A Stochastic RTL Circuit Generator for FPGA Architecture and CAD Evaluation

by

Motahareh Mashayekhi

B.Sc. Sharif University of Technology, 2014

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

Master of Applied Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

February 2017

Abstract

The performance and capacity of Field-Programmable Gate Arrays (FPGAs) have dramatically improved in recent years. Today these devices are emerging as massively reconfigurable and paralleled hardware computation engines in data centers and cloud computing infrastructures. These emerging application domains require better and faster FPGAs. Designing such FPGAs requires realistic benchmark circuits to evaluate new architectural proposals. However, the number of available benchmark circuits is small, outdated, and few of these are representative of realistic circuits.

A potential method to obtain more benchmark circuits is to design a generator that is capable of generating as many circuits as desired that are realistic and have specific characteristics. Previous work has focused on generating benchmark circuits at the netlist level. This limits the usefulness of these circuits in evaluating FPGA Computer Aided Design (CAD) algorithms since it does not allow for the evaluation of synthesis or related mapping algorithms. In addition, these netlist level circuit generators were calibrated using specific synthesis tools, which may no longer be state of the art. In this thesis, we introduce an Register Transfer Level (RTL) level circuit generator that can automatically create benchmark circuits that can be used for FPGA architecture studies and for evaluating CAD tools. Our generator can operate in two modes: as a random circuit generator or as a clone circuit generator.

The clone circuit generator works by first analyzing an input RTL circuit then it gen-

erates a new circuit based on the analysis results. The outcome of this phase is evaluated by measuring the distance between certain post-synthesis characteristics of the generated clone circuit and those of the original circuit. In this study we generated a clone circuit for each of the VTR set of Verilog benchmark circuits. We generate clones with post-synthesis characteristics that are within 25% of the corresponding characteristic of the original circuits. In the other mode, the random circuit generator extracts the analysis results from a set of RTL circuits and uses that data to generate a random circuit with post-synthesis characteristics in an acceptable range.

Preface

This dissertation is original, independent work by the author, M. Mashayekhi, under the supervision of Professor Steve Wilton.

Table of Contents

List of Tables

List of Figures

List of Codes

Acknowledgments

I would like to thank Professor Steve Wilton for his patience and guidance.

Chapter 1

Introduction

1.1 Motivation

Recent years have seen dramatic improvements in the capabilities of Field-Programmable Gate Arrays (FPGAs). Early FPGAs were optimized to implement glue logic. The availability of larger FPGAs enabled entire systems on programmable devices, and this lead to FPGAs containing embedded memories, digital signal processing (DSP) blocks, specialized I/O interface circuitry, and full-featured embedded processors. Today, we are seeing the emergence of FPGA technology in the cloud and in data centers, as as evidenced by Intel's recent aquisition of Altera and Microsoft's efforts to bring FPGA technology into the cloud [35]. This emerging application domain has the potential to change the way FPGAs are used and built. As the capabilities and use cases of FPGAs expand, there is an increasing need to design new FPGAs. Not only do new devices need to be larger and faster, but it is conceivable that both the architecture (internal structure) of the FPGA as well as the associated computer-aided design algorithms need to change. Current FPGA CAD tools take hours (or even a day) to compile a large design; while this *may* be acceptable for hardware designers, it may not be acceptable for the new breed of designers using FPGAs to accel-

erate cloud-based applications. Providing new compilation tools requires both a change in the algorithms as well as the FPGA fabric to which the algorithm is mapping.

Designing a new FPGA, however, is challenging. FPGA architects must provide just enough programmability in their devices; too much programmability leads to wasted power and slow devices, while too little programmability (or misplaced programmability) leads to devices that are not flexible enough. Moreover, FPGA architects need to balance the desire to include new embedded blocks (such as new embedded computational units) with the desire to create truly general devices that can be used by a wide variety of their customers. Finally, it is well-known that the design and optimization of FPGA architecture and the associated compilation tools cannot be performed in isolation. An architectural feature that cannot be efficiently be used by the CAD tools represents wasted silicon. Only by co-optimizing the architecture and CAD can efficient programmable system and ecosystems be developed.

Although some work has been performed on analytical modeling of FPGAs [11], most FPGAs today are designed using experimental techniques. Engineers create models of potential architectures and CAD tools, and use experimental CAD tools to map a set of benchmark circuits to each potential device [27]. Detailed area and delay models provide estimates of the efficiency of each architecture, allowing the architect to make informed decisions regarding the trade off between flexibility and efficiency. The selection of suitable benchmark circuits is critical in this process. Benchmark circuits must be representative of circuits that will eventually be implemented on the FPGA being developed. Many studies use circuits that are far too small to adequately exercise any modern CAD tool or architecture. Researchers have recently made progress towards releasing larger benchmark suites [32], however, even these are typically representative of existing circuits, not circuits that will be used on future devices.

A potential solution is to use automatically-generated synthetic benchmark circuits

[31], [15], [21], [23] [41]. Typically, these circuits are created using a circuit generator which creates synthetic 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: a researcher 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 (eg. dataflow circuits [23]). This latter advantage is critical during early evaluation, when it is important to understand what types of circuits work well and what types do not.

1.2 Contribution

In this thesis, we describe a synthetic benchmark circuit generator which unlike previous generators, generates circuits at the register-transfer level (RTL); as we will describe in the next chapter, benchmark circuits expressed in RTL are much more suitable for the types of architecture and CAD studies that researchers often want to perform. In order to ensure our generator produces realistic circuits, we base our generation on a set of statistics obtained from existing circuits and then generate synthetic circuits guided by those statistics. In addition our generator works in two different modes, generating a clone from one specific circuit or it generating a random circuit based on the information gathered from multiple circuits. It worth mentioning that our generated circuits are based on patterns and statistics rather than implementing a specific functionality. Therefore our circuits are useful for evaluating the impact of architectural or CAD algorithm enhancements on major FPGA design metrics such as delay and area. However, evaluating the impact of new designs on power consumption using our circuits is not practical since the major source of power consumption in FPGAs are dynamic switching which is dependent on the functionality of the circuit.

1.3 Thesis Organization

This thesis is organized as follows. In Chapter 2, we provide background on FPGA architecture, CAD algorithms, FPGA experimental evaluation techniques, and discuss previous work on generating synthetic circuits. Chapter 3 then describes our characterization process, where we gather information about common RTL designs. Chapter 4 shows how we then use this information to generate a clone from an RTL design and Chapter 5 shows how we generate a random RTL design. The suitability of the circuits obtained from our generator is evaluated in Chapter 6. Finally, Chapter 7 concludes the thesis and suggests the future work.

Chapter 2

Background and Previous Work

2.1 Overview

In this chapter, we first briefly introduce FPGAs and two major area of related research: architecture and CAD algorithms. Second, we review FPGA architecture and the works that have focused on architectural refinements. Third, we discuss different stages of CAD algorithms and the attempts to improve the FPGA efficiency based on improving each stage. Then we explain the necessity of proper benchmark circuits for FPGA research validation. Finally we summarize the previous studies on generating benchmark circuits to facilitate FPGA experimental evaluation.

2.2 Field Programmable Gate Arrays

Field-Programmable Gate Arrays (FPGAs) have gained significant popularity for fast prototyping of digital systems in a variety of application domains such as embedded systems, cloud data bases, networking, and cryptography due to their high performance, high flexibility, fast time-to-market along with massive parallelism. The flexibility of FPGAs allows FPGA-based designs to be easily upgraded, recover from failures, and adopt to new stan-

dards. Such high flexibility, however, comes at considerable cost. FPGAs consume more power and require more area to implement a circuit than their Application-Specific Integrated Circuits (ASIC) counterparts. The area gap is reported to be about 40 times between FPGAs and ASICs [24]. Increased area directly increases the static power consumption of the device. In addition, it results in longer interconnects and therefore lower performance. Previous works have tried to reduce the area, performance, power, and reliability gap of FPGAs and ASICs by enhancing the FPGA architecture, or modifying Computer Aided Design (CAD) algorithms.

2.2.1 FPGA Architecture and Experimentation

The island style routing architecture is commonly used in commercial FPGAs such as Xilinx Virtex-6 [43] and Altera Cyclone-V [2]. This routing architecture which is also commonly used in academic FPGA CAD tools such as VPR [5], consists of a pool of interconnect resources surrounding a two-dimensional array of cluster logic blocks (CLBs). In modern FPGAs, there are also various full-custom blocks available as logic resources such as Digital Signal Processing (DSP) processors, block RAMs, and multipliers.

As shown in Figure 2.1 there are four programmable resources in island style FPGAs: CLBs, Connection Blocks (CBs), Switch Matrix or Switch Boxes (SBs), and Input/Outputs (IOs). CLBs are goups of logic blocks (LBs) and LBs consists of either reconfigurable Logic Elements (LEs) or prefabricated full-custom complex blocks performing specific operations such as multiplication. A typical LE is made of a Look-Up Table (LUT) and a sequential element such as Flip-Flop (FF). LBs are connected to interconnect resources via CBs. Routing interconnects are also connected to SBs and IOs. SBs provide the connections between vertical and horizontal interconnects.

The amount of silicon area dedicated to the routing fabric is usually dominant in FPGAs. Since routing resources do not perform any computation by themselves, they are

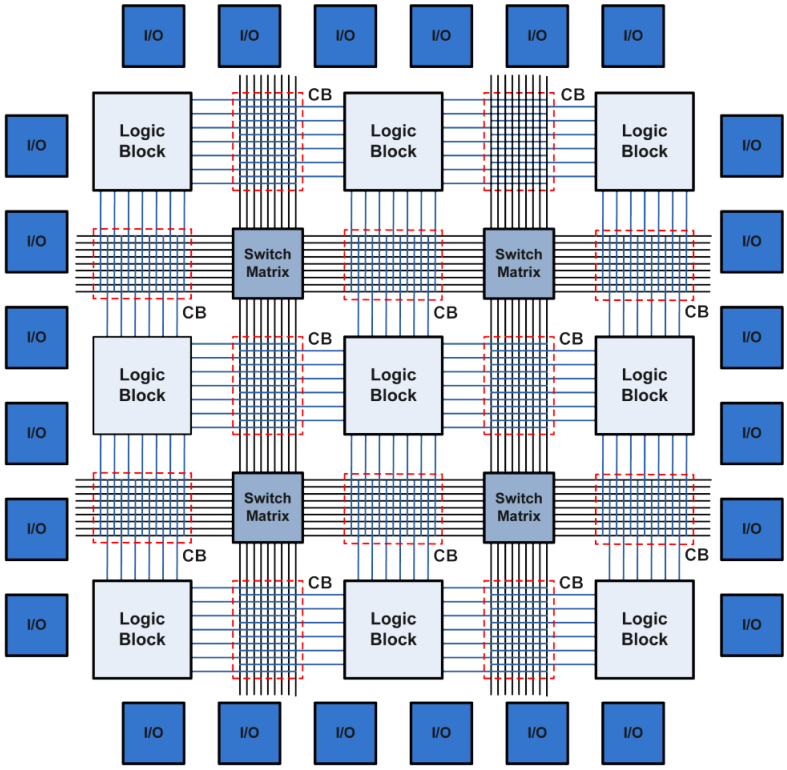

Figure 2.1: Overview of an FPGA routing structure and logic resources (CB = Connection Block, I/O = Input and output).

usually considered circuit overhead.

Logic Fabric Experimentation

The high level of flexibility provided by LUTs results in an excessive usage of silicon area compared to hard logic blocks. In addition, soft blocks are slower and less reliable than hard blocks. LUT size is an important parameter in FPGAs. Bigger LUTs result in lack of utilization and slower circuits when implementing simple functions while smaller LUTs require significant usage of the routing fabric to implement large functions and are therefore slower. A mixture of LUTs with different sizes can be used to improve LUT utilization and enhance performance by preventing LUT cascading to implement complex functions such as [9] and [10]. Hard logic cells have been used to improve the area efficiency of FPGAs

by employing efficient logic block that are capable of efficiently employing a limited set of functions [33] and [19]. These studies try to design efficient logic block that are capable of covering a fraction of all functions. The idea of hard logic blocks is widely employed in todays industrial FPGAs to the extent that full processors, memory blocks, and DSP blocks are implemented as hard cores [2] and [43]. A different approach in logic block optimization was taken by [17] which employed Clos networks between intra-cluster routing and logic element inputs to provide further flexibility in logic clusters. This approach has managed to reduce area and increase utilization by a proposed logic cluster also keeps the same performance despite increased logic depth. All of the above mentioned solutions have only contributed to a small fraction of the total silicon area. Hence, optimizing logic blocks in terms of area without addressing the routing fabric cannot effectively alleviate the FPGA/ASIC gap.

Routing Fabric Experimentation

Due to importance of the routing fabric, previous studies have aimed to improve performance, power, or dependability by modifying the routing fabric. Shadow clustering has been proposed to utilize the routing fabric in areas where hard logic has remained unused. Its main goal is to reduce the area overhead imposed by the routing fabric when employing hard logic cells [22]. To some extent, this technique manages to use silicon area more efficiently and avoids waste of resources. The method presented in [42], called Hard Wired Routing Patterns (HARP), reduces the area of routing fabric by using hard wired switch box patterns [38]. The optimum distribution of segments and combination of routing buffers with pass transistors have also been found to make the routing fabric more effective [6]. The use of short interconnect segments can reduce both power consumption and net delays in FPGAs at the cost of logic density [26]. Different modes of operation can also be employed in FPGAs to reduce power consumption in routing switches [4].

2.2.2 CAD Algorithms and Experimentation

The binary sequence used to program an FPGA is called a bit-stream. However engineers create their designs at higher levels of description such as register transfer language (RTL) level. The process of converting an RTL design to an FPGA bit-stream is a multi-stage and complex process which is done by CAD Algorithms including:

- RTL synthesis: The process of converting an RTL level design to a gate level and applying technology dependent and independent optimization.
- Technology mapping: Finding the optimal mapping solution of a gate level design to the available gate library of the target technology.
- Clustering: A clustering algorithm groups LBs into cluster logic blocks(CLBs) such that interconnect pattern of LBs within a cluster are similar to each other and those of LBs from different clusters are dissimilar.
- Placement: Locates each CLB on a specific resource of the target technology while attempting to minimizing the total interconnect length required or the critical path.
- Routing: Making the required connections between CLBs of the target technology using its available routing resources.

At any of these stages, modifications to existing algorithms can help reduce the FPGA-ASIC gap. While this approach may not reduce the gap as significantly as architectural modifications, it may still prove useful due to its low NRE cost and flexibility of modifications. Technology mapping algorithms can be modified to optimize any of the design parameters. Manohararajah et al. has proposed a technology mapping tool that can optimize designs for performance [28]. Cong and Ding [7] have investigated the trade-off between depth and area in technology mapping . It is also possible to optimize technology

mapping for dependability [8] or power [3]. Such modifications can also be made during clustering. [37] has proposed a clustering scheme to reduce area and power consumption of FPGAs. Placement and routing algorithms can also optimize a design. Sterpone and Violante have proposed a reliability oriented placement and routing algorithm to enhance FPGA reliability [39]. Wang et al. have also presented a power-efficient placement and routing algorithm [42].

2.2.3 FPGA Evaluation

In the previous section we discussed the previous research on architectural refinements and CAD Algorithm modifications to mitigate the FPGA and ASIC gap. All these potential enhancements require verification using experimental techniques. In another words researchers first need to model their potentially enhanced version of the FPGA architecture and CAD tool then synthesize benchmark circuits using them and measure the area, performance and power. Research validate their potential enhancements by comparing their new measurements to those obtained from the available technologies. This design and evaluation require which is proper benchmark circuits. Ideally a benchmark circuit is a customer circuit or a representative of the circuits that costumers are implementing on FPGAs. Both industrial and academic researchers indicate that obtaining such examples is challenging.

A solution to the lack of real circuits is to design a circuit generator that is capable of generating an arbitrary number of circuits with specific properties. Although there have been previous circuit generators, all of them characterize and generate circuits at the net list level or lower. This limits the usefulness of these circuits to evaluate physical design CAD algorithms (such as place and route algorithms) and does not allow for the evaluation of synthesis or related mapping algorithms. In addition, these gate-level circuit generators were calibrated using specific synthesis tools, which may no longer be state of the art. Thus, the gate level circuits may be unrealistic. In contrast, most designers specify circuits

at a higher-level of abstraction (RTL level) As will be described in the next sections, our work characterizes and generates circuits at the RTL level, which is fundamentally different than these previous works. In the next section we review some of these previous circuit generators, then in the following chapters we introduce a model of RTL designs, explain the algorithm of our RTL level circuit generator and verify its results.

