Enhancing Productivity with Back-End Similarity Matching of Digital Circuits for IP Reuse

Kevin Zeng

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Peter M. Athanas, Chair
Patrick R. Schaumont
Cameron D. Patterson

April 30, 2013
Blacksburg, Virginia

Keywords: FPGA, Productivity, Digital Circuits, Graph Matching, Similarity, IP Reuse

Copyright 2013, Kevin Zeng

Enhancing Productivity with Back-End Similarity Matching of Digital Circuits for IP Reuse

Kevin Zeng

(ABSTRACT)

Productivity for digital circuit design is being outpaced currently by the rate at which silicon is growing such as FPGAs. Complex designs take a large amount of engineering hours to complete. Reuse of existing design can potentially decrease this cost and increase design productivity. However, existing digital hardware designs are not being effectively reused by the hardware community due to the inability of designers to have knowledge of all the attributes of designs that can be reused. In addition, designers will have to accustom themselves to designs in the hardware library. By having a back-end system that looks for similar circuits, there is little to no effort for the designer to reuse the design. This thesis provides an overview and comparison of different methods for characterizing and comparing digital circuits in order to suggest candidate circuits that engineers can reuse. Several of these methods are implemented, modified, and compared to show the feasibility of utilizing this work for increasing overall productivity.

Acknowledgments

First and foremost, I would like to thank my advisor, Dr. Athanas, for the opportunities to grow and learn. With his guidance, I have learned so much and understood the meaning of what research is all about. I would like to also thank Dr. Schaumont and Dr. Patterson for serving on my committee. All three of them have guided me since I was an undergraduate student and have had a great impact on my academic career.

I would like to thank each and every member of the CCM lab: Ritesh for those late nights at the lab, Tony for his guidance and support, Shaver for consistently pushing me, Kavya for always having someone to talk to, Krzysztof for giving me new ideas to explore, and to Xin Xin, Wen Wei, Rama, Ali, David, Ryan, Madison, Jason, Richard, Kiran, and Andrew for being such great friends and peers to work with.

I would also like to thank each and every teacher, mentor, and professor I have had throughout my entire academic career. All of you have taught and inspired me tremendously.

I am extremely thankful to Dan Duong for constantly reaching out to me. Because of you, I have become more outgoing, engaging, and the overall person I am now. Through you, I got to know many people that I can call my friends. I'd like to also thank Kenney, George, Ian, Justin, Brian, Erick, Andy for being such great friends, to Aaron, Nate, Matt, and Brandon for studying with me, to the library group for a place and people to study with outside of lab, to Daniel, Zihan, Jenny, Diana, and Garrett, for giving me a life outside of school, and to the neighborhood and YG for having a place I can call home.

Lastly, I dedicate all my work to my family, my mom, my dad, and my two sisters. I have not forgotten the sacrifices you have made in order to give me this opportunity. I hope that this work becomes an inspiration to my two younger sisters, Linda and Tina, as they have become my source of inspiration and motivation when all seemed lost. Thank you all.

Contents

1 Introduction **1**

1.1 Motivation 1

Use Case 4

1.2 Contributions 4

1.3 Thesis Organization 5

2 Background **6**

2.1 IP Reuse 6

2.2 Graph Representation 7

2.2.1 Graph Isomorphism 7

2.2.2 Subgraph Isomorphism 9

2.2.3 Maximum Common Subgraph 9

2.2.4 Graph Similarity 11

2.2.5 Edit Distance 11

2.2.6 Spectral Graph Theory 12

2.3 Functional Isomorphism 13

2.3.1 Canonical Representation 13

2.3.2 Rule-Based Detection 14

2.3.3 Boolean Isomorphism 14

2.4 Existing Work Related to Circuit matching 15

2.4.1 Circuit Matching: Gemini and SubGemini 15

2.4.2 Reconfigurable Computing 16
2.4.3 Rule-Based Matching 16
2.4.4 Combinational Equivalence Checking 17
2.4.5 Other Matching Techniques 17
2.4.6 Optimizations by Reducing Search Space 17
2.4.7 Reusing IP Cores 18
2.5 Summary 19

3 System Implementation **20**
3.1 System Overview 20
3.1.1 Front-End (Azido) 21
3.1.2 Netlist 22
3.1.3 Graphical Representation 22
3.1.4 Matcher 23
3.2 Netlist to Graph Converter 24
3.2.1 TORC 24
3.2.2 Graph Format 24
3.2.3 I/O Ports 25
3.3 Displaying Results 26

4 Matcher Implementation **28**
4.1 Data Structure 29
4.1.1 Adjacency List and Adjacency Matrix 29
4.1.2 Incidence List and Incidence Matrix 31
4.2 Custom Subgraph Isomorphism 31
4.2.1 Candidate Vector 32
Page Rank 33
4.2.2 Determining Isomorphism 33
4.2.3 Similarity 34

4.2.4 VF2 36
4.3 Maximum Common Subgraph 36
4.3.1 Compatibility Graph 37
4.3.2 Modular Product 37
4.3.3 Clique Detection 38
4.3.4 Similarity 39
4.4 Decomposition 39
4.4.1 Decomposer 40
Partitioner 40
Decomposition Tree 41
4.4.2 Subgraph Isomorphism 43
4.4.3 Matching 43
Similarity 45
Optimizations 46

5 Results and Analysis 48

5.1 Benchmark 48
5.2 Accuracy 49
5.2.1 Simple Benchmark 49
CSI and VF2 50
MCS 52
DSI 53
5.2.2 Larger Benchmark 54
5.3 Performance 56
5.3.1 Custom Subgraph Isomorphism 56
5.3.2 Maximum Common Subgraph 57
5.3.3 Decomposition 58
5.3.4 Scalability 61

6 Conclusion **64**

6.1 Future Work 65

6.2 Extended Applications 67

Bibliography **68**

Appendix A: Detailed Description of Circuits in Simple Benchmark **72**

Appendix B: Database Circuits **74**

List of Figures

1.1 Design time breakdown of a FPGA design cycle 2
1.2 Proposed usage model 3

2.1 Full adder circuit and its graphical representation 8
2.2 An example of subgraph isomorphism 9
2.3 An example of MCS 10
2.4 Two cliques of different sizes 10
2.5 Edit distance transformation 12
2.6 Three different implementations of a 2-input XOR gate 13

3.1 Flow diagram of overall system 21
3.2 EDIF netlist of a 1-bit counter 23
3.3 Simplified representation of counter circuit 25
3.4 Screenshots of the interactive interface for displaying results of the matcher . 26
3.5 Video demo of the interactive interface. Click to play media 27

4.1 Block diagram for matcher implementation 29
4.2 Adjacency list and matrix of full adder circuit 30
4.3 Incidence list and matrix of full adder circuit 30
4.4 Example Circuit 35
4.5 Nodes of the compatibility graph 37
4.6 Complete compatibility graph 38
4.7 Largest cliques of the compatibility graph 38

4.8 Decomposition of NAND, NOR, and XOR gate 42

4.9 DSI Matching with counter as input circuit 44

5.1 4-bit kogge-stone counter with highlighted 1-bit counter sub-circuits 51

5.2 Execution time of CSI and VF2 for varying pattern circuit sizes 57

5.3 Execution time MCS with varying input and pattern circuit sizes 58

5.4 Execution time of matchers for varying input circuit sizes for a database of thirteen circuits 59

5.5 Execution time of decomposer for different database sizes 60

5.6 Execution time for varying database sizes 61

5.7 Execution time for DSI with varying input circuit sizes 62

List of Tables

5.1 Matches found using CSI Algorithm 50
5.2 Matches found using VF2 Algorithm 50
5.3 Matches found using MCS Algorithm 52
5.4 Matches found using DSI Algorithm 53
5.5 Comparison of results between CSI, VF2, and DSI 55
5.6 Comparison of decomposition matcher using two different decompositions trees 60

Chapter 1

Introduction

Field programmable gate arrays (FPGAs) have become a targeted platform for many computationally intensive applications due to its high parallelism and ability to quickly process massive amounts of data. On the other hand, a significant limitation in using FPGAs is the design phases of the hardware, particularly the debug and verification stages. For example, each module of the hardware has to be designed and verified to work correctly. However, due to the large overhead of the physical design, the number of turns per day is limited, about one to three turns a day depending on the size of the design [1]. Turns is defined as the total number of design iterations performed. A solution to reduce the amount of time in the debug and verification stage is to reuse existing designs. The overall breakdown time spent in each area of the FPGA design cycle is show in Figure 1.1 [2].

1.1 Motivation

The idea of reusing existing designs is heavily seen in software development. Reusable objects, methods, and functions are compiled into software libraries that programmers can

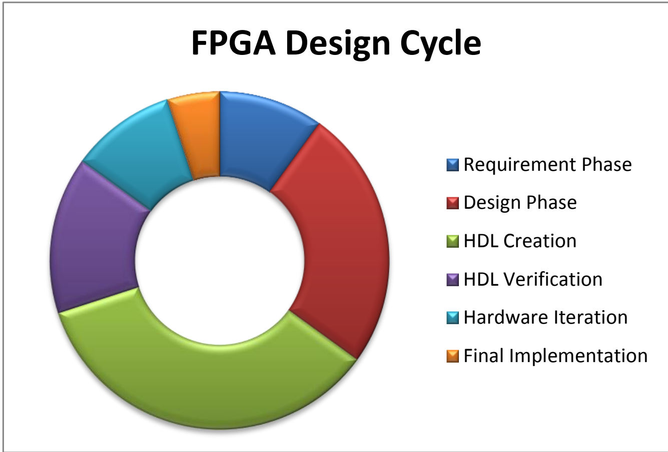


Figure 1.1: Design time breakdown of a FPGA design cycle

import and reuse [3]. The same can be applied to hardware as well. Similar to software, the reused hardware does not have to be verified functionally assuming the design works as intended. Therefore, most of the time and effort goes toward the interfacing and integration of the existing hardware into the overall design. Nelson et al. [1] suggested that, depending on the fraction of the design that is being reused and the overhead of reusing the design, using existing hardware can lead to a significant increase in FPGA productivity.

Despite positive benefits of reusing software, design reuse in the hardware community has not gained widespread acceptance. Reasons include protected intellectual property (IP), high computation costs for comparing existing hardware, overhead of designing a module to be reusable, different hardware platforms [1], performance [3], etc. Many standards such as OCP-IP [4] and IP-XACT [5] are available to help facilitate the reuse of IP cores across various sources; yet, many designers are unwilling to conform to these standards just yet because of the overhead and complexity associated with them. For example, the XML data format of IP-XACT is difficult to read and modify without the support of additional tools [6]. Moreover, there are hardware libraries that provide reusable IP cores such as OpenCores

[7]. However, to use the constituents of these library-based approaches, the designer must become familiar with the contents of the library, or spend time repetitively browsing through the libraries. If a majority of the disadvantages are made transparent to the user during the design phase, reuse of existing hardware can be a viable solution to increase productivity for designers.

There are challenges in the domain of design entry in FPGA tools. Current design environments for FPGAs lack interactive features that can potentially assist and promote design reuse. For example, Microsoft's Intellisense [8] in Microsoft Visual Studio provides extensive features for software programmers that allow the user to find and access library elements and references. The user then does not have to leave the context of the page to look up the application programming interface (API) of the software components. Furthermore, Intellisense has an auto-complete feature that automatically inserts library elements into the code . The same effect can be had with hardware design. By having a compiled library or

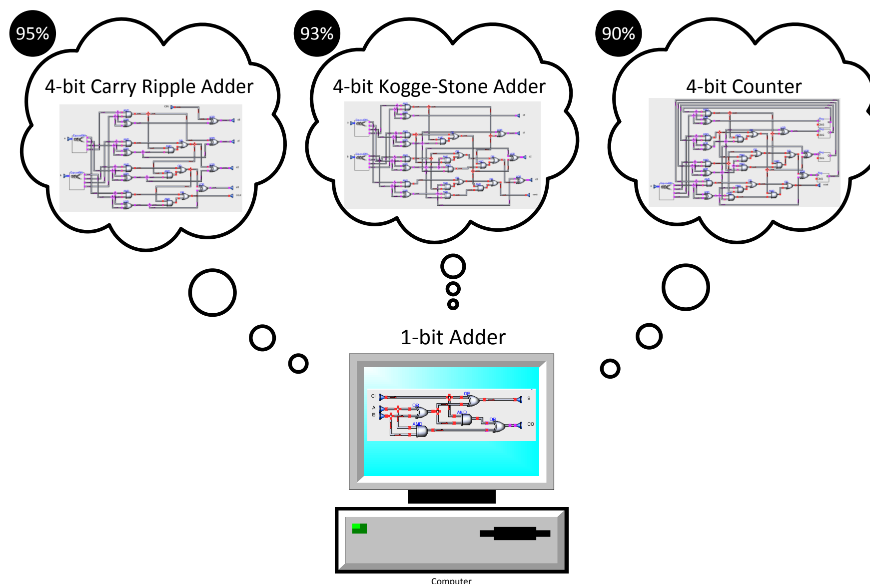


Figure 1.2: Proposed usage model

database that contains a variety of different circuits, the system can attempt to find circuits that have similar traits as the reference.

Use Case

A use case is presented in Figure 1.2. In Figure 1.2, an engineer is designing a new circuit from scratch using a design entry tool. As the user designs a circuit, the tool will continuously monitor the circuit design during the entry process. As the designer builds and modifies the circuit, the tool will perform a vast number of comparisons to see how closely the emerging design is to archives of circuits that have already been designed. Similar candidate designs are suggested and ranked similar to how Google's PageRank [9] ranks their Internet search results. By presenting similar circuit alternatives to the designer, the designer does not have to search through hardware libraries. In addition, the user is more likely to use the candidate circuits if they are suggested automatically. Even if the suggested circuits are not used, the results can be used as a guide during the design process. Therefore, the best way of improving the design process is to turn it into a discovery process. This thesis explores different possibilities for a back-end that determines the reusability of circuits in order to provide an environment where reusing existing hardware is essentially integrated into the design process, requiring little to no effort from the user.

1.2 Contributions

Determining suitable existing hardware for reuse requires a system that is able to compare a design against a multitude of patterns in order to search for a similar match. The system would require a library or database to store and keep track of the pattern circuits. These patterns can be compiled from a variety of sources such as Altera [10], HiTech Global [11],

OpenCores [7], or even custom circuits within the organization. The work presented in this thesis is an overview and proof of concept of a system that is able to determine hardware designs that the user can reintegrate given a reference design. Furthermore, extensions of the system for other applications are explored such as classifying and organizing circuits based on similarity to construct application-specific libraries. Different models for representing circuits as well as various methods for comparing circuits are discussed and analyzed. The idea of comparing two circuits does not necessarily imply that the search looks for an exact match between a circuit and pattern, but rather a similarity metric to decide how similar two circuits are. Therefore, the matches implemented will be focused on determine how similar two circuits are. A usage model and overview of the overall system is presented. Additionally, the system can potentially promote the idea of collaboration within the digital design community as users contribute and learn from one another.

1.3 Thesis Organization

The remainder of this thesis explores several possibilities to try and provide a suitable and efficient system to compare and match similar digital circuits for reuse. Chapter 2 looks at different ways circuits can be characterized as well as different methods to try and compare either the structure or function of the circuit. Chapter 3 outlines the overall back-end system for the circuit detection. The detailed implementations of the matchers are explained in Chapter 4. Chapter 5 presents the experiments and results of the different implementations. Finally, Chapter 6 concludes this work.

