5-2004

A Reconfigurable Computing Solution to the Parameterized Vertex Cover Problem

Dorai Mahesh
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a thesis written by Dorai Mahesh entitled "A Reconfigurable Computing Solution to the Parameterized Vertex Cover Problem." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Dr. Gregory D. Peterson, Major Professor

We have read this thesis and recommend its acceptance:

Dr. Donald W. Bouldin, Dr. Michael A. Langston, Dr. Chandra Tan, Dr. Philip Locasio

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by Mahesh Dorai entitled "A Reconfigurable Computing Solution to the Parameterized Vertex Cover Problem". I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Dr. Gregory D. Peterson

(Major Professor)

We have read this thesis and recommend its acceptance:

Dr. Donald W. Bouldin

Dr. Michael A. Langston

Dr. Chandra Tan

Dr. Philip Locasio

Accepted for the Council:

Anne Mayhew

Vice Provost and

Dean of Graduate Studies

(Original signatures are on file with official student records)

A Reconfigurable Computing Solution to the Parameterized Vertex Cover Problem

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Mahesh Dorai

May 2004

Acknowledgements

At the outset, I am thankful to my advisors, Dr. Gregory D. Peterson and Dr. Michael A. Langston for their guidance and support throughout the course of my education. This work is a result of their encouragement, timely ideas and constructive criticism. I would like to thank Dr. Donald W. Bouldin for the entire series of Digital Design courses, which gave me a complete overview of the fundamentals underlying Digital VLSI design. Many of the ideas and concepts that I learnt in those classes served me at critical points and stumbling blocks in this work. I am thankful to Dr. Chandra Tan for those invaluable suggestions, which he had to offer, from that seemingly endless tank of ideas that he possesses, at key junctures in my research and course work. I would like to thank Dr. Philip Locascio for having read and provided useful suggestions to my thesis.

Further, I am also thankful to my friends Faisal Abu-Khzam, Chris Symons, Pushkar Shanbhag, Fuat Karakaya, Adam Miller, Sampath Kothandaraman, Mike McCollum, Kirk Baugher, Melissa Smith, Saumil Merchant and others for their helpful suggestions.

I am grateful to the National Science Foundation for their financial support. Further, I am thankful to Ho Chun Hok at the Chinese University of Hong Kong for giving me an insight into the Pilchard reconfigurable environment.

I would like to thank the National Science Foundation, under grants NSF 0075792 and NSF 9972889, the Office of Naval Research grant N00014-01-1-0608, The Air Force Research Laboratories grant F30602-00-D-0221, for having supported me. I would also like to thank SRC Computers, Inc for the technology licenses to the Pilchard machines.

I am most indebted to my parents, Sri. Dorai Krishnamurthy and Smt. Subhalakshmi Dorai, and my brother Rajesh Dorai, for their unrelenting support, love, and encouragement. I also wish to express my thanks to all my uncles, aunts and cousins for providing me with those lighter months and "much-wanted distraction", that one needs at times. Many thanks are due to my best friend and soon to be wife, Priyya Natarajan for her support and never-say-die attitude.

Finally, I wish to thank the Almighty and my ancestors for their love and blessings, without which none of this would have been possible.

Abstract

Active research has been done in the past two decades in the field of computational intractability. This thesis explores parallel implementations on a RC (reconfigurable computing) platform for FPT (*fixed-parameter tractable*) algorithms.

Reconfigurable hardware implementations of algorithms for solving NP-Complete problems have been of great interest for research in the past few years. However, most of the research that has been done target exact algorithms for solving problems of this nature. Although such implementations have generated good results, it should be kept in mind that the input sizes were small. Moreover, most of these implementations are instance-specific in nature making it mandatory to generate a different circuit for every new problem instance.

In this work, we present an efficient and scalable algorithm that breaks out of the conventional instance-specific approach towards a more general parameterized approach to solve such problems. We present approaches based on the theory of *fixed-parameter tractability*. The prototype problem used as a case study here is the classic vertex cover problem. The hardware implementation has demonstrated speedups of the order of 100x over the software version of the vertex cover problem.

Table of Contents

List of Tables

List of Figures

Chapter 1

Terminology and Introduction to Computational Complexity

The graphs studied in this work are simple and undirected graphs. Graphs with self-loops and vertices with no edges are not discussed here. Some of the properties of graphs are described here. We restrict the terminology and notation to the scope of the study and those relevant to the work. In this chapter, we also discuss the fundamentals underlying the concept of *fixed-parameter tractability.*

1.1 Terms and Definitions

A graph is a set of vertices and the edges that connect them [8]. A graph is defined by a vertex set V and an edge set E and is denoted by $G(V, E)$. In the following text, the vertices V might also be referred to as nodes. Similarly the edges E might also be referred to as branches.

Graph theory is the branch of mathematics that examines the properties of graphs. Depending on the applications, edges may or may not have a direction; edges joining a vertex to itself may or may not be allowed, and vertices and/or edges may be assigned weights. If the edges have a direction associated with them (indicated by an arrow in the graphical representation) we have a **directed graph.** From the point of view of digital system design, many CAD algorithms are based on directed graphs. Directed

graphs are also used to represent finite state machines. The development of algorithms to handle graphs is therefore of major interest.

Removal of a certain number of vertices and (or) edges from the graph results in what are known as subgraphs. It should be noted that the removal of a vertex implies the removal of all its edges from the graph.

The degree of a vertex represents the number of edges that are incident on it.

1.2 Data Structures for the Representation of Graphs

For the purpose of implementing graph algorithms and search space techniques, one often uses a data structure that makes it easier to manipulate the graph. In computers, a finite directed or undirected graph (with n vertices) is often represented by its **adjacency matrix**: an n-by-n matrix whose entry in row i and column j gives the existance of an edge from the i^{th} to the j^{th} vertex. In this regard, it has to be kept in mind that different algorithms may have different requirements and hence the need for a data structure that suits is requirements. The data structure used has to be suitable to represent the graph in any computing environment, be it in software or custom hardware.

Figure 1.1 depicts a simple undirected graph and figure 1.2 gives the adjacency matrix representation of the graph. Given a graph $G(V,E)$ with n vertices, the individual elements of the adjacency matrix are constructed with the condition that [8]

$$A_{ij} = 1 \text{ if } (v_i v_j) \in E, \text{and } A_{ij} = 0 \text{ if } (v_i v_j) \notin E$$

It is evident from the adjacency matrix representation shown in figure 1.2, that the adjacency matrix representation of any graph is symmetric for undirected graphs. We use undirected graphs for the vertex cover problem in this thesis and describe the graphs using adjacency matrices.

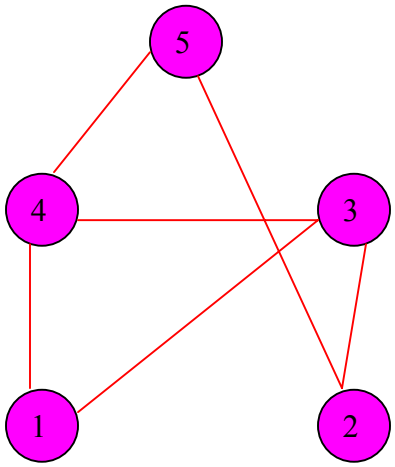

Figure 1.1 An example of a simple undirected graph

$$\begin{bmatrix} 00110 \\ 00101 \\ 11010 \\ 10101 \\ 01010 \end{bmatrix}$$

Figure 1.2 Adjacency matrix representation of graph shown in figure 1.1

1.3 Computational Complexity

One of the main concerns regarding the design of an algorithm is the efficiency of the algorithm. The computational complexity describes the asymptotic performance or speed with which the algorithm produces the final result as a function of problem size [8]. The input size of an algorithm is the number of elements that are necessary to describe the input. The input size of a graph algorithm operating on a graph *G(V,E)* is characterized by two parameters –

1. *The size of the vertex set |V|*
2. *The size of the edge set |E|*

In the fields of algorithm analysis and computational complexity theory, the runtime or space requirements of an algorithm are expressed as a function of the problem size. Computational complexity is of two types:

1. *Time complexity*
2. *Space complexity*

The time complexity of a problem asymptotically describes the number of steps required to solve an instance of a problem, as a function of the input size. The space complexity on the other hand asymptotically describes the amount of memory required to solve the instance of the problem. In this thesis we focus on the time complexity of graph algorithms.

An algorithm that grows exponentially as the problem size grows would take more time to find a solution than an algorithm that takes polynomial time. Hence, algorithms with polynomial time complexity are preferred over algorithms with exponential time complexity. Polynomial time algorithms are considered computationally

tractable or efficient, whereas exponential time algorithms are computationally intractable.

We know that the notion of time complexity is extremely important in designing an algorithm. We also discussed that an algorithm that grows in a polynomial fashion takes lesser time in comparison to an algorithm that grows in an exponential fashion. Any problem that can be solved in polynomial time is considered **tractable**. It is **intractable** otherwise. While exact algorithms can be used to find optimal solutions for tractable problems, in the case of intractable problems, often one has to be satisfied with algorithms that do not guarantee optimal solutions.

In complexity theory, the class P(P stands for polynomial) consists of all those decision problems that can be solved using an algorithm on a deterministic sequential machine in polynomial time. Before we discuss the class of NP, we need to understand the meaning of a nondeterministic computer. The class **NP** consists of all those decision problems that can be verfied(we purposely do not use the word "solved", we use the word "verified" as most NP problem are decision problems) in polynomial time on a deterministic machine. In this context, it will be beneficial to discuss the whole notion of Decision Problems.

1.4 Decision Problems

Simply put, a Decision problem is one whose solution is either a "Yes" or a "No". To illustrate the notion of NP-Complete, here is an example from [9] to get an idea for the question.

> "Given two large numbers *X* and *Y*, we might ask whether *Y* is a multiple of any integers between 1 and *X*, exclusive. For example, we might ask whether 69799

is a multiple of any integers between 1 and 250. The answer is YES, though it would take a fair amount of work to find it manually. On the other hand, if someone claims that the answer is YES because 223 is a divisor of 69799, then we can quickly check that with a single division. Verifying that a number is a divisor is much easier than finding the divisor in the first place".

Since all polynomial time algorithms that can be executed on a deterministic computer will definitely execute on a non-deterministic computer, the class **P** set of problems belong to the domain of the class **NP**.

With these ideas in mind, we now introduce the notion of **Parameterized Complexity**.

1.5 Parameterized Complexity

Currently, no polynomial-time algorithm has been found to solve any NP-complete problem. It is rather unlikely that a polynomial-time algorithm will exist for these kind of problems. Numerous techniques using approximation techniques and heuristic techniques are used to attempt to solve NP-complete problems[8].

There have been cases of exact algorithms being used to find solutions[1]. But, in the cases, where exact algorithms were used, the input sizes were either small or modest at best.

The work of Fellows and Langston proved that certain intractable problems become tractable when the input parameters are fixed [11,12,13,14]. Later the work of Downey and Fellows [37] led to the creation of a solid base for Parametrized Complexity theory.

1.6 Fixed-Parameter-Tractability

From the definition of *fixed-parameter tractability* in [2],given a parametrized problem (*I,k*) with an instance I and a parameter k, if there exists an algorithm such that the problem instance (*I,k*) executes in time $O(f(k)|I|^c)$,where $|I|$ is the size of I, *f(k)* is an arbitrary function, and *c* is a constant, then the problem (*I,k*) becomes tractable. The algorithms that can execute in the time $O(f(k)|I|^c)$ are called *fixed-parameter-tractable* algorithms. Some of the well known *fixed-parameter-tractable* algorithms are listed below[8].