2.3 Previous Work

We categorize the previous circuit generator approaches into graph-based, glue of logic and mutation. In this section we review some of the previous work from each category.

2.3.1 Graph Based

There have been several earlier efforts to generate synthetic circuits. Previous work in this area first model netlist circuit using graphs and then generates new netlist circuits based on these graphs.

Gnl

Stroobandt et al. in [40] presents a generator based on a multi-terminal net model which can generate netlists with a precise Rent exponent value [25], the relationship between the number of pins and the blocks in a partition of a logic design. The Rent exponent value, the number of LBs and input and outputs of each LB is defined by user. The algorithm first initiates the user defined number of LBs, then based on a bottom-up approach it pairs the LBs together to generate a CLBs. The number of pins after each pairing is decided based on the preset Rent exponent value. After all the LBs were grouped into CLBs, it pairs CLBs to generate a netlist. However a number of constraints have to be satisfied since not all set of user defined inputs will lead to a feasible netlist circuit. Stroobandt et al. in [41] add timing to their pervious work by using predefined cells as their lowest level cells. The

predefined cells can be a FF or any gates. As a result this approach can generate circuits with a functionality in contrast with their previous work that merely generates directed graphs.

GEN

Hutton et al. have proposed a method of generating a netlist of LUTs based on pre-processing MCNC benchmark circuits [1]. They model netlist of a combinational circuit using a directed acyclic graph and describe it using the following characteristics:

- The circuit size and the number of inputs and output pins.
- Combinational delay: Combinational delay of a node in the netlist is the longest path to reach that node starting from an input pin.
- Circuit shape: Circuit shape is the distribution function of nodes with different combinational delays.
- Edge length distribution: When an edge connects a node with a combinational delay a to another node with combinational delay b, the length of the edge is $|a-b|$. The edge length distribution is the distribution function of edges with different lengths.
- Fanout distribution: Fanout is the number of edges leaving a node and fan-out distribution is the distribution function of this value.
- Reconvergence: When edges that have a common ancestor have the same node as a sink.

In order to generate a new netlist all the aforementioned items are given as an input to the generator. Then the generator algorithm construct a netlist based on the inputs: the number of nodes at each combinational delay is known using the shape function and using

the edge length distribution the number of edges and the length of each edge are also known. In addition based on the fanout distribution, a set of valid fanouts for the nodes is known. As a result the generation problem is now formulated as the problem of constructing a graph given these constraints on the delays, edge lengths, and the fanouts. In this study a heuristic algorithm is described to address this problem. In their following work [21] they add backedges to their netlists graph in order to be able to generate sequential circuits.

2.3.2 Glue of Logic

Work in PartGen [34] and Mark [31] intuited that circuits are composed of several different kinds of logic connected using structured interconnect patterns (such as a bus or network on chip). By varying the proportion of these various types of logic, as well as the way they are interconnected, these generators are able to mimic different kinds of circuits in a realistic manner. More details on these previous work is as follows:

Mark

Mark et al. modeled circuits at netlist level as a network of modules [31]. Modules are categorized as processors such as CPUs, interfaces such as a UART, controllers such as USB controllers, and others. Networks categories are:

- Bus: A bi-directional data transfer between three or more modules.
- Star: A bi-directional data transfer between a master and a few slave modules.
- Dataflow: A uni-directional data transfer between a chain of modules.
- Others

The input circuit library in this work is the netlist of 66 circuits. According to their model each circuit is hierarchy of modules connected to each other using one of the network types. Based on these 66 circuits, they collected the following set of data:

- Distribution of different module types. For example 12% of the modules existing in their input circuit library are processors.
- Distribution of different network types.
- Distribution of hierarchy depth. For example 80% of their input circuit library have a zero hierarchy depth, i.e. 80% of them are consisting of a few leaf modules connected to each other by a single network.
- The average number of networks at a hierarchy level. The number of networks for the circuits with a hierarchy depth of zero is always one and for circuits with a hierarchy depth of one is two. The number of networks for the circuits with a maximum hierarchy depth of one is 1.81.
- Distribution of the number of leaf modules for each network type.

They generate netlist level circuits by using networks to glue leaf modules while using the collected data to ensure the generated circuits mimic the input circuits. They have divided their input circuit library to the four module category and used them as the leaf modules.

To validate this work, they have demonstrated that the trend of the post-synthesis results (eg. critical path versus number of LUTs) of their generated circuits is more realistic than previous generators, GEN and Gnl. They validation process is based on the *trends* of characteristics because the size of largest circuit that they can generate has 72625 number of four-input-LUTs which is much smaller than their available set of realistic circuits, eASIC circuits [12]. The eASIC circuits are a set of industrial circuits commonly used for verifying placement algorithms. These circuits are broken into 10 parts to be able to get simulated using academic CAD flows and fit in FPGAs [30].

They were successful in generating new circuits to match the eASIC post-synthesis trends. However, the largest size of circuit that they can generated is small. This can be

improved by updating their collected data using a library of larger circuits, which is not practical since that phase is done manually based on the distributed datasheets. In addition they have decomposed their input circuits up to three levels of hierarchy and used the result as leaf modules to generate new circuits.

PartGen

PartGen is a generator based on GEN which can generate netlist benchmarks with an arbitrary size [34]. They categorized partial netlists into five different categories: regular combinational logic, irregular combiational logic (the bridge between large functional blocks), memory blocks, controller logic (consists of both combinational and sequential eg. cache controller), and interconnections. They propose a generator algorithm that connects different number of blocks from each categories to make a complete netlist. These new circuit are suitable for evaluating the partitioning algorithm phase of the CAD flow.

2.3.3 Mutation

Another method of generating circuits is the mutation approach, as presented by [18], [13] and [15]. Portions of the logic are modified, but structural characteristics such as path length, I/O, and wirelength are kept the same. This method is effective at generating a family of circuits similar to an existing circuit, but they lack the ability to generate new circuits of different size or of different structure. More details on each approach is as follows:

Harlow

A combinational circuit is an implementation of the truth table of a Boolean function which can also be represented by a binary decision tree. A reduced ordered binary decision diagrams(BDD) is the optimized version of a binary decision tree [36]. [18] models combinaitonal netlists by their BDDs. In this work netlists are classified based on their entropy

which is directly related to probability of their outputs being high. As a result two one-output functions with the same probability of being high will be in the same class. Generally an entropy invariant mutation is any modification to a function that results in a new function with the same entropy. Using these definitions new classes of BDDs are generated to measure the sensitivity of CAD algorithms among different classes of circuits.

Ghosh

Ghosh et al. develop a graph-based canonical representation for netlists of a combinational circuits which is a bipartite uni-directional graph based on the topological ordering of the wires and gates [13]. Using this model they introduce circuit perturbation and mutation. Perturbation is randomly eliminating a percentage of wires at each topological order level. Since perturbation might disconnect all the fan-in of a node and leave it floating, a process of adding wires to revive such nodes is required. This process is named mutation. Perturbation and mutation can greatly transform a netlist. In order to prevent this, they define classes of circuits based on the characteristics of their canonical representation and impose a limit on the type of perturbation and mutation that can be performed so that the new netlist is in the same class of the original one. By generating a number of equivalence class with many circuit members they evaluate the performance of CAD algorithms.

Grant

Grant and Lemieux in [14] generate new circuits by partially substituting a real circuit with its mutated version while preserving the post-synthesis characteristics. Such classes of partially different circuits are suitable for evaluating incremental place and route algorithms. In this work a netlist is modeled as a graph. Its nodes represent LUTs or FFs and its edges represent wires. In order to create a mutated version of a circuit, first the height of all nodes of the graph is calculated while assigning height *one* to the inputs and the outputs of FFs.

Second a height h is selected and a list of edges that connect the nodes of height h to height $h+1$ is created. Third, a percentage of edges of the list swap sinks with each other. Swaps may change switch the structure of the circuit because it is not a locality aware algorithm. For example, consider swapping wires of two independent buses. In order to prevent such swaps they limited the candidate nodes to swaps with each other to the ones that have a common ancestor within a certain depth.

Grant et al. in [16] generate new circuit by stitching partial circuits to each other while avoiding combinaitonal loops. Stitching together circuits may cause a combinational loop because of an input to output dependency. In this work such connections are avoided by formulating and solving it as a graph monophormism problem.

Grant and Lemieux in [15] combine their two previous works while introducing a new mutation algorithm which can scale a circuit to become larger or smaller than the original circuit. Scaling down is performed by removing a specific number of nodes of the graph. Scaling up is done by replicating a selected portion of the graph.

2.4 Summary

In this chapter we first introduced FPGAs and the reseach areas of CAD algorithm and architecture. Then we discussed the importance of benchmark circuits for an effective FPGA research and the fact that a lack of these sample circuits is imposing a barrier on the advancement of the field. Finally we categorized and discussed some of the previous work. The previous work are not good enough since they are generating circuits at netlist level or lower levels of abstraction. In addition their generators are tuned using outdated tools and the circuits that they generate are small.

Chapter 3

Circuit Analysis

3.1 Overview

In this chapter we explain how we analyze all the VTR 7.0 set of benchmark circuits in order to extract the necessary data to tune our circuit generators. The input to our analyzer is an RTL design from which a set of parameters, patterns and graphs will be extracted. In this chapter, first we define a terminology to be able to properly refer to different parts of an RTL design in study. Using the stated terminology, a model for different parts of RTL circuits is then introduced and analyzed.

3.2 Definitions and Terminology

In this section we define terms that are used in our circuit analysis and generator.

Code 3.1: A Sample Verilog Circuit

```
1 module power(clock, reset, pow, X, result, DONE);
2       input [7:0] pow, X;
3       input clock, reset;
4       output [7:0] result;
5       output DONE;
6       wire [1:0] state;
7       wire Z;
8
```

```
9       StateMachine(clock, reset, Z, state, DONE);
10      DataPath(clock, state, X, pow, result, Z);
11 endmodule
12
13 module DataPath(clock, state, X, pow, result, Z, W);
14   input clock;
15   input [1:0] state;
16   input [7:0] pow, X;
17   output reg [7:0]result;
18   output reg Z;
19   output wire W;
20   reg [7:0] cnt;
21   wire Y;
22   assign Y = 3'b100;
23   assign W = Y * result + 2'b11;
24   always @(posedge clock)
25     if (state == 0) begin
26       result <= 1;
27       cnt <= pow;
28     end else if(state == 1) begin
29       result <= X * result;
30       cnt <= cnt - 1;
31     end
32   always @(*)
33     if (cnt <= 0)
34       Z = 1;
35     else
36       Z = 0;
37 endmodule
38
39 module DataPath(clock, state, X, pow, result, Z);
40   input clock, reset;
41   input [1:0] state;
42   input [7:0] pow, X;
43   output reg [7:0]result;
44   output reg Z;
45   reg [7:0] cnt;
46
47   always @(posedge clock)
48     if (state == 0) begin
49       result <= 1;
50       cnt <= pow;
51     end else if(state == 1) begin
52       result <= X * result;
53       cnt <= cnt - 1;
54     end
55
56   always @(*)
57     if (CNT <= 0)
58       Z = 1;
59     else
```

```
60      Z = 0;
61  endmodule
62
63  module StateMachine(clock, reset, Z, state, DONE);
64    input clock, reset, Z;
65    output reg DONE;
66    output reg [1:0] state;
67    reg [1:0] next_state;
68
69    always @(posedge clock, negedge reset)
70      if (reset == 0) begin
71        state <= 0;
72        DONE <= 0;
73      end else begin
74        case (state)
75          0: next_state <= 1;
76          1: if (Z == 1) begin
77               next_state <= 2;
78               DONE <= 1;
79             end
80          default: next_state <= 0;
81        endcase
82        state <= next_state;
83    end
84  endmodule
```

- Module and Instance: The building blocks of Verilog circuits are modules. Modules are connected to each other via instantiation. The module that is not instantiated in any modules of the circuit is called the *top module*. A Verilog circuit must have a top module and possibly one or more lower level modules. For example, Code 3.1 has three modules. Module *power* is the top module which has two instances, *DataPath* and *StateMachine*.

- Operands: There are two types of operands in a Verilog circuit; ports and local operands.

  - Port: Ports are the means that a module interfaces with other modules and its surroundings. A port can be an input, output, or inout. Inputs are used to set values and outputs are to read values of a modules from the outside. Inout ports

can be used to do both. For example in Code 3.1, module *power* has six ports, four inputs and two outputs.

  - Local operand: To describe a circuit using Verilog, it is often necessary to have local operands in addition to the ports. Local operands can be used inside of a module but they cannot be accessed or modified by other modules. For example in Code 3.1, module *power* has two local variables called $state$ and Z.

- Wire and register: Ports and local variables of a verilog circuit can be defined as a wire or a register. A Wire provides the connection between two points in the circuit, as a result it does not have a storage ability. Registers are implemented by flip-flops and can store a value if there are used in a sequential process (defined in the next item) or they are implemented as wires if they are used in a combinational process (defined in the next item). For example in Code 3.1, module *power* has a local wire called $state$ which is used to make a connection between modules *DataPath* and *StateMachine* and module *DataPath* has an output register called *result* which is used to store a value.

- Process and Sensitivity List: There are two types of processes; initial blocks and always blocks. Initial block are used for writing test benches and they are not synthesisable. In addition, the code inside of an initial block executes only once at the beginning of a test bench. On the contrary, Always blocks can be implemented by hardware and they are executed whenever one of the items in their following parenthesis changes. Such items are called the $sensitivity\ list$ of an always block. The following items are the instructions that can be used in a process:

  - Assignment statement or statement: An Assignment statement is simply assigns an expression to another expression inside a process. For example in line 26 of

Code 3.1, the right hand side (RHS) expression which is a multiplication of two operands, $X * result$, is being assigned to the left hand side (LHS) expression which is an operand, $result$. To simplify this discussion we use the term $statement$ instead of $assignmentstatement$. There are two types of statements, blocking and nonblocking:

  - Blocking statements (LHS expression = RHS expression): All $blockingstatements$ in an always block are executed sequentially. In another words the execution of the next statements are blocked until the execution of current $blockingstatement$ is finished.
  - Non-blocking statements (LHS expression <= RHS expression): All $non-blockingstatement$ are executed without blocking the other statements as a result all $nonblockingstatements$ are executed in parallel.

- Control statements
  - if-statement, conditional-statement, and else-statement: For example line 31 of Code 3.1 is an $if-statement$ and line 33 is an $else-statement$.
  - case-statement and case-item: For example a $case-statement$ starts at line 48 of Code 3.1 and it has two $case-items$ in lines 49 and 50.

- Assignment: An $assignment$ is used to assign a LHS expression to a RHS expression outside of an always block ($assign\ LHS\ expression = RHS\ exprssion$).

3.3 Circuit Analysis

3.3.1 Module Topology Graphs

As already described in Section 3.1 every RTL design is built of one or more modules, connected to each other in a hierarchical fashion. The topology of modules connections can be

modeled by a directed acyclic graph (DAG) in which nodes represent modules and edges represent instantiations. For example there is an edge from *node a* to *node b* if *module b* was instantiated in *module a*. Figure 3.1 shows the module topology of the *bgm* circuit (one of VTR benchmark circuits) which has 15 modules. The name of its root is *bgm* which is the top module of *bgm* circuit. There are three outgoing edges from the *bgm* node to three other nodes which are the modules that are instantiated in the *bgm* module. Our circuit analyzer reads all VTR benchmark circuits and generates their corresponding module topology graph and stores them in a pool called the pool of module topology graphs.

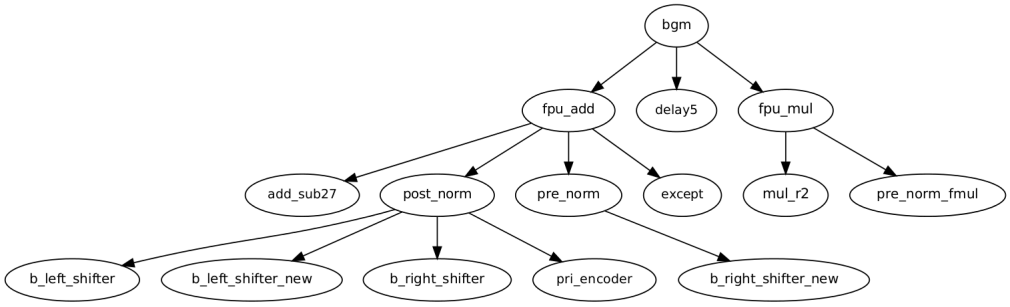

Figure 3.1: Module topology of bgm circuit.