Chapter 2

Background

In this chapter, an overview of how digital circuits can be represented and compared is presented. The following sections introduce the idea behind circuit similarity matching as well as two common ways to model a digital circuit for comparison: structurally and functionally. Graph theory concepts and notations are provided in order to acquire a better understanding of the underlying algorithms.

2.1 IP Reuse

FPGAs are integrated circuits that can be reconfigured based on the specifications of the hardware designer. The flexibility of FPGAs makes it an attractive, quick prototyping solution for many designers. However, as designs become more and more complex, the overall design time can increase significantly. In some cases, the complexity may out-pace the productivity levels [12]. The design productivity gap mentioned in [1] shows that design capabilities are unable to keep up with the doubling of silicon density every two years according to Moore's Law. As a result, many engineering hours are spent designing complex

and large circuits. One solution to increase productivity is to reuse existing digital circuits, or IP cores.

Many existing designs are available for reuse whether they are from a third-party, a vendor, or internal to a company. Since many of these resources are located across various sources, many new as well as experienced designers are most likely unfamiliar of the resources available. Redesigning hardware components will increase the overall design time. Furthermore, searching through all possible sources for IP cores to reuse can be time consuming as well.

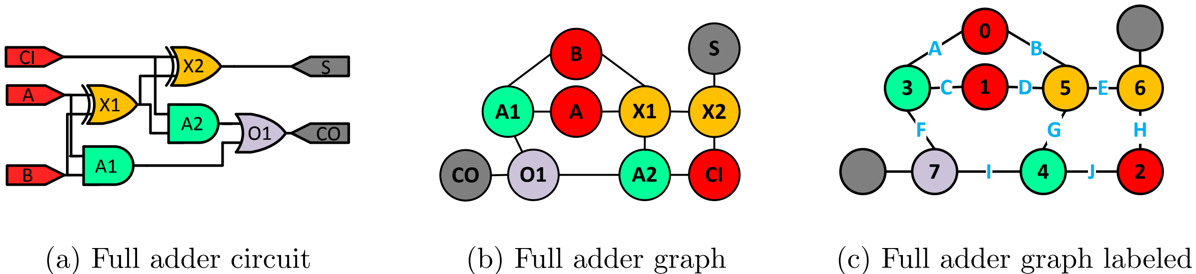
In order to suggest possible hardware to reuse, a comparison is performed between the reference circuit and the existing circuit. If two designs are similar, then a possible match for an existing design can be suggested. The idea is then extended to a database of pattern circuits where possible matches between the reference circuit and those in the database are determined. The following sections discuss possible methods of comparing two circuits.

2.2 Graph Representation

A common way to represent a circuit is with a graph, where the vertices represent the logic components and the edges represent the wires connecting the components together. A circuit and its graphical representation can be seen in Figure 2.1. There are many existing algorithms for extracting structural data from graphical representations. These algorithms can also be used to compare the how similar two circuits are.

2.2.1 Graph Isomorphism

When two graphs, C_1 (V_1, E_1, l_1) and C_2 (V_2, E_2, l_2), are structurally identical, a graph isomorphism is said to exist, where C is the graph, V is the vertex set, E is the edge set, and



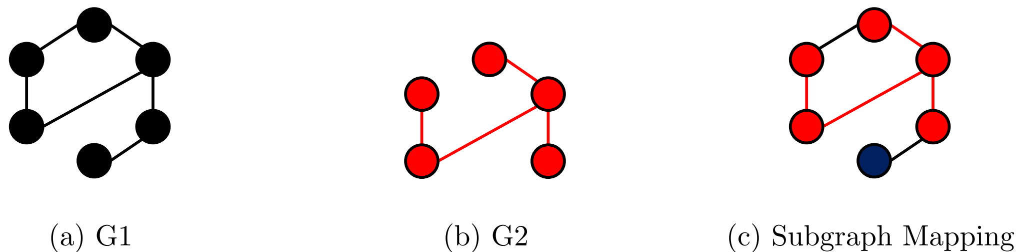
(a) Full adder circuit (b) Full adder graph (c) Full adder graph labeled

Figure 2.1: Full adder circuit and its graphical representation

l is the labeling function for the vertex set. In other words, there exists a bijective function that maps $C_1(V_1)$ to the $C_2(V_2)$.

The time complexity of graph isomorphism is not yet known. It is stated in [13] that the complexity in determining whether an isomorphism exists between two graphs is between P and NP complete. NP complete means that the time complexity increases significantly as the problem size grows. As a result, the problem cannot be solved within polynomial time. However, there have been a number of studies performed on algorithms that try to improve the timing of the isomorphism problem by reducing the search space. Ullmann's algorithm [14], one of the most widely used graph isomorphism algorithms for graph matching, attempts to reduce the search space by using a backtracking technique. In addition, the VF2 [15] algorithm has become popular because of its depth-first search technique used to prune the search tree for a more efficient search [16].

Graph isomorphism finds only exact matches and does not necessarily tell the user anything about the circuit such as what it is composed of or what circuit it might appear to be. The next section explores a more special case of graph isomorphism.



(a) G1 (b) G2 (c) Subgraph Mapping

Figure 2.2: An example of subgraph isomorphism

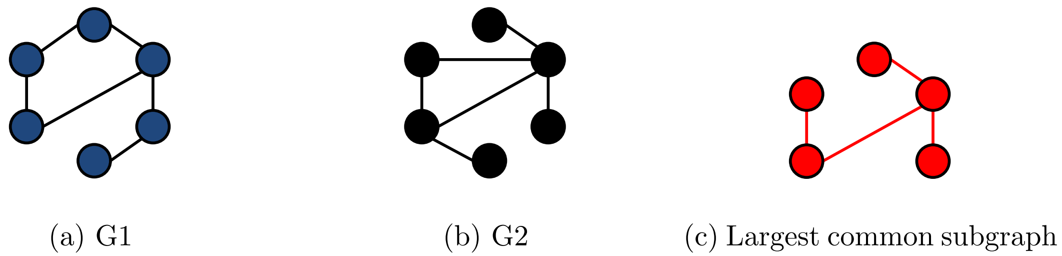
2.2.2 Subgraph Isomorphism

Given two graphs $G_1(V_1,\ E_1,\ l_1)$ and $G_2(V_2,\ E_2,\ l_2)$ with $|V_2| < |V_1|$, a subgraph isomorphism is said to exist between G_1 and G_2 if V_2 is a subset of V_1, where $V_2 \subset V_1$ and $E_2 \subset E_1$ [17]. In other words, this determines if the smaller graph, G_2, is contained with the larger graph G_1. The example in Figure 2.2 shows that Figure 2.2b is a subgraph of the circuit in Figure 2.2a.

Unlike graph isomorphism, subgraph isomorphism is NP-complete; however, by restricting the search space by applying a labeling function to the nodes and edges, the time complexity of the problem can be reduced. The algorithms listed in the previous section (Ullmann's and VF2) can not only detect graph isomorphism, but can also be extended for subgraph isomorphism. Like graph isomorphism, subgraph isomorphism is too restricting. Only circuits that make up or are made up of the reference are returned.

2.2.3 Maximum Common Subgraph

Maximum common subgraph (MCS) is a type of subgraph isomorphism where the largest subgraph that is common to two given input graphs is determined. This is extremely useful in circuit matching because there could be a large sub-circuit that two circuits may have in common, but neither is a sub-circuit of one another. An example of MCS is shown in Figure

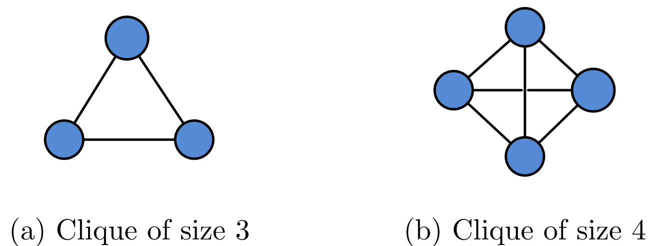


(a) G1 (b) G2 (c) Largest common subgraph

Figure 2.3: An example of MCS

2.3 where Figure 2.3c is the largest subgraph common to both the circuits in Figure 2.3a and Figure 2.3b. The size of the subgraph found relative to both the input graphs can then be used to calculate similarity between the two circuits.

The MCS problem can be described as a maximum clique problem. In order to find a MCS, the product graph of the two input graphs is needed. The product graph G_p of the two input graphs shows the possible compatibility between the vertices and edges of the two graphs. With the product graph, the MCS problem is then determining the largest clique of G_p. A clique is a graph where all the vertices are connected to every other vertex in the graph. An example of a clique can be seen in Figure 2.4. The clique problem determines if the there is a *complete* subgraph that exists in G_p of size k, where k is the largest complete subgraph. By finding the largest clique in G_p, the mapping of the common edges and nodes between the input graphs can be determined resulting in the maximum common subgraph. Bron and Kerbosch's (BK) algorithm [18] is one of the most efficient algorithms for clique detection



(a) Clique of size 3 (b) Clique of size 4

Figure 2.4: Two cliques of different sizes

available. Furthermore, there are variations of the BK algorithm that make the detection more efficient since maximum clique detection is considered NP-complete [19].

2.2.4 Graph Similarity

Graph isomorphism and subgraph isomorphism only find exact matches and any slight discrepancies between either of the input or pattern graph will result in a negative match found. However, since not one circuit is expressed and designed exactly the same, exact matching would not be desired. Noise, errors, and inconsistencies exist in the data the graphs are trying to model. Therefore, isomorphism, in general, is too strict and will fail to detect possible matches that are possibly similar. The idea behind graph similarity is that a vertex V_1 in graph G_1 is similar to V_2 in G_2 if the neighbors of V_1 and V_2 are similar [13]. In other words, a distance metric is calculated in order to depict the similarity between two circuits usually between the values of 0 and 1.

2.2.5 Edit Distance

Edit distance is one of the most widely accepted methods for error-tolerant graph matching. Given two graphs, edit distance is the total number of edits that are needed in order to transform one graph to the other. The edits can include adding and deleting vertices and edges, as well as the relabeling of vertices or edges. There is a cost associated with each edit operation. After all necessary edits are completed, the total distance between the two graphs is the total cost of the edits that were performed. The distance metric is then used as a way to indicate how closely related the two graphs are. The most widely used approach is the A* (A-Star) method, a best-first algorithm, which means that it will try to find the match with the lowest overall cost based on some heuristics [20]. Other heuristics have been applied to

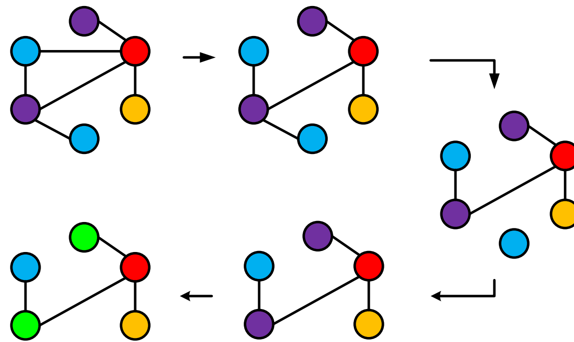
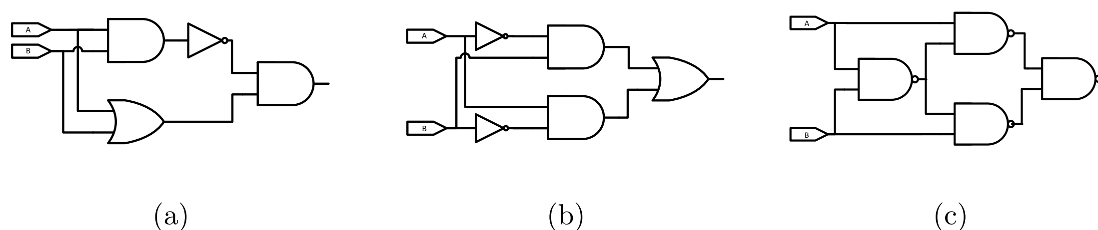


Figure 2.5: Edit distance transformation

the A* method in order to significantly speed up computation such as only expanding upon a certain number of paths that have the lowest cost. However, these variations of A* are suboptimal in that they return an approximate edit distance. Moreover, the complexity of edit graph is also NP-complete and was recommended by [20] to avoid graphs with more than twelve vertices. Therefore, applying graph edit to circuit matching would not be a probable choice considering circuit consists of hundreds if not thousands of logic components.

2.2.6 Spectral Graph Theory

The concept of using spectral theory on graphs is still fairly new and has been gaining a lot of popularity in research especially in the field of computer vision. Spectral theory uses adjacency and Laplacian matrices as well as eigenvalues and eigenvectors in order to characterize the structural properties of graphs [21]. The spectra of a graph (the eigenvalues) can provide ways to easily assess the similarity between two graphs when compared with the spectra of another graph. Despite its popularity, it is not yet widely accepted. One reason is because the spectra of a graph are not unique. Two completely different graphs can have the same exact spectra. Furthermore, small structural differences can significantly influence the



(a) (b) (c)

Figure 2.6: Three different implementations of a 2-input XOR gate

spectrum of the graph [22]. Nonetheless, spectral graph theory provides powerful methods to efficiently assess the structure of graphs.

2.3 Functional Isomorphism

Structural properties can be extracted from graphs that describe the topology of the circuit; however, graphs do not contain adequate detail to easily extract the functionality of a circuit. Two circuits may be structurally different but functionally isomorphic. Figure 2.6 shows an example of how circuits can be structurally different yet functionally the same by depicting three different implementation of the XOR gate using primitive gates. Figures 2.6a and 2.6b both use AND, OR, and NOT gates whereas Figure 2.6c uses only NAND gates. The topology and structure of all three circuits are different yet functionally equivalent.

2.3.1 Canonical Representation

One way to check for functional isomorphism is to transform the circuit into a canonical representation based on its logic equation. It is possible to transform any logic circuit to a circuit that uses just AND and NOT gates. By having this canonical form, the logic equations of the two circuits can be directly compared. However, this would make the circuit larger and more complex. On the other hand, binary decision diagrams (BDDs) provide a compact

and efficient canonical form. BDDs are directed acyclic graphs (DAG) that represent a single output of a boolean function. There are two sink nodes: 0 and 1 that indicate the result of the output given an input vector [23]. Two functionally equivalent circuits will produce the same BDDs; however, the input variable ordering would have to be the same for the two circuits in order for the BDDs to be identical. With an unknown circuit, there is no easy way of determining the variable ordering without trying all possible permutations. Heuristics may be applied to look for a feasible match; however, a feasible match is not guaranteed to be an actual match.

2.3.2 Rule-Based Detection

Rule-based detection using expert systems is another way to determine the functionality of a circuit. The idea is that for a given circuit, rules describe the function and behavior of the circuit. Rather than matching the circuit against every pattern in the database structurally, a match is considered if a similar set of rules is shared between the input and pattern [17]. This method makes using a rule-based system extremely favorable. Expert systems are designed to solve problems with knowledge and reasoning that a human expert might possess. The basis of an expert system is a knowledge base where all the rules (knowledge) are stored, and the inference engine is where the system ultimately decides on an answer from the rules in the knowledge base [24].

2.3.3 Boolean Isomorphism

The idea behind boolean isomorphism is that given two boolean functions f_1 and f_2, there is a permutation of the input variables such that the f_1 and f_2 are equivalent. To determine if the two boolean functions are equivalent, the truth tables of f_1 and f_2 are represented as

a hypergraph. A hypergraph is a special graph where the edges of the graph are a subset of the nodes [13]. After the hypergraphs are built, a hypergraph isomorphism is performed between f_1 and f_2 to determine if the two boolean functions are isomorphic. The boolean isomorphism problem is considered NP-complete with a time complexity of $2^{O(n)}$, where n is the number of vertices in the hypergraph [25].