1. The Vertex Cover Problem(The prototype problem studied in this work)
2. The Face Cover Problem
3. The Disk Dimension Problem
4. The Planar Dominsating Set Problem

In this chapter, we have discussed some of the key terms in graph theory related to this thesis. We have discussed the theory of *fixed-parameter tractability.* In the next chapter, we discuss some of the research done in acceleration of optimization algorithms in a reconfigurable computing platform.

Chapter 2

Introduction and Background

2.1 Reconfigurable Architectures

In the last several years, reconfigurable architectures have been used in a variety of methods to speedup combinatorial problems. More specifically, a lot of research has gone into effectively harnessing the power of reconfigurable logic and its inherent properties that includes concurrency. The research community targeted many problems that were NP-complete and devised algorithms to solve them. Normally, the very fact that the problem is NP-complete would deter persons from pursuing an exact algorithm for them. Although exact algorithms are not usually pursued for solving NP-complete problems, several exact algorithms were proposed. Some of these algorithms targeted modest input sizes or problem instances with a very low parameter. The reader will recollect that a FPT problem is defined with the problem I and the parameter k.

2.2 Models of Reconfiguration

The models of configuration are broadly classified as follows.

1. *Generic computation engine*
2. *Instance-specific reconfiguration*

Shown in Figure 2.1 are the steps involved in the *generic computation engine* [16][35][36]

2.2.1 Compile-Time Reconfiguration

In this model of reconfiguration, the circuit is compiled, synthesized and loaded once. The same configuration file is used for testing and processing different sets of data. This is the model used for most custom-computing machines. The configuration remains in the FPGA for the duration of the application. The same engine can be used and reused for different inputs. Hence for each application or algorithm, a new configuration is built that can be downloaded to the FPGA.

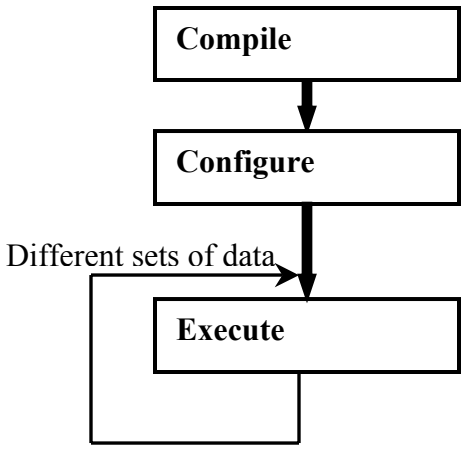

Figure 2.1 Generic graph engine compute model [16]

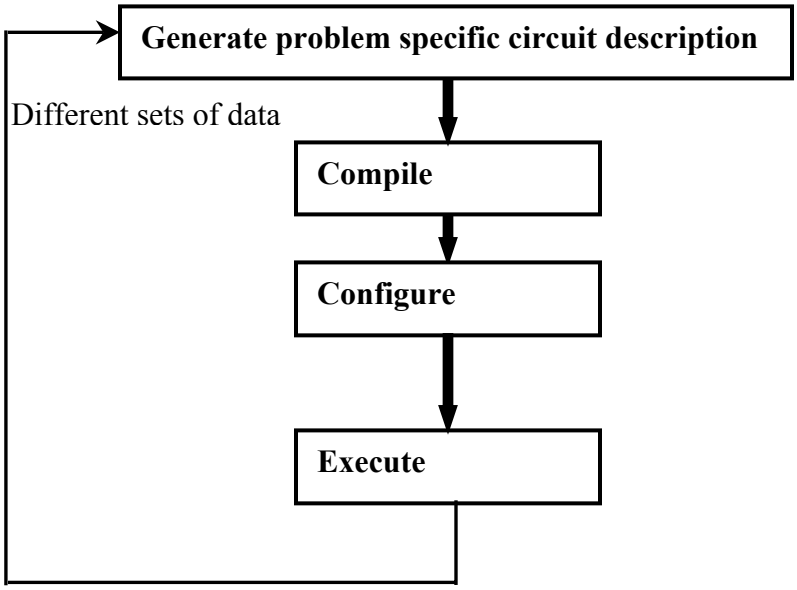

Figure 2.2 Instance-specific reconfiguration [16]

2.2.2 Instance Specific Reconfiguration

The other model of reconfiguration called the instance-specific reconfiguration, is based on the idea that the hardware circuit is optimized to the specific graph instance. It is also denoted as *dynamic compilation* whereas our approach uses *static compilation*. Shown in Figure 2.2 are the steps involved in *instance-specific reconfiguration*.

Suyama et al.[33] were the first to propose the use of reconfigurable computing power to solve hard problems such as the SAT. They developed an *instance-specific* logic circuit specialized to solve each problem instance of the SAT problem. Suyama et al[33] proposed a new parallel checking algorithm that would assign all variable values concurrently and scan all the clauses (constraints) simultaneously. They implemented a

hard random 3-SAT problem with 300 variables and ran the logic circuits at about 1 MHz. They reported that the time taken for logic circuit generation from a problem description to be in the order of hours.

Suyama et al.[34] later developed a series of algorithms suitable for logic circuit implementation. The circuit implemented was able to solve a 400 variable problem within 1.6 minutes at a clock rate of 10 MHz. The aim of most of the then existing algorithms was to find just one solution, if it existed. An important improvement of their work over the then existing methods was that they aimed at finding all or multiple solutions.

Hamadi and Merceron [26] implemented the GSAT algorithm on FPGA's to speedup the resolution of SAT problems. The GSAT algorithm, a greedy local search procedure searches for satisfiable instantiations of formulas under conjunctive normal form. They proposed an incomplete algorithm, which dealt with formulas of large size. They argued that though the algorithm was incomplete, the existing technology was out of bounds for an exhaustive search with regards to large formulas. Incomplete algorithms are those that may not find a solution even if it does exist. Complete algorithms on the other hand are guaranteed to find a solution if it indeed existed.

In the initial years of using reconfigurable computing to solve hard problems, the SAT or the Satisfiability problem and numerous flavors of the same were explored to a great deal.

In particular Plessl and Platzner [15] discuss an instance-specific reconfigurable architecture for "*minimum covering"*. It should be noted that the algorithm used is an exact algorithm, targeting an instance-specific architecture. Plessl and Platzner [15] have

demonstrated raw speedups of several orders of magnitude over the software versions. However they were constrained by the long synthesis and compilation times, as the architecture was instance-specific. Also, their approach uses a NP-Complete algorithm which limits scaling the problem size.

Numerous reconfigurable architectures were proposed for the boolean SAT problem. Zhong et al.[30] proposed a reconfigurable accelerator to accelerate problems in the CAD domain. This work too targeted the algorithm on an "input specific"[30] basis rather than a parameterized form.

Platzner et al. [17] also proposed different architectures to solve the boolean satisfiability problem. Overall speed-ups (taking into account the hardware compilation time of Xilinx design implementation tools) of 6.5x have been achieved. An exact algorithm was implemented in this case as well.

One of the limitations of all the above-discussed implementations is that a new circuit customized to the problem is developed for every problem instance. In hardware terms, this translates to a huge overhead from factors such as compilation time, synthesis time, mapping and place and route to name a few. For each new set of problem instances, the entire cycle of processing from a high level description to a bit-level generation is repeated.

Leong et al.[32] were the first to propose an implementation in 2001, which discussed this limitation of the architectures. They broke away from the architectures that were in vogue till then, by proposing an implementation that was devoid of the overheads involving re-synthesis, and repeated cycles of place and route for each problem instance.

Leong et al.[32] chose the WSAT algorithm as the prototype for implementing this new approach.

All of the discusses approaches use NP-complete algorithms. This thesis uses a computationally efficient algorithm. Also, the implementation approach in this thesis is a generic computation engine and not an instance specific engine.

2.3 The Pilchard Reconfigurable Platform

The Pilchard Reconfigurable computing platform was developed by Leong et al. [41] at the Chinese University of Hong Kong. The Pilchard houses a Xilinx Virtex 1000E FPGA, which has close to a million gates on it. Unlike other reconfigurable platforms that are based on a PCI interface, the Pilchard board resides in the DIMM (dual In-line memory module) slot of a standard personal computer. The Pilchard interface offers higher bandwidth, and lower latency [41]. One of the key features of the Pilchard board is the built-in clock generator. The built-in clock generator is capable of generating clocks whose periods are 1.5, 2, 2.5, 3, 5, 8 and 16 times that of the main clock. This way the user need not generate a clock divider circuit on chip. The Pilchard supports a 64-bit data bus and a 14-bit address bus. The main system clock can be either set to a frequency of 100 or 133 MHz. Shown in figure 2.3 is a snap shot of the Pilchard board.

2.4 Case Study – The Vertex Cover Problem

The Vertex Cover problem can be defined as follows. Given a graph $G(V,E)$ and a parameter k, the objective is to find a subset S of the graph G, that will cover every edge of G. An edge is covered if either or both of its endpoints are present in S. In other words, removal of the vertices that are in S, amounts to the non-existance of the graph G. (Please note that, when a vertex is removed from the graph, all the edges that are

Figure 2.3 The Pilchard board

incident on it are removed, and hence the notion of the non-existance of the graph, when such a subset *S* is found.)

2.4.1 Algorithmic Reduction Techniques for FPT Problems

Pre-processing techniques prove very useful in handling large graph inputs. The objective of any pre-processing technique is to reduce the size of the graph instance before the actual process of branching. Abu-Khzam [2] in his work has mentioned a variety of reduction techniques to FPT problems. In particular, he established a suite of algorithmic tools to demonstrate the fact that FPT problems are in general amenable to reduction in size by use of suitable reduction techniques. He also introduces a new idea known as re-processing or interleaving. More information on this can be found in [2].

Some of the commonly used pre-processing techniques [2] are discussed below.

The discussed techniques are based on the properties of the graphs themselves. Of late, a variety of heuristics are in use, some of which have been used in this work.

(i) Checking to see of the input graph is fully connected. Dealing with a fully connected graph is easier. Most algorithms assume that the input graph is already connected

(ii) Dealing with high degree vertices: High degree vertices play an important role in the reduction techniques involved in the vertex cover problem. The fundamental concept behind the branching algorithm is that any randomly chosen vertex or all of its neighbors have to be in the cover. Let us assume that that we have a problem instance (G,k). Now if we chose a vertex P at random and it has $(k+1)$ neighbors, then P has to be in every vertex cover of size k. This can be reasoned as follows. Let us assume that the selected vertex P is not in the cover. This would mean that all the neighbors of P are in the cover. But the number of neighbors it has is ($k+1$). Since the number of neighbors exceeds the requested parameter k, to guarantee that we get a cover of a maximum size of k, our assumption that the highest degree vertex is not in the cover is wrong. To give us a chance of finding a cover of maximum size k, either the highest degree vertex or all of its neighbors have to be in every vertex cover of size k.

(iii) Dealing with low degree vertices: Abu-Khzam [2] has shown that if an instance (G,k), of the vertex cover problem has vertices of degree less than 3, then (G,k) can be pre-processed into a graph, (G',k') such that $\delta(G') > 2$ and $k' < k$. The author has also shown that a pendant vertex can be deleted in almost all problem instances. A pendant vertex is a vertex of degree one.