3.3.2 Basic Enumerated Features

A typical RTL module is made up of a number of assignments, processes, blocking/non-blocking statements, case-conditions, and if-conditions. Table 3.2 reports the number of all aforementioned features of VTR benchmark circuits. Its first column is the name of the benchmark circuit and the second column is number of modules that each circuit contains. The numbers reported in rest of the columns are the cumulative number of the feature among all modules for each benchmark circuit. For example as it is shown in Table 3.1, *bgm* circuit has 15 modules and the sum of number of processes among those 15 modules are 119. Another set of data that can be collected from a circuit is the number of its operands, their types and whether they are declared locally or as a port. Table 3.3 reports

this information for the top module in the VTR benchmark circuits. It worth mentioning that the reported number of inputs and outputs might be different from what a CAD tool reports. The reason is our study is based on the circuit at RTL level but the CAD reports the number after the complete CAD flow and several possible optimizations.

Table 3.1: Number of assignments, processes, assignments, blocking/nonblocking statements, case-conditions, and if-conditions for each module of *bgm* circuit

Module Name	#assignment	#process	#blocking	#nonblocking	#case	#if
add_sub27	1	0	0	0	0	0
b_left_shifter	0	1	49	0	1	0
b_left_shifter_new	0	1	57	0	1	0
b_right_shifter	0	1	49	0	1	0
b_right_shifter_new	0	1	28	0	1	0
bgm_top	1	0	0	0	0	0
delay5	1	1	0	1	0	0
except	4	24	0	24	0	0
fpu_add	14	30	0	38	0	0
fpu_mul	24	33	1	40	0	0
mul_r2	0	2	0	2	0	0
post_norm	109	2	8	0	0	0
pre_norm	29	15	44	12	0	0
pre_norm_fmul	28	7	4	6	0	0
pri_encoder	1	1	49	0	0	49
sum	212	119	289	123	4	49

3.3.3 Process Patterns

As already mentioned an RTL module consists of one or more processes. In this study we have limited our focus to synthesizable processes. In another words only always blocks that fit in one of the following categories are analyzed and later generated by our RTL circuit generator. There are three categories of synthesizable processes; purely combinational,

Table 3.2: Number of assignments, processes, assignments, blocking/nonblocking statements, case-conditions, and if-conditions of VTR benchmark circuits

Circuit	#module	#assignment	#process	#blocking	#nonblocking	#case	#if
bgm	15	212	119	289	123	4	49
blob_merge	2	3	2	61	379	2	183
boundtop	13	40	22	465	341	7	166
ch_intrinsics	2	1	4	7	36	4	4
diffeq1	1	1	1	0	16	0	3
diffeq2	1	1	1	0	6	0	2
LU32PEEng	29	313	58	500	826	14	335
LU64PEEng	29	441	58	564	890	14	367
LU8PEEng	29	217	58	452	778	14	311
mcml	36	240	121	2593	3053	20	297
mkDelayWorker32B	16	760	87	106	430	17	280
mkPktMerge	7	76	30	2	108	0	106
mkSMAdapter4B	8	576	51	100	252	17	154
or1200	16	126	78	331	272	38	124
raygentop	15	34	23	595	238	12	145
sha	1	11	4	0	1604	3	13
spree	29	154	16	21	425	14	18
stereovision0	25	58	27	0	851	4	66
stereovision1	16	11	24	61	643	9	50
stereovision2	24	9	26	0	367	5	28
stereovision3	1	1	5	135	274	2	37

sequential, and sequential with asynchronous reset. [1]

- Combinational: Process p is a combinational process if it does not have any non-blocking statements and its sensitivity list includes all the signals that are on the right hand side of blocking statements. For example in Code 3.1, the always block in lines 30 to 34 is a combinational process.

- Sequential: Process p is a sequential process if it does not have any blocking state-

[1] Some tools might handle other patterns, but this is the simplified version that is commonly used by RTL designers.

Table 3.3: Number of assignments, processes, assignments, blocking/nonblocking statements, case-conditions, and if-conditions of VTR benchmark circuits

Circuit	#Output Regs	#Output Wires	#Inputs	#Local Regs	#Local Wires
bgm	0	192	1548	3283	5619
blob_merge	100	0	132	1684	70
boundtop	0	386	550	1821	2069
ch_intrinsics	98	32	99	231	0
diffeq1	96	0	162	97	32
diffeq2	96	0	66	0	32
LU32PEEng	0	306	342	13954	33997
LU64PEEng	0	306	342	27301	67319
LU8PEEng	0	306	342	3900	8953
mcml	990	0	1080	185309	154536
mkDelayWorker32B	0	8848	8208	5028	18622
mkPktMerge	0	1092	2177	278	3810
mkSMAdapter4B	0	1640	1584	2279	4290
or1200	0	394	388	0	909
raygentop	0	2745	2295	986	1715
sha	32	4	38	879	423
spree	0	864	3564	620	4263
stereovision0	2163	1974	3549	20051	8674
stereovision1	0	1160	1064	10720	11185
stereovision2	462	3360	3129	20136	10054
stereovision3	29	1	23	128	0

ment [2] and it is sensitive to edge of only one signal. This signal represents clock and is should not be used as an operand at any statements in the body of p. For example in Code 3.1, the always block in lines 21 to 28 is a sequential always block.

- Sequential with asynchronise reset: Process p is of the third category if it does not have any blocking statements and it is sensitive to the rising or falling edge of exactly

[2]Processes can have mixed blocking and non-blocking statements. However none of the processes in our input set circuits consists of a mix blocking and non-blocking statements. As a result we assume that all statements in a process are either blocking or non-blocking.

two signals. One of these signals is the clock signal and another signal represents reset. The reset signal must be used as a condition for an if-statement. For example in Code 3.1, the always block in lines 43 to 57 is a sequential with asynchronise reset always block.

Table 3.4: Instructions and corresponding keywords

Instruction Type	Instruction Example	Keyword
if statement	if (reset == 1)	conditional
non-blocking statement	q = 4'b0;	nonblocking
blocking statement	out <= q + 1;	blocking
case statement	case state:	case
case item	1:	caseitem
sequential block	begin	seqblock

In Table 3.2 we report the number of processes and different instructions that are used inside a process for each VTR benchmark circuit. Although this data is useful for deciding the new the numbers for a new RTL circuit, it is not enough information for our circuit generator to come up with the RTL code for each new process. In addition the hardware that an RTL circuit will synthesize to is dependent on type and sequence of statements in its processes. It possible to combine or separate the statements and modify the number of processes in a module without affecting the module functionally or its synthesized hardware. To model the processes of a RTL circuits we came up with an approach that considers type and sequence of statements in each process and the operands and operators in expressions. We developed an RTL parser on top of Invio which converts the body of each process to a sequence of a keywords and numbers. Keywords represent the statement type and numbers represent the nesting level. For example, the obtained sequence from Code 3.2 is: *0:conditional - 1:seqblock - 2:nonblocking - 2:nonblocking - 1:seqblock - 2:nonblocking - 2:nonblocking* and the obtained sequence from Code 3.3 is *0:conditional - 1:seqblock -*

2:nonblocking - 2:nonblocking - 1:seqblock - 2:nonblocking - 0:nonblocking

Using a number to specify the nesting level of instruction as a part of each element of the sequence is necessary. To elaborate, consider Code 3.2 and Code 3.3; the only difference between these examples is the nesting level of the last non-blocking statement. In Code 3.2 the last non-blocking statement is nested under *else* but in Code 3.2, it is simply a part of the main body. As a result the nesting level of last nonblocking assignment in code 3.2 is 2 but in Code 3.3 is 0.

Code 3.2: A Process Pattern Sample 1

```
1  always @(posedge clock) begin
2    if (reset == 1)   //0:conditional
3    begin   //1:seq block
4       q = 4'b0; //2:nonblocking
5       out = 1'b0; //2:nonblocking
6     end else begin  //1:seq block
7       q = 4'b0101; //2:nonblocking
8       out = q + 1; //2:nonblocking
9    end
10 end
```

Code 3.3: A Process Pattern Sample 2

```
1  always @(posedge clock) begin
2  if (reset == 1)  //0:conditional
3    begin   //1:seqblock
4      q = 4'b0; //2:nonblocing
5      out = 1'b0; //2:nonblocing
6    end else begin  //1:seqblock
7      q = 4'b1010;  //2:nonblocking
8    end
9    out = q + 1;  //0:nonblocking
10 end
```

To analyze processes in existing benchmark circuits we categorized each them into the three type of synthesizable processes (combinaitonal, sequential and sequential with asynchronise reset) and converted their instruction to the aforementioned sequence. Interestingly our experiments shows that most processes convert to the same sequence. This suggests

that RTL designers tend to repeatedly use the same patterns while designing RTL circuits. Table 3.7 shows the result of studying 815 processes. The third column of this table is the *processpattern*, the second column states the process type that the pattern is used and the first column is a percentage that shows how common each pattern is.

Some process patterns are more common than others. The naive approach to calculate how common each process pattern is (first column of Table 3.7), to divide how many times is it repeated by the total number of processes. The outcome of this approach will be biased to the module that has the highest number of processes. For example, suppose we study only two sequential modules, one with 99 processes of pattern $p1$ and with one has one process of pattern $p2$. The table reporting this case study is Table 3.5, biased to the first circuit. In order to avoid this issue instead of counting each repetition of a pattern as 1 it is counted as $\frac{1}{number\ of\ processes\ in\ module}$. With this modification the reported data of our case study will become Table 3.6.

Table 3.5: Calculation results of how common different process patterns of two sequential modules (one with 99 processes of pattern $p1$ and the other with one process of pattern $p2$) based on the naive approach

Percentage	Process Category	Pattern
99%	Sequential	P1
1%	Sequential	P2

Table 3.6: Calculation results of how common different process patterns of two sequential modules (one with 99 processes of pattern $p1$ and the other with one process of pattern $p2$) based on the $\frac{1}{number\ of\ processes\ in\ module}$ approach

Percentage	Process Category	Pattern
50%	Sequential	P1
50%	Sequential	P2

Table 3.7: Pattern of Processes

Percentage	Process Category	Pattern
13.05%	Sequential	$0 : nonblockingassign$
9.25%	Sequential	$0 : conditional - 1 : nonblockingassign - 0 : conditional - 1 : nonblockingassign -$ $0 : conditional - 1 : nonblockingassign - 0 : conditional - 1 : nonblockingassign$
4.31%	Sequential	$0 : seqblock - 1 : conditional - 2 : seqblock - 3 : nonblockingassign$
4.18%	Combinational	$0 : seqblock - 1 : case - 2 : caseitem - 3 : blockingassign -$ $2 : caseitem - 3 : blockingassign - 2 : caseitem - 3 : blockingassign -$ $2 : caseitem - 3 : blockingassign$
3.67%	Sequential	$0seqblock - 1 : nonblockingassign - 1 : nonblockingassign - 1 : nonblockingassign$
3.04%	Sequential	$0 : conditional - 1 : nonblockingassign - 0 : conditional - 1 : nonblockingassign$ $-0 : conditional - 1 : nonblockingassign - 0 : conditional - 1 : nonblockingassign$
2.66%	Sequential	$0 : seqblock - 1 : conditional - 2 : nonblockingassign - 1 : conditional$ $-2 : nonblockingassign - 1 : conditional - 2 : nonblockingassign - 1 : conditional$ $-2 : nonblockingassign$
2.40%	Sequential with Sync Reset	$seqblock - 1 : conditional - 2 : nonblockingassign - 2 : conditional$ $-3 : nonblockingassign$
1.64%	Sequential	$0 : seqblock - 1 : nonblockingassign - 1 : nonblockingassign$
1.52%	Combinational	0:case - 1:caseitem- 2:blockingassign - 1:caseitem- 2:blockingassign - 1:caseitem- 2:blockingassign - 1:caseitem- 2:blockingassign - 1:caseitem- 2:blockingassign)
54.24%	-	$otherpatterns$

3.3.4 Expression Patterns

Assignments and statements have a RHS and a LHS expression. These expressions can be modeled by the pre-order traversal of their corresponding expression tree. For example the expression pattern of the right hand side of the first blocking statement in Code 3.4 is:

binary concat constant signalref range constant constant constant

Results for the right hand side of non-blocking statements are reported in Table 3.8. Similarly to process patterns, if we report the percentage according to the number of repetitions of each expression pattern the outcome will be biased to the circuit with the highest number of expressions. In order to avoid this issue we count each repetition of an expression pattern as $\frac{1}{number\ of\ expressions\ in the\ module}$. Table 3.8 shows the

Code 3.4: Sample Verilog code 3.

```
1 always @(*) begin
```

```
2     q <= {1'b1, c[2:0] >> 2}; //0)blocking
3     out <= q + 1;  //0)blocking
4 end
```

Table 3.8: Pattern of Expressions

Percentage	Pattern	Example
35.00%	constant	a = 1'b1;
34.08%	signalref	a = c;
9.10%	binary constant signalref	a = c + 1;
4.55%	signalref range constant constant	a = c[15:8];
2.43%	binary concat constant signalref range constant constant constant	a = 1'b1, c[7:0] >>3'b100;
2.24%	concat signalref range constant constant constant	a[7:0] = c[15:8], 1'b0;
2.24%	concat signalref range constant constant constant	a = b[7:0] <<5'b00001;
1.23%	paramref	a = some_parameter;
1.12%	concat binary signalref range constant constant signalref range constant constant constant	a = d[46:23] - c[23:0], 23'b0;
1.02%	concat binary signalref range constant constant signalref signalref range constant constant	a = c[94:63] - d, e[62:0];
6.29%	other patterns	-

3.4 Data Flow Graph

3.4.1 Definition

We model the flow of data in assignments and statements of an RTL circuit by a directed acyclic graph (DAG). This method is similar to Hutton's approach in [20]. Each node of the graph represents a signal in the RTL circuit and each directed edge indicates a dependency. For example in Code 3.5 *a* is dependant on *b* and *c*. In the second assignment c is dependant on *d* and *b*. These data relationships are modeled by the DAG in Figure 3.2.

Code 3.5: A basic combinational Verilog code to demonstrate a basic DFG

```
1 always @(*) begin
2     a = b + c;
3     c = d - b;
```

```
4 end
```

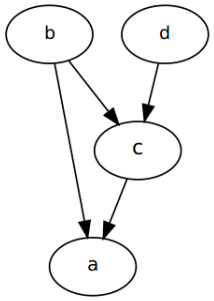

Figure 3.2: Data flow Graph of Verilog Code 3.5

It is important to note that if the statements in code 3.5 were nonblocking the data flow model would be different. In a process with nonblocking statements, all the left hand side signals will keep their old value from the previous clock cycle but the right hand side signals will get updated. In order to represent the data flow graph of a nonblocking statement, two nodes are necessary for demonstrating a signal that is being used both on the LHS and RHS, one to represent the old value and the other one to represent the new value. We can see such a sequential version in Code 3.6. Signal *c* has the dual usage description and it is represented by two different nodes in its data flow graph in Figure 3.3.

Code 3.6: Sample Verilog code to demonstrate a basic data flow graph

```
1 always @(posedge clock) begin
2    a <= b + c;  // a <= b + c_old;
3    c <= d - b;  // c_new <= d - b;
4 end
```

RTL circuits are usually include controlling statements such as if-conditions and case-conditions. For example in code 3.7, based on value of reset only parts of code will get executed. Hence the value of *a* and *c* are dependent on *reset* in addition to the operands of their LHS expressions. In order to keep our data flow graph simple, we assume that all

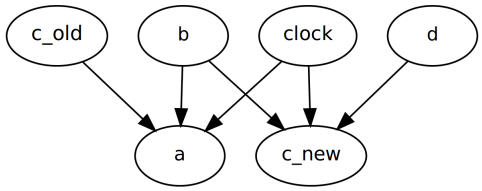

Figure 3.3: Data flow graph of Verilog code 3.6

instruction are executed regardless of the controlling instructions. Later on we show that this assumption does not materially affect our collected data.