2.4 Existing Work Related to Circuit matching

Since graph theory is heavily studied in many applications such as biology, social networks, chemistry, the Internet, and more, there is great interest in utilizing graphs as a way to characterize and model circuits for a wide variety of applications.

2.4.1 Circuit Matching: Gemini and SubGemini

The tool Gemini [26] uses graph isomorphism in order to validate layout versus schematic (LVS) in VLSI circuit designs. Given a circuit specification or schematic and a circuit layout, Gemini tries to determine if the two representations are isomorphic. Olrich et. al [27] further extends Gemini in a tool called SubGemini in order to identify sub-circuits that exist in a design. The goal of SubGemini is to make searching for sub-circuits technology independent so various CAD programs can utilize the tool. A new subgraph isomorphism technique was developed in order to be able to search through large circuits of over ten thousand transistors. Whitham [17] used SubGemini as a basis for an electronic circuit repository with improvements in the underlying data structure of SubGemini. The repository allows students who are learning about electronic circuits to easily search for complete or partial circuits that they have designed.

2.4.2 Reconfigurable Computing

Shi et. al [28] use an iterative similarity algorithm in order to compare design differences between iterations in order to speed up re-synthesis time. The idea is that the previous circuit that was synthesized is saved and compared with the newly synthesized design. By comparing the similarity of the two designs, the author is able to prevent unnecessary placing and routing of unchanged sections in the design.

2.4.3 Rule-Based Matching

Rule-based functional matching was extensively studied by Takashima et. al [29]. Their approach was to use rule-based detection only when the structure of two circuits failed isomorphism. By doing so, Takashima ensured that his program not only checked for structural isomorphism, but also functional isomorphism as well. This is because designers may adjust certain parts of their circuit for performance benefits where the structure of the circuit changes but functionally remains the same. On the other hand, functional isomorphism is slow in general and only feasible for smaller circuits. Therefore, functional isomorphism was avoided unless structural isomorphism returned negative. Eckmann et. al [23] use a rule-based system called OTTER (Organized Techniques for Theorem Proving and Effective Research) to assign functional meaning to circuits when given a detailed circuit description. OTTER takes facts, or features about the circuit and uses rules to canonicalize them in order to compare if the function of the circuit is similar. There are two canonical forms that [23] discussed: reduced ordered BDDs (ROBDD) and XOR and AND gate representation.

2.4.4 Combinational Equivalence Checking

Combinational equivalence checking (CEC) is a specific form of the formal equivalence checking focused on the verification of digital circuits. The idea of CEC is to compare the functionality and structural similarity between two digital circuits [30]. Typical implementation of CEC consists of using BDDs in conjunction with a satifiability (SAT) solver. SAT solvers can be applied to a wide variety of applications regarding electronic design automation [31]. They work by trying to find an assignment that can *satisfy* a certain assignment problem [32] such as the CEC problem.

2.4.5 Other Matching Techniques

There have been other numerous methods for graph matching that many researchers have also explored. Portegys from Illinois State University [33] used MD5 hashing as a way to store and quickly identifies graphs. Graph with identical hashes are isomorphic. Fuzzy attributed graphs (FAG) were used by Zhang and Wunsch in order to extract sub-circuits as a part of LVS in VLSI testing.

2.4.6 Optimizations by Reducing Search Space

In addition to graph matching algorithms, there has been research that tried to minimize the search space for pattern graphs in a database. Most applications require the ability to match an input graph against a database of pattern graphs. Since many of graph matching algorithms are computationally expensive, reducing the number of patterns is crucial in reducing the search time as well. The machine learning algorithm C4.5 was used in [34] as a way to extract certain features from the graphs in order to determine which graph could

match with a given input graph. The authors in [35] recursively decomposed the pattern graphs into subgraphs where these similar subgraphs are only represented once to prevent redundant searches. By having this simple database representation, the search can be quick and efficient. Whithams repository [17] takes advantage of the transitivity of sub-circuits to sort the circuits in a partial order so that if certain sub-circuits are not present, super-circuits of the sub-circuits will not even be considered as a possible match.

2.4.7 Reusing IP Cores

Reusing IP Cores is not a new concept. There have been great amounts of effort in increasing the re-usability of IP cores for FPGAs. OpenFPGA [36] has specific teams that work on application specific and core libraries for FPGAs such as their CoreLib. Their CoreLib group's primary goal is to develop and create a standard for hardware libraries in order to promote interoperability so that existing cores can be seamlessly integrated into FPGA tools. Open Core Protocol International Partnership (OCP-IP) [4] focuses on developing standards for the interfaces of IP cores. IP-XACT [5] also developed standards for defining IP cores so that they can be used with automated integration techniques. Furthermore, private IP cores may be available within a company that internal hardware designers could possibly leverage.

Vendors provide many pre-designed hardware modules that are optimized for their specific products. For example, Altera [37] provides a library of parameterized modules (LPM) whose purpose is to provide efficient designs of specific functions that are independent of the technology they are on. Such functions are commonly used operations such as addition or multiplication. Altera also designed Megafunctions which are IP blocks that are optimized specifically for Altera FPGAs [10]. Xilinx [38] also provides a unified library of numerous primitives and macros for their specific FPGAs. There are also open source communities

that provide IP cores for designers to use. OpenCores [7], the leading community currently for open source hardware IP cores, provides several hundred existing IP cores ranging from a wide range of applications such as cryptography and video controllers.

2.5 Summary

This chapter explained different matching techniques that can be used to characterize and compare digital circuits. Additionally, several different approaches were introduced to try to characterize the function of the circuit. However, because the time complexity of many of the algorithms described are exponential as the problem size grows, many heuristics are applied to try and limit the search space.

Chapter 3

System Implementation

This chapter introduces the basic structure of the overall system for hardware reuse. A high-level system overview and usage models are presented. The circuits will be modeled as graphs in order to utilize many of the graph tools and algorithms that already exist. From the graphs, the structure of the circuit can be analyzed and compared. The focus of the comparison is on the similarity between two circuits. The main focus of this work is the back-end circuit similarity matcher of the overall system.

3.1 System Overview

The overall system was designed as a proof of concept to promote the idea hardware reuse. In order to make the system as flexible as possible, the back-end is designed such that it can be easily integrated to a FPGA design environment. The high-level flow of the entire system can be seen in Figure 3.1. A reference design is converted to a netlist. The circuit netlist is then transformed into a graphical representation. Finally, the matcher compares

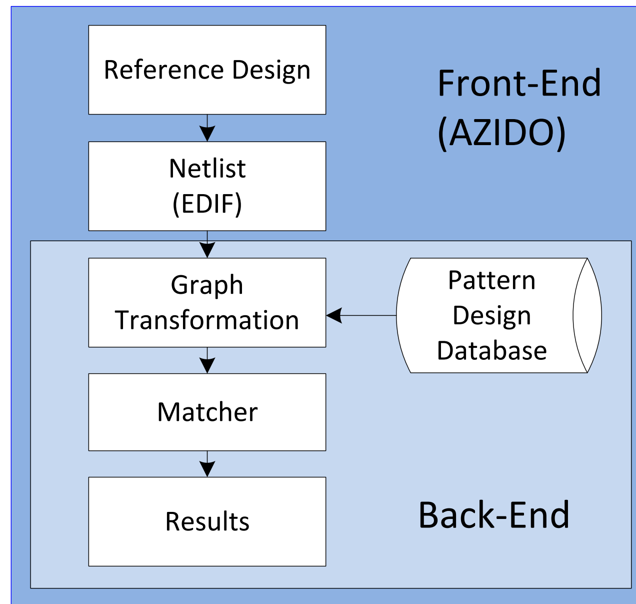


Figure 3.1: Flow diagram of overall system

the patterns in the database with the reference to determine how similar the reference and the circuits in the database are.

3.1.1 Front-End (Azido)

The design environment for hardware currently explored is the front-end graphical IDE called Azido [39]. Azido was chosen because of its unique interface and that it allows designers to reuse designs with an expandable library of predefined components. The user can also add custom hardware into the library for future use. Furthermore plug-ins are fairly simple to integrate into the overall environment. With a reference hardware circuit designed in Azido, a netlist can be generated and used to search for similar hardware.

The overall system was not designed and based around Azido. The back-end was designed

with a generic front-end in mind and therefore can potentially be extended to other tools such as National Instrument's Labview FPGA Module [40]. In addition to graphical environments, the back-end can be adapted to non-graphical environments as well such as the tools provided by Xilinx and Altera. As long as a netlist is generated, the back-end can be easily integrated with any front-end.

3.1.2 Netlist

The netlist that Azido generates is a generic netlist in Electronic Design Interchange Format (EDIF). EDIF is a standard format for representing a netlist of a design. Figure 3.2 shows the EDIF netlist for the full adder circuit described in Figure 2.1a. With the netlist, the topology of the circuit can be extracted and represented as a graph.

Just like the front-end, the netlist generate is not limited to EDIF. As long as the netlist contains all the interconnections between components, the graphical representation can be extracted.

3.1.3 Graphical Representation

With the netlist, a model can then be constructed by representing the circuit as a graph where the components of the circuits are vertices and the wires or nets as edges of the graph. The circuit will be represented as a directed acyclic graph. Parsing netlist files can be a daunting task due to the complexity of the format such as EDIF or IP-XACT. Furthermore, netlists may contain more information than is actually needed for similarity matching. Reading in these netlist can then be time-consuming especially if the database contains thousands of patterns. Therefore, the netlist is converted into a simpler representation that can be easily parsed and stored in memory.

3.1.4 Matcher

The main component of the system is the matcher. The matcher compares two circuits and determines if they are similar in structure. Several graph matching algorithms are implemented and compared. Algorithms include a custom subgraph isomorphism, MCS, and decomposition subgraph isomorphism. Subgraph isomorphism VF2 was integrated in as a comparison to see how well the implemented matchers are. A reference circuit will be passed into the matcher in a simplified graphical format. The matcher will then compare the design against the patterns in the database. After the circuits are compared, the results

```
(cell COUNTER
  (cellType GENERIC)
  (view net
    (viewType netList)
    (interface
      (port CLK(direction INPUT))
      (port IN1(direction INPUT))
      (port OUT(direction OUTPUT))
    )
    (contents
      (instance A1 (viewRef net (cellRef AND2)))
      (instance N1 (viewRef net (cellRef NOT)))
      (instance O1 (viewRef net (cellRef OR2)))
      (instance A2 (viewRef net (cellRef AND2)))
      (instance A3 (viewRef net (cellRef AND2)))
      (instance CLOCK (viewRef net (cellRef VCC)))
      (instance DFFE (viewRef net (cellRef DFFE)))
      (instance INPORT1 (viewRef net (cellRef VCC)))
      (instance CLR (viewRef net (cellRef VCC)))
      (net OUT(joined (portRef O(instanceRef A3)) (portRef OUT)))
      (net NET1(joined (portRef O(instanceRef A2)) (portRef D(instanceRef DFFE))))
      (net IN1(joined (portRef IN1) (portRef I0(instanceRef A1)) (portRef I0(instanceRef O1)) (portRef I0(instanceRef A3))))
      (net NET2(joined (portRef O(instanceRef A1)) (portRef I(instanceRef N1))))
      (net NET3(joined (portRef O(instanceRef N1)) (portRef I0(instanceRef A2))))
      (net NET4(joined (portRef O(instanceRef O1)) (portRef I1(instanceRef A2))))
      (net CLK(joined (portRef CLK) (portRef CLK(instanceRef DFFE))))
      (net NET5(joined (portRef Q(instanceRef DFFE)) (portRef I1(instanceRef A1)) (portRef I1(instanceRef O1)) (portRef I1(instanceRef A3))))
      (net NET6(joined (portRef VCC(instanceRef CLR)) (portRef CLRN(instanceRef DFFE))))
      (net NET7(joined (portRef VCC(instanceRef INPORT1)) (portRef PRN(instanceRef DFFE))))
      (net NET8(joined (portRef VCC(instanceRef CLOCK)) (portRef ENA(instanceRef DFFE))))
    )
  )
)
```

Figure 3.2: EDIF netlist of a 1-bit counter

are displayed for the user. Suggestions are ordered based on how similar the patterns are to the current design of the user.

3.2 Netlist to Graph Converter

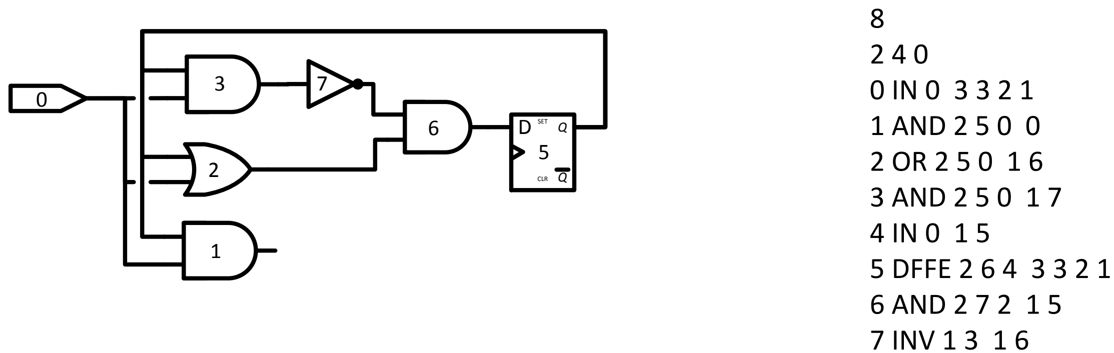
The format of the netlist Azido generates is EDIF. EDIF format contains bloated information and is organized in a way that makes it difficult to parse quickly. A simpler representation is needed to reduce the overall execution time.

3.2.1 TORC

In order to quickly develop a parser program, the Tools for Open-source Reconfigurable Computing (TORC) framework was used [41]. TORC contains tools that allow users to efficiently access and manipulate generic netlists, such as EDIF, by importing the netlist data into an organized data structure. Once the EDIF has been imported, the user has access to all the design elements described in the EDIF and can then re-describe the data in a more condensed graphical format.

3.2.2 Graph Format

An example of the simplified graph format can be seen in Figure 3.3b describing the circuit shown in Figure 3.3a. The first line indicates how many components make up the circuit and the second line has the total number of inputs followed by the vertex number of all the inputs. Since circuits are considered directed graphs, the output nodes are omitted and only the input nodes remain intact. The format of the remaining lines indicate the vertex number, the component type, number of inputs, inputs, number of outputs, and the outputs



(a) Counter circuit (b) Simplified graph format of counter

Figure 3.3: Simplified representation of counter circuit

respectively. The simple format allows for the back-end to efficiently and quickly parse in the graphical data, especially for languages such as C and C++.

The conversion from EDIF to the simplified graph format is only performed once as a pre-processing step. However, the reference circuit will have to be converted from EDIF to the simplified format each time the updated netlist is generated by Azido. After the conversion, the pattern graphs are stored into a database where they will be later compared to the input graph by the matcher.

