williams, T., et al.	A Programmable 512 GOPS Stream Processor for Signal, Image, and Video Processing
Kim, S. H., Yoon, J.S., et al.	A 36fps 3D Display Processor with a Programmable 3D Graphics Rendering Engine
Nagano, K., Okamoto, K., et al.	A 0.13um CMOS Ultra-compact DVD SoC employing a Full Digital Equalizing PRML Read Channel
Huang, C., Ravi, S., et al.	Eliminating Memory Bottlenecks for a JPEG Encoder Through Distributed Logic-Memory Architecture and Computation-unit Integrated Memory
Park, S., Cho, H., et al.	An Implemented H.24 Video Decoder using Hardware and Software

A.4 Miscellaneous

Infineon	Security & Chip Card ICs SLE 66CX360PE
Innova Card	USIP Professional IC
Lewellen, T., Miyaoka, R., et al.	System Electronics for the MiCES Small Animal PET Scanner
Park, J., Hwang, J.-T., Kim, Y.-C.	FPGA and ASIC implementation of ECC Processor for Security on Medical Embedded System
Bainbridge, W. J., Plana, L.A., Furber, S.B.	The Design and Test of a Smartcard Chip Using a CHAIN Self-timed Network-on-Chip

Appendix B

Bus Interface Pins

The widths may differ if the sub-module is either the reset source or interrupt sink.

Table B.1: Slave Bus Interface

Bus Side	Module Side
RW signal	IP load
Address [1..Bus width]	IP Data Input [1..Module inputs]
Data to Slave [1..Bus width]	IP Data Output [1..(Module outputs - 1)]
Data to Master [1..Bus width]	Clock
Clock	Reset
Reset	

* The RW signal is managed by the interface

Table B.2: Master Bus Interface

Bus Side	Module Side
RW signal	IP load
Address [1..Bus width]	RW signal
Data to Slave [1..Bus width]	IP Data Input [1..Module inputs]
Data to Master [1..Bus width]	IP Data Output [1..(Module outputs - 2)/2]
Clock	IP Address Output [1..(Module ouputs - 2)/2]
Reset	Clock
	Reset

* The RW signal is managed by the interface

Appendix C

Network Type Experiment Results

(Table follows on next page)

Table C.1: Characterisation of Network Types: Circuit Properties

	No. LE (k)			No. I/O			Rent Parameter			No. Clusters (k)			No. Nets (k)			FPGA size		
No	B	D	S	B	D	S	B	D	S	B	D	S	B	D	S	B	D	S
1	17.3	15.8	15.8	450	343	544	0.75	0.75	0.75	4.4	4.0	4.0	12.5	11.3	11.5	67	64	64
2	3.4	1.4	1.4	42	215	95	0.67	0.64	0.65	0.9	0.4	0.4	2.2	1.1	1.1	30	19	19
3	44.0	37.3	37.3	613	854	2739	0.78	0.77	0.77	11.1	9.5	9.4	31.3	26.8	27.8	106	98	229
4	8.1	6.4	6.4	88	114	463	0.72	0.70	0.70	2.0	1.6	1.6	5.1	4.2	4.3	46	41	41
5	10.9	8.0	8.0	103	477	647	0.73	0.70	0.71	2.8	2.0	2.0	7.7	6.0	6.2	53	45	54
6	37.9	29.5	29.5	287	1516	1265	0.78	0.77	0.76	9.6	7.5	7.5	26.9	22.4	21.9	98	127	106
7	7.3	4.1	4.1	31	157	153	0.72	0.70	0.70	1.8	1.0	1.0	4.6	2.7	2.7	43	33	33
8	4.2	3.2	3.2	79	187	193	0.70	0.69	0.70	1.1	0.8	0.8	2.7	2.2	2.2	33	29	29
9	18.6	13.8	13.8	146	695	467	0.75	0.74	0.74	4.7	3.5	3.5	13.1	10.7	10.4	69	60	60
10	14.0	11.6	11.6	325	339	747	0.74	0.73	0.73	3.5	3.0	2.9	10.1	8.6	8.7	60	55	63
11	32.7	28.0	28.0	989	2470	1671	0.77	0.76	0.76	8.3	7.1	7.1	24.1	21.9	21.2	92	206	140