Naming ports solely based on their name in the Verilog code is not sufficient since signal names are localized to their module scope, meaning two different signals can have the same name as long as they are defined in different modules. For example signal *e* in code 3.7 is an input port in the *module top* and a local wire in *module instance0*. To avoid merging these two signals we incorporate the module name into each node name. The data flow graph of code 3.7 is shown by Figure 3.4.

Code 3.7: Sample Verilog code to demonstrate a basic data flow graph

```
1  module top (b, e, d, a, c, f, reset);
2    input b, e, d, reset;
3    output a, c, f;
4    reg a, c;
5    always @(*) begin
6      if(reset) begin
7        a = e + 1;
8        c = e + 2;
9      end else begin
10       a = b + c;
11       c = d - b;
12     end
13   end
14   instance0 inst(a, c, f, reset);
15 endmodule
16
17 module instance0(x, y, z, reset);
18   input x, y, reset;
19   output z;
20   wire e;
21   e = 1'b1;
22   assign z = x * y >> e;
23 endmodule
```

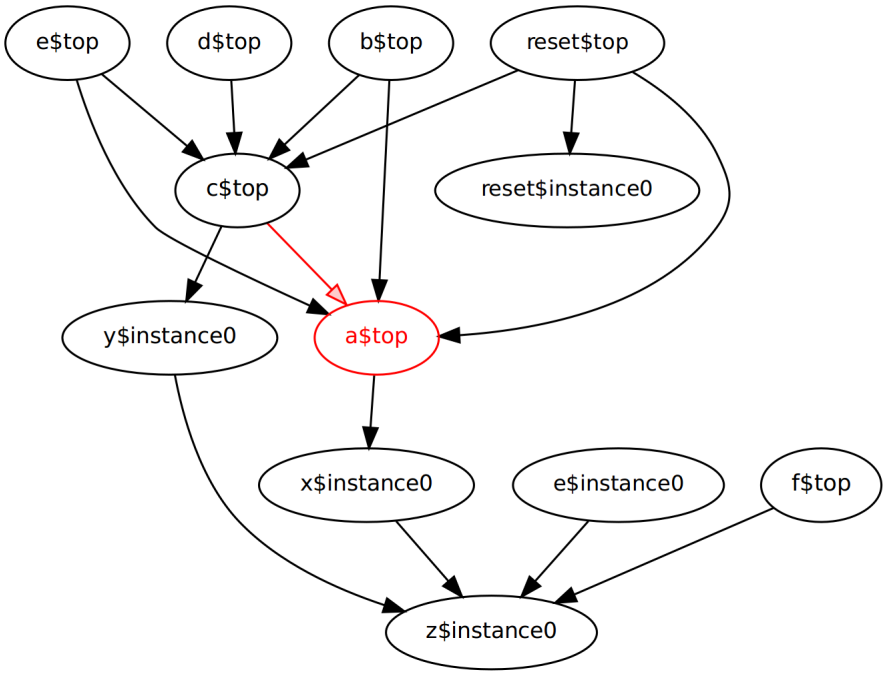

Figure 3.4: Data flow Graph of Verilog code 3.7

3.4.2 Analysis

The data flow graph of an RTL design contains a lot of information. However, these graphs tend to have many nodes and edges that makes identifying specific features difficult. Hence in this study we focus on two important features extracted from each graph. Table 3.9 shows the number of nodes and the longest path of the DFG of the VTR benchmark circuits.

3.5 Implementation

The circuit analysis phase of this study resulted in a software package, implemented in the Python programming language consisting of 3 files containing 1297 lines of code. One file of this software package is a python script to get advantage of an Industrial platform called

Table 3.9: longest path and number of nodes of DFG of all VTR benchmark circuits.

Circuit	#Nodes	Longest Path
bgm	471	25
blob-merge	49	19
boundtop	537	3
ch-intrinsics	19	3
diffeq1	3	1
diffeq2	3	1
LU8PEEng	630	25
LU32PEEng	725	25
LU64PEEng	854	25
mcml	6022	7
mkDelayWorker32B	898	9
mkPktMerge	160	6
mkSMAdapter4B	624	11
or1200	545	8
raygentop	494	2
sha	9	3
spree	361	4
stereovision0	780	2
stereovision1	471	3
stereovision2	324	2
stereovision3	9	1

Invio, from Invionics Inc. The Invio platform is an RTL processing engine, which allows designers to quickly parse, search, and modify RTL designs.

This software package gets a Verilog file and the name of its top module as the input, analyzes it, and generates the proper output. The input file needs to be self-contained i.e. it should contain all module definitions that have been instantiated except for common primitive modules. The modules that are defined but not instantiated will be ignored. Our

feature analysis technique is a combination of a parser and using the Invionics platform, called Invio.

3.6 Summary

In this chapter we represented our analysis techniques and results of our input set of circuits. We modeled circuits at RTL level and then profiled our input set of circuits using our model using different profiling schemes. Our first profiling scheme collects basic numeric information such as number of ports, assignments, processes, and different statements. Since it is not possible to generate a new RTL circuit merely based on these numerical information, we employed a second profiling scheme which collects sequences of statements in processes and sequences of operands and operators in expressions. Lastly we gathered graph-based information such as the topology of module instantiations and the DFG of an RTL circuit and its longest path.

Chapter 4

Clone Circuit Generation

4.1 Introduction

This describes the details of our clone circuit generator and how it uses the data gathered from analyzing one Verilog circuit to generate a clone circuit. Using our clone circuit generator researchers can reproduce a class of circuits similar to specific circuit that they are interested in. For example if a specific circuit does not work well on a proposed FPGA architecture, it is possible to investigate the reasons by using a class of similar circuits.

This chapter is organized as follows. In Section 4.2 first discusses the difference between a clone circuit and a random circuit. Then it provides a high level explanation of our clone circuit generation algorithm. Section 4.4 provides details on how we generate a clone module for each node of a given topology graph. Section 4.5 discuss how we connect the generated clone modules to create a complete clone circuit.

4.2 Overview

We first differentiate between clone circuit generator (this chapter) and random circuit generator (Chapter 5). The difference between a clone circuit and a random circuit is that

the clone circuit is generated based on the analysis of one specific circuit and is validated based on how close its post-synthesis characteristics are to those of the original circuit. In contrast, a random circuit is generated based on the analysis results of all available benchmark circuits and is validated based on the acceptable ranges for each of its post-synthesis characteristics. Figure4.1 shows the flow of these two different generators.

4.3 Motivation

Cloned circuits are interesting for

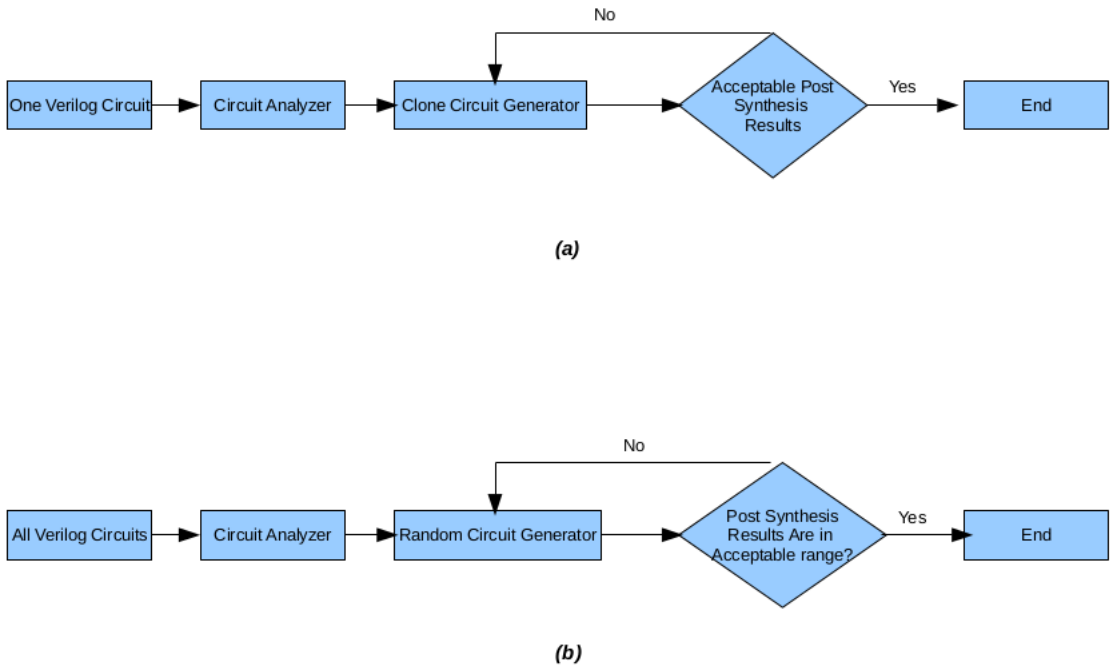

Figure 4.1: (a) Clone Circuit Generation Flow. (b) Random Circuit Generation Flow.

The input to our clone circuit generator is the set of analysis results for one Verilog circuit as we described in Chapter 4, which are:

- Parameters consisting of a module topology graph, similar to Figure 3.1, and the number of assignments, processes, and different operands for each node of that graph,

similar to Table 3.3.

- Process patterns, similar to Table 3.7
- Expression patterns, similar to Table 3.8.

The pseudocode shown in Code 4.1 is an overview of our clone circuit generator algorithm. It starts by generating as many single modules as the number of nodes of the input topology graph, *clone_module_generator*, then it connects them to each other to make them a circuit, *connect_module*. In Section 4.4 *clone_module_generator* is explained in detail and Section 4.5 presents the details of *connect_module*.

Code 4.1: Pseudocode of the clone circuit generator

```
1  clone_circuit_generator(){
2    for each node of the input topology graph:
3      clone_module_generator()
4      connect_module()
5  }
6
7  clone_module_generator(){
8    ports_and_variable_selection
9    assignment_generation
10   process_generation
11   operator_selection
12   operands_selection
13 }
```

4.4 Clone Module Generator

The following summarizes how we generate a clone module:

- Step one: Our module generator begins by determining the number and width of inputs, outputs, local register, and local wires.
- Step two: It choose the number of assignments and determines a left hand side (LHS) and a right hand side (RHS) expression pattern for each assignment.

- Step three: It determines how many processes this module is going to have, and then selects the patterns for each process. After that it completes each process pattern by choosing expression patterns.

- Step four: The generator then selects an operator for each *binary* or *unary* keyword in the selected expression patterns.

- Step five: It then chooses an operand for each *signalref* keyword in the selected expression patterns.

4.4.1 Step One - Ports and Variables Selection

Generating a clone module starts with selecting the same number and width as the original circuit for the inputs, output wires, output registers, local wires and local registers. Such as the example in Code 4.2.

Code 4.2: The progress of generating a clone module *M* after the first step.

```
1 module M (output_2, input_1, output_1, input_2, output_3, output_4,
2             reset, clock);
3   input reset, clock;
4   input [7:0]input_1;
5   input [3:0]input_2;
6   output [7:0]output_1, output_2;
7   output [3:0] output_3, output_4;
8   reg [3:0] output_3, output_4;
9 endmodule
```

4.4.2 Step Two - Assignment Generation

At this step as many assignments as the original circuit is generated. For each assignment an LHS and an RHS expression pattern is chosen from the given expression table, using the repetition percentages as weights. For example, based on Table 3.8, the repetition percentage of pattern *constant* is 35.00%, *signalref* is 34.08%, and *binary constant signalref* is 9.10%. As a result the chance of choosing pattern *constant* is $\frac{35.00}{9.10}$ times higher than

binary constant signalref and the chance of choosing *signalref* is $\frac{34.08}{9.10}$ times higher than *binary constant signalref*. The generator progress up to this step is shown in Code 4.3.

Code 4.3: The progress of generating module *M* after second step.

```
1  module M (output_2, input_1, output_1, input_2, output_3, output_4,
2              reset, clock);
3    input reset, clock;
4    input [7:0]input_1;
5    input [3:0]input_2;
6    output [7:0]output_1, output_2;
7    output [3:0] output_3, output_4;
8    reg [3:0] output_3, output_4;
9
10   assign signalref = binary constant signalref
11   assign signalref = concat signalref signalref
12 endmodule
```

4.4.3 Step Three - Process Generation

In the third step, the clone module generator chooses as many processes as the original circuit from the given process pattern table while using the repetition percetages as the weight for its random decision. For example suppose that it is determined that module *M* has one sequential process and the following pattern is chosen for this process:

$$0 : conditional, 1 : seqblock2 : nonblocking2 : nonblocking1 : seqblock2 : nonblocking \\ 0 : nonblocking \tag{4.1}$$

In this example the process pattern consists of one conditional and four non-blocking statements so five pairs of proper LHS and RHS expression patterns need to be selected. The generator progress up to this point is shown in Code 4.4. Note that the process category is decided simultaneously with the process pattern. In other words, for each process, one row of the Table 3.7 is chosen; The row indicates the process category as well as the process pattern.

Code 4.4: The progress of generating module *M* after process pattern selection

```
1  module M (output_2, input_1, output_1, input_2, output_3, output_4,
2              reset, clock);
3    input reset, clock;
4    input [7:0]input_1;
5    input [3:0]input_2;
6    output [7:0]output_1, output_2;
7    output [3:0] output_3, output_4;
8    reg [3:0] output_3, output_4;
9
10   assign signalref = binary constant signalref
11   assign signalref = concat signalref signalref
12
13   always @(posedge clock)
14   begin
15      conditional(conditional expression) begin
16        nonblocking: LHS expression <= RHS expression
17        nonblocking: LHS expression <= RHS expression
18      end else begin
19        nonblocking: LHS expression <= RHS expression
20        nonblocking: LHS expression <= RHS expression
21      end
22   end
23 endmodule
```

As shown in Code 4.4 the expression patterns used in the process statements are still unknown. The next step is to choose a pattern for each of these expressions based on the data that is collected for each expression type such as Table 3.8. This leads to a code such as Code 5.3. As we described in Chapter 2, if the process category is sequential with an asynchronous reset then the process pattern is always an if-else statement with reset as the condition. This means that in these cases the conditional expression pattern is always $signalref$.

Code 4.5: The progress of generating module *M* after step 3.

```
1  module M (output_2, input_1, output_1, input_2, output_3, output_4,
2              reset, clock);
3    input reset, clock;
4    input [7:0]input_1;
5    input [3:0]input_2;
6    output [7:0]output_1, output_2;
7    output [3:0] output_3, output_4;
8    reg [3:0] output_3, output_4;
9
```

```
10   assign signalref = binary constant signalref
11   assign signalref = concat signalref signalref
12   always @(posedge clock)
13   begin
14     conditional(unary signalref)
15     begin
16       signalref <= constant
17       signalref <= constant
18     end else begin
19       signalref <= signalref range constant constant
20       signalref <= binary signalref signalref
21     end
22   end
23 endmodule
```

4.4.4 Step Four - Operator Selection

An operator, such as +, -, %, *, etc needs to be chosen for every *binary* and *unary* keyword in the selected expression patterns. The selection can be done randomly or based on a distribution. Alternatively, it would be possible to modify the expression pattern to include the operators. In another words, while the post order tree traversal is being done we store exactly which operator is being used. After this modification, the expression pattern percentage also shows how often a specific type of logic (add, sub, etc) have been used. For example an expression pattern will look like *add constant signalref* instead of *binary constant signalref*. All these three approaches are included in our generator implementation. The default approach is the latter, storing the operator type, unless modified by user.

4.4.5 Step Five - Operands Selection

At this point, the skeleton of the circuit is generated and all inputs, output wires, output registers, local wires and local registers are declared. In this step we choose an operand from the ports or local variables for each *signalref* keyword. A naive approach to operand selection would be to simply select operands randomly for each *signalref* keyword. This

method will not work well, because not all resources match all *signalref*s (e.g. the LHS of an assignment must be a *wire*). Using an unsuitable operand causes several issues. The following is a list of possible issues and how we resolve each of them:

- Invalid operand type: When a wire is used as the LHS in a process or a register is used on the LHS of an assignment.

  This is easily prevented by making selections out of the proper pools of operands. We define a proper pool for each *signalref* based on whether is being used in an assignment or a statement, also whether its on the LHS or RHS. Given these points, the proper pools used by our generator are as follows:

  - LHS of assignments pool = all wires = [local wires + output wires]
  - LHS of statements pool = all registers = [output registers + local registers]
  - RHS of assignments pool = RHS of statement pool = wires + registers = all operands = [local wires + local registers + output wires + output registers + inputs]

- Combinational loops: If the DFG of the generated circuit is not acyclic it means at least one combinational loop exists.