(iv) Detecting special subgraphs: Abu-Khzam [2] has shown that detection of special subgraphs can simplify the path to finding a solution to the problem instance to a great extent. In the case of the Vertex Cover problem, the presence of a simple path of length *(2k+1)* in an instance *(G,k)* implies that *(G,k)* is a no instance or no cover of size k_{max} exists for the instance *(G,k)*.

Several other reduction or preprocessing techniques are discussed in [2]. Downey, Fellows and Stege [37] give a comprehensive outlook of the notion of Parameterized Complexity with special emphasis on the Vertex Cover problem.

However, these reduction techniques or preprocessing techniques are not computationally intensive. This thesis does not implement these techniques on hardware. Rather we concentrate on the computationally intensive part, namely branching.

2.4.2 Search Techniques for Finding a Solution to the Vertex Cover Problem

The fundamental idea behind finding an optimal cover to the graph lies in the fact that any vertex (chosen at random) or all of its neighbors have to be in the cover for a solution to be obtained. This property of the vertex cover problem is exploited to find an optimal solution given a graph G(V,E) and a parameter k.

In order that we minimize the number of iterations to find a solution, we choose vertices based on degree (rather than choose vertices at random). In this regard, it has been observed (from solutions) that, more often than not, the vertex of highest degree ends up being in the cover. By the property stated above, we can now start the algorithm with the assumption that the highest degree will be in the cover.

The algorithm then proceeds in a recursive fashion by adding more vertices or the neighbors of the vertices to the cover. Since there are two possible ways or forking or branching at each selected vertex, search tree algorithms are often referred to as branching algorithms [2].

2.4.3 Obtaining an Initial Solution and the Backtracking Approach

Rather than find a solution by an exhaustive search method, the branching algorithm proceeds by finding an initial partial solution, which may or may not represent the final correct solution. The algorithm then systematically proceeds by either finding a subset of the graph that represents the solution or by hitting a constraint that makes it impossible to process more nodes in the graph. In either case, the algorithm proceeds by returning to an earlier partial solution (stored in a stack) and taking the alternate choice. Thus we call this as a backtracking approach.

Remark 1

During the backtracking process, if the assumption that " the maximum degree vertex is in the cover" does not hold and if the number of neighbors of the highest degree vertex is greater than the parameter k, then we can safely declare that no solution is possible for the requested parameter k.

Remark 2

During the backtracking process, if all the possible nodes (dictated by the algorithm) have been visited and no solution has been found, we can again declare that no solution is possible for the requested parameter k

2.4.4 Algorithmic Formulation

The algorithmic formulation of obtaining an initial solution and the backtracking approach is described below. Given a graph G(V,E) and a parameter k, the algorithm for finding a cover of size $\leq k$ is as follows

while vertex_count $\leq k$ {

vertex of highest degree added to the cover

vertex_count = vertex_count + 1

if edgeless{

solution found ➔ *done}*

}

k_edit = k

backtracking starts / continues:

neighbors of k_edit vertex added to the cover

k_new = k_new + 1

if number_of_neighbors of most recently added vertex > k_new {

parameter value condition violated

k_edit = k_edit –1

}

elsif number_of_neighbors of most recently added vertex = k_new {

if edgeless{

solution found ➔ *done}*

else {

k_edit = k_edit –1}

backtracking continues

}

else{

number_of_neighbors of most recently added vertex < k_new {

while vertex_count ≤ *k* {

vertex of highest degree added to the cover

vertex_count = vertex_count + 1

if edgeless {

solution found ➔ *done}*

}

k_edit = k_edit –1

backtracking continues

}

if top of stack reached (

declare no solution for requested parameter

}

close

In this chapter, we discussed some of the key aspects of reconfigurable computing related to the hardware acceleration of optimization problems. In the next chapter, we discuss the actual implementation of the branching algorithm on the Pilchard reconfigurable platform.

Chapter 3

Approaches to Branching Implementations

We seek to devise and develop efficient algorithms for solving large problem instances. Techniques such as the Brute-force and Bounded search trees are used to implement this. The bounded search tree technique is a commonly used approach for solving many interesting problems. The Brute-force technique as discussed below is a totally exhaustive technique in comparison to the bounded search technique that is selective in its search space.

3.1 The Brute-Force Branching Technique

The brute-force branching technique as the name suggests, is an algorithm that performs a truly exhaustive search of the search space without exploiting any properties or regard to any sort of logical conclusions that can be derived from a graph. For example, In the Vertex Cover problem, given a graph *G(V,E)* and a parameter *k*, any vertex chosen or all of it neighbors have to in the cover.

The brute-force technique does not take into account any such property. Instead what it does is a fully exhaustive search of the search space. This is illustrated with the help of the following example. The graph considered in the example is shown in figure 3.1.

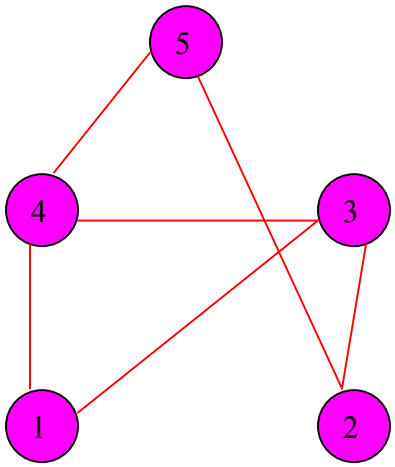

Figure 3.1 A simple graph to illustrate branching techniques

3.1.1 Why is the Brute-Force Technique Inefficient?

The search space that the brute force algorithm goes through before finding a solution is shown in table 3.1. The brute-force technique execution time grows exponentially with the value of the parameter *k*. For a graph of size k, the number of possible iterations or search spaces that the algorithm has to go through is 2^k. For large problem instances, the brute force algorithm introduces redundancy. Table 3.1 shows an example of the exhaustiveness of the search approach.

From a hardware perspective, the brute force algorithm can be easily implemented as a modified counter. However, the catch is that the time required to find a solution also grows exponentially with the problem size. In the table shown below, the highlighted parts of the text represent cases, in which the brute-force algorithm does find a solution, although the number of vertices in the cover exceeds the parameter *k*.

Table 3.1 Search space for an instance *(I,k) where k=2.*

Number of Iteration	Cover Vector					Edgeless (Yes/No)	Cover < *k*(Yes/No)
	1	2	3	4	5		
1	0	0	0	0	1	No	NA
2	0	0	0	1	0	No	NA
3	0	0	0	1	1	No	NA
4	0	0	1	0	0	No	NA
5	0	0	1	0	1	No	NA
6	0	0	1	1	0	No	NA
7	0	0	1	1	1	Yes	No
8	0	1	0	0	0	No	NA
9	0	1	0	0	1	No	NA
10	0	1	0	1	0	No	NA
11	0	1	0	1	1	No	NA
12	0	1	1	0	0	No	NA
13	0	1	1	0	1	No	NA
14	0	1	1	1	0	Yes	No
15	0	1	1	1	1	No	NA
16	1	0	0	0	0	No	NA
17	1	0	0	0	1	No	NA
18	1	0	0	1	0	No	NA
19	1	0	0	1	1	No	NA
20	1	0	1	0	0	No	NA
21	1	0	1	0	1	Yes	No
22	1	0	1	1	0	No	NA
23	1	0	1	1	1	Yes	No
24	1	1	0	0	0	No	NA
25	1	1	0	0	1	No	NA
26	1	1	0	1	0	No	NA
27	1	1	0	1	1	Yes	No
28	1	1	1	0	0	No	NA
29	1	1	1	0	1	Yes	No
30	1	1	1	1	0	Yes	No
31	1	1	1	1	1	Yes	No

We can infer from table 3.1 that the brute force algorithm required 32 steps to arrive at a conclusion that no cover of size less than or equal to *k* exists.

In table 3.1, the entire search space for the brute-force branching is shown. In the succeeding sections, we shall see how the bounded search technique is more efficient than the brute-force technque The search space of the brute-force technique grows exponentially as the size of the problem. In fact, adding just one more node to the example shown in figure 3.1 would double the existing search space. Hence the brute force is a computationally intensive algorithm that is impractical as the problem size scales up-to even modest graph sizes of 50 vertices

3.1.2 Why the Bounded Search Technique?

It is imperative that we maintain a balanced decomposition of the search space to achieve scalability [38]. In a worst-case scenario, the asymptotically fastest ***FPT*** algorithm currently known for vertex cover is due to the work of Chen at al [39][38], and runs in $O(1.2852^k + kn)$. The brute force technique in comparison takes $O(n^k)$, to examine all subsets of size *k*. The bounded search tree technique consists of an exhaustive search in a tree whose size is bounded by a function of the parameter. The search for finding the cover is usually done using a depth-first search. The basis for selecting nodes to be in the cover is based on the highest current degree node. The tree branches at every selected node. At every selected node, there are two ways of branching. The first path is to assume that the selected node is in the cover and proceed. The second path is to assume that the neighbors of the selected vertex rather than the selected vertex are in the cover.

Thus the left subtree denotes the path that the selected vertex is in the cover. The right subtree on the other hand denotes the path that the neighbors of the selected vertex are in the cover. At this point, it is interesting to note that solutions are found faster if the neighbors of an earlier selected vertex are in the cover. This is because, when the selected vertex *v* is assumed to be not in the cover, all of its neighbors must be in the cover. If the degree of *v* is high, we converge faster to the solution.

If *(G,k)* is an instance of the vertex cover problem, the search for an answer(Yes/No) proceeds using the following search technique. Let *xy* be an edge in the graph *G*. Either *x* or *y* or both belong to the cover. We can take one of two paths here. We can either assume *x* to be in the cover and proceed or assume *y* to be in the cover and proceed recursively. If we assume *x* to be in the cover, the search proceeds with a new graph *(G-x,k-1)*. Similarly, if we assume *y* to be in the cover, the search proceeds with a new graph *(G-x,k-1)*. If *(G-x,k-1)* is edgeless, then we add *x* to the solution and stop. If not, we keep iteratively adding nodes or vertices of highest current degree and proceed. If the number of vertices added exceeds *k*, we retract (backtrack) the steps that we came through, and add the neighbors of the nodes that we had most recently added. Thus the number of possible covers in this particular search tree is 2^k.

3.2 Backtracking

The process of retracting the steps that the search tree came through initially and taking the path of the right subtree that was not taken previously is called backtracking. To illustrate this idea, we use the graph shown earlier in figure 3.1. This technique is computationally less intensive in comparison to the brute-forcce technique. The graph is

shown again in figure 3.2. Shown in figure 3.3 is the pictorial representation of the backtracking process. The reader will observe that the search space is now visibly reduced and that an answer (Yes/No) is found much quicker, in comparison to the brute-force approach. This effect is more profound is large graph instances, wherein the brute force algorithm takes a longer time to find an answer.

3.3 Hardware Implementation on the Pilchard

The branching process is found to be split-up into the following functions.

1. Function to select the highest degree vertex based on the current graph
2. Function to check if the graph is edgeless
3. Function for backtracking and adding the neighbors of the most recently added vertex
4. Function to maintain and update the stack (to store intermittent values of the cover vector at each leaf node)

It is important that we design each of the above steps in such a way that we obtain maximum concurrency and thus generate an appreciable speed-up over the software version of branching. Keeping this mind, the above-mentioned blocks were designed to obtain maximum parallelism and concurrency. On closer analysis of the graphs, it was clear that one could obtain considerable speedups by improving upon those modules in which the software versions of branching consumed a lot of time. The four points mentioned above fell into this category and hence the motivation to devise efficient hardware implementation of the same.