B=Bus; D=Dataflow; S=Star

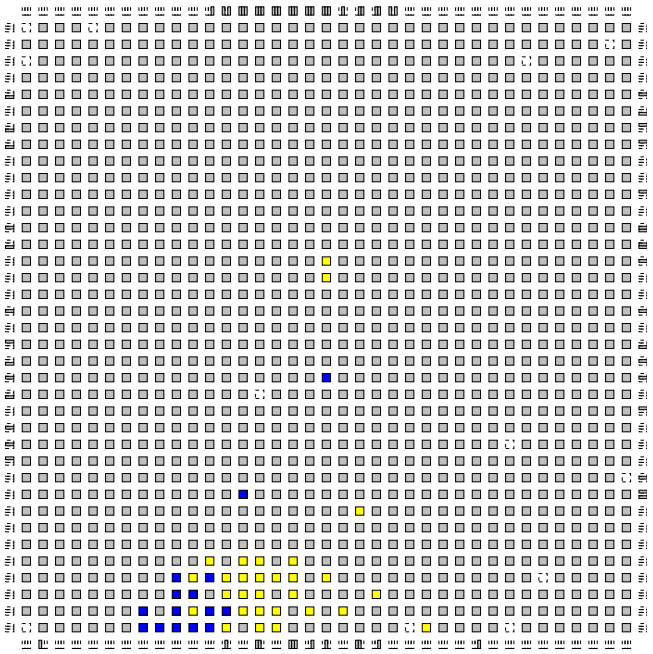
Table C.2: Characterisation of Network Types: Post-Routing Results

	Average Net Length			Channel Width			Critical Path (ns)		
No	B	D	S	B	D	S	B	D	S
1	18	17	18	52	52	48	39	37	38
2	9	10	11	20	24	26	9	16	17
3	16	17	32	44	44	44	60	62	54
4	17	18	18	42	40	42	24	17	18
5	13	14	14	36	36	36	41	42	40
6	16	21	20	52	54	58	54	47	44
7	14	15	15	30	30	34	23	17	14
8	13	14	14	30	30	32	15	16	12
9	14	16	16	38	42	40	38	39	39
10	14	15	16	38	36	36	38	42	40
11	16	29	23	42	52	44	48	54	46