3.2.3 I/O Ports

Every single component that a circuit is made up of is a logic component. On the other hand, I/O ports are technically not considered as a logic component and therefore, have to be treated differently. One reason is that output ports can be ignored completely during the matching process because the graphical representation is directed; however, inputs are different. The entire circuit may be disconnected if input ports are removed. Moreover,

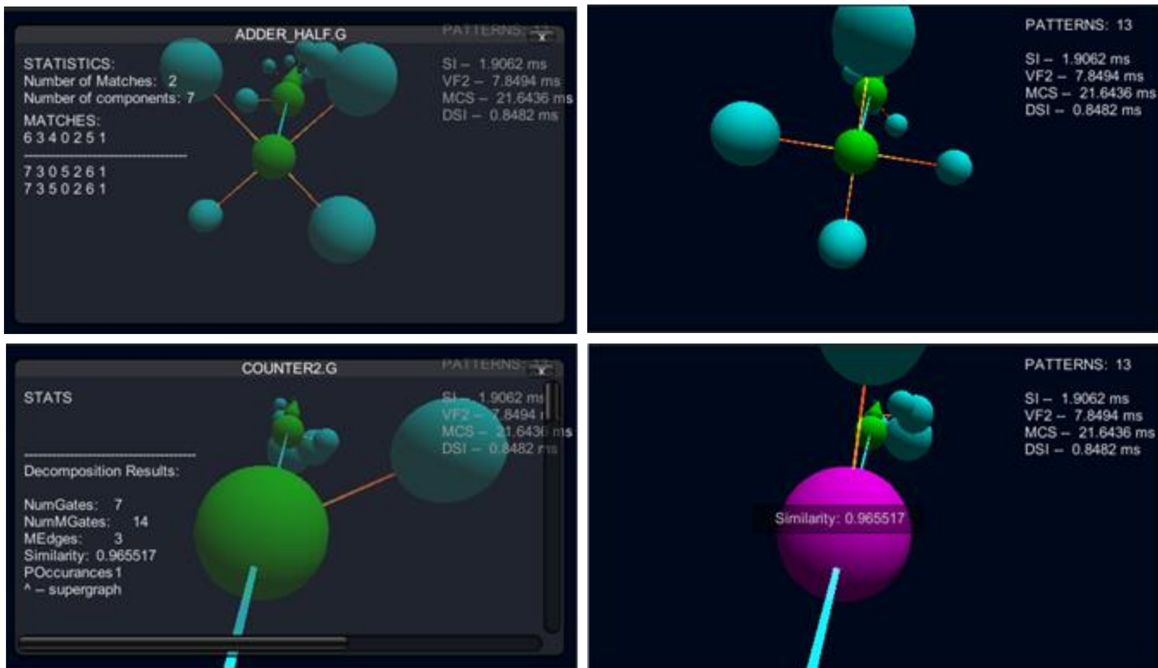


Figure 3.4: Screenshots of the interactive interface for displaying results of the matcher

input ports can be matched to any logic component if a circuit is a sub-circuit of a larger one. Details on the three different matchers will be discussed in the next chapter.

3.3 Displaying Results

After the matchers complete the search to find circuits similar to the input, the results are passed to the output to be displayed for the user. The matches returned are ranked based on how similar the pattern circuit and the input are with the best matched circuit listed out front. In order to provide a better sense of similarity, a graphical representation of the results was explored. The game engine, Unity [42] was used to develop the interactive interface. Game engines contain libraries and tools that make game development simple such

as renderers for 2D and 3D graphic. In this case, the tools were used to create a visualization of the similarity between an input circuit and a multitude of patterns.

The results from the matcher are saved to a file. Each time the file is updated, the interface will read in the results and update the display. Circuits with the same similarity ranking are grouped together with each circuit presented as nodes. Going down the structure displays circuits with consecutively lower similarity. Matcher execution statistics are shown. Furthermore, additional information on the matching nodes, mapping of logic gates, and other statistics can be viewed. By having a visual aid and an interactive interface, data is compressed and expressed in a manner where the user will be able to identify and compare circuits quickly and easily. A video demo can be seen below.

Figure 3.5: Video demo of the interactive interface. Click to play media

Chapter 4

Matcher Implementation

This chapter explains the details of the four main different graph comparison algorithm implemented. Data structures used are described in detail.

From the database, each of the pattern graphs needs to be compared with the reference design. The importer of the matcher will import both the reference and pattern circuit. Afterwards, the two graphs will be passed into the different comparators where the similarity of the two circuits will be determined. The results of the comparator will be displayed on the output so that the user will be informed of possible similar designs. The block diagram of the matcher can be seen in Figure 4.1. The core of the matcher is implemented in C++ in Ubuntu 12.04 on a Dell Vostro with a 2.8 GHz Intel Core2 Duo processor and 2.9 GB of RAM.

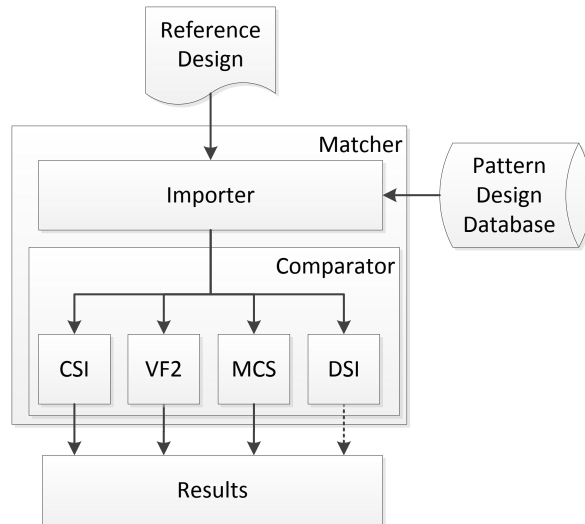


Figure 4.1: Block diagram for matcher implementation

4.1 Data Structure

There are two main ways of representing graphs: adjacency lists and matrices, and incidence lists and matrices. Each type of representation has their own benefits in terms of efficiency depending on the operations that are needed to be performed. Other structures include Laplacian matrix which is primarily seen in spectral graph theory.

4.1.1 Adjacency List and Adjacency Matrix

Adjacency list and matrix both indicate the adjacent vertices of a specific vertex. In an adjacency list, each vertex of the graph contains a list of all other vertices that are adjacent to it. Therefore, the size of an adjacency list is $V + E$. However, since the graphs are directed, only the nodes that the edges direct to are shown. The adjacency matrix is of size $V \times V$ and has a *one* in the cell if the two vertices are connected to each other by an edge with the row vertices as source vertex and the column vertices as the destination vertex.

0	3	5
1	3	5
2	4	6
3	7	
4	7	
5	4	6
6		
7		

(a) Adjacency list

	0	1	2	3	4	5	6	7
0	0	0	0	1	0	1	0	0
1	0	0	0	1	0	1	0	0
2	0	0	0	0	1	0	1	0
3	0	0	0	0	0	0	0	1
4	0	0	0	0	0	0	0	1
5	0	0	0	0	1	0	1	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0

(b) Adjacency Matrix

Figure 4.2: Adjacency list and matrix of full adder circuit

A	0	3
B	0	5
C	1	3
D	1	5
E	5	6
F	3	7
G	5	4
H	2	6
I	4	7
J	2	4

(a) Incidence list

	0	1	2	3	4	5	6	7
A	-1	0	0	1	0	0	0	0
B	-1	0	0	0	0	1	0	0
C	0	-1	0	1	0	0	0	0
D	0	-1	0	0	0	1	0	0
E	0	0	0	0	0	-1	1	0
F	0	0	0	-1	0	0	0	1
G	0	0	0	0	1	-1	0	0
H	0	0	-1	0	0	0	1	0
I	0	0	0	0	-1	0	0	1
J	0	0	-1	0	1	0	0	0

(b) Incidence Matrix

Figure 4.3: Incidence list and matrix of full adder circuit

Figure 4.2 shows the adjacency list and the adjacency matrix of the graph shown in Figure 2.1c.

4.1.2 Incidence List and Incidence Matrix

The incidence list and matrix describes the edges of the graph and indicates the vertices that are connected to each edge. For the matrix, it is a $V \times E$ size matrix where the value *1* represents the vertex that the connected edge is going into and the value *-1* represents the vertex that the connected edge is going out of. Since edges only have a source and destination vertex, the incidence list just contains two entries for each edge with the first column as the source vertex and the second as destination vertex. Therefore the size of an incidence list is $2 \times E$. Figure 4.4 shows the incidence matrix. The incidence list of the graph is shown in Figure 2.1.

Depending on the algorithm, one representation may be more efficient in retrieving data than the other. Lists are a more compact representation which makes it more efficient than matrices in terms of space. Adjacency lists takes *O(V + E)* of space as opposed to $O(V \times V)$ for the adjacency matrix. The incidence list takes $O(2 \times E)$ of space where the incidence matrix occupies $O(V \times E)$. Since circuits are considered sparse graphs, the list data structure is preferred. However, which matching algorithm used will depend on whether the adjacency list or incidence list is chosen.

4.2 Custom Subgraph Isomorphism

One of the graph comparison methods that are going to be implemented and compared is subgraph isomorphism. Generic subgraph isomorphism algorithms such as VF2 [15] or Ull-

mann's [14] are both widely used and accepted. There exists many variations of subgraph isomorphism tailored to specific applications by adding certain heuristics and/or optimizations such as [27], [43], and [44]. VF2 and Ullmann are both guaranteed to return exact matches if one exist. The nature of the custom implemented algorithm will not be as strict as VF2 or Ullmann. The algorithm focuses more on the logic component and its neighbors rather than the edges that connect them and will return structures that are similar. For the custom subgraph isomorphism algorithm (CSI), the adjacency list representation was used for subgraph isomorphism because no information about its edges is needed and only the neighboring nodes are analyzed.

4.2.1 Candidate Vector

The implementation of the CSI takes from [27] and [15]. First a candidate vector is determined between two circuits, G_1 and G_2. The candidate vector is a list of possible matches between the vertices in G_1 and G_2, or a candidate pair. By finding the candidate vector, only the most likely matches are explored, limiting the search space. The candidate pairs are determined by the most uncommon logic component between the two circuits. The more unique the component in the circuit, the more likely that a match can be found with the least amount of searching. Therefore, the candidate vector contains the components that appear the least in the input circuit. For example, for G_1 and G_2 shown in Figure 4.4, the candidate pairs in the candidate vector would be 5 and 11, 5 and 13, 1 and 2, and 1 and 7 because the inverter and OR gates are the only two gates that are most uncommon between the two circuits. Therefore, the four candidate pairs are used as a starting point when testing for a subgraph isomorphism.

Page Rank

Other possible methods for determining possible starting points were explored. PageRank [9] by Google, primarily seen with searching for web pages, determines the overall importance of a web page based on the number of sites that goes out from it and the number of sites that comes into it. However, based on how the algorithm works, the PageRank of a node is largely affected by the PageRank of the nodes that come into a web page. Therefore, the nodes close to the output of a circuit always have a higher PageRank than those near the input. Feedback loops will increase the PageRank for that node, but not all circuits will have feedback loops. For this reason, PageRank is not a suitable method for determining which node of a circuit is most important.

4.2.2 Determining Isomorphism

With each candidate pair as a starting point, the two graphs can now be compared. To do so, both G_1 and G_2 need to be traversed. Traversal is done recursively so that if a mismatch is found, the matcher can backtrack to a valid state and check the next possible node for a match. At each level of traversal, the type of the logic component is compared. If the component of the nodes in question in G_1 and G_2 are identical, then the match counter is incremented, the node is marked, and then next node is compared.

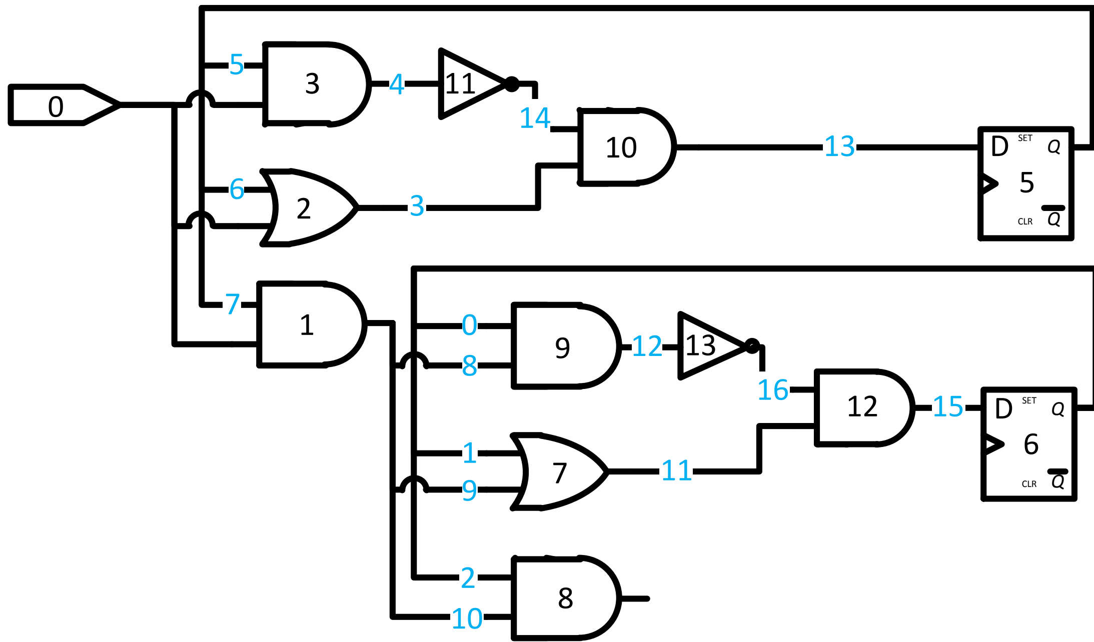
Each matching node is marked so that if a feedback loop exists, it doesn't pair a vertex with an existing match. The next nodes to be searched are determined by looking at all the connected outputs. If there is more than one possible match, the matcher peeks at the next level down and decides which path to take so that the input and pattern matches. If the matcher reaches a dead end, it backtracks and attempts to find a valid matching pair to continue searching. In addition, if the matcher finds a mismatch, not only does it backtrack,

but it also pops the last result off the matching node list. The matcher continuously scans the nodes until the match counter reaches the number of nodes in the smaller graph. This indicates that all the nodes in the smaller graph were successfully mapped to the nodes on the other graph and a subgraph exists. If all possible paths have been scanned and the number of nodes in the smaller graph does not match the match counter, no subgraph isomorphism is found. Furthermore, this algorithm can also be extended to graph isomorphism as well. If the match counter and the total number of nodes in the pattern are equal and the total number of nodes in the pattern is the same as the number of nodes in the input graph, then an isomorphism exists.

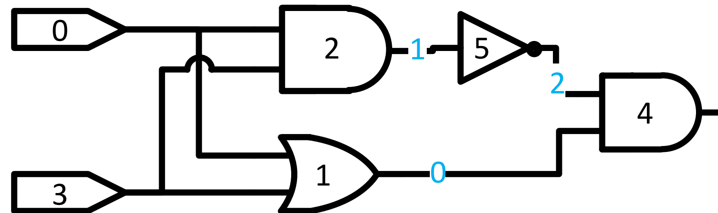
Due to how the traversal and backtracking is done, the subgraph matching has heuristics applied that won't necessarily return an exact match and matches that exist may not be an actual result. Because of this, it may give the user more information about a circuit due to the inexact nature of the search. Additionally, CSI can be extended to calculate a similarity metric between two circuits based on the traversal during the search.