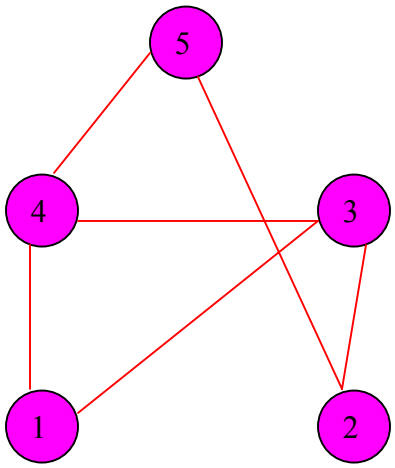

Figure 3.2 A simple graph to illustrate the backtracking approach

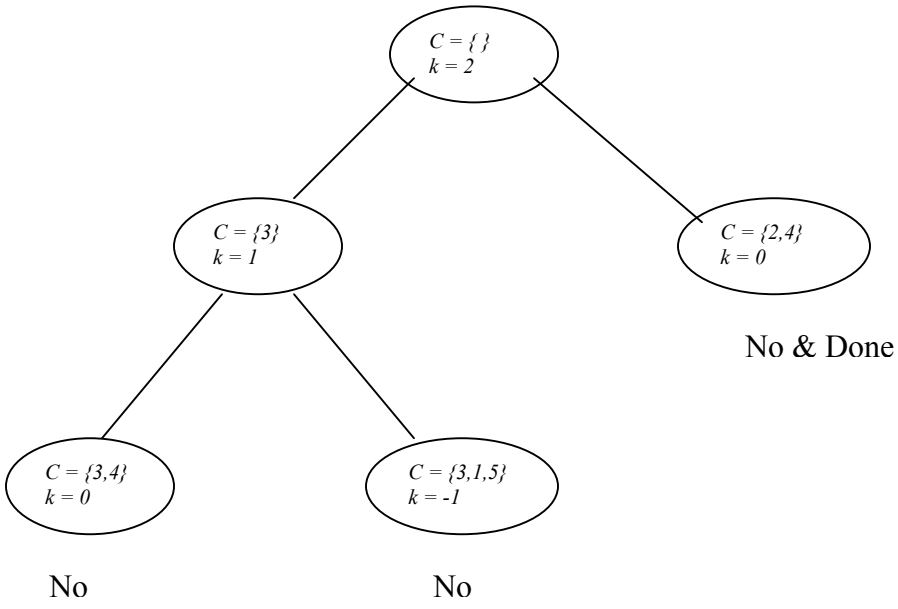

Figure 3.3 The backtracking process

3.3.1 Design of the Select Function- Ones Counting, an Important Combinational Block

The select vertex function systematically scans through each node of the graph and computes the degree of each node and thereby finds the maximum degree vertex based on the "current graph". The word "current graph" is important here because the graph is assumed to be devoid of all edges that emanate from a vertex that has already been added to the cover. For example, for the graph instance shown in figure 3.1, at the end of the first iteration, the maximum degree vertex is 3. After vertex 3 has been added to the cover, all the edges that are incident/emanate on/from it are removed and the graph is modified as shown in figure 3.4. Figure 3.5 shows a further modified graph, after node 4 has been removed. Now the maximum degree vertex is 4. In instances where there are more than one node that have the same maximum degree, the vertex that appears earlier in the search is added to the cover. For example, if in an instance, node *8* and *11* shares the same degree of say 56, node *8* is chosen ahead of *11*.

The degree of a vertex is found by counting the number of incident edges it has. In an adjacency matrix, a '1' represents the existence of an edge between any two nodes and a '0' represents the absence of an edge. Hence to ascertain the degree of a node, we have to count the number of edges (represented by a '1' in the adjacency matrix) that are currently not covered by any node in the graph. There are a number of ways to do this and the most commonly used ways are

1. Using a sequential counter to count the number of ones
2. Using look-up tables
3. Using adder trees

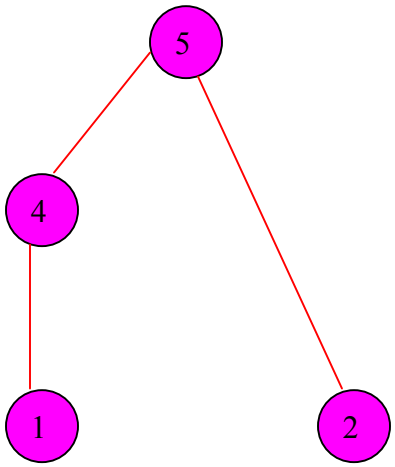

Figure 3.4 After node 3 has been removed

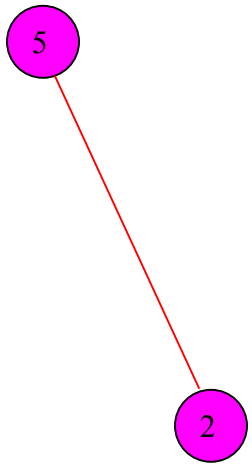

Figure 3.5 After node 4 has been removed, graph is still not edgeless

All the above methods are discussed in the following sections

3.3.1.1 Using a Sequential Counter to Count the Number of Ones

Several important algorithms include the step of counting the number of "1" bits in a data word. Shown in figure 3.6 is the pictorial arrangement of the adders for the proposed 16 bit ones counter. A behavioral VHDL program, as shown in figure 3.7, can describe ones counting very easily. The RTL description shown in figure 3.7 is that of an ones counter that capable of counting the number of ones in a 16 bit data word. Although, this program is fully synthesizable, it generates a very slow, inefficient realization with 15 4-bit adders in series.

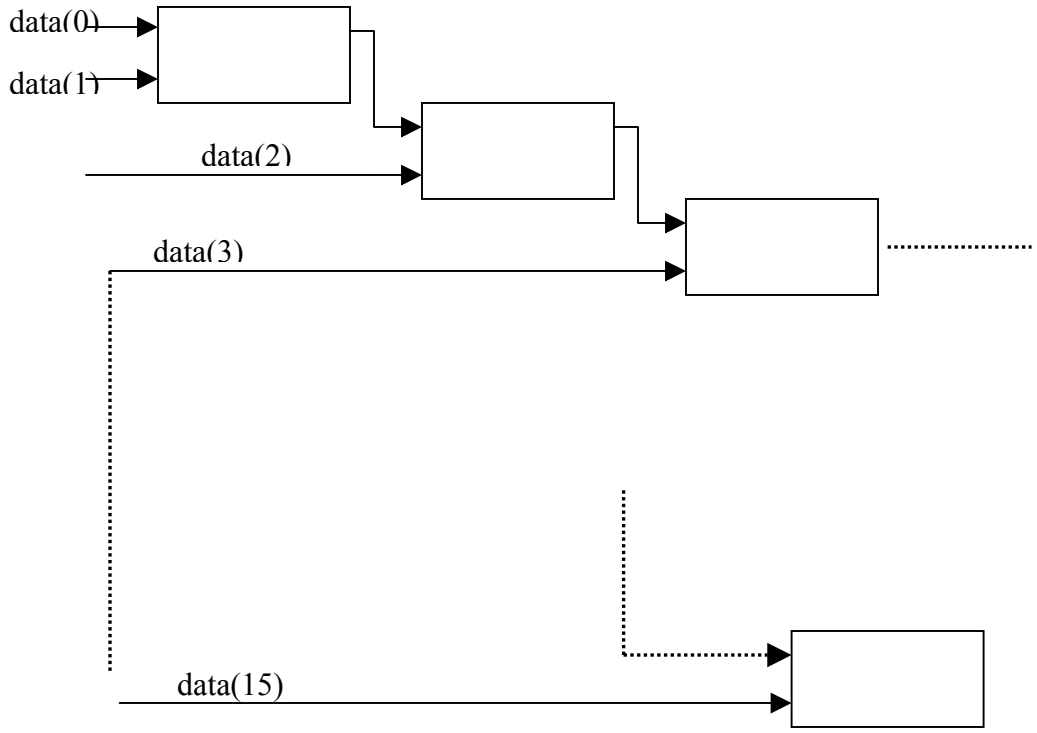

Figure 3.6 Schematic of a sequential ones counter

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity seq_count is
port    (
        data_in: in STD_LOGIC_VECTOR (15 downto 0);
        ones_count: out STD_LOGIC_VECTOR (3 downto 0)
        );
end seq_count;

architecture seq_count_a of seq_count is

begin
process (data_in)
        variable tmp_ones_count : STD_LOGIC_VECTOR(3 downto 0);

begin

tmp_ones_count := "00000";

for i in 0 to 15 loop
        if (data_in(i) = '1' ) then
                tmp_ones_count := tmp_ones_count + "0001";
        end if;
end loop;

ones_count <= tmp_ones_count;

end process;
end seq_count_a;
```

Figure 3.7 Sequential ones counter

3.3.1.2 Using Look Up Tables for Counting the Number of Ones

As the name suggest, look up tables "look up" the value for a set of data inputs, from a pre-determined list of values. Since they do not need to explicitly perform calculations, they possess very little delay.

However, the drawback in using look up tables is their size. A complete look up table has to contain all the combinations of the possible inputs. In the case of counting the number of ones from a data word of 16 bits, there are 2^{16} possibilities.

Shown in figure 3.8 is the layout of the 16-bit look up table.To generate this look up table, MATLAB was used as a scripting tool. This script is shown in figure 3.9.

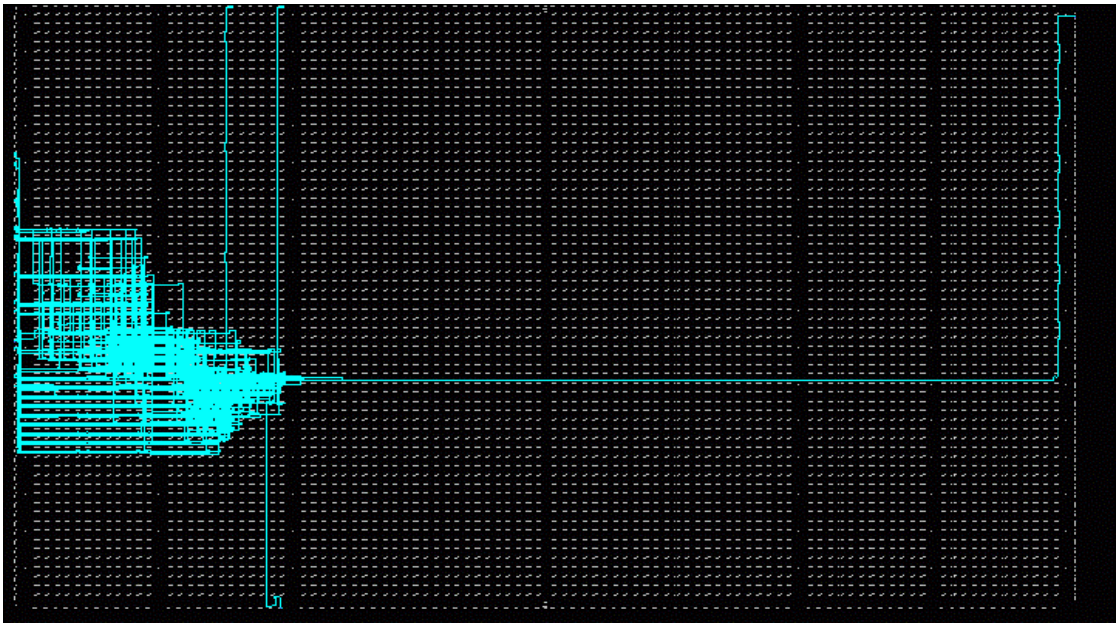

Figure 3.8 Layout of a 16 bit look up table