B=Bus; D=Dataflow; S=Star

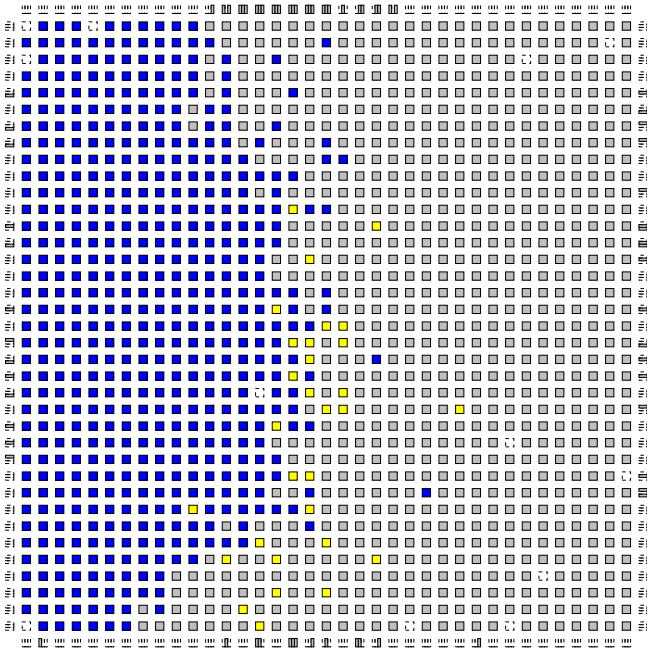
Appendix D

SoC Post-Placement Example

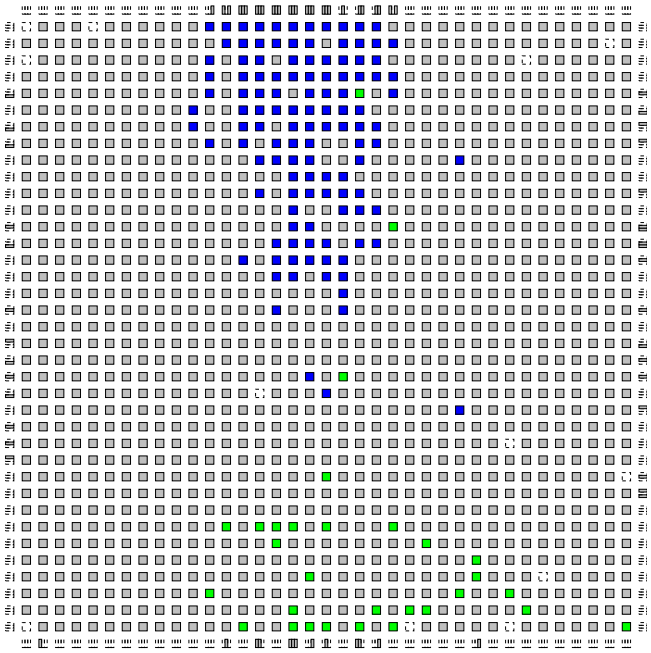


(a) Module 1

Figure D.1: Example of SoC Post-Placement Logic Locality: Blue squares are clusters containing that module's logic; Yellow squares are clusters containing slave interface logic; Green squares are clusters containing master interface logic; Grey squares are clusters that are unused by that module; Hollow squares are clusters that are unused by any module

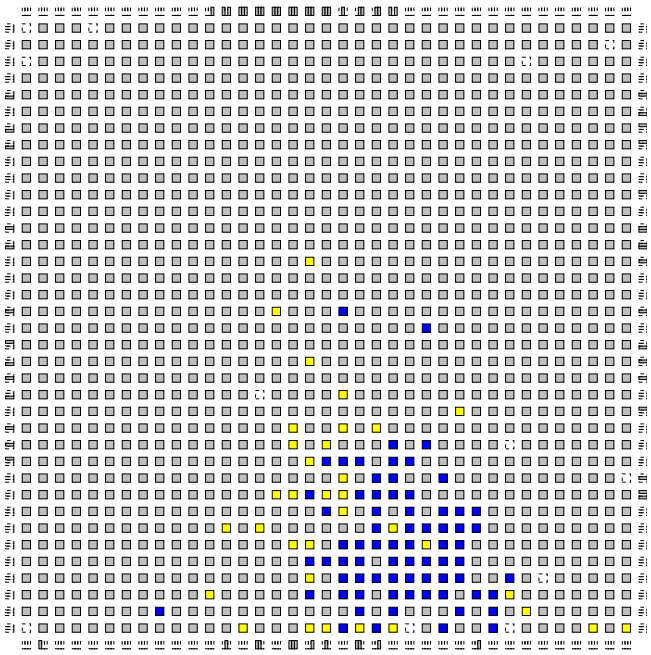


(b) Module 2

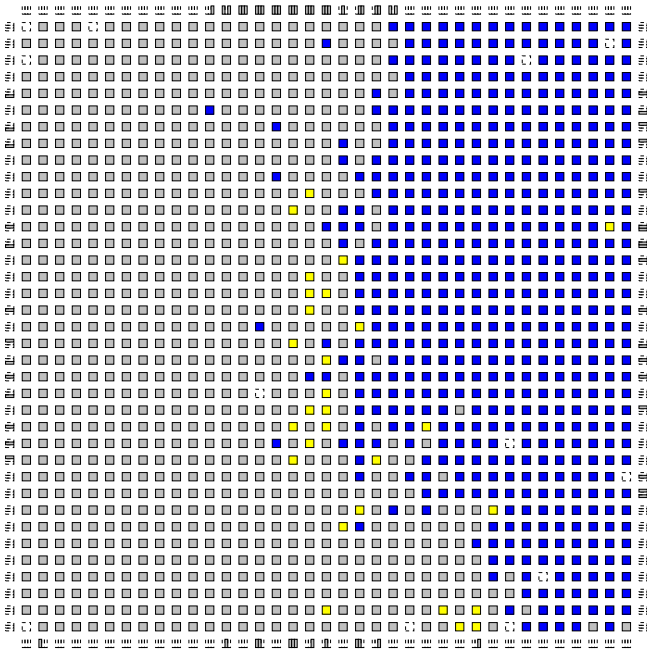


(c) Module 3

Figure D.1: Example of SoC Post-Placement Logic Locality (continued)