  To address this issue, a greedy algorithm could be used to build the DFG after selecting all operands and checking it for cycles. If the DFG is not acylic, the algorithm could redo all the operand selections. This solution will find a valid selection but it is not runtime efficient.

  Instead, our solution is to incrementally create the DFG and make sure it remains acyclic after each update. In other words, all operands of one assignment or statement are selected and the DFG is updated by adding the necessary edges and nodes. If the updated DFG is acyclic we move on to the next operand selection, otherwise we redo

the current operand selection. The most time consuming part of this algorithm is checking for cycles after each update by running a depth first search (DFS) algorithm on each node to check if any back edges exist. However, if the updated DFG is cyclic this means that at least one of the edges from the cycle was added in the previous update. Therefore, we only need to run the DFS from the nodes for which the number of incident edges were modified in the previous update of the DFG. Code 4.6 shows a pseudocode of this algorithm.

Code 4.6: Pseudocode of preventing combinational loop algorithm.

```
1  for all assignments and statements:{
2        done = 0
3        while(done == 0){
4              select operands from the proper pool
5              update DFG
6              if DFG is acyclic:
7                    done = 1
8              else:
9                    revert DFG
10       }
11 }
```

- Optimizable code: We are generating code based on patterns and statistics rather than implementing a functionality like an RTL designer. In addition if a piece of generated code does have any effects on outputs, it will be optimized away by the compiler. In addition typically a real RTL circuit does not include a large portion of optimizable code. As a result we need to avoid generating optimizable code.

  A piece of Verilog code is optimizable if it consists of an operand that is assigned a value but it has never been used on the RHS as a driver. This occurs when a local operand is used on the LHS but never on the RHS. Similarly, when a local operand is used on the RHS as a driver while it has never received a value. We use two constraints to guarantee that all parts of the generated code will synthesize to a piece of hardware. At each iteration of the operands selection (Code 4.6) only those operands

of an RHS pool can be selected to be used on the RHS that are inputs or have already been used on the LHS and received a value. In addition, each of the operands in the RHS pool is given an initial weight to be used as the probability in the random RHS selection. Each time an operand is selected to be used on the RHS, its weight is divided by two to minimize the possibility of selecting one operand many times and leave some operands unused.

- Inferred Latches: A latch is inferred when in a combinational process:
  - case-statements and case-items:
    - There must be a case-item for all possible values of a case-condition. In other words, the number of case-items must be equal to the $2^{case-condition\ width}$ otherwise there must be a $default$ case-item.
    - Any operands that are assigned in one case-item must be assigned in all case-items.
  - if and else-statements:
    - There must be an else-statement for each if-statement.
    - Any operands that are assigned in an if-statement must be assigned in the else-statement.

Code 4.7: The progress of generating module *M* after step 5.

```
1 module M (output_2, input_1, output_1, input_2, output_3, output_4,
2 reset, clock);
3   input reset, clock;
4   input [7:0]input_1;
5   input [3:0]input_2;
6   output [7:0]output_1, output_2;
7   output [3:0] output_3, output_4;
8   reg [3:0] output_3, output_4;
9   assign output_2 = input_1 ^ 8'b01010101;
```

```
10   assign output_1 = output_3 & output_4;
11   always @(posedge clock)
12   begin
13     if(! reset)
14     begin
15       output_3 <= 4'b0;
16       output_4 <= 4'b0;
17     end else begin
18       output_3 <= output_2[3:0];
19       output_4 <= output_3 + input_2;
20     end
21   end
22 endmodule
```

4.5 Connecting Modules

After generating as many modules as the number of nodes of the topology graph using the aforementioned steps, we need to pair each with a node of the topology graph. Then we make the connections between them based on the directed edges in another words for each directed edge we add an instance of the sink module to the source module. The important points to keep in mind while pairing modules with nodes is as follows:

- A module containing a sequential process cannot be a successor of a module without a clock port.
- A module without a reset port cannot be a predecessor of a module with a reset port.

In order to come up with a proper paring, we make an ordered list of generated modules starting from sequential modules with asynchronous reset then sequential modules and finally the combinational modules. Finally we traverse the topology graph in level order fashion and pair each node with the ordered list satrting from the begining. The pseudocode of this algorithm is shown in Code 4.8.

Code 4.8: Pseudocode of the pairing algorithm.

```
1 Connecting_modules(single_modules_to_connect, topology_graph){
2   for m in single_modules_to_connect:{
```

```
3     if m is sequential with reset:
4       add m to the end of ordered_list_1
5     if m is sequential without reset:
6       add m to the end of ordered_list_2
7     if m is combinational:
8       add m to the end of ordered_list_3
9   }
10
11  ordered_list_all = ordered_list_1 + ordered_list_2 + ordered_list_3
12
13  for n in level-order traversal of topology_graph:{
14    pair n with the front of ordered_list_all
15    remove the front of list_all
16  }
17 }
```

4.6 Implementation

We implemented the described clone circuit generation algorithm in the Python programming language consisting of 21 files containing 8001 lines of code. This software package gets the outputs of the circuit analysis package that is described in Chapter 3, which are a set of parameters, patterns and constraints tunes the clone generator algorithm and to outputs a clone of the input circuit.

4.7 Summary

In this chapter we explained our algorithm for generating a clone circuit using the analysis data of an input circuit. Our clone generation algorithm generate as many modules as the input circuit. After that it connects them to each other based on the topology graph of the input circuit. A clone module is generated in five steps. First the number and width of inputs, outputs, local register, and local wires is determined. Then the number of assignments and an LHS and an RHS expression pattern for each assignment is chosen. After that the number of processes and their patterns is determined. Finally the operators and operands are selected for each chosen expression pattern.

Chapter 5

Random Circuit Generation

5.1 Introduction

In this chapter we describe how we generate a random circuit based on the analysis results of all available benchmark circuits. Our random circuit generator can be used to generate many different circuits. FPGA researchers can use such a variety of circuits to evaluate the impact of their CAD algorithm or FPGA architectural innovations. This chapter is organized as follows. First we present an overview of our random circuit generation algorithm. Section 5.2 provides details on how we generate one random module. Finally, in Section 5.3, the implementation details are discussed.

5.1.1 Random Generator Overview

We describe a random circuit generation algorithm that takes the analysis results (process and expression patterns and distribution of the number of assignments and the number of processes in modules) of all available benchmark circuits and generates a new circuit which has post-synthesis characteristics in a valid range. The validation range is defined based on the minimum and maximum of a post-synthesis characteristics of the input circuits.

For example, the minimum critical path of all VTR benchmark circuits is 2.67 ns and the maximum critical path of all VTR benchmark circuits is 115.29 ns. This means if the critical path of the new circuit is more than 115.29 ns or less than 2.67 ns then we discard it and repeat the algorithm until we generate a random circuit that fit to the specified range. This flow is shown in Figure 4.1.

The input to our random circuit generator is the analysis results of a set of Verilog circuits as we described in Chapter 4 which are:

- A pool of module topology graphs similar to Figure 3.1 and the number of assignments, processes, and different operands for each node of that graph similar to Table 3.3.
- All process patterns that were used in the input circuits, similar to Table 3.7
- All expression patterns that were used in the input circuits, similar to Table 3.8.

The pseudocode shown in Code 4.1 is an overview of our random circuit generator algorithm. It starts with choosing a topology graph then it generates as many random modules as the number of nodes of the chosen topology graph, *random_module_generator*. It then connects them to each other to make them a circuit, called *connect_module*. In Section 4.4 details of *clone_module_generator* is explained in detail. Section 4.5 present the details of *connect_module*.

Code 5.1: Pseudocode of the random circuit generator

```
1  random_circuit_generator{
2    topology_graph = random(pool of available topology graphs)
3    for each node of topology_graph:
4         random_module_generator()
5         connect_module()
6  }
7
8  random_module_generator(){
9    random_assignment_generation()
10   random_process_generation()
```

```
11   random_ports_and_variables_selection()
12   random_operator_selection()
13   random_operand_selection()
14   random_operand_width_selection()
15 }
```

5.2 Random Module Generator

Our random module generator begins by selecting a topology graph. This graph can be user specified or chosen randomly from the pool of topology graphs of available benchmarks described in Section 3.3.1. After selecting the topology graph, our generator creates one module for each node of the chosen topology graph.

To generate one module a naive approach is to first choose the number of ports and local variables. This imposes a limit on the pattern and number of processes or assignments can be in the module. For example, if we begin by determining that the number of output ports is three wires and the number of local wires is zero, then the number of assignments must be three. Choosing more than three assignments results in an output wire with multiple drivers and choosing fewer than three outputs results in at least one wire without a driver. As a result our random module generator after choosing a topology graph, determines the number of processes and expressions and a pattern for each. Then it chooses the necessary operands.

A simplified overview of our approach for generating a module is as follows. The pseudocode of our module generator is shown in Code 5.1 and the details of each step are discussed in Section 4.4.

- Step one: Our random module generator begins by determining the number of assignments. It then decides a LHS and a RHS expression pattern for each assignment.
- Step two: It then determines how many processes this module is going to have, and then selects a pattern for each process. After that it completes each process pattern

by choosing expression patterns based on the statement type (conditional, blocking, nonblocking, etc).

- Step three: It determines the number of input and output operands as well as local wires and registers.
- Step four: Selects an operator for each *binary* or *unary* keyword in the selected expression patterns.
- Step five: Chooses an operand for each *signalref* keyword in the selected expression patterns.
- Step Six: Finally it specifies the width of each operand.

5.2.1 Step one - Assignment Generation

Generating a module starts with determining how many assignments it contains using the distribution of number of assignments in the input set of circuits. However, this decision can also be made by picking a random number from a valid range or simply using a user defined number. After that we take the same approach as the second step of the clone circuit generator to choose a RHS and LHS expression pattern for each assignment as described in Subsection 4.4.2. In our example we decided that module *M* has two assignments. The generator progress up to this step is shown in Code 5.2.

Code 5.2: The progress of generating module *M* after the first step.

```
1 module M();
2   assign signalref = binary constant signalref
3   assign signalref = concat signalref signalref
4 endmodule
```

5.2.2 Step Two - Process Generation

In the second step, our generator determines the number of processes for the module then chooses a pattern for each process based on the distribution of the number of processes in the input set of circuits. In our example we determined that module *M* has one sequential process and with the following pattern:

$$0 : conditional1 : seqblock2 : nonblocking2 : nonblocking1 : seqblock \\ 2 : nonblocking0 : nonblocking. \tag{5.1}$$

Then we determined the number of processes we take the same approach as Subsection .This leads to a code such at what is shown in Code 5.3.

Code 5.3: The progress of generating module *M* after step 2.

```
1  module M();
2    assign signalref = binary constant signalref
3    assign signalref = concat signalref signalref
4    always @(posedge clock)
5    begin
6      if(unary signalref) begin
7        signalref <= constant
8        signalref <= constant
9      end else begin
10       signalref <= signalref range constant constant
11       signalref <= binary signalref signalref
12     end
13   end
14 endmodule
```

5.2.3 Step Three - Ports and Variables Selection

By the end of step two the skeleton of the random module is generated and there is enough information to choose the number of inputs, output wires, output registers, local wires and local registers. In order to simplify the explanation of this step, all of following rules and equations are based on the assumption that the LHS expression pattern is always has a $signalref$ pattern and the length of all operands is one bit. Later, we generalize our findings to cover all LHS patterns and bus operands.

The LHS of an assignment must be a wire. To avoid multiple drivers a wire must be used only once as a LHS operand of assignments. As a result the number of wires used on the LHS is the same as number of assignments of the module, as stated in Equation 5.2. In addition, a wire can be declared as a local variable, an output port or an input port. Since inputs cannot be used on the LHS we can write Equation 5.3. Equation 5.4 follows from Equation 5.2 and 5.3. Since, the number of assignments was determined in step 1, so this equation has two unknowns. For example, the sum of the total number of local wires and outputs wires of Code 5.3 is two.

$$\#assignments = \#left\_hand\_side\_wires \tag{5.2}$$

$$\#left\_hand\_side\_wires = \#local\_wires + \#output\_wires \tag{5.3}$$

$$\#assignments = \#local\_wires + \#output\_wires \tag{5.4}$$

The LHS of a statement must be a register. A register can be used as a LHS only in a single process otherwise it result in multiple drivers for the LHS. On the other hand, in a single process, a register can be used multiple times on the LHS as long as only one

of them executes according to the if-statements and case-statements.[1] It is also important to consider that according to specified conditions only some statements get executed. As a result in a sequential processes if a register gets a value only in some cases and not in others, it will get synthesized using an inferred latch to maintain its value when it is unspecified, which is usually not the intended outcome.

In order to avoid inferred latches we came up with an algorithm to label all the *signalref* keywords used on the LHS in a way that if a register was driven in one condition according to the *if* and *case* statements it will be driven in all others as well. In addition when two different *signalref* have the same label it means they will be mapped to the same register so the number of unique registers can be counted, as in Equation 5.5. Our example code is as shown in Code 5.4 after the labeling algorithm is performed. Since a register can be declared as both a local variable and an output port it is possible to write Equation 5.6 leading to Equation 5.7. The total number of local registers and output registers of Code 5.4 is two.

$$\#unique\_left\_hand\_side\_registers = \#registers \tag{5.5}$$

$$\#registers = \#local\_registers + \#output\_registers \tag{5.6}$$

$$\#unique\_left\_hand\_side\_registers = \#local\_registers + \#output\_registers \tag{5.7}$$

Code 5.4: The progress of generating module *M* after labeling each LHS *signalref*.

```
1  module M();
2    assign signalref = binary constant signalref
3    assign signalref = concat signalref signalref
4
5    always @(posedge clock)
6    begin
7      if(unary signalref)
8      begin
9        signalref_label1 <= constant
10       signalref_label2 <= constant
```

[1] Overriding registers is not considered in this study since academic synthesis tools do not support it.

```
11    end else begin
12      signalref_label1 <= signalref range constant constant
13      signalref_label2 <= binary signalref signalref
14    end
15  end
16 endmodule
```

Both wire and register operands can be used on the RHS of assignments and statements and each of them can be used multiple times, as shown in Equation 5.8.

$$\#wires + \#registers = \#inputs * A + \#local\_wires * B + \#local\_registers * C + \#outputs\_wires * D + \#outputs\_registers * E \tag{5.8}$$

Since we have three equations and ten unknown, there are more than one solution for the unknowns. Our generator randomly selects one of them. For example in sample Code 5.4 we have:

$2 = \#local\_wires + \#output\_wires$

$2 = \#local\_registers + \#output\_registers$

$6 = \#inputs * A + \#local\_wires * B + \#local\_registers * C + \#outputs\_wires * D + \#outputs\_registers * E$

One possible solution is as follows:

$\#inputs = 2$

$\#outputs\_wires = 2$

$\#outputs\_registers = 2$

$\#local\_wires = 0$

$\#local\_registers = 0$

$A = B = C = D = E = F = 1$

Also clock and reset signals need to be declared as inputs if they were used in the sensitivity list of any of the selected process patterns. The progress of generating module M by the

end of step three is shown in Code5.5.

Code 5.5: The progress of generating module *M* by the end of step 3.

```
1  module M(input_1, input_2, output_1, output_2, output_3, output_4,
2              clock, reset);
3    input reset, clock;
4    input input_1;
5    input input_2;
6    output output_1, output_2;
7    output output_3, output_4;
8    reg output_3, output_4;
9
10   assign signalref = binary constant signalref
11   assign signalref = concat signalref signalref
12
13   always @(posedge clock)
14       begin
15     if(unary signalref)
16     begin
17       signalref_label1 <= constant
18       signalref_label2 <= constant
19     end else begin
20       signalref_label1 <= signalref range constant constant
21       signalref_label2 <= binary signalref signalref
22     end
23   end
24 endmodule
```

5.2.4 Step Four - Operator Selection

In this step we select an operator for each *binary* or *unary* keyword of the selected expression patterns while using the same approach as the operator selection of our clone circuit generator, described in Subsection 4.4.4.