```
%function vhd_gen(n,bit_width,LUT_size)
%profile on -detail builtin
clc
clear;
close all;
home;
LUT_size=16;
n=2048;
bit_width=16;
i=0:n;
s=dec2bin(i,bit_width);
d = sum(s,2);
temp=d(1);
final_one=dec2bin(d-temp);
[x,sum_width]=size(final_one);

%opening file for writing

fname=sprintf('vhd_gen%d.vhd',LUT_size);
fprintf('creating file %s\n',fname);
fid=fopen(fname,'w');
%writing beginning stuff to the file

fprintf(fid,'-- vhdl file for 16 bit LUT \n');
fprintf(fid,'-- %s',fname);
fprintf(fid,' contains %d points of %d bit width \n',n,bit_width);

fprintf(fid,'LIBRARY ieee;\nUSE ieee.std_logic_1164.ALL;\nUSE
ieee.std_logic_arith.ALL;\n');
fprintf(fid,'\n\nENTITY lut16 IS\n    GENERIC(\n');
fprintf(fid,'        bit_width : integer :=%d;\n',bit_width);
fprintf(fid,'        sum_width : integer :=%d\n',sum_width);
fprintf(fid,'    );\n    PORT(\n');
fprintf(fid,'        bit_vector :in std_logic_vector (%d downto
0);\n',bit_width-1);
fprintf(fid,'        one_count : OUT std_logic_vector ((sum_width-1)
DOWNTO 0));\n');
fprintf(fid,'end lut16;\n');

%begin writing architecture

fprintf(fid,'ARCHITECTURE behavior OF lut16 IS\n\n BEGIN\n\n');
fprintf(fid,'process(bit_vector)\nbegin\n    case bit_vector is\n');

for i=1:n+1
        fprintf(fid,'        when "');
```

Figure 3.9 Matlab code to generate a 16 bit look up table

```
        for j=1:bit_width
            fprintf(fid,'%s',s(i,j));
        end
        fprintf(fid,'" => ');
        fprintf(fid,'one_count <= "');

        for k=1:sum_width
            fprintf(fid,'%s',final_one(i,k));
        end

        fprintf(fid,'";\n');
end

fprintf(fid,'        when others => one_count <= "11111";\n');