(d) Module 4



(e) Module 5

Figure D.1: Example of SoC Post-Placement Logic Locality (continued)

Appendix E

eASIC Post-Routing Results

Table E.1: eASIC 1 Post-Routing Results

	LE	No. of Nets	Net Length	Channel Width	Critical Path (ns)
0	98884	69175	22.2429	112	136.182
1	95694	67289	23.31	92	124.685
2	97893	58347	21.3289	66	78.8533
3	91313	57582	18.6057	88	*
4	98834	68525	23.2005	102	87.5604
5	92434	64865	24.0966	90	124.31
6	97906	67294	26.2198	82	130.489
7	86710	56592	20.8404	74	90.4157
8	86658	56607	22.0885	78	89.7965
9	100620	68850	24.2235	76	109.458

* Synthesized using timing invariant methods

Table E.2: eASIC 2 Post-Routing Results

	LE	No. of Nets	Net Length	Channel Width	Critical Path (ns)
0	78031	48227	21.5491	68	77.9169
1	92216	57444	25.9866	102	82.0337
2	83360	51633	22.9695	76	96.247
3	90105	56987	32.3982	80	79.7303
4	93469	57931	42.9972	104	104.229
5	82816	51543	21.7935	70	85.3263
6	92284	57470	24.8204	80	80.4697
7	89677	55472	23.5067	80	93.2832
8	82794	48965	30.243	80	61.7551
9	90978	59629	36.8413	116	111

Table E.3: eASIC 3 Post-Routing Results

	LE	No. of Nets	Net Length	Channel Width	Critical Path (ns)
0	87862	41521	26.9248	60	44.261
1	92158	49797	25.5695	74	75.6516

Table E.3: eASIC 3 Post-Routing Results (continued)

	LE	No. of Nets	Net Length	Channel Width	Critical Path (ns)
2	102642	51694	29.1663	68	52.7585
3	107143	51277	31.5987	66	52.437
4	98595	53141	30.5125	82	58.6319
5	87236	44613	25.0459	60	49.3659
6	90567	46133	25.742	64	73.3277
7	101021	52159	28.5409	74	58.2856
8	97490	52676	30.5529	78	56.5732
9	105490	50304	30.9852	102	50.2337

Table E.4: eASIC 4 Post-Routing Results

	LE	No. of Nets	Net Length	Channel Width	Critical Path (ns)
original	125637	67841	23.2918	72	66.9218
0	13169	7009	15.8132	36	25.8787
1	11168	5231	13.3855	26	13.9489
2	14523	8677	17.1418	46	45.5342
3	11820	6028	12.6745	30	13.5887
4	11714	4990	13.7438	24	11.0972
5	12496	7372	14.5045	40	17.4525
6	11186	7037	14.6926	38	16.4419
7	13043	7395	14.2031	36	14.4713
8	12987	6523	15.8534	36	19.4205
9	13531	7803	14.6069	38	21.3247

Table E.5: eASIC 5 Post-Routing Results

	LE	No. of Nets	Net Length	Channel Width	Critical Path (ns)
0	101497	68978	20.4291	82	*
1	115249	74865	36.1794	126	60.4511
2	100703	61389	25.3643	76	87.8322
3	99503	63150	18.5145	76	*
4	96888	62367	23.4991	88	84.4122
5	97766	52146	20.4501	62	65.337
6	93425	55366	19.8603	68	136.487
7	95187	59441	25.7129	82	101.227
8	106052	74963	23.5209	106	92.9103
9	101851	74896	24.8286	92	144.563

* Synthesized using timing invariant methods