5.2.5 Step Five - Operands Selection

At this point, the skeleton of the circuit is generated and all necessary inputs, output, local wire, and local registers are declared. In this step we choose an operand from the ports or local variables for each *signalref* keyword based on the same approach as described in Subsection 4.4.5 of our clone circuit generator.

Code 5.6: The progress of generating module *M* after step 5.

```
1  module M(output_2, input_1, output_1, input_2, output_3, output_4,
2              reset, clock);
3    input reset, clock;
4    input input_1;
5    input input_2;
6    output output_1, output_2;
7    output output_3, output_4;
8    reg output_3, output_4;
9
10   assign output_2 = input_1 ^ constant;
11   assign output_1 = output_3 & output_4;
12
13   always @(posedge clock)
14   begin
15     if(! reset)
16     begin
17       output_3 <= constant;
18       output_4 <= constant;
19     end else begin
20       output_3 <= output_2;
21       output_4 <= output_3 + input_2;
22     end
23   end
24 endmodule
```

5.2.6 Step Six - Operand Width Selection

Typically, Verilog circuits deal with operands which are wider than a bit. To explain how our generator handles buses, first we describe a width selection algorithm that we have developed then we present the modifications to the ports and variables deceleration, and operand selection steps to make them support bus operands.

in this step we randomly choose a number as the width for the LHS of each assignment and statement from an acceptable range, defined by the user. Then we use these randomly selected numbers to indicate the width of RHS and LHS operands based on the selected operators in the expression pattern.

- The RHS operator is a unary: The LHS width is always one and the selected number determines the width of LHS operand.

- The RHS operator is a binary operator other than multiplication: The LHS and RHS width are the same and the selected number determines all of the width of all LHS operands.

- Concatenation and Multiplication: The LHS width is the sum of the width of the RHS operands as a result the randomly selected width must be more than the number of RHS operands. The width of RHS operands will be determined in a way that sum of them equals the randomly selected width. For example if the randomly selected width for the LHS is 10 and there are three RHS operands, any three numbers more than zero that sum up to 10 are acceptable as the RHS widths:

  $a[9:0] = b[1:0] * c[2:0] * d[4:0]$

  Our example code after the final step is shown in Code 5.7.

Code 5.7: The progress of generating module *M* after step 6.

```
1  module M (output_2, input_1, output_1, input_2, output_3, output_4
2              , reset, clock);
3    input reset, clock;
4    input [7:0]input_1;
5    input [3:0]input_2;
6    output [7:0]output_1, output_2;
7    output [3:0] output_3, output_4;
8    reg [3:0] output_3, output_4;
9
10   assign output_2 = input_1 ^ 8'b01010101;
11   assign output_1 = output_3 & output_4;
12
13   always @(posedge clock)
14   begin
15     if(! reset)
16     begin
17       output_3 <= 4'b0;
18       output_4 <= 4'b0;
19     end else begin
20       output_3 <= output_2[3:0];
21       output_4 <= output_3 + input_2;
22     end
23   end
24 endmodule
```

5.3 Implementation

We implemented the described random circuit generation algorithm in the Python programming language on top of our clone circuit generation, described in Chapter 4. We added 2 files containing 2055 lines of code to the previous package. This software package gets the outputs of analysis package of all available circuits, which are a set of patterns, acceptable ranges and constraints. The output of this package is a random circuit that has the post-synthesis characteristics which fall within the acceptable range.

5.4 Summary

In this chapter we altered our clone generator algorithm to generate random circuits using the analysis information extracted form all available RTL circuits.

Chapter 6

Results and Validation

6.1 Introduction

In this chapter we present our experimental methodology and results. This chapter is organized as follows. In Section 6.2 we describe our experimental methodology and present the results using our clone circuit generator. In Section 6.3 we define an acceptable random circuit and present results using our random circuit generator showing how using correlated input data will enhance the chance of generating an acceptable random circuit. In Section 6.4 we compare the circuits generated using our random circuit generator to earlier generators.

6.2 Clone Results and Validation

6.2.1 Overview of Experimentation Methodology

We generated a clone circuit for 19 Verilog circuits from the VTR set of benchmark circuits, shown in Table 6.2 and explained in detail in Subsection 6.2.3. We synthesized, packed, placed, and routed all clone and original circuits using VPR 6.0 on an architecture based on the Altera Stratix IV, characterized by logic cluster size of 10, 33 inputs per cluster, 6-input

LUTs, cluster input and output flexibilities of $F_c$in = 0.33 and $F_c$out = 0.33, respectively, and channels with segment length of 4, summarized in Table 6.1.

As shown in Figure 4.1, our clone circuit generator starts by analyzing an input Verilog circuit. It then repeatedly generates clone circuits until it generates an acceptable clone based on the distance of the clone's post-synthesis characteristics from those of the original. As explained in Chapter 4, a clone circuits is acceptable if its constrained post-synthesis characteristics are within a specific range. Note that the higher the number of constrained post-synthesis characteristics and the smaller acceptable range, the more attempts will typically be required to find an acceptable clone circuit.

Table 6.1: CAD Flow and the architecture setup used in clone generator and random generator experiments

CAD Flow Property	**Value**
CAD Flow	VPR 6.0
Target FPGA	Similar to Altera Stratix IV
LUT Size	6
Cluster Size	N = 10
$F_c in$	0.33
$F_c out$	0.33
Channel Segment Length	4
Input Pins per Cluster	33
Optimization	Timing (assuming minimum channel width)

In this experiment the minimum channel width, critical path and number of cluster logic blocks (CLBs) of all generated clone circuits are constrained to be within 25% of the original Verilog circuit, computed using Equation 6.1. We also imposed a time limit of 48 hours on the amount of time that is spent to generate an acceptable clone. If a clone circuit with acceptable characteristics it not generated before the time limit, the program will be termi-

nated. In this case, the best clone circuit generated in the past 48 hours is considered as the accepted clone circuit. The best clone circuit is the one that the has the minimum sum of the computed value of Equation 6.1 for all of its constrained post-synthesis characteristics.

$$\frac{Original\ Postsynthesis\ Characteristics - Cloned\ Postsynthesis\ Characteristics}{Original\ Postsynthesis\ Characteristics} * 100\% \tag{6.1}$$

6.2.2 Runtime Optimization

The time required to analyze a given circuit, as described in Chapter 3, and to generate a clone circuit, as described in Chapter 4, is negligible in comparison with running a circuit through the CAD flow. In addition, as described in Section 4.4, we are generating a circuit based on patterns and statistics rather than implementing a specific functionality. As a result, the generated clone circuit may contain an unreasonably long critical path. Such a clone circuit is not an acceptable clone and running it through the CAD flow to measure its critical path is highly time consuming.

As shown in Table 3.9 the longest path of the data flow graph is correlated with the critical path of the circuit. This observation can be used to discard the generated clone circuits with unreasonably high critical path without running them through the CAD flow. Hence after generating a clone circuit, the longest path in its DFG is measured. If it is longer than a user defined limit the generated circuit is discarded. By setting the limit to a low value the chance of discarding circuits with an unreasonably long critical path is higher, however there is a higher risk of discarding an acceptable clone. The default longest path limit is set to 30 DFG nodes since the longest path among all experimental set of circuits is 25 DFG nodes. The pseudocode of our acceptable clone generator is shown in Code 6.1.

An alternative approach is to avoid unreasonably long critical path is to modify our clone circuit generation algorithm to prevent generating a circuit that its DFG has a longest

path higher than a limit. In another words, at each iteration of operand selection, described in Subsection 4.4.5, in addition to updating the DFG and checking it for cycles, we would also update its longest path and check if it is higher than a limit. If it is, the DFG would be reverted to its previous version and operand selection is performed again to select new operands. This approach was not persuaded because of concerns about the convergence of the algorithm.

Code 6.1: Pseudocode of the clone circuit generation algorithm.

```
1
2  Clone_circuit_generator(input_circuit){
3        parameters, patterns, constraints = analyze(input_circuit)
4     while(time < 48 hours){
5        new_circuit = clone_circuit_generator (parameters, patterns)
6        DFG_longest_path = DFG_longest_path(new_circuit)
7        if(DFG_longest_path < 30 nodes){
8              post_synthesis_characteristics = CAD_flow(new_circuit)
9           if(|post_synthesis_characteristics| < 25%){
10             found = 1
11             accepted_clone_circuit = new_circuit
12             break
13          }
14       }
15    }
16    if (found == 0)
17       accepted_clone_circuit = best generated clone
18 }
```

6.2.3 Experimental Results

Number of Attempts, Number of Discards, Clone Time and Time Limit

Column 11 of Table 6.2 shows the *number of attempts* to generate an acceptable clone for each benchmark circuit. Column 12 shows the *number of discards* which is the number of generated circuits that are discarded because is has a DFG with a longest path more than 30 nodes. Column 13 shows the *normalized clone time*, which is the time that it takes to generate an acceptable clone circuit divided by the time spent to run the original

Table 6.2: Results of the Clone Circuit Generator for generating a clone for each Verilog circuit of the VTR benchmark suit

	Original Circuit Results			Accepted Clone Circuit Results			Percentages Using 6.1			Other			
Circuit	Channel Width	Critical Path Delay (ns)	#CLBs	Channel Width	Critical Path Delay (ns)	#CLBs	Channel Width	Critical Path Delay (ns)	#CLBs	#Attemps	#Discards	Normalized Clone Time	Time Limit
bgm	116	26.46	2930	120	21.39	3781	-3.45	19.16	-29.04 %	34	7	4.36x	1
blob_merge	74	10.34	543	88	12.44	653	-18.92	-20.29	-20.26 %	25	2	3.24x	0
boundtop	60	6.46	233	50	4.93	285	16.67	23.69	-22.32 %	56	0	71.32x	0
ch_intrinsics	50	3.94	37	38	2.70	33	24.00	31.34	10.81 %	76	0	47.16x	-1
diffeq1	52	22.52	36	43	16.53	47	17.31	26.61	-30.56 %	51	1	91.37x	1
diffeq2	52	16.53	27	68	18.59	35	-30.77	-12.49	-29.63 %	43	1	52.41x	0
LU8PEEng	114	115.29	2104	122	137.03	1527	-7.02	-18.86	27.42 %	53	11	7.56x	1
LU32PEEng	174	115.06	7128	129	121.97	6102	25.86	-6.01	14.39 %	31	18	1.42x	1
mcml	104	79.64	6615	140	55.55	4391	-34.62	30.25	33.62 %	53	27	1.73x	1
mkDelayWorker32B	76	7.30	447	60	7.18	416	21.05	1.63	6.94 %	42	0	13.49x	0
mkPktMerge	46	4.57	15	41	5.28	19	10.87	-15.45	-26.67 %	67	0	8.34x	-1
mkSMAdapter4B	56	5.65	165	49	5.72	148	12.50	-1.28	10.30 %	89	0	6.32x	0
or1200	74	13.34	257	60	14.99	250	18.92	-12.44	2.72 %	32	1	35.72x	0
raygentop	68	5.04	173	53	4.10	200	22.06	18.70	-15.61 %	75	0	24.83x	0
sha	50	13.64	209	37	11.71	236	26.00	14.17	-12.92 %	94	3	52.07x	1
stereovision0	60	4.36	905	68	5.89	968	-13.33	-35.03	-6.96 %	32	0	21.62x	0
stereovision1	104	5.75	889	124	7.43	623	-19.23	-29.16	29.92 %	112	0	74.36x	1
stereovision2	154	17.39	2395	127	11.38	2972	17.53	34.56	-24.09 %	93	38	1.74x	1
stereovision3	34	2.67	13	38	3.13	17	-11.76	-17.24	-30.77 %	88	0	30.76x	-1

circuit through the CAD flow. As mentioned in Subsection 6.2.2, the dominant portion of time spent for generating an acceptable clone is running the generated clones through the CAD flow. As a result Equation 6.2 is approximately the same as Equation 6.3. The *time limit* (Column 14), is 1 if an acceptable clone is not generated in less than 48 hours and 0 otherwise.

$$\frac{total\ time\ spent\ to\ generate\ an\ acceptable\ clone\ circuit}{time\ spent\ to\ run\ the\ original\ circuit\ through\ the\ CAD\ flow} \tag{6.2}$$

$$\frac{time\ spent\ to\ run\ the\ generated\ clones\ through\ the\ CAD\ flow}{time\ spent\ to\ run\ the\ original\ circuit\ through\ the\ CAD\ flow} \tag{6.3}$$

For example to come up with an acceptable clone for *blob_merge* circuit, shown in Row 3 of Table 6.2, 25 clone circuits are generated and 2 of them are discarded. Hence only 23 of them are run through the CAD flow. Running these 23 clone circuit takes 3.24 times longer than running *blob_merge* circuit through the CAD flow. An acceptable clone circuit is generated in less than 48 hours.

The clone time is less than number of circuits that are run through the CAD flow, except for two cases (circuits $bound\_top$ and $or1200$). The reason is that most generated clone circuits have fewer CLBs and a lower critical path than the original circuit. This can have

two reasons:

- The generated clone is based on patterns and statistics rather than to implement a specific functionality as a result its DFG might be not as complex as the DFG of the original circuit.

- There are other situations that cause optimizable code which are not considered in Subsection 4.4.5. For example 6.2 will be optimized to $b = b + a;$.

- Mixing certain process patterns and expression patterns creates unsynthesisable Verilog code or a code that is not supported by the CAD flow, hence the CAD flow fails at early stages.

Code 6.2: An Optimizable Example Code

```
1 case(a) begin
2   0: b = b;
3   1: b = b + 1;
4   2: b = b + 2;
5   3: b = b + 3;
6 end
```

The clone time for circuits *bound_top* and *or*1200 is more than the number of circuits that are run through the CAD flow. The reason is that for both of these, a few generated clones have an unreasonably long critical path that were not eliminated using our runtime optimization approach described in Subsection 6.2.2.

The number of discarded generated clone circuits is higher when the critical path of the original circuit is longer. This is because those original Verilog circuits are more likely to contain an expression and process patterns that results in a DFG with the longest path more than the limit.

Minimum Channel Width, Critical Path and Number of CLBs

Columns 2 to 4 of Table 6.2 show the post-synthesis characteristics of the original Verilog circuits. Columns 5 to 7 show the post-synthesis characteristics of the generated clone circuit. Columns 8 to 10 show the result of the Equation 6.1 for each post-synthesis characteristics of the generated clone circuit. These percentages are less than 25% unless the 48 hours time limit is reached while generating the clone circuit.

In addition, if the critical path delay of the original circuit is short, generating an acceptable clone circuit will take several attempts since the acceptable range is also small. As a result the generated clone circuits for which the critical path delay is not within 25% of the original circuit but their difference is less than 2 ns are considered as acceptable clones, such as *ch_intrinsics* in row 2. For the same reason the generated clone circuits for which the number of CLBs is not within 25% of the original circuit but their difference is less than 5 CLBs are also acceptable clones, such as *mkPktMerge* in row 12 and *stereovision*3 in the last row.

6.3 Random Results and Characterization

6.3.1 Overview of Experimentation Methodology

We analyzed 19 Verilog circuits from the VTR set of benchmarks as the input set of circuits to obtain the necessary set of data. This data consists of a pool of module topology graphs, a table of process patterns, a table of expression patterns, the distribution of the number of assignments and the distribution of the number of processes. We used this set of data to tune our random circuit generator and generate random circuits as described in Chapter 5. One of challenges of designing a random circuit generators is verifying the realism of the circuits that it generates, since there is no specific definition of *real* circuit. One possible solution could be demonstrating that the random generated circuits have the same

trend as the input set of circus. However our studies show that it is difficult to identify trends in our input set of circuits. For example, Figure 6.1 shows the relationship between $size\ of\ circuit(CLBs)$ and $critical\ path\ delay$ of our input set of circuits and Figures 6.3 to 6.6 show the relationships between the size of circuit and four different post-systhesis characteristics of eASIC circuits which occupy a region rather forming a trend.