%fprintf(fid,'        when others => \n');
fprintf(fid,'    end case;\n\n');
fprintf(fid,'end process;\nEND behavior;\n');
fclose(fid);
disp('done')
%profile report
```

Figure 3.9 (Continued)

3.3.1.2.1 Synthesis and Timing Results for the 16 bit Look Up Table

As expected, the look up table turned out be very bulky and occupied a sizeable part of the FPGA. The 16-bit look up table occupied 151 out of the available 12288 slices. Although this appears as a small number, this number would pose a severe bottleneck when the problem size is scaled up. For example, when the problem size is 256, we would require a minimum of 16 look up tables. This translates to the look up tables occupying 2416 slices, or 20 % of the chip, a certainly unacceptable number. Moreover, the bulky nature of the look up tables makes it difficult for the place and route tool to efficiently place and route the design for obtaining good speed.

In fact, these look up tables themselves take a large amount of time to go through the synthesis, mapping and place and route process. A hierarchical synthesis method was used to synthesize them. Synopsys FPGA compiler was used to synthesize them. Synthesis alone took close to 36 hours.

Even though the already synthesized look up table was used in the overall design, the final design exhibited huge synthesis and place and route times. Hence the design flow from a RTL level description to a bit-level generation took close to 2 hours at times.

Due to all these factors, the adder tree approach discussed in the next section was used to count the number of ones.

3.3.1.3 Using Adder Trees to Count the Number of Ones

To synthesize a more efficient realization of the ones counter, we must come up with an efficient structure and then write an architecture that describes it. Synopsys Designware components were used to implement the individual adders.

The adder trees occupied a very low percentage area of the chip in comparison to the look up table. The number of slices that the adder tree occupied was a mere 18 slices in comparison to the 151 occupied by the look up table. Since the adder trees were not bulky, the processes of synthesizing and place and route became easier and more importantly faster!

3.3.1.4 Function to Select the Highest Degree Vertex Based on the Current Graph

The “select highest degree vertex” function is implemented based on the current graph and the current cover vector. At any point, nodes that are already present in the

cover are not considered towards determining the current highest degree vertex. To illustrate this further, in the example shown in figure 3.10, the highest degree vertices are 3 and 4.

By our search technique, since 3 appears earlier in the search, node 3 is assumed to be included in the cover. Now all the edges that are incident on 3 are removed and a new graph is constructed. It can be seen from the modified graph shown in figure 3.11 that all edges incident on node 3 have been removed. The next call to the function "select highest degree vertex is based on this new graph as shown in figure 3.11. In this new modified graph, the highest degree vertex is 4. It is this evident, as to the choise of high degree vertices. Shown in figure 3.12 is the flowchart for the implementation of the "select vertex" function.

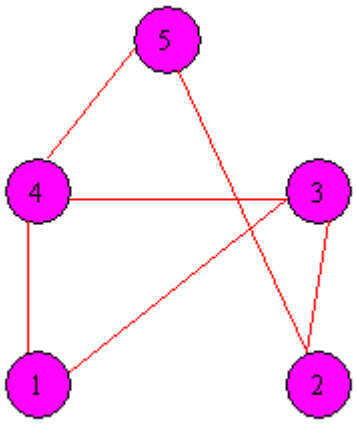

Figure 3.10 Graph to illustrate "select highest degree vertex" process

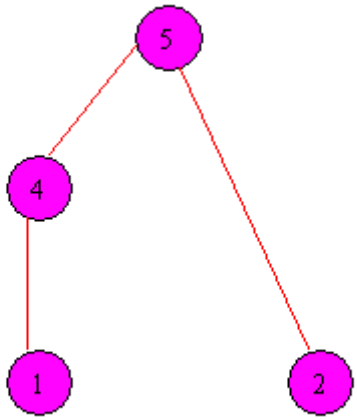

Figure 3.11 Modified graph

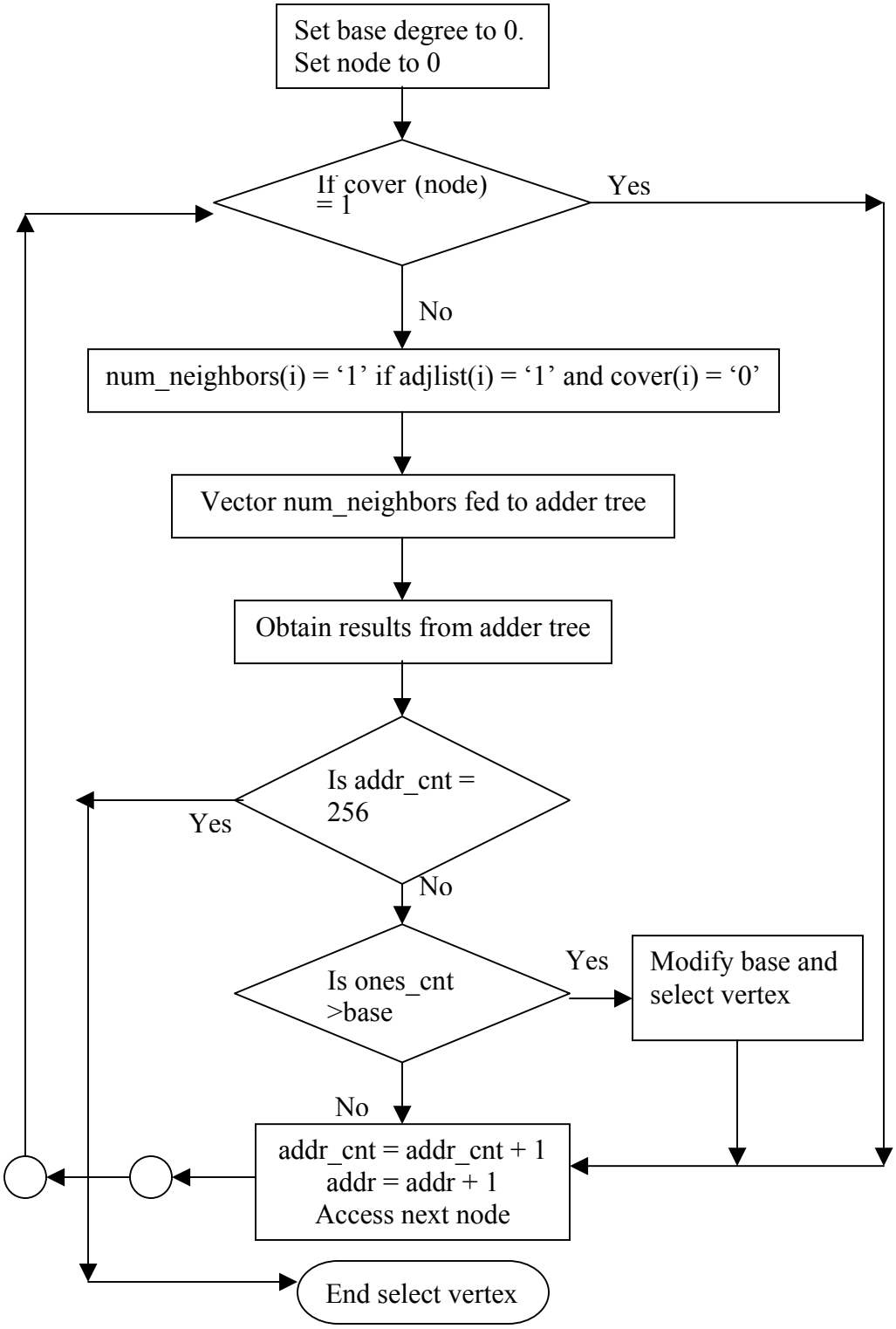

Figure 3.12 Flowchart for implementing the function "select vertex"

The state machine representation of the “select highest degree” function is shown in figure 3.13. Also shown in table 3.2 is the state machine encodings of the “select highest degree vertex” function.

3.3.2 Function to Check if the Graph is Edgeless Based on the Current Cover Vector

The “edgeless check” function is implemented based on the current graph and the current cover vector. If a node is found to be in the cover vector, all the edges incident on it are covered, and this is a forgone conclusion. However, if a node is not present in the cover, we will have to check if all the edges that are incident on it are covered. Even if one of the edges is not covered, we declare that the graph is not fully edgeless and the branching process is continued from the point it was stopped. The flow chart for the implementation of the edgeless check function is shown in figure 3.14. Shown in figure 3.15 is the state machine representation of the “edgeless check” function. Also shown in table 3.3 is the state machine encodings of the same.

3.3.3 Recursive Implementation - Maintaining and Updating the Stack

Unlike the C programming language that dynamically updates and stores the stack for each recursive function call, VHDL or for that matter, no hardware description language supports arbitrary depth recursion. Any kind of recursive implementation must have a bound on it at run time.

Hence, stacks have to be exclusively created in advance for implementation that are recursive. The branching process being an inherently recursive implementation, warrants the creation of such a stack to store the intermittent values of the cover vector.

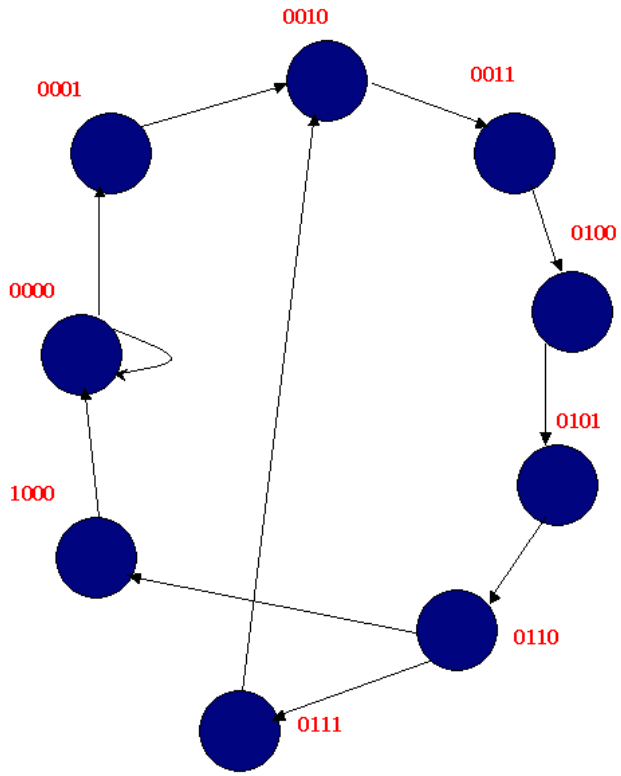

Figure 3.13 State machine implementation of the "select highest degree vertex function"

Table 3.2 State machine implementation of the "select highest degree vertex function"

State Machine Encoding	Function of state
0000	Idle State
0001	Initialization State
0010	Adder Pipeline stage 1
0011	Adder Pipeline stage 2
0100	Adder Pipeline stage 3
0101	Adder Pipeline stage 4
0110	Address counter check state & Degree check state
0111	Wait state & Address increment state
1000	Degree check of final address

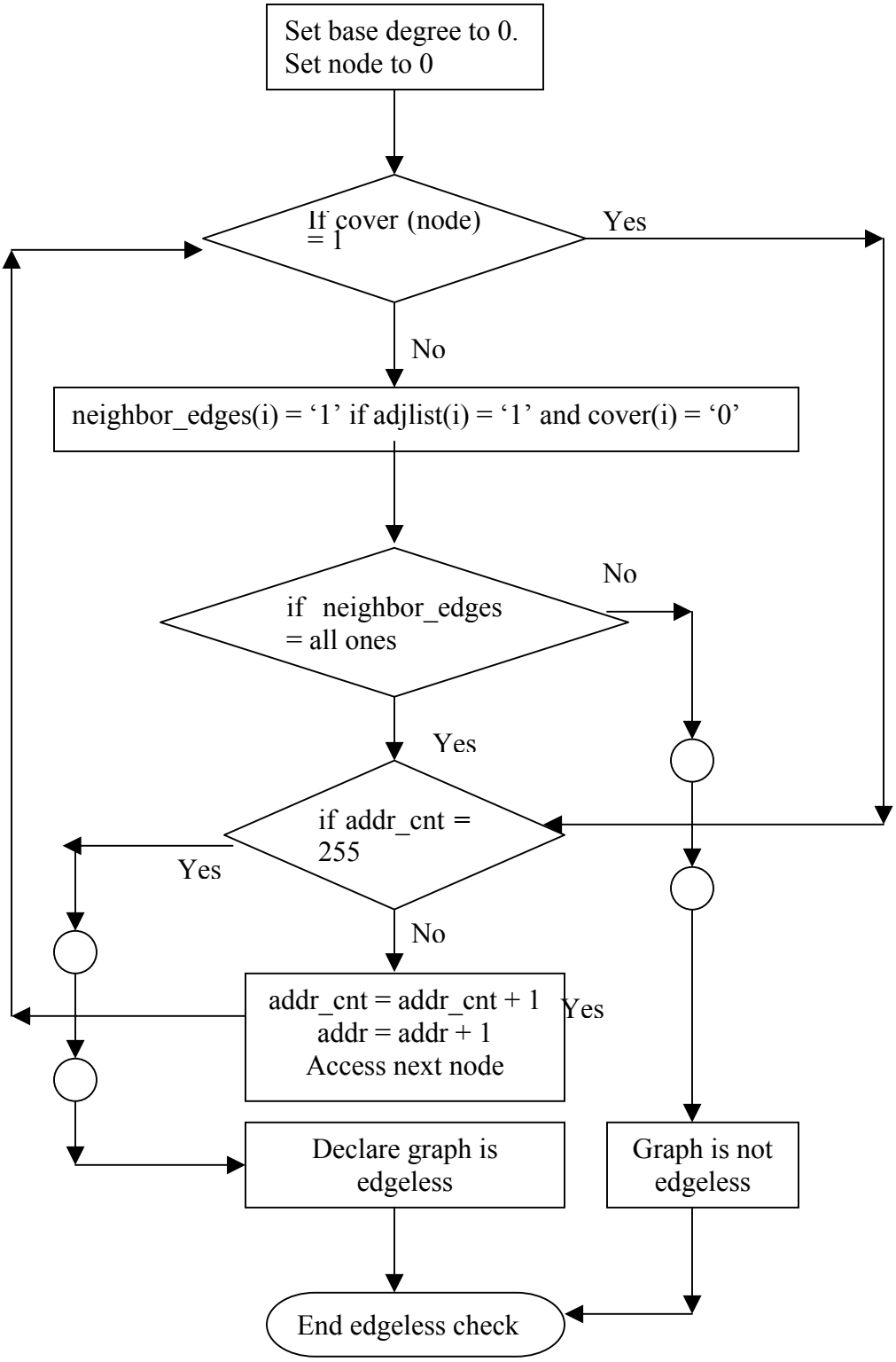

Figure 3.14 Flowchart for implementing the function "edgeless check"

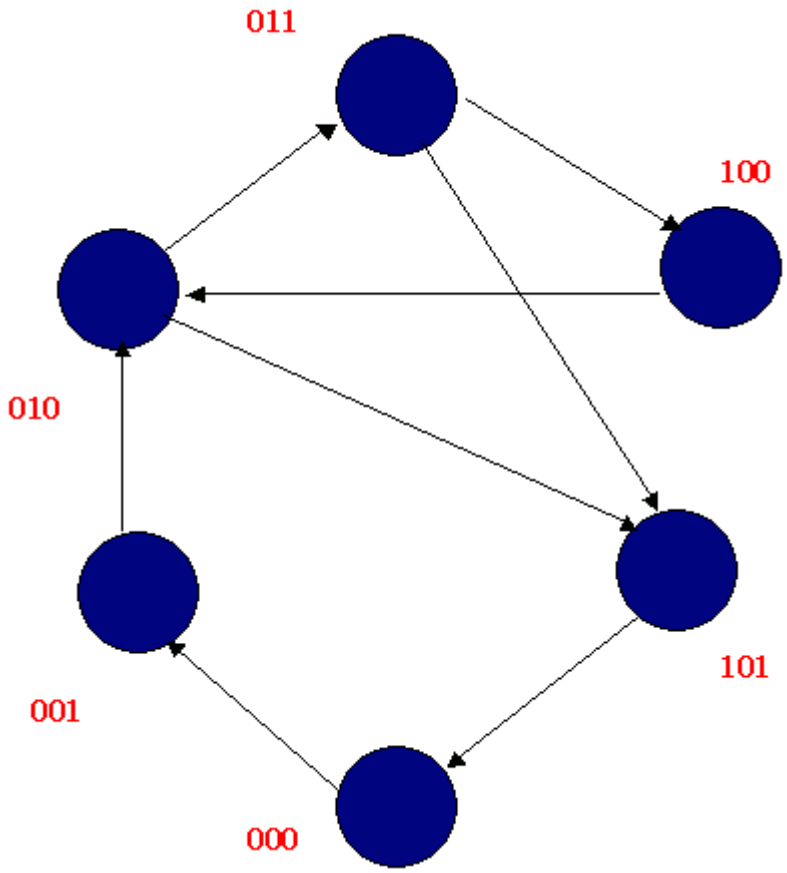

Figure 3.15 State machine implementation of the "edgeless check" function

Table 3.3 State machine implementation of the "edgeless check" function

State Machine Encoding	Function of state
000	Idle state
001	Initialization state
010	Counter check state
011	Edgeless vector check
100	Address increment and wait state
101	Edgeless check of last address