4.2.3 Similarity

The similarity of the circuit can be determined using CSI algorithm; however, the problem is that CSI is not guaranteed to find a match. Just like the matches, the similarity will depend on the candidate pairs that are determined initially. Therefore, if the candidate pair determined initially is completely off, then it will assume that there is no similarity between the input and pattern. On the other hand, if the candidate pair is a good match, then even if the circuits do not match, a reasonable similarity metric will be returned. The similarity metric is calculated by the ratio of the number of nodes in the best possible match between



(a) Two-bit counter (G_1)



(b) Two input XOR gate (G_2)

Figure 4.4: Example Circuit

the input and pattern and the size of the pattern graph. The equation to calculate the similarity is shown in Equation 4.1.

$$\frac{sizeof(bestPossibleMatch)}{sizeof(patternCircuit)} \tag{4.1}$$

4.2.4 VF2

The VF2 algorithm [15] was added as a fourth method into the list of matchers. VF2 is used more as a basis for the comparison between the different methods implemented. The implementation of VF2 is provided by Boost libraries [45]. In order to utilize the algorithm for matching digital circuits, the algorithm was slightly modified to take into account the nodes that represent inputs. Because inputs can map to any logic component, it automatically maps the node as a feasible match and continues the search.

4.3 Maximum Common Subgraph

MCS allows for a more inexact form of matching circuits by determining the largest circuit common to two graphs G_1 and G_2. Different from the CSI, the incidence list representation was used. MCS relies on the possible edge matches to determine if a match exists. In other words, if two edges have the same source and destination type, then it is said that the two edges are compatible. These compatibilities are used to form the compatibility graph. The algorithm implemented is based on [19].

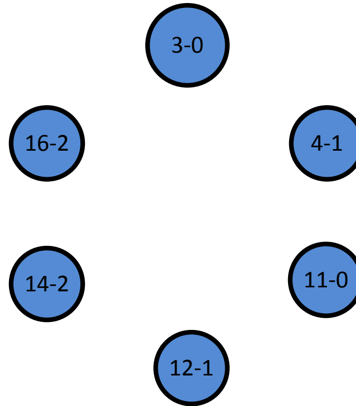


Figure 4.5: Nodes of the compatibility graph

4.3.1 Compatibility Graph

The compatibility graph is a graph that indicates the adjacencies of common edges given two graphs G_1 and G_2. The nodes of the compatibility graph are all possible edges in G_1 whose edges have the same source and sink component type as the edges in G_2. For example, Edge 3 in Figure 4.4a would be compatible to Edge 0 in 4.4b because the edges have a common source and sink type. By finding all the edges that are compatible, a compatibility graph of six nodes is formed as shown in Figure 4.5. With the nodes of the compatibility graph, the relationship between these nodes need to be determined by finding the modular product of G_1 and G_2 with respect to the edges indicated in the nodes of the compatibility graph

4.3.2 Modular Product

The modular product of two graphs is used to determine the relationship of the nodes in the compatibility graph. Two vertices in the compatibility graph, cv_1 and cv_2, are adjacent if edge e_1 of cv_1 is incident on the same vertex as e_1 of cv_2 and e_2 of cv_1 is incident on the same vertex as e_2 of cv_2, or both e_1 and e_2 of cv_1 are not incident to e_1 and e_2 of cv_2 respectively.

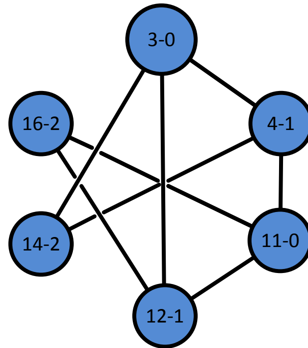


Figure 4.6: Complete compatibility graph

The modular product of G_1 and G_2 with respect to the nodes in the compatibility graph is shown in Figure 4.6.

4.3.3 Clique Detection

After the modular product of G_1 and G_2 has been determined, the cliques in the compatibility graph needs to be found. The largest cliques found are the largest subgraphs that are common to both G_1 and G_2. The clique detection algorithm implemented is a variation of a recursive backtracking algorithm called Bron and Kerbosch (BK-algorithm) in [18]. A variation of

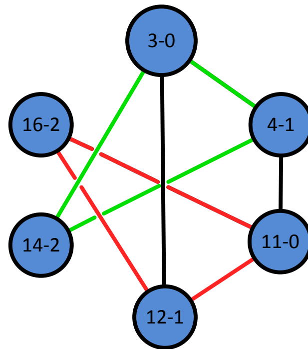


Figure 4.7: Largest cliques of the compatibility graph

the BK-algorithm in [19] was used in order to reduce the amount recursive calls required. The largest cliques detected by the BK-algorithm are shown in Figure 4.7. The nodes of the compatibility graph represent the matching edges between the input and the pattern. Therefore, in Figure 4.7, the green clique represents a match found with the edges 3, 4, and 14 in G_2 matching 0, 1, and 2 in G_1 respectively. The same can be seen with the matches in the red clique.

4.3.4 Similarity

The largest common circuit of the input and pattern circuit can be used to determine a similarity metric. The similarity is calculated by finding the ratio between the size of the largest common circuit and the size of the input circuit. The same ratio is calculated again but with the size of the pattern circuit. The larger ratio of the two is chosen as the similarity between the input and the pattern circuit. The equation is shown below in Equation 4.2.

$$max\left(\frac{sizeof(MCS)}{sizeof(inputCircuit)}, \frac{sizeof(MCS)}{sizeof(patternCircuit)}\right) \tag{4.2}$$

4.4 Decomposition

The CSI and MCS algorithms both work on two graphs at a time. Therefore, to determine a similar reusable design from a database of patterns, the input circuit will have to be compared individually with each pattern. As the number of pattern grows, the search time will increase with the size of the patterns. Also, if the patterns become increasingly complex, then the search time will rise significantly as well.

The idea of decomposition subgraph isomorphism (DSI) is where the pattern graphs are

recursively decomposed by a decomposer into smaller graphs until only one vertex remains. Patterns with similar subgraphs are represented once by reusing the same subgraph in their decomposition. The entire decomposition hierarchy of the pattern database is stored in a decomposition tree structure. Each node in the tree is a decomposition of a pattern graph. After the all the pattern graphs have been decomposed, the matching subgraphs and be searched for. Subgraph detection is done by trying to combine and build the circuit from the bottom up. The general algorithm implemented is described in [35]. The rest of this section discusses the basics of the algorithm as well as improvements and modifications to the existing algorithm to incorporate similarity matching for the application of matching digital circuits

4.4.1 Decomposer

Before searching for matching circuits, the entire database of patterns needs to be decomposed. By decomposing the patterns, circuits with similar subgraphs can be represented once leading to a more compact representation of the database. Furthermore, the new tree-like structure will allow a more efficient search for similar circuits. The decomposition of the pattern database is a one-time incremental procedure that is done offline where a pattern is decomposed into smaller subgraphs and added to the decomposition tree. Moreover, additional patterns can be added without having to re-decompose the entire database. The first step to decomposition is the partitioning of the circuits.

Partitioner

The partitioner decomposes or partitions the pattern into two connected graphs. The partitioner chooses a vertex at random and then traverses the circuit until the number of vertices

traversed is equal to half of the total number of vertices in the circuit. This will create two partitions fairly equal in size. The first partition is the part of the circuit that was traversed initially and is therefore a connected graph. To ensure that the second partition is connected, the second partition is traversed from a starting point. All the nodes that are not connected are moved to the first partition. This can be guaranteed because if the nodes are not connected to any of the nodes in the second partition, they have to be connected to nodes in the first partition assuming the pattern graph itself is connected. Once there are two connected partitions, the decomposer then recursively partitions the already partitioned subgraphs until there is only a single vertex left.

Decomposition Tree

All decompositions decomposed by the partitioner are stored in a decomposition tree. The tree structure shows how the each subgraph is decomposed and will provide an effective search method for finding matches and similarities. Figure 4.8 shows the decomposition tree of a NAND, NOR, and XOR gate. The decomposition nodes with the blue rim indicate that decomposition is a complete pattern. Furthermore, the decompositions are related to each other by an edge set. The edge set is a set of edges that are removed in order to partition the subgraphs into two smaller graphs. In other words, the edge set indicates how the two partitions are related to each other. Since there are multiple patterns in the database, the ID that is assigned to each vertex is unique. However, when subgraphs share the same decomposition, a mapping is needed to keep track of the matches that are found. If no unique labeling is given, then there is no way of differentiating which vertex the source or destination node is given two vertices that have the same ID.

The decomposition tree generated by the decomposer is not unique. If the order of the patterns being decomposed is rearranged, this would generate a different decomposition tree.

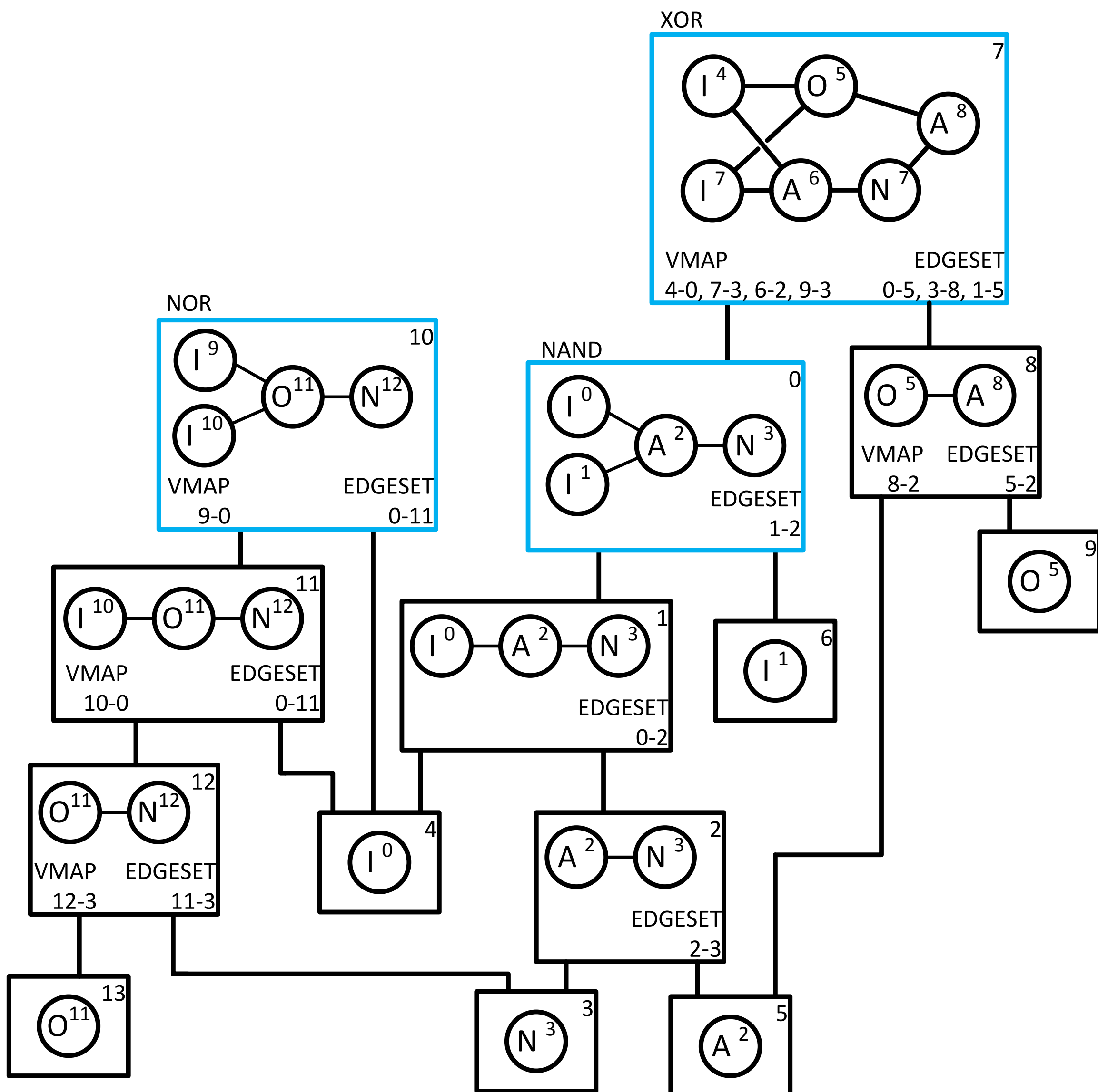


Figure 4.8: Decomposition of NAND, NOR, and XOR gate

Because the tree is not unique, the performance of the matcher may vary during the search. The structure of the decomposition tree all depends on how the partitioner partitions the circuits. On the other hand, according to [35], despite the decomposition not being optimal, there is little to no significant influence to the performance of the matching.

4.4.2 Subgraph Isomorphism

At each level of the decomposition process, the new partitions are matched with each node in the decomposition tree to see if there exists an identical subgraph. This will make it so that identical subgraphs of the patterns will be represented once to provide a compact representation of the database. Consequently, this will make the search for matches more efficient due to a smaller search space. If a subgraph match exists with a decomposition node, then the decomposer will link the first partition as the node with the same subgraph and the other as the difference between the current decomposition and the subgraph match. However, if the difference causes the second partition to become disconnected, the subgraph is ignored. All possible subgraph matches are tested and if no possible decomposition can be used, then the subgraph is partitioned using the partitioner. The subgraph isomorphism method used is the CSI algorithm.

4.4.3 Matching

When the patterns in the database are all decomposed, the core of DSI can then be performed given an input circuit. An example is shown in Figure 4.9. There are three states which each decomposition can reside in: unsolved (orange), dead (red), or alive (green). Each decomposition node is labeled unsolved initially. The search first focuses on the decompositions that consist of a single vertex. For each of the decompositions with a single

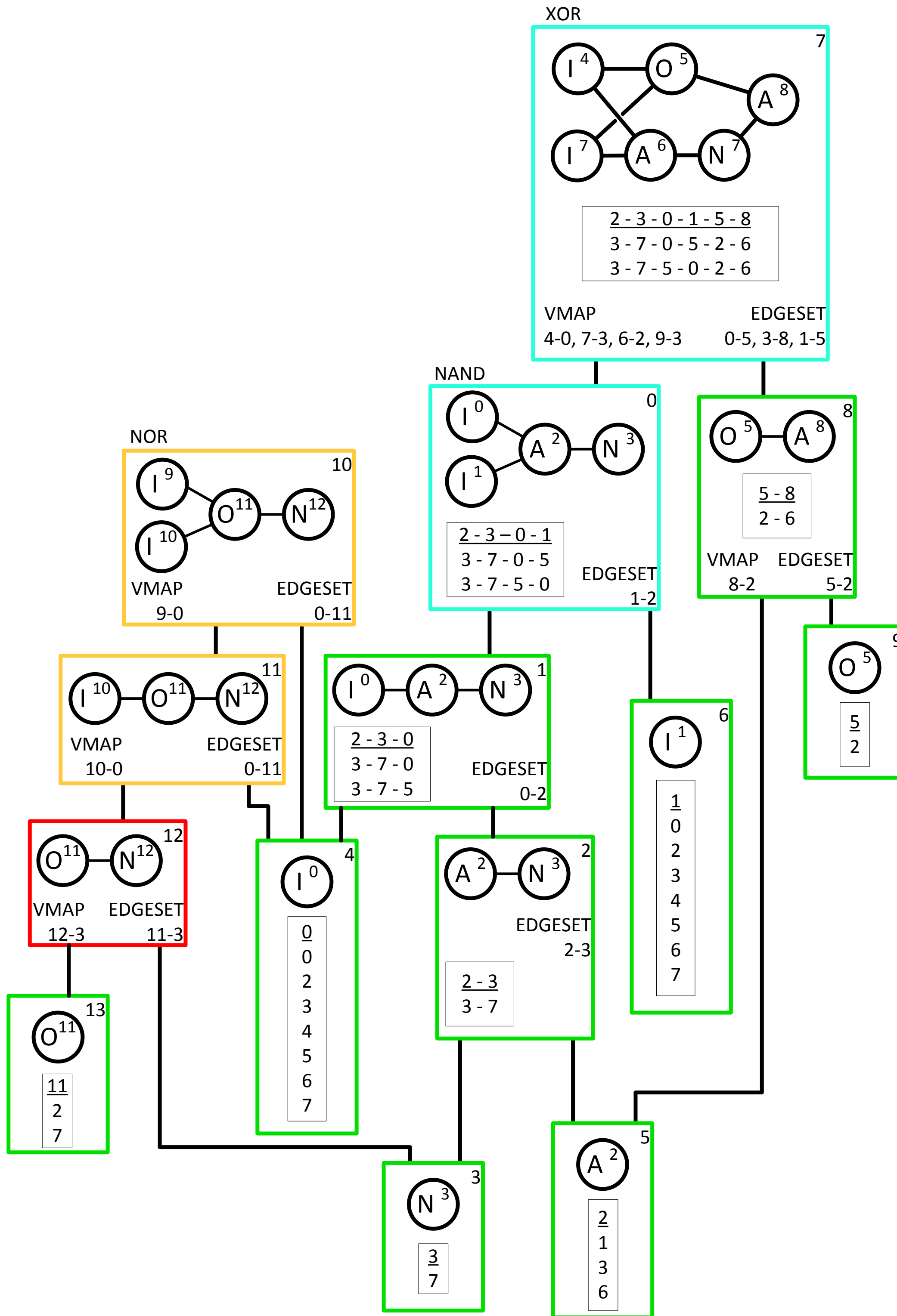


Figure 4.9: DSI Matching with counter as input circuit

vertex, the vertices with the same component type in the input graph is added to a list of possible matches. The mapping seen in Figure 4.9 is from the input 1-bit counter circuit depicted in Figure 3.3a.