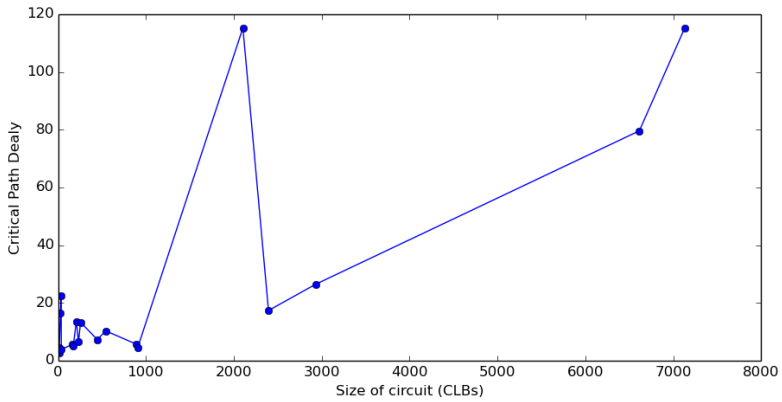

Figure 6.1: The relationship between size of circuit (CLBs) and critical path delay of our input set of circuits to demonstrate that this relation has no specific trend.

An alternative verification approach is determining if a randomly generated circuit has post-synthesis characteristics are within a predetermine range.For example we could verify that the circuit's critical path, number of CLBs and minimum channel width are within the corresponding renges for the input set of circuits. Based on this verification approach the acceptable ranges are (2.667ns, 115.29ns) for the critical path, (34, 174) for the channel width, and (13, 7128) for the number of CLBs. However these acceptable ranges are wide, and not all circuits in these ranges may be realistic. For example, a randomly generated circuit with channel width of 40 and critical path of 100ns is not realistic, as illustrated in Figure 6.2. Hence we divide our input set of circuit into two groups: those with fewer than 1000 CLBs and those with more than 1000 CLBs, and define two separate acceptable ranges based on each group, as shown in Table 6.3.

Table 6.3: Acceptable ranges of critical path, minimum channel width and number of CLBs based on set of input circuits divided into two groups.

Circuit Group	Critical Path Range	Minimum Channel Width Range	#CLBs Range
VTR circuits with less than 1000 CLBs	(2.667ns, 22.523ns)	(34, 104)	(13, 1000)
VTR circuits with more than 1000 CLBs	(17.385ns, 115.29s)	(104, 174)	(1000, 7128)

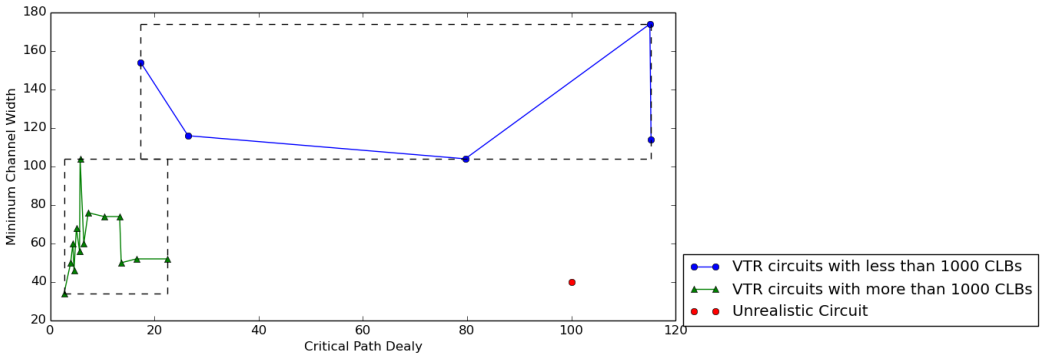

Figure 6.2: Demonstrating that a randomly generated circuit with channel width of 40 and critical path of 100ns is not realistic. Dividing the input set of circuits into two groups based on their size.

6.3.2 Correlation

Out of 50 circuits that we generated using our random generation algorithm described in Chapter 5, 19 are in acceptable ranges, from Table 6.3. Generally, the generated random circuits that are not in the acceptable range have fewer CLBs or a shorter critical path than the circuits that are in the acceptable range. Hence our random circuit generator is generating unrealistic circuits. the following subsections describe the modifications that we applied to our random circuit generation algorithm to make it more realistic. Table 6.6 shows how many circuits were acceptable when we generated 50 different circuits using our random circuit generator in each different mode.

Correlation Between Number of Processes in a Module and the Pattern of its Processes

Our random circuit generator may generate a module with one process. In this case its process pattern is most likely a sequential process containing a non-blocking assignment, according to Table 3.7. The post-synthesis characteristics of this module, including the number of CLBs and critical path, are small. However, such a small module does not exist in our input set of circuits. There are 11 patterns that are found in the modules with one process; Table 6.4 shows how common each of them are, their process category, number of keywords that each process pattern has and the first five keywords. As a result choosing a process pattern solely based on how common each pattern is in the input set of circuits is not realistic.

In order to make our process pattern decision more realistic, we create tables similar to Table 6.4 for each number of processes that a module can have. Then in our random circuit generator algorithm while choosing a process pattern for a module with p processes, we choose from the table that stores the process pattern from modules with p processes. We generated 50 random circuits using this modified algorithm. The outcome is shown in Table 6.6. The number of accepted random generated circuits is 7 more than that from theprevious experiment using the unmodified algorithm.

We applied the same modification on expressions patterns of assignments, however the improvement to number of accepted circuits was not significant.

Correlation Between Process Patterns and Expression Patterns

We also address the correlation between process pattern and expression patterns. As an example, consider a sequential process with the $0 : sequential - 1 : conditional - 2 : seqblock - 3 : nonblockingassign$ pattern. This is the third most common process pattern, according to Table 3.7. As described in Subsection 5.2.2, if our random circuit generator chooses this

Table 6.4: Pattern of Processes of Modules with One Process

Percentage	Process Category	#Keywords	First Five Keywords of the Process Pattern
25%	Sequential	11	0:seqblock - 1:conditional - 2:seqblock - 3:nonblockingassign - 3:nonblockingassign
25%	Sequential	24	0:seqblock - 1:conditional - 2:seqblock - 3:nonblockingassign - 3:nonblockingassign
6.25%	Sequential	29	0:seqblock - 1:conditional - 2:seqblock - 3:nonblockingassign - 3:nonblockingassign
6.25%	Sequential	1765	0:seqblock - 1:conditional - 2:seqblock - 3:nonblockingassign - 3:nonblockingassign
6.25%	Sequential	12	0:seqblock - 1:conditional - 2:seqblock - 3:nonblockingassign - 2:seqblock
6.25%	Sequential	9	0:seqblock - 1:conditional - 2:nonblockingassign - 2:conditional - 3:nonblockingassign
5%	Sequential	412	0:seqblock - 1:case - 2:caseitem - 3:seqblock - 4:nonblockingassign
5%	Sequential	11	0:seqblock - 1:nonblockingassign - 1:nonblockingassign - 1:nonblockingassign - 1:conditional
5%	Combinational	269	0:seqblock - 1:case - 2:caseitem - 3:seqblock - 4:blockingassign ...
5%	Sequential	27	0:seqblock - 1:conditional - 2:seqblock - 3:nonblockingassign - 2:seqblock ...
5%	Sequential	20	0:seqblock - 1:conditional - 2:seqblock - 3:nonblockingassign - 1:nonblockingassign ...

process pattern then it needs to choose an expression pattern for $3 : nonblockingassign$ part of this pattern. According to Table 3.8 this would be a $constant$ or a $signalref$ with 69.08% possibility such a pattern is implemented by wiring and does not require any complicated circuitry. However in our input set of circuits the expression pattern of this non-blocking statements contains at least one binary operation, as shown in Table 6.5. As a result choosing an expression pattern solely based on how common each pattern is in the input set of circuits is not realistic.

In order to make our expression pattern decision more realistic we created a table similar to Table 6.5 for each process pattern. Then we modified our clone circuit generation algorithm in a way that it chooses the expression pattern for each process pattern from its specific table. For example the expression pattern for the non-blocking statement of $0 : sequential - 1 : conditional - 2 : seqblock - 3 : nonblockingassign$ process will be decided based on Table 6.5. We generated 50 random circuits using this modified algorithm and 23 of the generated circuits were in the acceptable range, 4 which is more that the original version our random circuit generator.

Table 6.5: Expression Patterns that were found in processes with $0 : sequential - 1 : conditional - 2 : seqblock - 3 : nonblockingassign$ pattern

Percentage	Pattern
16.27%	binary signalref signalref
16.27%	binary binary signalref signalref constant
67.45%	binary signalref constant

Table 6.6: Random Circuit Generator Results

Random Generator Mode	#Attempts	#Accepted
No Correlation	50	19
#Processes in a Module and Process Patterns	50	26
Process Patterns and Expression Patterns	50	23

6.4 Comparison to Earlier Circuit Generators

In this section, we discuss why generating circuits at the RTL level is fundamentally advantageous compared to generating circuits at lower levels of abstraction. Then we compare the post-synthesis characteristics of our random circuit generator against previous benchmark generators: Mark [31], GEN [21] and Gnl [41] and eASIC circuits [30].

6.4.1 RTL Circuit Generator vs. Netlist Circuit Generator

Unlike previous generators, ours generates circuits at the RTL which are much more suitable for the types of architecture and CAD studies that researchers often want to perform. The reason is that CAD tools are created to synthesize circuits at RTL level which is the level of abstraction at which designers specify their circuits. However, all previous works characterize and generate circuits at the netlist level or gate level which limits the usefulness of their generated circuits for evaluating physical design CAD algorithms and does not allow for the evaluation of synthesis related mapping algorithms.

6.4.2 Comparison against previous benchmark generators

This section compares results obtained form the eASIC circuits to those obtained using our random circuit generator as well as Mark, GEN, Gnl. We introduced eASIC circuits in Subsection 2.3.2 and more details on the data collected from the eASIC circuits can be found in [30].

The Mark generator validated its results by demonstrating that her generator is capable of generating circuits with post-synthesis characteristics that scales well to mimic eASIC circuits in comparison with previous generators, GEN and Gnl, with respect to circuit size.

Mark's, GEN and Gnl were calibrated using specific input set of circuits and synthesis tools, which may no longer be state of the art. As a result in order to compare our results we used the numbers reported in [31] for 18 circuits of varying sized from 5687 to 72625 four-input LUTs. These circuits were synthesized using T-VPACK and VPR 5.0's timing-driven clustering, placement, and mapped to a minimum-sized FPGA with minimum channel width. Moreover, we generated 5 random circuits using our random generator and synthesized them using CAD Flow and the architecture setup as shown in Table 6.7.

Table 6.7: CAD Flow and the architecture setup used in comparison to earlier circuit generators experiment

CAD Flow Property	**Value**
CAD Flow	VTR 5.0
Target FPGA	Similar to Altera Stratix
LUT Size	4
Cluster Size	N = 6
$F_c in$	0.33
$F_c out$	0.33
Channel Segment Length	4
Input Pins per Cluster	15
Optimization	Area (assuming minimum channel width)

The relationship of the number of nets versus the size of circuit based on number of four-input LUTs of these four different generators and eASIC circuits are shown in Figure 6.3. A dataset provided by Mark was used to obtain data for the previous generator as well as the eASIC circuits [29]. Each group of circuits is labeled by its name. Circuits generated by our random circuit generator are labeled as *New*. The eASIC circuits are shown as scattered dots however the generated circuits of each generator are connected together using a solid line in order to demonstrate trends. Figures 6.4, 6.5 and 6.6 were produced in the same manner and show the relationship of a different post-synthesis characteristics to the size of circuit. In the following subsections we discuss each of these figures.

Number of Nets

Figure 6.3 shows the relationship between the number of nets and the size of circuit for each of eASIC circuits and the circuits generated by the four generators, Mark, GEN, Gnl, and *New*. The largest circuit generated by Mark, GEN, and Gnl are much smaller than eASIC circuits; as a result the only possible way to evaluate them is extrapolating their trends. Doing so indicates that all three of them have a high possibility of colliding with the region that eASIC circuits are scattered. However our generator is capable of generating circuits almost as big as the eASIC circuits; moreover the number of nets in our circuits are in the range of eASIC circuits.

Critical Path and Minimum Channel Width

Figures 6.4 and 6.5 represent the relationships of the critical path delay and the minimum channel width versus the size of circuit. Trends for GEN and Gnl are not likely to reach the region of the eASIC circuits. However the Mark's trend will reach eASIC circuits and our random generated circuits have the correct ranges.

Average Net Length

Figure 6.6 shows the average net length of eASIC circuits and the circuits generated by each of the four generators. Mark's circuits show correct trend related to the eASIC circuits. However, the trends of GEN and Gnl are not likely to reach that region. One of our random generated circuits has a higher average net length compared to the eASIC circuits. The other four of our circuits are in a less crowded region. This suggests that our random generator is generating circuits with a higher average net length than the realistic circuits. Since their number of nets were reasonable according to Figure 6.3, we can conclude that nets of our circuits are longer than realistic circuits. Long nets are used to connect parts of circuits that are futhur away form each other. As a result our circuits have less locality.

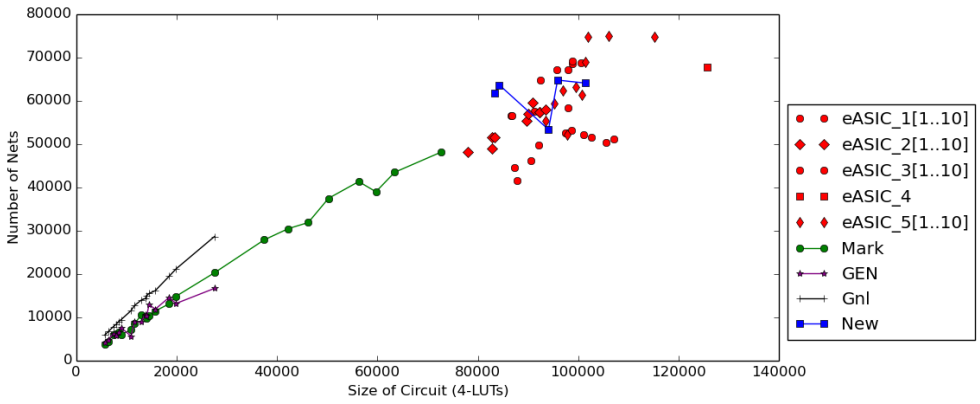

Figure 6.3: Number of Nets Comparison

6.5 Summary

In order to evaluate the efficiency of our clone circuit generator algorithm we generated a clone circuit for each of the VTR set of Verilog benchmark circuits. We were able to generate clones with post-synthesis characteristics (critical path, channel width and number of cluster logic blocks) that fall within 25% range of corresponding characteristic of the original circuits. Generating these circuits requires, on average, 23.89x the run-time of

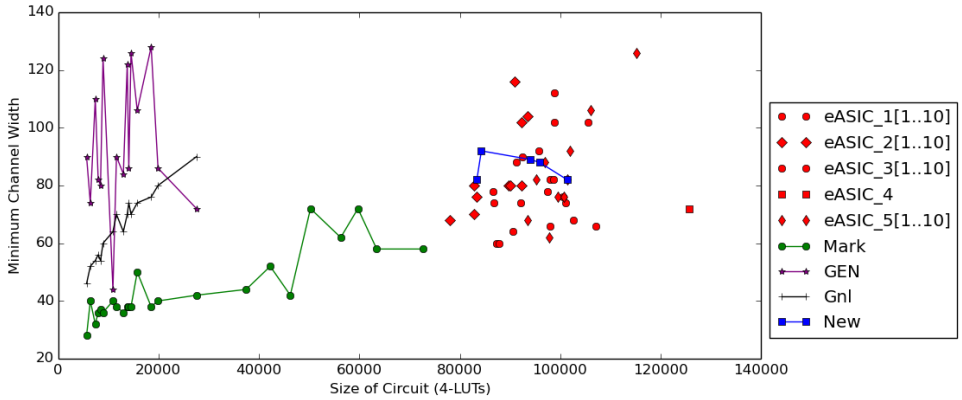

Figure 6.4: Minimum Channel Width Comparison

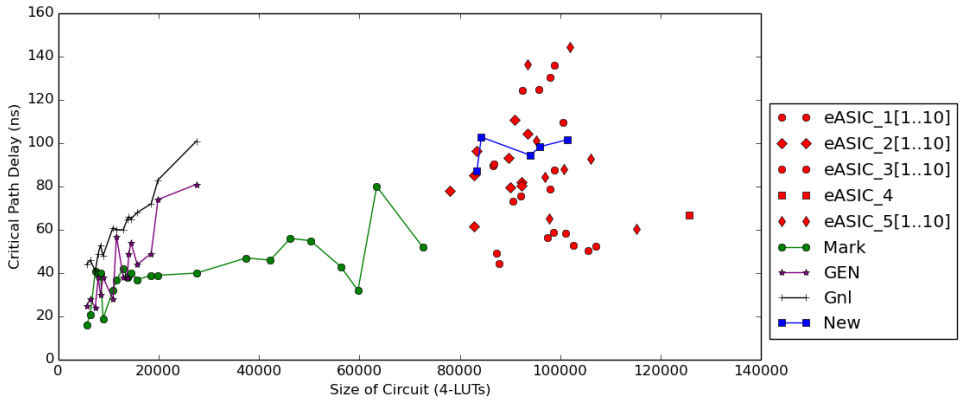

Figure 6.5: Critical Path Comparison

synthesizing the original circuit using the VTR flow.