```
subtype reg is std_logic_vector(63 downto 0);
type regArray is array (integer range <>) of reg;
signal registerFile : regArray(0 to 63);
```

Figure 3.16 Creating a stack on chip

One of the simplest methods of creating a stack on chip is shown in figure 3.16. The stack shown in figure 3.16 has a width of 64 and a depth of 64. This approach did not pose any problems for small problem instances. For small problem instances of size 16 and 32, the total area occupied on the chip was not an appreciable one. There were no errors or discrepancies in timing too. Shown in table 3.4 and table 3.5 is the respective area and timing report's of the 16 and 32 bit problem instances. However, when the problem size was scaled up to a size of 64, an exponential increase in the area occupied was observed. The timing results were still worse, with the timing even failing to meet the minimum required speed of 6 MHz!

One of the other drawbacks of using this approach was that the time taken for synthesis and place and route was agonizingly huge. It turns out any kind of exercise to build a large memory on chip is just not worth it, be it an FPGA or an ASIC. Shown in table 3.6 is the time taken for the place and route process for different problem instances.

It was apparent that, building a stack or memory on chip, using the real estate on chip was a futile exercise. It was beneficial to use this approach for small instances, but a "strict no" for bigger problem sizes. One of the possible alternatives was to use a memory component from external vendors such as Synopsys Designware. However, documentation manuals [42]

Table 3.4 Area occupied by each problem instance

Problem Instance Size	Number of slices	Percentage occupation on chip
16	709 out of 12288	5%
32	2079 out of 12288	16%
64	9315 out of 12288	75%

Table 3.5 Timing report for each problem instance

Problem Instance size	Attempted Maximum Speed (MHz)	Tool Generated Maximum Speed (MHz)	Timing Failure/Success
16	33	35	Successful
32	20	22	Successful
64	6	On the order of a few kilohertz	Failed

Table 3.6 Time for place and route for each problem instance

Problem Instance size	Time for Place and Route
16	7 minutes
32	27 minutes
64	2 hours and 17 minutes

from Designware suggested that their RAM's and ROM's were to be used only as a scratch-pad memory and not for implementing huge data-paths on chip.

The only other viable alternative was to use the Xilinx Dual Port RAM on the chip. This approach was not pursued in the beginning because of latency issues. Shown in figure 3.17 and figure 3.18 are the read and write timing diagrams [43] for the Xilinx Dual Port RAM. It is evident from the figure that there is a definite lag (delay) between the onset of an address on the address bus and the appearance of the contents of the address on the output data bus.

3.4 Memory Issues for Implementation of Graphs of Size Greater than 64

One of the main limitations of the Pilchard reconfigurable platform is the limited addressing capability. Although, 14 address lines are provided, only 8 of then can actually be used. Hence, the designer is limited to addressing just 2^8 or 256 addresses from the console. Compounded to this problem is the capability of the data bus of the Pilchard. The input and output data bus of the Pilchard reconfigurable platform being

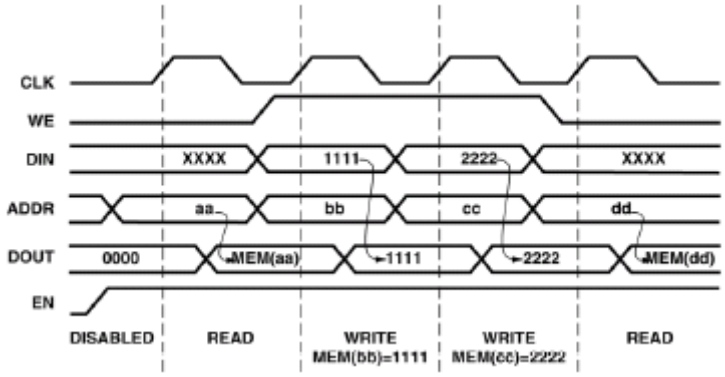

Figure 3.17 Timing diagram of writing to the dual port RAM [43]

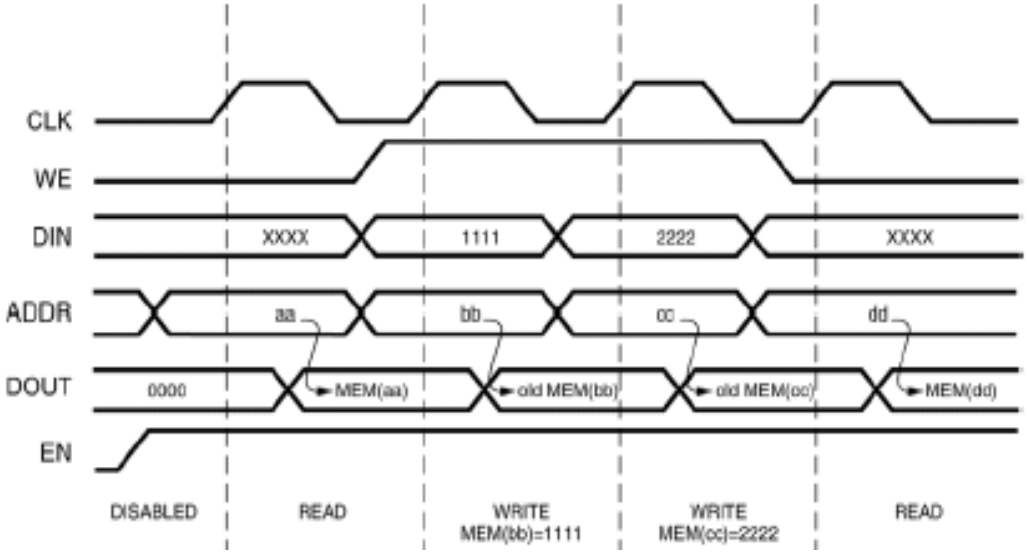

Figure 3.18 Timing diagram of reading from the dual port RAM [43]

limited to a width of 64 bits provides a serious impediment to the efficient execution of the algorithm.

For graphs of size 64 and less, this was never a problem. Trouble begins when we target graphs of size greater than 64. The data structure used in the work here is an adjacency matrix, essentially a square matrix. Once the size of the adjacency matrix exceeds 64, we cannot transfer the entire contents of a row or a column of the matrix in a single transaction. Questions then arise as to a suitable method of transferring the entire adjacency matrix onto the onboard Xilinx Virtex RAM. Several ideas were experimented with. They are discussed in the sections that follow.

3.4.1 Method 1: Using the Symmetry of the Adjacency Matrix

One of the first methods to be discussed was the exploitation of the symmetries of the adjacency matrix. Since the adjacency matrix is symmetrical about the diagonal, it naturally becomes a choice. Given either the upper triangular or the lower triangular matrix of any adjacency matrix, it is easy to reconstruct the graph because of the symmetries. The following algorithm extracts the row of the vertex without re-constructing the entire graph. With the example graph and adjacency matrix shown in figure 3.19 and figure 3.20, the algorithm is verified.

In the adjacency matrix representation of the sample graph shown in figure 3.16, 1,2,3,4,5 represent the vertices. A, B, C, D, E are variables that are either 1's or 0's that represent the existence of a connection between the vertices. Hence the dotted lines denoted the existence of an edge.

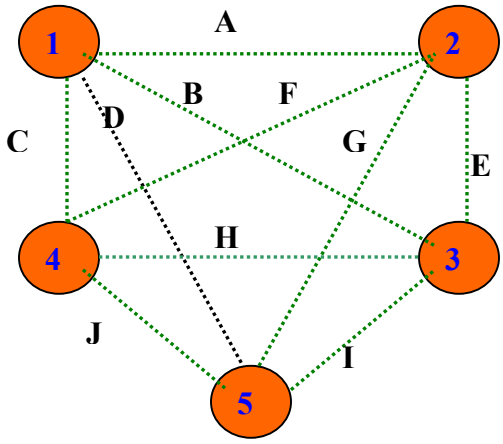

Figure 3.19 Sample graph of size 5

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

Figure 3.20 Adjacency matrix representation of figure 3.19

The number of elements required to represent the 1st row of an adjacency matrix of n vertices, excluding the element along the diagonal of the matrix is *(n-1)*.

Similarly the number of elements required to represent the 2nd row of an adjacency matrix of n vertices, excluding the element along the diagonal of the matrix is *(n-2)*.

Going by the same lines of reduction, the number of elements required to represent the m^{th} row of an adjacency matrix of n vertices, excluding the element along the diagonal of the matrix is *(n-m)*.

Therefore, the total number of elements required to represent the entire adjacency matrix

$$(n-1)+(n-2)+n-3)+.....(n-m)+1+0$$

Note that the last row needs 0 unique elements to represent it.

Hence, it is fairly evident that the number of elements required to represent a graph of size *n* is just the elements of the upper triangular matrix and is given by

$$Number\ of\ elements = [n*(n-1)\div 2]$$

In the graph shown in figure 3.17, the graph is of size 5 and hence the number of elements required is

$$[5*(5-1)\div 2]=10$$

Having derived this, we now aim to obtain the row vector corresponding to a particular vertex "*i*". The row vector corresponding to any vertex is divided into two parts, the divider being the "0" along the diagonal. We shall use this property to extract the row vector corresponding to the vertex *"i"*.

This process is split into 3 stages:

Pick until *(i-1)* elements of a total of *n* elements in the order shown below, where *n* represents the size of the square matrix.

$(i-1)^{th}$ element, $(i+2)^{nd}$ element, $(i+4)^{th}$ element, $(i+6)^{th}$ element..

……..(i-1)elements

- After we extract the above elements, we then append a zero this result
- All that we are left with is to add the rest of the elements. We have already extracted *i* elements. We have to extract the remaining *(n-i) elements.* So we add the remaining *(n-i)* elements starting from the element represented by the expression

$$[(i-1)*(n-(i/2))+1]$$

- Exceptions to handle

The algorithm will hold for all the vertices except the last element of the last vertex. But in any case, we will be handling the first and the last vertex separately.

3.4.1.1 Limitations of Using this Approach

The limitation of using this approach is that a lot of time is wasted in reconstructing the matrix every time a row of the matrix needs to be addressed. The branching algorithm is a highly data intensive algorithm in the sense that the access to the adjacency matrix is frequent. Any approach that wastes a lot of time reconstructing the

matrix would add a large overhead to the algorithm. Hence this approach was not used to address the memory problem that we were facing.

3.4.2 Method 2: Using More than one Address to Hold the Contents of a Row of an Adjacency Matrix

The maximum number of addressable locations in the Pilchard reconfigurable platform is 256. The input data bus of the Pilchard supported 64 bits of data transfer. A work around solution had to be thought of to address this data width problem as the adjacency matrices are square in nature. Hence an adjacency matrix of size greater than 64 would have data width greater than 64. Rather than use the address lines for addressing, the data input lines were used both for addressing and data input. The first 10 bits of the input data bus was used for addressing and the last 32 bits were used for data transfer. The rest of the bits were unused. Figure 3.18 gives an accurate idea of the addressing and data transfer process.

$$\underbrace{63,62,61,60,59,.....32}_{Data\_input},\underbrace{31,29.....10}_{Unused\_bits},\underbrace{9,8,.....0}_{Addres\sin g}$$

For example, a row of an adjacency matrix of size 128 would be broken up into 4 segments each of 32 bits, before transferring it to the onboard Xilinx Virtex RAM. In the example, the 128 bit wide vector is split into four contiguous segments of 32 bits each.

To access a row of the adjacency matrix, one would have to address a number of address location, depending on the problem size. For example, to access the contents of a single row of a graph of size 128, we would require 4 address reads. For a graph of size 256, one would require 8 address reads.

3.4.2.1 Advantages and Disadvantages of Using this Approach