The decomposition node is then marked alive if a mapping is found. An unsolved decomposition node contains a possible match if its two children nodes are both marked alive. The matcher then tries to combine the mapping of the two decomposition nodes by analyzing the edge set. If the edge set between the two partitions are all contained in the input graph, then the node is marked alive, otherwise, it is marked dead. As seen, since there is no edge from an OR gate to an inverter, Node 12 is marked dead. This prevents further decomposition nodes up the tree to be searched. The reason is if the decomposition is not a subgraph of the input graph, then any graph with the same decomposition subgraph is not a possible subgraph of the input graph. When no more unsolved decomposition nodes can be combined, all the decompositions that are marked alive are subgraphs of the input graph. The decomposition nodes that contain the full pattern circuit that are marked alive are passed to the output as indicated with the cyan border in Figure 4.9.

Similarity

A similarity metric can be derived from the result of the matching. From each full pattern graph in the decomposition tree, the distance from the current position to the first node that is alive can be used as the distance metric. Furthermore, the number of matching nodes and the number of edges missing are used to further refine the similarity metric. Equation 4.3 below shows the similarity metric used.

$$\frac{2 \times matchingNodes}{2 \times subgraphSize + \frac{missingEdges}{4}} \tag{4.3}$$

A heavier weight is placed on the matching nodes that were matched rather than the missing edges. This is because the nodes that were matched exist already somewhere in the circuit. However, just because the nodes of the circuit match the input, the interconnections between the nodes might not be the same. Therefore, the missing edges variable was added to the metric to differentiate the interconnections.

The similarity metric is not the same for every decomposition tree and depending on the decomposition of the pattern graphs, the similarity metric might be entirely different. One decomposition may provide a better metric than the other. However, the metric does not differentiate much when tested with two different decompositions when the patterns were rearranged. Therefore, the metric can be used to provide decent information as to how closely related the two circuits may be.

Optimizations

Several optimizations were put into place to try and speed up the overall decomposition and the matcher.

- After the pattern database has been decomposed, the decomposition tree goes through a simplification process. Single vertex decompositions are merged and remapped. This will save significant amounts of memory especially for larger circuits.
- Nodes that go straight to output are not included in the initial mapping for the single vertex decompositions that contain the input component. This is because inputs output a signal into the circuit. Nodes without any outputs cannot be input nodes and are omitted to help limit the number of comparisons performed.
- Instead of constantly searching through all of the decompositions for a node that is alive from the beginning after one is found, the program will try and look ahead in the

tree to see if the decomposition above can be combined. This way, the matcher does not have to search again from the beginning for a node that is alive.

- The reason why the decomposition was limited to connected graphs was due to the matching process. If the graphs are disconnected, every possible combination that can be made with the disconnected graphs has to be mapped. This is because the relationship between the disconnected graphs is unknown. Furthermore, depending on the standard component library used, the number of different logic components is very few compared to the number of total components in a circuit. If an AND gate and an OR gate are disconnected with the input circuit containing a hundred of each gate, there would then be a possible match of 100,000 because the relationship between the two partitions is unknown. Therefore, all possible mappings between every AND and OR gate have to be taken into account which can greatly reduce the efficiency of the matcher during run-time. By limiting the decomposition to connected graphs, the matcher will be much more efficient with a much more limited search space. Even though the decomposition will be more complex with a larger decomposition tree, the performance of the matcher is greatly increased.

Chapter 5

Results and Analysis

This chapter discusses the results obtained from the implemented matching algorithms with respect to digital circuits. The accuracy and performance of each algorithm are analyzed.

5.1 Benchmark

A simple benchmark was constructed and used for initial testing of accuracy and functionality. The initial benchmark contains about 20 circuits ranging from 8 to 50 components. The circuits designed consist of logic gates, arithmetic operators, and counters. All circuits were designed and tested using Azido. Additional information about the benchmark can be found in Appendix A. Two different databases of different size were constructed from these circuits. Databases using circuits from the simple benchmark are described in Appendix B. To test scalability on a larger scale, different benchmarks were used.

Two of the International Workshop for Logic Synthesis (IWLS) 2005 benchmarks were used as larger datasets: The International Symposium on Circuits and Systems (ISCAS) bench-

mark and the International Test Conference (ITC99) benchmark [46]. Both the ISCAS and ITC99 benchmark consist of around 30 circuits ranging from about 20 nodes to over several thousand nodes and were used to test the scalability and accuracy of the different matchers. Four different databases of different sizes were constructed from the circuits in these benchmarks. The circuits in the databases are listed in Appendix B.

The netlists from the IWLS benchmark were in structural Verilog format. Xilinx ISE was used to synthesize the netlist to produce a Xilinx specific netlist file called NGC. However, since the overall back-end is based off of Azido's front end system, an EDIF netlist file is required. NGC2EDIF was used to convert the NGC files after synthesis to EDIF. I/O buffers were ignored and removed from the final netlist. From the EDIF, the files were converted to the simplified graph format and stored into the database of patterns.

5.2 Accuracy

The dataset used to test how accurate the matchers are was the SIMPLE-2 benchmark. This was used primarily to see and compare circuits that were known to have matches so that the results can be accurately analyzed. Therefore the simple dataset was used first to try and compare the results of the matches. The larger dataset in IWLS-40 will be used for the CSI, VF2, and DSI matcher. MCS is left out of the larger dataset test due to performance issues describe in a later section.

5.2.1 Simple Benchmark

The SIMPLE-2 benchmark was used with a 1-bit counter circuit as the input circuit. Table 5.1 - Table 5.4 display the results of the four different matchers.

Circuit	Instances	Similarity (%)
XOR	1 Pattern	100
AOI	-	37.5
NAND2	1 Pattern	100
Half adder	1 Pattern	100
1-Bit Counter	1	100
2-Bit Counter	2 Input	100
4-Bit Carry ripple Adder	-	87.5
4-Bit Kogge-stone Adder	-	87.5
4-bit Kogge-stone Counter	4 Input	100

Table 5.1: Matches found using CSI Algorithm

CSI and VF2

The results seen in Table 5.1 and Table 5.2 shows the difference in matches between CSI and VF2. The instances column is the total number of possible instances found. *Pattern* means the total number of instances the circuit was found as a subgraph in the input and *input* is the total number of instance the input was found as a subgraph in the pattern. The similarity column indicates the similarity between the two circuits.

Even though both VF2 and CSI are both considered subgraph isomorphism algorithms, the results returned differ. The CSI algorithm returned more results due to the inexact methods and heuristics used. However, false positives may appear and matches aren't guaranteed

Circuit	Instances
XOR	2 Pattern
AOI	-
NAND2	2 Pattern
Half adder	-
1-Bit Counter	1
2-Bit Counter	2 Input
4-Bit Carry ripple Adder	-
4-Bit Kogge-stone Adder	-
4-bit Kogge-stone Counter	1 Input

Table 5.2: Matches found using VF2 Algorithm

to be found whereas if a match does exist, VF2 will find it. On the other hand, the CSI algorithm does provide more insight in the compatibility between the input and pattern circuit by calculating a similarity metric.

Some complete matches found in the CSI were not seen in the VF2 matcher because of the edge relationships between the two circuits. For example, with VF2, only one instance of the counter was found in the 4-bit kogge-stone counter; yet, four instances were found with the CSI matcher. From Figure 5.1, the sub-circuit highlighted in red indicates the complete counter found by VF2 and CSI. The other highlighted sub-circuits are technically also considered counters and have the same structure as the 1-bit counter. The only difference is that there is additional logic between the output of the XOR gate and the register. Because of this inexactness, the CSI matcher is able to identify the other three instances of the counter.

There are also several occasions where the VF2 found two instances of a circuit and the CSI found one, such as the XOR gate. The reason is the inputs can be in any order going into an XOR gate. Therefore, there are two different permutations for a two input XOR gate.

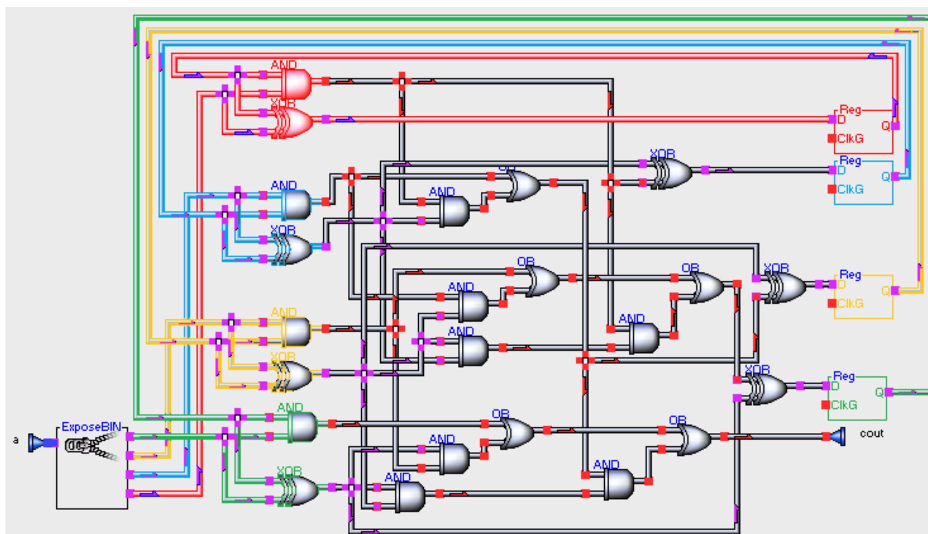


Figure 5.1: 4-bit kogge-stone counter with highlighted 1-bit counter sub-circuits

Because CSI marks a node when a match is found, any consecutive matches will ignore any nodes that are matched. Therefore, CSI returns a more accurate result in terms of instances.

CSI also has the capability to determine the similarity of two circuits whereas VF2 does not. The similarity metric the CSI algorithm returned is a pretty accurate representation. Since the AOI circuit has a completely different structure, the similarity measure should be fairly low. The kogge-stone adder and counter have a similar feature in that they both contain adders. Therefore, the adders should have a higher similarity metric compared with AOI as seen in Table 5.1.

MCS

Table 5.3 shows the results of the MCS matcher. Compared with the results in Table 5.1 and Table 5.2, MCS provides more data on the circuits that did not return exact matches. For example, the two 4-bit adders returned a possible match of seven input circuits. However, it is not a complete match since the similarity between the input and pattern is only 66 %. Counters have adders in them, more specifically, XOR gates. Figure 5.1 shows that the 4-bit kogge-stone adder contains seven XOR gates. Therefore, based on the number of potential XOR gates found, the 4-bit adders contain seven possible counter circuits.

Circuit	Instances	Similarity (%)
XOR	1 Pattern	100
AOI	-	-
NAND2	1 Pattern	100
Half adder	1 Pattern	80
1-Bit Counter	1	100
2-Bit Counter	2 Input	100
4-Bit Carry ripple Adder	7 Input	66
4-Bit Kogge-stone Adder	7 Input	66
4-bit Kogge-stone Counter	1 Input	100

Table 5.3: Matches found using MCS Algorithm

DSI

The results from Table 5.4 are from the DSI matcher. One main problem with DSI is that decomposition is primarily a subgraph isomorphism matcher. With CSI and VF2, the two graphs in question can be passed in based on which graph is larger. Therefore, super-circuits can also be determined. However, because DSI decomposes the patterns as an offline step, only exact subgraph matches can be found when using the decomposition tree during matching. Trying to find exact super-circuits won't work as intended. For the VF2 and CSI algorithm, the size of the two circuits are checked and passed to the matcher based on their size. In order to try and detect super-circuits of the input, the similarity metric is used. From Table 5.4, exact matches for the subgraph were found along with the number of possible instances. A similarity metric is given to all circuits that don't have a complete match, including super-circuits. There is no way to definitively determine if a super circuit exists or not given the results from DSI; however, the similarity can be used to suggest if a circuit is a possible super-circuit. For example, the 2-bit counter has the highest similarity score among the other circuits and is indeed a super-circuit of the input. At the same time, false positives may occur as well. For example, the matcher returned that the similarity

Circuit	Instances	Similarity (%)
XOR	2 Pattern	100
AOI	-	95
NAND2	2 Pattern	100
Half adder	2 Pattern	100
1-Bit Counter	1	100
2-Bit Counter	-	96
4-Bit Carry ripple Adder	-	89
4-Bit Kogge-stone Adder	-	82
4-bit Kogge-stone Counter	-	81

Table 5.4: Matches found using DSI Algorithm

between the AOI gate and the input is 95 % even though the structure of the AOI gate is fairly different from a counter.

One solution to try to find a more exact match for super-circuit is to decompose the input and test each pattern with the input to try and find a pattern circuit the input is a subgraph of, though this would defeat of purpose of decomposing the database initially. Furthermore, the search time for DSI would then depend on how many circuits are larger than the input.

5.2.2 Larger Benchmark

In order to test the accuracy of the larger benchmark, a circuit in the ISCAS benchmark was modified. Circuit s5378 was chosen and had one gate removed. By placing the actual s5378 circuit into the database, the matchers are expected to be able to find a match with the modified circuit to the original. IWLS-40 was used as the database.

From the results shown in Table 5.5, all three matcher were able to successfully identify an exact match when comparing the input with itself as the pattern. Only DSI and CSI matchers gave additional information on the similarity between the input and pattern circuit. However, one difference as mentioned in the previous section, CSI was able to detect that s5378mod is a sub-circuit of s5378 whereas DSI was not able to. On the other hand, due to the high similarity score given to s5378 by DSI, s5378 is most likely a possible super-circuit of s5378mod. Comparing the similarity results of CSI and DSI, DSI provides more insight between two circuits. For CSI, the matcher returns similarity values of zero for over half of the circuits because of the initial candidate pair used for the search. CSI doesn't look for all possible matches that can occur and just looks at locations where it thinks a likely match may exist.