Our random circuit generator is evaluated based on the number of the candidate circuits with characteristics that fall within an acceptable range random circuits. By exploring possible correlations we increased the number of acceptable circuits. In our experiments 46% of our random generated candidate circuits fell in the acceptable range bounded by the minimum and maximum of post-synthetic characteristics in our input set of circuits. We also demonstrated that our generated circuits have characteristics that fell within the range of large industrial circuits while the circuits generated by the previous work are much smaller than large industrial circuits and do not scale well.

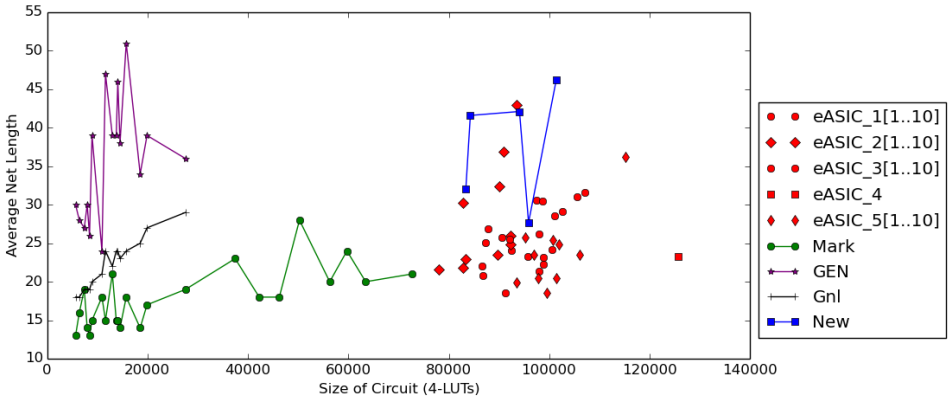

Figure 6.6: Average Net Length Comparison

Chapter 7

Conclusion

7.1 Summary

Today FPGAs are emerging as essential components of data centers and cloud computing infrastructures. In order to keep up with such quickly changing application domains, FPGAs that are more dencse consume less power and run faster are required. However, the process of enhancing FPGAs is hindered by the lack of proper benchmark circuits (required for evaluation of new designs). In this thesis we introduced an approach for generating RTL level benchmark circuits.

The first phase of this thesis describes our analysis techniques and presents the analysis results from our input set of circuits. In this phase we first modeled circuits at RTL level. We then profiled our input set of circuits based on our model using different profiling techniques. Our first profiling technique collects basic numeric information such as number of ports, assignments, processes, and different types of statements. Since it is not possible to generate a new RTL circuit merely based on this numerical information, we employed a second profiling technique in which we collected graph-based information such as the topology of module instantiations and the DFG of an RTL circuit and its longest path.

Lastly we studied possible sequences of statements in processes (process patterns) and possible sequences of operand and operators in a expressions (expression patterns).

In the second phase of this thesis, we developed an algorithm to generate a clone for an input circuit. Our algorithm first analyzes the input circuit and extracts all the numerical and graph-based information as well as process and expression patterns that exist in the circuit. Then it generates new candidate circuits using the extracted information and runs them through the CAD flow. When a candidate circuit fits within a predefined range of selected post-synthesis characteristics, it will be outputted as the clone circuit and the algorithm terminates. In order to evaluate the efficiency of our clone circuit generator algorithm we generated a clone circuit for each of the VTR set of Verilog benchmark circuits. We were able to generate clones with post-synthesis characteristics (critical path, channel width and number of cluster logic blocks) that fall within 25% of the corresponding characteristic in the original circuits. Generating these circuits requires, on average, 23.89 times the run-time of synthesizing the original circuit using the VTR flow.

In third phase we altered our clone generator algorithm to generate random circuits using the analysis information extracted form all available RTL circuits. Our random circuit generator is evaluated based on the number of the candidate circuits with characteristics that fall within an acceptable range. By exploring possible correlations we increased the number of acceptable circuits. In our experiments 46% of our random generated candidate circuits fell in the acceptable range bounded by the minimum and maximum of post-synthetic characteristics in our input set of circuits. We also showed that our generator is able to generate random circuits with post-synthesis characteristics that fall within the range of large industrial circuits.

It is known that not all circuits written at RTL level are synthesizable and only a subset of them are supported by all CAD flows. Since our generator mixes different process and expression patterns it is capable of generating circuits that are not supported by CAD

tools. This issue increases the number of necessary attempts to generate an acceptable circuit. However, the discarded circuits can be used as examples in studies targeting CAD algorithm improvements.

7.2 Limitations and Future Work

The circuit generator algorithm introduced in this thesis generates circuits based on collected data and statistics rather than specific functionalities. One issue with this approach is the it is possible to generate a circuit with an unusually long critical path (addressed in 6.2.2). A second issue is that the locality of our generated circuits may be lower than realistic circuits. Hence their average wirelength will be higher than normal as shown in Figure 6.6. However, the gap is not large since our generator creates a circuit by connecting individually generated modules based on the module topology graph. As a result the communication between modules is limited to ports and all the internal signals are localized within a module. However, to address this issue, one possible solution is to add the average wire-length to the list of selected post-synthesis characteristics. This approach is not desirable since increasing the number of selected post-synthesis characteristics will increase the required number of attempts to generate an acceptable circuit. A better approach is to study the DFGs of available modules and generate new module with DFGs that are isomorphic to the DFGs of the available modules.

The complexity of process and expression patterns that will be used in a generated circuit is limited to those that are found in the input set of circuits. For example none of our input set of RTL circuits have a $for\ loop$. Hence none of the circuits generated using this set of data has a $for\ loop$s.

One area of potential improvement is decreasing the generation runtime by eliminating the number of necessary attempts. The unsuccessful attempts are the result of generating circuits containing unrealistic patterns. We eliminated some of them by applying the corre-

lations between process pattern and number of processes, as well as expression patterns and process patterns, as described in Section 6.3.2. The same idea could be applied to the other input parameters. However, no other significant correlations were found due to the small number of input circuits. We expect that exploring possible correlations within a much larger set of circuits and using the findings to tune the generator significantly decrease the average number of attempts to generate an acceptable circuit.

Another area of future research is to enhance our benchmark circuit generator algorithm so that it can generate circuits consisting embedded blocks such as memories, processors, and multipliers. Embedded blocks are an important part of realistic RTL circuits. In order to achieve this, a careful circuit analysis study on input and output parameters of these blocks and the type of circuits and hierarchy level that they appear in is required. In addition the generation algorithm needs to be modified to be able to properly instantiate these blocks.

Bibliography

[1] S. N. Adya, M. C. Yildiz, I. L. Markov, P. Villarrubia, P. N. Parakh, and P. H. Madden. Benchmarking for large-scale placement and beyond. volume 23, pages 472–487, 2004. → pages 12

[2] Altera. Cyclone v device handbook vol. 1: Device interfaces and integration. 2015. → pages 6, 8

[3] J. H. Anderson and F. N. Najm. Power-aware technology mapping for lut-based fpgas. In *Proceedings of the 2002 IEEE International Conference on Field-Programmable Technology, FPT 2002, Hong Kong, China, December 16-18, 2002*, pages 211–218. → pages 9

[4] J. H. Anderson and F. N. Najm. Low-power programmable FPGA routing circuitry. volume 17, pages 1048–1060, 2009. → pages 8

[5] V. Betz and J. Rose. VPR: A new packing, placement and routing tool for FPGA research. In W. Luk, P. Y. K. Cheung, and M. Glesner, editors, *Field-Programmable Logic and Applications, 7th International Workshop, FPL '97, London, UK, September 1-3, 1997, Proceedings*, volume 1304 of *Lecture Notes in Computer Science*, pages 213–222. Springer, 1997. → pages 6

[6] V. Betz and J. Rose. FPGA routing architecture: Segmentation and buffering to optimize speed and density. In *FPGA*, pages 59–68, 1999. → pages 8

[7] J. Cong and Y. Ding. On area/depth trade-off in lut-based FPGA technology mapping. volume 2, pages 137–148, 1994. → pages 9

[8] J. Cong and K. Minkovich. Lut-based FPGA technology mapping for reliability. In S. S. Sapatnekar, editor, *Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010*, pages 517–522. ACM, 2010. → pages 9

[9] J. Cong, C. Wu, and Y. Ding. Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution. In *FPGA*, pages 29–35, 1999. → pages 7

[10] J. Cong and S. Xu. Delay-optimal technology mapping for fpgas with heterogeneous luts. In *DAC*, pages 704–707, 1998. → pages 7

[11] J. Das, A. Lam, S. J. E. Wilton, P. H. W. Leong, and W. Luk. An analytical model relating FPGA architecture to logic density and depth. volume 19, pages 2229–2242, 2011. → pages 2

[12] ePrize1. 2008. → pages 14

[13] D. Ghosh, N. Kapur, F. Brglez, and J. E. H. III. Synthesis of wiring signature-invariant equivalence class circuit mutants and applications to benchmarking. In P. Dewilde, F. J. Rammig, and G. Musgrave, editors, *1998 Design, Automation and Test in Europe (DATE '98), February 23-26, 1998, Le Palais des Congrès de Paris, Paris, France*, pages 656–663. IEEE Computer Society, 1998. → pages 15, 16

[14] D. Grant, S. Chin, and G. G. Lemieux. Semi-synthetic circuit generation using graph monomorphism for testing incremental placement and incremental routing tools. In *Proceedings of the 2006 International Conference on Field Programmable Logic and Applications (FPL), Madrid, Spain, August 28-30, 2006*, pages 1–4. IEEE, 2006. → pages 17

[15] D. Grant and G. Lemieux. Perturber: semi-synthetic circuit generation using ancestor control for testing incremental place and route. In G. A. Constantinides, W. Mak, P. Sirisuk, and T. Wiangtong, editors, *2006 IEEE International Conference on Field Programmable Technology, FPT 2006, Bangkok, Thailand, December 13-15, 2006*, pages 189–196. IEEE, 2006. → pages 16

[16] D. Grant and G. G. Lemieux. Perturb+mutate: Semisynthetic circuit generation for incremental placement and routing. volume 1, pages 16:1–16:24, 2008. → pages 3, 15, 17

[17] J. W. Greene, S. Kaptanoglu, W. Feng, V. Hecht, J. Landry, F. Li, A. Krouglyanskiy, M. Morosan, and V. Pevzner. A 65nm flash-based FPGA fabric optimized for low cost and power. In J. Wawrzynek and K. Compton, editors, *Proceedings of the ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays, FPGA 2011, Monterey, California, USA, February 27, March 1, 2011*, pages 87–96. ACM, 2011. → pages 8

[18] J. E. Harlow, III, and F. Brglez. Synthesis of esi equivalence class combinational circuit mutants. Technical report, CBL, CS DEPT., NCSU, BOX 7550, 1997. → pages 15

[19] Y. Hu, S. Das, S. Trimberger, and L. He. Design and synthesis of programmable logic block with mixed LUT and macrogate. volume 28, pages 591–595, 2009. → pages 8

[20] M. D. Hutton, J. Rose, and D. G. Corneil. Automatic generation of synthetic sequential benchmark circuits. volume 21, pages 928–940, 2002. → pages 3, 13, 72

[21] M. D. Hutton, J. Rose, J. P. Grossman, and D. G. Corneil. Characterization and parameterized generation of synthetic combinational benchmark circuits. volume 17, pages 985–996, 1998. → pages 12, 31

[22] P. A. Jamieson and J. Rose. Enhancing the area efficiency of fpgas with hard circuits using shadow clusters. volume 18, pages 1696–1709, 2010. → pages 8

[23] P. D. Kundarewich and J. Rose. Synthetic circuit generation using clustering and iteration. volume 23, pages 869–887, 2004. → pages 3

[24] I. Kuon and J. Rose. Measuring the gap between fpgas and asics. volume 26, pages 203–215, 2007. → pages 6

[25] B. S. Landman and R. L. Russo. On a pin versus block relationship for partitions of logic graphs. volume C-20, pages 1469–1479, Dec 1971. → pages 11

[26] M. Lin and A. E. Gamal. A low-power field-programmable gate array routing fabric. volume 17, pages 1481–1494, 2009. → pages 8

[27] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz. VTR 7.0: Next generation architecture and CAD system for fpgas. volume 7, pages 6:1–6:30, 2014. → pages 2

[28] V. Manohararajah, S. D. Brown, and Z. G. Vranesic. Heuristics for area minimization in lut-based FPGA technology mapping. volume 25, pages 2331–2340, 2006. → pages 9

[29] C. Mark. A system-level synthetic circuit generator for fpga architectural analysis. 2006. → pages 74

[30] C. Mark, S. Y. L. Chin, L. Shannon, and S. J. E. Wilton. Hierarchical benchmark circuit generation for FPGA architecture evaluation. volume 11, pages 42:1–42:25, 2012. → pages 3, 13, 72, 73

[31] C. Mark, A. Shui, and S. J. E. Wilton. A system-level stochastic circuit generator for FPGA architecture evaluation. In T. A. El-Ghazawi, Y. Chang, J. Huang, and P. Saha, editors, *2008 International Conference on Field-Programmable Technology, FPT 2008, Taipei, Taiwan, December 7-10, 2008*, pages 25–32. IEEE, 2008. → pages 14, 72, 73

[32] K. E. Murray, S. Whitty, S. Liu, J. Luu, and V. Betz. Titan: Enabling large and complex benchmarks in academic CAD. In *23rd International Conference on Field programmable Logic and Applications, FPL 2013, Porto, Portugal, September 2-4, 2013*, pages 1–8. IEEE, 2013. → pages 2

[33] Y. Okamoto, Y. Ichinomiya, M. Amagasaki, M. Iida, and T. Sueyoshi. *COGRE: A configuration memory reduced reconfigurable logic cell architecture for area minimization*, pages 304–309. 12 2010. → pages 8

[34] J. Pistorius, E. Legai, and M. Minoux. Partgen: a generator of very large circuits to benchmark thepartitioning of fpgas. volume 19, pages 1314–1321, 2000. → pages 13, 15

[35] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. R. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. volume 59, pages 114–122, 2016. → pages 1

[36] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In M. R. Lightner and J. A. G. Jess, editors, *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993, Santa Clara, California, USA, November 7-11, 1993*, pages 42–47. IEEE Computer Society / ACM, 1993. → pages 15

[37] A. Singh and M. Marek-Sadowska. Efficient circuit clustering for area and power reduction in fpgas. In *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-programmable Gate Arrays*, FPGA '02, pages 59–66, New York, NY, USA, 2002. ACM. → pages 9

[38] S. Sivaswamy, G. Wang, C. Ababei, K. Bazargan, R. Kastner, and E. Bozorgzadeh. HARP: hard-wired routing pattern fpgas. In H. Schmit and S. J. E. Wilton, editors, *Proceedings of the ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays, FPGA 2005, Monterey, California, USA, February 20-22, 2005*, pages 21–29. ACM, 2005. → pages 8

[39] L. Sterpone and M. Violante. A new reliability-oriented place and route algorithm for sram-based fpgas. volume 55, pages 732–744, 2006. → pages 10

[40] D. Stroobandt, J. Depreitere, and J. V. Campenhout. Generating new benchmark designs using a multi-terminal net model. volume 27, pages 113–129, 1999. → pages 11

[41] D. Stroobandt, P. Verplaetse, and J. M. V. Campenhout. Generating synthetic benchmark circuits for evaluating CAD tools. volume 19, pages 1011–1022, 2000. → pages 3, 11, 72

[42] G. Wang, S. Sivaswamy, C. Ababei, K. Bazargan, R. Kastner, and E. Bozorgzadeh. Statistical analysis and design of HARP fpgas. volume 25, pages 2088–2102, 2006. → pages 8, 10

[43] Xilinx. Virtex-6 fpga configuration user guide. 2015. → pages 6, 8