Circuit	CSI (%)	VF2	DSI (%)	
s13207	0	-	79	Pattern
s5378	100	1	91	Input
s5378mod	100	1	100	
s15850	0	-	76	Pattern
s1494	0	-	83	Pattern
s1488	2.78	-	84	Pattern
s1423	0	-	80	Pattern
s1238	0	-	85	Pattern
s1196	12.02	-	84	Pattern
s838.1	0	-	83	Pattern
s832	0	-	86	Pattern
s820	6.46	-	86	Pattern
s713	0	-	90	Pattern
s641	0	-	89	Pattern
s526n	1.28	-	85	Pattern
s510	0	-	87	Pattern
s526	0	-	86	Pattern
s444	0	-	85	Pattern
s420.1	0.96	-	85	Pattern
s400	0	-	84	Pattern
s386	1.70	-	84	Pattern
s382	0	-	79	Pattern
s349	1.49	-	84	Pattern
s344	0	-	84	Pattern
s298	0	-	86	Pattern
s208.1	0	-	81	Pattern
s27	40.0	-	93	Pattern
b01	0	-	86	Pattern
b02	4.16	-	87	Pattern
b03	0	-	84	Pattern
b04	0	-	84	Pattern
b05	0	-	83	Pattern
b06	0	-	88	Pattern
b07	0	-	84	Pattern
b08	0	-	84	Pattern
b09	0	-	84	Pattern
b10	0	-	82	Pattern
b11	0	-	81	Pattern
b12	0	-	81	Input
b13	14.4	-	81	Pattern

Table 5.5: Comparison of results between CSI, VF2, and DSI

5.3 Performance

The results and timing obtained were averaged after five runs. Database and input circuit used depends on the type of matchers used.

5.3.1 Custom Subgraph Isomorphism

The CSI was tested alongside with the VF2 algorithm [15]. The input circuit used was s5378 which is a decently large circuit of over a thousand nodes in the ISCAS benchmark. IWLS-40 is the database used.

Performance of both the CSI and VF2 algorithms are not necessarily dependent on the size of the patterns or input circuit, to a certain degree. From the results shown in Figure 5.2, there is no apparent relationship between the execution time and the size of the pattern. For the CSI algorithm, this is due to how the search is being performed. The CSI algorithm looks for the component that is most uncommon in both input and pattern circuit and uses that as a starting point. If different candidate pairs do not produce a result, then the algorithm assumes that there is no match to be found and exits. Therefore, if the most uncommon node between the input and the circuit appears only occurs once, then only one scan is done with the pair as a starting point. However, if there are many candidate pairs, then the search will take longer due to having to search through every single pair.

The VF2 algorithm also looks for a candidate pair as a starting point and uses defined feasibility rules to try and find a correct match by looking ahead in the circuit. If a circuit has a similar repeating structure, even with the feasibility rules, there is still a large search space that needs to be covered. This is because, one wrong match, and VF2 has to backtrack to a valid state to try another path [15]. Moreover, because VF2 is an exact algorithm, if

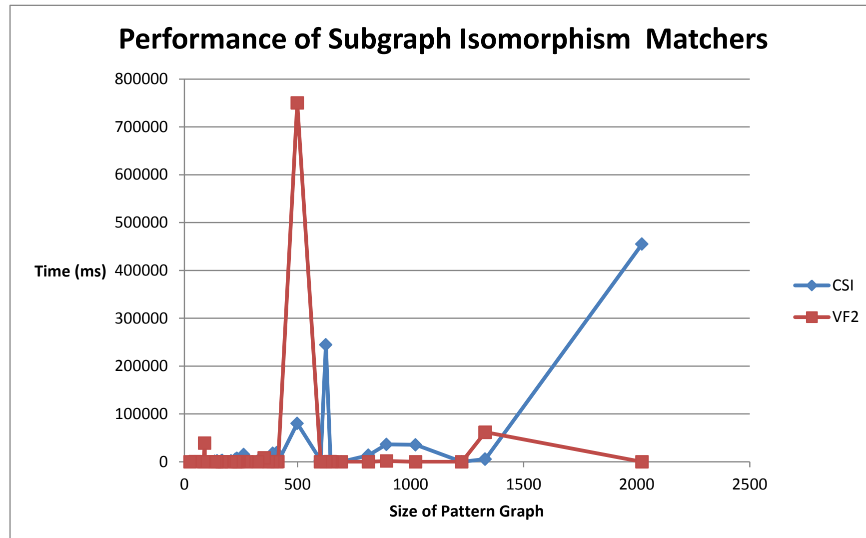


Figure 5.2: Execution time of CSI and VF2 for varying pattern circuit sizes

there is a match, then VF2 is guaranteed to find it. This means that VF2 will attempt all possible combinations until every single possible match has been searched.

5.3.2 Maximum Common Subgraph

To test the MCS matcher, the SIMPLE-1 database was used. The matcher used three different sized input circuits: counter, counter2, and counter4k.

The results the MCS matcher returned can be seen in Figure 5.3. By indicating the largest circuit that was common to both the input and pattern, the similarity between the two can be determined. For small circuits of probably less than 50 nodes, MCS performed exceptionally well, but as the number of components increase to more than 50, the execution time increases significantly compared to the other matchers. This is not only true for the patterns, but for the input circuit as well. If the input circuit is fairly small, and the pattern circuit is a couple hundred, MCS still runs reasonably well. However, if both the input and the pattern

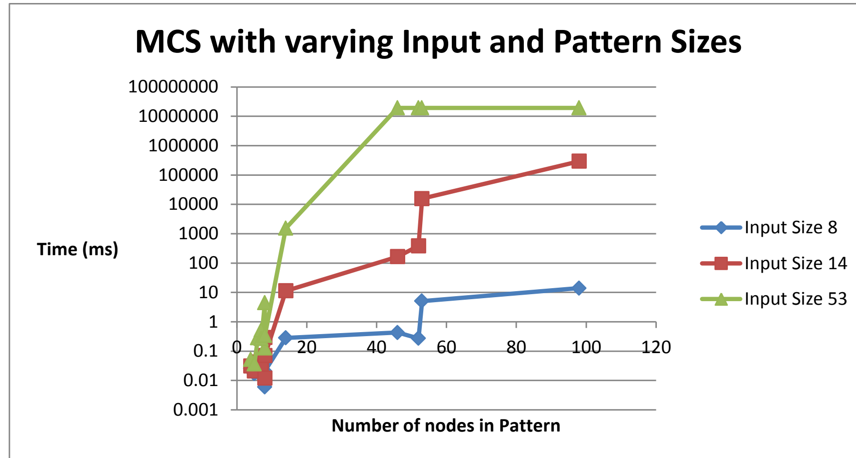


Figure 5.3: Execution time MCS with varying input and pattern circuit sizes

are large circuits, run time will increase significantly as seen in Figure 5.3 The input circuit of size 53 was terminated due to the execution time being several magnitudes higher than the previous input. One possible cause of this drastic increase is the clique detection. As the number of nodes increases, the size of the compatibility graph increases as well. The largest clique detection is also known as an NP-Complete problem and can be seen from the results. Furthermore the modular product of the two compatibility graphs will produce an extremely dense graph. On the other hand, the BK-algorithm also has a significant bottleneck due to the copying of arrays within each level of recursion. Run time comparison between the three different matchers can be seen in Figure 5.4 using the simple dataset with varying input graph sizes. MCS run time increases significantly after doubling in size whereas the other matchers performed decently well. Because of this, the simple benchmark was used.

5.3.3 Decomposition

The decomposition method has two main parts: the decomposition of pattern graphs and the subgraph matching. The decomposition takes a significant amount of time as the de-

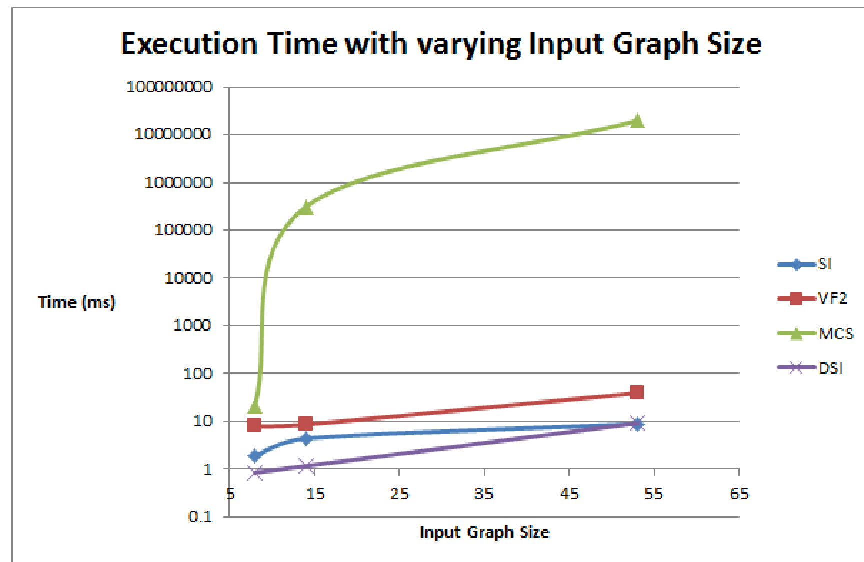


Figure 5.4: Execution time of matchers for varying input circuit sizes for a database of thirteen circuits

composition tree gets larger. This is because the algorithm does a subgraph match with every single node in the decomposition tree. Moreover, the larger the circuit, the more time it will take to decompose due to the amount of partitioning required to break the entire circuit down to a single vertex. To test the decomposer, the average execution time of the databases IWLS-5, IWLS-10, IWLS-20, and IWLS-40 was obtained after being decomposed a total of five times each. Because decomposition is not unique, the databases were ordered from smallest circuit to largest and largest circuit to smallest in order to test how well DSI performed using different decompositions. Figure 5.5 shows the overall execution timing for decomposing the databases. Subsequent patterns added onto the circuit will take longer to decompose due to the search for a possible sub-circuit in the entire decomposition tree. On the other hand, there was a noticeable difference in execution time depending on the order of the patterns that were passed into the decomposer. It can be seen in Figure 5.5 that decomposing the larger circuits first yields a lower overall decomposition time.

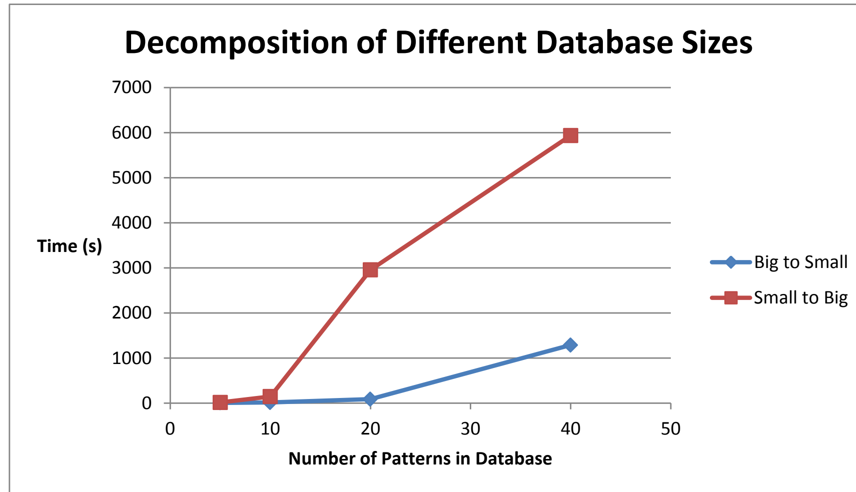


Figure 5.5: Execution time of decomposer for different database sizes

Database Size	Large to Small (sec)	Small to Large (sec)	Percent Difference (%)
5	28.826	28.956	0.44
10	36.450	35.919	1.46
20	58.836	59.145	0.523
40	179.126	179.210	0.046

Table 5.6: Comparison of decomposition matcher using two different decompositions trees

This is probably because with the first circuit, a majority of the operation done is partitioning. If the larger circuit is inserted at the end, there will be a significant amount of CSI tests running. Despite the two different decompositions, it can be seen that the overall DSI time is unaffected in Table 5.6. Therefore, to make the decomposition as efficient as possible, the larger circuits are decomposed first followed by consecutively smaller ones. This only applies to the initial decomposition rather than incremental decomposition when an entire database is set to be decomposed.

5.3.4 Scalability

In order to test how well the program scales as the number of patterns in the database grows, each matcher was tested with the IWLS-5, IWLS-10, IWLS-20, and IWLS-40 database.

From the results shown in Figure 5.6, decomposition scales extremely well compared to the subgraph isomorphism matchers. Again the primary reason is that similar sub-circuits are represented once wherever possible, and the tree like structure of the decomposition allows an efficient search for similarities.

The CSI matcher scales linearly as expected. For VF2, there was one circuit in the IWLS10 database that took exceptionally long to determine if a sub-circuit exists. As explained early, this is due to the topology of the circuit and how there are many similar components between the input and circuit. Nonetheless, the overall execution time for VF2 is still several times larger than DSI. To further analyze how well the algorithm will scale for databases larger

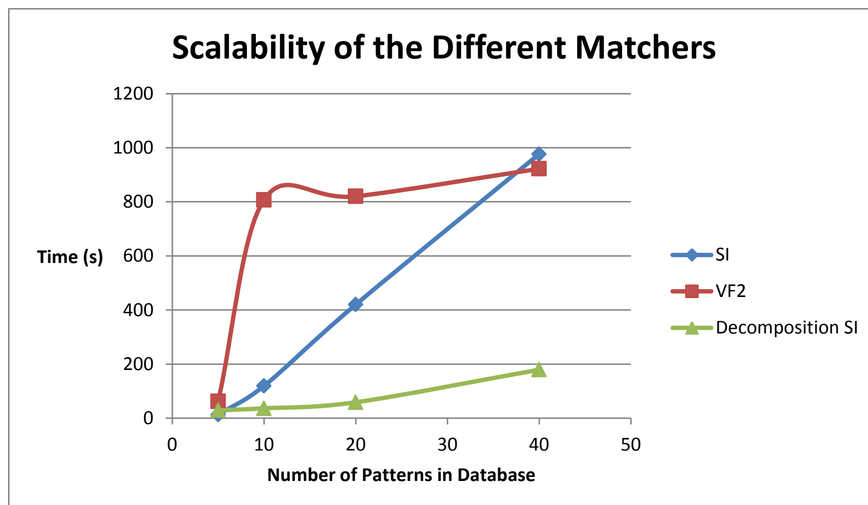


Figure 5.6: Execution time for varying database sizes

than 40 circuits, a polynomial regression was calculated from the data points of the DSI results. The polynomial regression is shown below in Equation 5.1.

$$0.1132x^2 - 0.8403x + 31.408 \tag{5.1}$$

If the database grows to about a size of 1000, then the overall execution time of the DSI matcher would be expected to take around 31.2 hours to try and find a match. However, this is also dependent on the size of the input graph. From Figure 5.7, as the size of the input graph increases, the execution time for DSI increases as well. This is different from the behavior of CSI and VF2 because DSI attempts the build the circuit from the bottom up. Furthermore as the input circuit becomes larger, there are more possible matches that can occur. Due to the number of possibilities, if a match is found, the mapping from the two child decompositions are combined and copied consuming more memory and time. One solution is to split the database up into several database depending on type, function, etc.

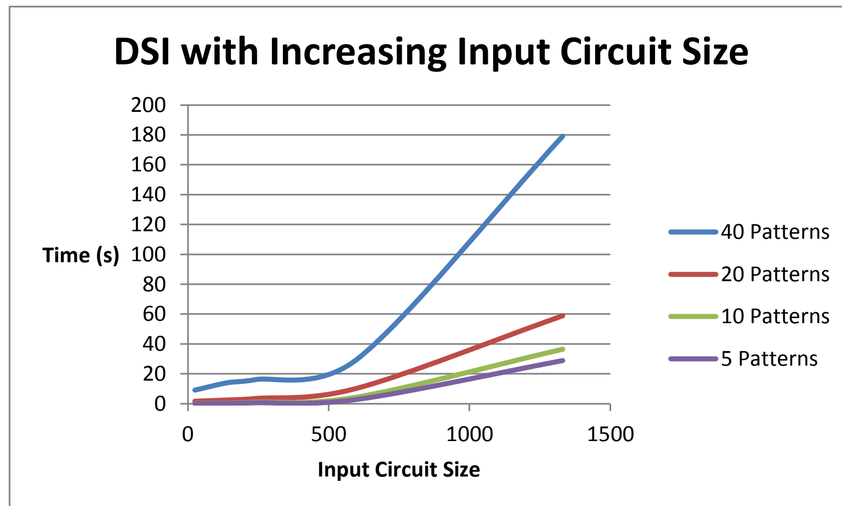


Figure 5.7: Execution time for DSI with varying input circuit sizes

This way, the matcher will have a smaller database to search. The databases can be organized by functionality or design, just like how software libraries are split up as well. With separate databases, the matcher has a much narrower window for searching. Additionally, decomposition can be parallelized to take advantage of multi-core architectures. The subgraph isomorphism search of the decomposer can easily be parallelized by having each core perform a CSI test with a decomposition node that has not been tested yet. The search for which nodes are alive during the matching phase can also be parallelized, giving more of a performance boost to the overall system.

Chapter 6

Conclusion

Circuit designers can reduce design time significantly by not having to redesign circuits that already exist. The IP discovery system uses different matching techniques to try and find similarities between digital circuits with an emphasis on design reuse. By reusing existing hardware, the designers can focus more on the application rather than the verification and debugging of existing hardware. The main focus of this thesis was the proof of concept of a back-end implementation of a system that can potentially increase FPGA productivity through reusing existing designs. This thesis presented an overview of possible implementations for matching a reference circuit against a database of pattern designs. It also discussed how reusing designs can increase FPGA productivity by incorporating the system into the design environment.

From the different circuit matching techniques that were explored, the DSI test is the best way to approach this in terms of performance and scalability. It allows an effective and efficient way of searching through a large database of circuits. MCS execution does not scale well at all for large circuits, but provides a detailed analysis on the similarity between two circuits. If some sort of hierarchy is inferred, the overall circuit can be simplified which can

lead to a more efficient MCS match. The CSI algorithm, even though it does not perform an exact match, returns fairly exact matches while indicating how similar two circuits may be. However, depending on the candidate pairs chosen, the CSI matcher may determine two circuits are not similar at all. One disadvantage of CSI is that it is not guaranteed to return a match if one exists. On the other hand, DSI provides a way of finding exact matches as well as comparing the similarity of the input across all the circuits in the database. DSI is also scales better than any of the other methods as the size of the database goes up. Unfortunately, there is no easy way to determine if the input circuit is a sub-circuit of those in the database. The similarity measure DSI returns is able to provide some insight as to which pattern could be a possible super-circuit, but does not indicate if one absolutely exists.

6.1 Future Work

This thesis presents a back-end system that determines possible reusable designs for a given reference design. Further improvements and enhancements to the overall system can be applied as follows:

- A front-end interface can be developed so that the back-end system is seamlessly integrated into the design environment. The interface can also be extended to be interactive so that the user can tailor the design that best fits.
- The overall system can also be extended to be able to recognize circuits based on their function. Currently, the feasibility of reusing similar circuits is determined by their structural properties. As seen, two circuits can be functionally identical and yet completely structurally different. This could provide a more accurate matching scheme for users.

- After an existing design has been suggested, the designer still has to make appropriate adjustments in order to utilize the hardware. The interface of the existing design may not be the same as what the designer intended. A possible extension would be to seamlessly integrate the existing hardware into the design trying to decrease the overall overhead of the reuse.

- Since the idea is that the reuse application will be integrated into the design environment, incremental design can potentially give substantial optimization in performance. The overall design doesn't change significantly during each iteration and can possibly limit the search space for possible matches in the database.

- Currently, all the netlists are flattened into primitives. Hierarchy of the circuit can be applied. By doing so, the overall netlist of the circuit is smaller resulting in a quicker overall search. Circuit hierarchy can be inferred by simplifying the circuit with exact components. Furthermore existing hierarchy can be searched as well; however, the patterns will have to take circuits that contains the same type of hierarchy as well.

- There is still quite a bit of performance that can be added to DSI. For one, the data structure can be modified to allow for a more efficient search. A major bottleneck of DSI is the copying of the mappings currently found from decomposition to decomposition. With larger circuits, the amount of mappings that need to be copied at each level increases significantly. Reducing the amount of copying would increase performance of the DSI.

- The development of an efficient database where users can search for and contribute designs to can be explored. Having a common repository allows the hardware community to grow and learn from one another while promoting a sense of collaboration. Furthermore having a database for the decomposition for DSI can help alleviate mem-

ory problems while providing a quick and easy way to find matches. Cloud-based systems can be examined. Having a central online system that performs vast amounts of comparisons for users can potentially provide quicker results and not consume all of the user's computing power.

6.2 Extended Applications

The overall system can be extended to a wider variety of applications. Finding similar circuits and then providing multiple implementations of the circuit can provide optimization suggestions. One design may be more efficient for a particular function and the user can choose the one that best suits their application. Similar to code-complete for software, the tool can automatically complete a given hardware circuit. This can reduce overall design time as well by providing the user with a working design if the designer has trouble getting the circuit to function correctly. In otherwords, it can be use as an autocorrect as well. In addition, the tool can be applied to reverse engineering. Given an unknown circuit, the function and properties of the circuit is determined by finding similar circuits that make up the circuit in question. On the other hand, there are vast amounts of circuits that can be reused across various sources. Some are not organized or classified as such as those in OpenCores [7]. The system can be used to build reliable hardware libraries by finding circuits similar in structure. It can also help broaden the community of hardware developers as designers contribute and learn as a whole. Users will consume from the community by using the system's resources to search for possible similarities in the circuit, and produce or give back to the community by submitting their reference design to help grow the database and/or use the feedback/selection of the user to help train the system in providing more accurate matches.

Bibliography

[1] B. Nelson, M. Wirthlin, B. Hutchings, P. Athanas, and S. Bohner, "Design productivity for configurable computing," in *ERSA08: Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pp. 57–66, 2008.

[2] MathWorks, "http://www.mathworks.com/company/newsroom/features/4-steps-to-smarter-fpga-design.html,"

[3] E. Girczyc and S. Carlson, "Increasing design quality and engineering productivity through design reuse," in *Design Automation, 1993. 30th Conference on*, pp. 48–53, 1993.

[4] O. C. P. I. Partnership, "http://www.ocpip.org,"

[5] C. André, F. Mallet, A. M. Khan, R. De Simone, and I. S.-A. Méditerranée, "Modeling spirit ip-xact with uml marte," in *Proc. DATE Workshop on Modeling and Analysis of Real-Time and Embedded Systems with the MARTE UML profile*, 2008.

[6] T. Schattkowsky, T. Xie, and W. Mueller, "A uml frontend for ip-xact-based ip management," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, (3001 Leuven, Belgium, Belgium), pp. 238–243, European Design and Automation Association, 2009.

[7] OpenCores, "http://www.opencores.org,"

[8] J. Sharp, *Microsoft® Visual C#® 2010 Step by Step.* Microsoft Press, 2010.

[9] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web.," Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[10] Altera, "Introduction to megafunctions," 2011.

[11] H. Global, "http://hitechglobal.com/,"

[12] R. Chandra, "Ip-reuse and platform base designs," 2002.

[13] L. Zager, *Graph similarity and matching.* PhD thesis, Massachusetts Institute of Technology, 2005.

[14] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, pp. 31–42, Jan. 1976.

[15] L. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 26, no. 10, pp. 1367–1372, 2004.

[16] P. Foggia, C. Sansone, and M. Vento, "A performance comparison of five algorithms for graph isomorphism," in *Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, pp. 188–199, 2001.

[17] J. Whitham, "A graph matching search algorithm for an electronic circuit repository," *Univ. of York*, 2004.

[18] P. Durand, R. Pasari, J. Baker, and C. Tsai, "An efficient algorithm for similarity analysis of molecules," *Internet Journal of Chemistry*, vol. 2, no. 17, pp. 1–16, 1999.

[19] I. Koch, "Enumerating all connected maximal common subgraphs in two graphs," *Theoretical Computer Science*, vol. 250, no. 1, pp. 1–30, 2001.

[20] M. Neuhaus, K. Riesen, and H. Bunke, "Fast suboptimal algorithms for the computation of graph edit distance," *Structural, Syntactic, and Statistical Pattern Recognition*, pp. 163–172, 2006.

[21] H. Elghawalby and E. Hancock, "Measuring graph similarity using spectral geometry," *Image Analysis and Recognition*, pp. 517–526, 2008.

[22] R. Wilson and P. Zhu, "A study of graph spectra for comparing graphs and trees," *Pattern Recognition*, vol. 41, no. 9, pp. 2833–2841, 2008.

[23] S. Eckmann and G. Chisholm, "Assigning functional meaning to digital circuits," tech. rep., Argonne National Lab., IL (United States), 1997.

[24] K. Anjaneyulu, "Expert systems: An introduction," *Resonance*, vol. 3, no. 3, pp. 46–58, 1998.

[25] V. Arvind and Y. Vasudev, "Isomorphism testing of boolean functions computable by constant-depth circuits," *Language and Automata Theory and Applications*, pp. 83–94, 2012.

[26] C. Ebeling, "Geminiii: A second generation layout validation program," in *Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers., IEEE International Conference on*, pp. 322–325, IEEE, 1988.

[27] M. Ohlrich, C. Ebeling, E. Ginting, and L. Sather, "Subgemini: identifying subcircuits using a fast subgraph isomorphism algorithm," in *Proceedings of the 30th International Design Automation Conference*, pp. 31–37, ACM, 1993.

[28] X. Shi, D. Zeng, Y. Hu, G. Lin, and O. Zaiane, "Enhancement of incremental design for fpgas using circuit similarity," in *Quality Electronic Design (ISQED), 2011 12th International Symposium on*, pp. 1–8, IEEE, 2011.

[29] M. Takashima, A. Ikeuchi, S. Kojima, T. Tanaka, T. Saitou, and J. Sakata, "A circuit comparison system with rule-based functional isomorphism checking," in *Design Automation Conference, 1988. Proceedings., 25th ACM/IEEE*, pp. 512–516, IEEE, 1988.

[30] E. I. Goldberg, M. R. Prasad, and R. K. Brayton, "Using sat for combinational equivalence checking," in *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, pp. 114–121, IEEE, 2001.

[31] J. Marques-Silva and T. Glass, "Combinational equivalence checking using satisfiability and recursive learning," in *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings*, pp. 145–149, IEEE, 1999.

[32] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman, "Satisfiability solvers," *Foundations of Artificial Intelligence*, vol. 3, pp. 89–134, 2008.

[33] T. Portegys, "General graph identification by hashing," *Normal, IL, Illinois State University*, 2008.

[34] M. Lazarescu, H. Bunke, and S. Venkatesh, "Graph matching: fast candidate elimination using machine learning techniques," *Advances in Pattern Recognition*, pp. 236–245, 2000.

[35] B. Messmer and H. Bunke, "Efficient subgraph isomorphism detection: a decomposition approach," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 12, no. 2, pp. 307–323, 2000.

[36] OpenFPGA, "http://www.openfpga.org,"

[37] Altera, "Lpm quick reference guide," 1996.

[38] Xilinx, "Virtex 4 libraries guide for hdl designs," 2009.

[39] Azido, "http://www.azido.net,"

[40] LabView, "http://www.ni.com/labview/fpga,"

[41] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French, "Torc: towards an open-source tool flow," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 41–44, ACM, 2011.

[42] Unity3D, "http://unity3d.com/,"

[43] Y. Tian, R. Mceachin, C. Santos, J. Patel, *et al.*, "Saga: a subgraph matching tool for biological graphs," *Bioinformatics*, vol. 23, no. 2, pp. 232–239, 2007.

[44] S. Zhang, S. Li, and J. Yang, "Gaddi: distance index based subgraph matching in biological networks," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pp. 192–203, ACM, 2009.

[45] Boost, "http://www.boost.org,"

[46] C. Albrecht, "Iwls 2005 benchmarks," in *International Workshop for Logic Synthesis (IWLS): http://www. iwls. org*, 2005.

Appendix A: Detailed Description of Circuits in the Simple Benchmark

The simple benchmark consists of thirteen circuits made up of adders, counters, and simple gates with sizes of 4 to 80 components. All circuits in the simple benchmark were designed with AND, NOT, OR, XOR, and D-flip flops as the primitive gates. Only circuits with an 'x' in the file name was designed using XOR gates.

- nand2- A NAND gate with two inputs and designed with an AND gate followed by an inverter.
- xor- A XOR gate with two inputs. The XOR circuit was designed similar to Figure 2.6a
- xnor- A XNOR gate with two inputs and designed with the expanded XOR circuit followed by an inverter
- aoi- A AOI circuit with four inputs going into two separate two-input and gates. The outputs of the AND gates are connected to an OR gate followed by an inverter.
- adder-half- A half adder circuit designed with the XOR gate as the sum and an AND gate as the carry out.

- counter- A counter circuit designed with the adder-half circuit with the sum connected to a D-flip flop. The output of the flip flop is looped back into a input of the adder.
- counterx- A counter designed with an XOR gate primitive.
- counter2- A 2-bit counter composed of two 1-bit counters with the carry out of one feeding into the input of the second.
- counter2x- A counter2 designed with XOR gate primitives.
- carryripple- A 4-bit carry ripple adder
- kogge-stone- A 4-bit kogge-stone adder
- counter4k- A 4-bit counter using a kogge-stone adder.
- coutcounter- A 4-bit carry ripple adder using the counter4k circuit to count the number of carry outs the adder produces.

Appendix B: Database Circuits

Below are the two databases consisting of circuits from the simple benchmark and the 4 databases consisting of circuits from the IWLS benchmark.

SIMPLE-1	SIMPLE2
xor	xor
aoi	aoi
nand2	nand2
adder-half	adder-half
counter	counter
counter2	counter2
carryripple	carryripple
kogge-stone	kogge-stone
counter4k	counter4k
counterx	
counter2x	
coutcounter	
xnor	

IWLS-5	IWLS-10	IWLS-20	IWLS-40
s5378	s5378	s5378	s5378
s526	s526	s526	s526
b13	s526n	s444	s444
s400	s400	s400	s400
b10	s1196	s1196	s1196
	b06	s526n	526n
	b08	s13207	s13207
	b09	s1494	s1494
	b10	s1488	s1488
	b13	s1423	s1423
		s832	s386
		s713	s382
		s641	s349
		b05	s344
		b06	s298
		b08	s208-1
		b09	s27
		b10	s1238
		b11	s713
		b13	s820
			s641
			s15850
			s510
			s5378mod
			s832
			s838-1
			s420-1
			b01
			b02
			b03
			b04
			b05
			b06
			b07
			b08
			b09
			b10
			b11
			b12
			b13