By using this approach, the overhead of reconstructing the matrix is removed. Each row of the adjacency matrix is stored as a separate entity and so no time is wasted in trying to reconstruct the contents of a row of the adjacency matrix.

While this does not pose a problem in respect of an implementation point of view, accessing a row requires multiple reads, again an overhead considering the frequency with which rows of the adjacency matrix are addressed. Hence this approach was dropped in favor of an approach discussed in the next section.

3.4.3 Using a State Machine to Re-construct the Entire Adjacency Matrix

We observed that methods 1 and 2, proved inefficient and possessed large overheads, as far as the final implementation of the branching algorithm was concerned. Methods (1) and (2) exposed the chink in the armory of the branching algorithm. We needed the data corresponding to a row in a single shot rather than in spurts.

Data had to be arranged such that each row of the graph occupied exactly one row of the RAM. This way, there would be no overhead on the Branching algorithm on chip, as there would be only one memory access corresponding to the adjacency list corresponding to a vertex. So method (2) was modified to facilitate the re-construction of the matrix before the actual branching implementation commenced.

Shown in figure 3.21 is the devided algorithm to use more than one address to store the contents of one row. The only overhead in this approach would be that of the initial concatenation process. This however can be safely neglected, as it is very small. The algorithm for method (3) is discussed in figure 3.22

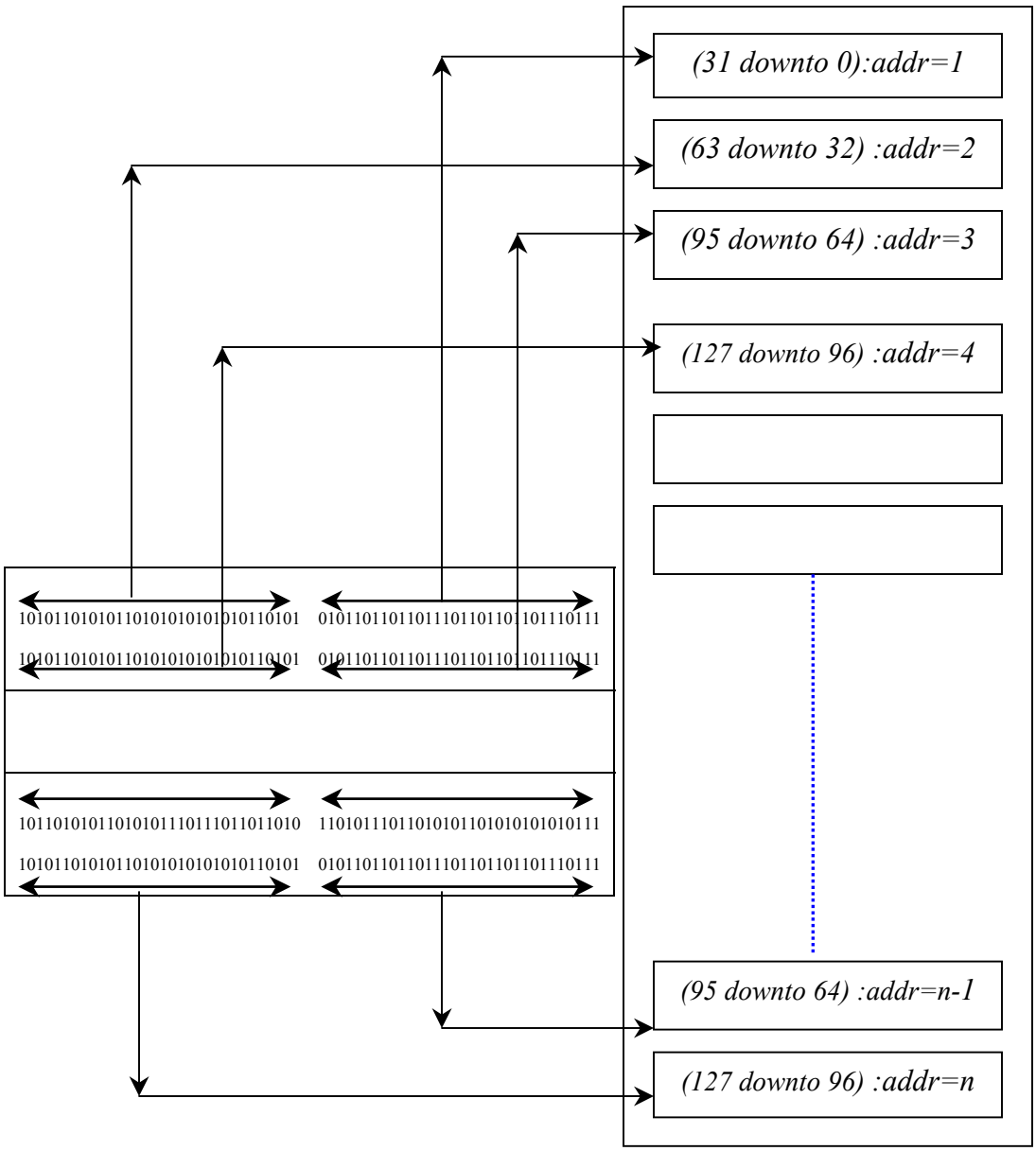

Figure 3.21 Using more than one address to store the contents of one row

The following algorithm lists the steps involved in writing an adjacency matrix of size 256 into the Xilinx Virtex RAM on the Pilchard.

1. *Break each row of the adjacency matrix into 32 bit chunks*
2. *Using the write64 C routine of the Pilchard Interface, write the entire contents of a row of the matrix, in 8 steps. For example, the first 32 bit chunk would be written to the 1st address, the 2nd 32 bit chunk to the 2nd address and so on.*
3. *After the entire matrix has been written in this fashion, use the "addr" line of the Pilchard to initiate the concatenation process*
4. *The concatenation process now starts.*
5. *After the state machine completes the entire process of concatenation, a "finish_load" signal is made high to signal the fact that the concatenation process is now complete and that the branching process can commence.*

Figure 3.22 Algorithm used for the RAM concatenation process

3.5 Reading the Final Output

In problems of sizes greater than 64, the final output, namely the cover vector is of size equal to the problem size. However, the output data line of the Pilchard platform supports just 64 bits. Hence, we have to read the final output in spurts of 64 bits. To do this, the final output has to be stored in some kind of a buffer or RAM in order that we read the bits in order.

For this purpose, an output RAM was created to store the cover vector before reading it out. Shown in table 3.7 is the breakup of the number of RAM blocks used for different problem sizes.

Table 3.7 Number of RAM blocks used for different problem sizes

Problem Size	Adjacency matrix size$^{\phi}$	Total No. of RAM blocks required
128	129 x 128	21
256	257 x 256	81
512	513 x 512	321
1024	1025x1024	1281
2048	2049x2048	5121
4096	4097x4096	20481

ϕ - Size of adjacency matrix is 129 x129 because the value of *k* too is fed into the initial input matrix

Chapter 4

Results

4.1 Test Vector Generation

For the purpose of debugging, test benches had to be built to simulate and debug in case of erroneous results. Unlike other test benches, which are written from scratch, in the branching implementation, automatic test bench generation became a necessity simply because of the huge amount of data involved. Scripts written in MATLAB® were used to generate test benches from the original adjacency matrix.

Some of the important signals or variables in the branching process are mentioned below.

1. order_vector – Stores the order in which the vertices are added to the cover.
2. stack_indicator – Serves to maintain the order in which the branching takes place. When the branching implementation steps to the backtracking process, it is imperative that we process all possible branches and do not miss any part of the search space. The stack indicator directs the implementation to the path it should take next, in an event of a solution not being found.
3. mask vector – Represents the cover of the process at any instant of time.

Table 4.1 Number of slices occupied by graphs of different sizes

Graph Size	Number of Slices occupied	Percentage Area Occupation
16	410 /12288	3
64	1287/12288	10
128	2613/12288	21
256	5898/12288	48

4.2 Hardware Implementation – Area Results

Shown in table 4.1 is the number of slices occupied by graphs of different sizes. Also shown in figure 4.1 is the area distribution for graphs of different sizes using a stack implemented with the following two methods:

1. Stack implemented using transistors on chip

2. Stack implemented using the Xilinx Virtex RAM

The Dual-Port Block Memory module for the Virtex 1000E part is composed of single or multiple 4Kilo-bits blocks called Select-RAM+™. The dual port memory has two independent ports that enable shared access to a single memory location. Simultaneous reads from the same memory location may occur, but all other simultaneous, reading-from, and writing to the same memory location will result in

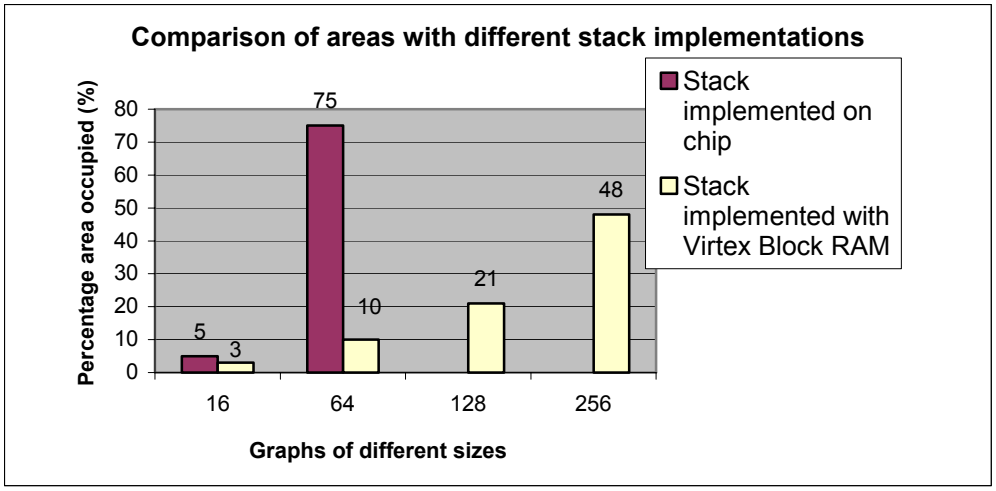

Figure 4.1 Percentage area occupancy with different stack implementation

correct data being written into the memory, but invalid data being read. The Virtex 1000E possesses 96 RAM blocks. It is interesting to note that the problem scales promisingly using a stack implemented with the Xilinx Virtex RAM. The data for the graphs of sizes 128 and 256 for the stack on chip implementation are not shown in figure 4.1 as they exceed the area of the chip. Hence these values were not shown in the figure

4.3 Hardware Implementation – Circuit Speed Results

In order that we obtain sufficient speeds of operation, critical paths in the design have to be broken to generate increased speeds of operation. Shown in table 4.2 are the speeds of operation with the stack implemented on chip.

It can be observed from table 4.2 that the 64 bit branching implementation with the stack implemented on chip fails to meet the timing requirements. Although the expected critical path in the design, namely the signal that computes the neighbor count of each vertex has been pipelined to increase the speed of the operations, the 64-bit

Table 4.2 Circuit speed of operation with stack implemented on chip

Graph Size	Percentage Area Occupation	Attempted Frequency (MHz)	Tool Generated Frequency (MHz)	Failure/ Success
16	5	33	35	Success
32	16	20	22	Success
64	75	6	In the order of a few KHz	Failure
128	Would have run out of space	----	---	---
256	Would have run out of space	---	---	---

implementation which fails miserably to meet even the lowest the timing constraint of 6 MHz. This timing failure can be attributed to the fact that the place and route process is severely impeded by the sheer volume of the design that it has to route.

However, in case of the implementation, with the stack being implemented on the Virtex RAM, the problem scales appreciably to allow for higher speeds of operation.

In direct contrast to the above seen results, the circuit implemented with the stack on the Virtex Block RAM, scales appreciably with good speeds of operation. Shown in table 4.3 are the speeds of operation for this approach.

4.4 Comparison of Software and Hardware Execution Time

The hardware and software branching implementations were executed and tested on randomly generated graphs. The hardware specifications of the machines on which the software implementation of branching was executed are shown in table 4.4

Shown in table 4.5 are the software and hardware execution times for random graphs. Speedups of several orders of magnitude have been obtained with the hardware implementation over the software implementation. The speedups obtained with the test graphs, range from a minimum of 59 to a maximum of 127. The minimum speedups were obtained on sparse graphs, which have relatively lesser edges. Lesser edges reduce the search space that the branching process has to cover and hence the lesser speedups.

Shown in figure 4.2 is a plot of the speedups obtained with the hardware branching implementation. The average speedup with the tested graphs was found to be 93

Table 4.3 Circuit speed of operation with stack implemented on the Virtex RAM

Graph Size	Percentage Area Occupation	Frequency (MHz)	Failure/ Success
16	3	40	Success
64	10	33	Success
128	21	33	Success
256	48	25	Success
512	25-35 on the latest Virtex2 Pro	25(expected)	Expected success
1024	Close to 75 on the latest Virtex2Pro	12.5(expected)	Expected success

Table 4.4 Hardware specifications of the software platform

Machine hardware	Sun4u	Pentium III
OS version	5.8	Mandrake Linux 2.4
Processor type	Sparcv9 @ 450 MHz, Dual processors	Pentium III @ 800 MHz
Memory	2048 Mbytes	2048 Mbytes

Table 4.5 Comparison of hardware and software execution times

Graph Size	Cover Size	Software Runtime- Sun SparcV9 @ 450 Mhz (seconds)	Software Runtime- Pentium III @ 800 MHz (seconds)	FPGA Runtime (seconds)	Instance Type	Speedup in comparison to the Sun Sparc machines
256	248	1.959389	0.702033	.016131	Yes	121
256	247	2.154869	0.923886	.023092	Yes	93
256	246	3.624747	1.324847	.034942	Yes	103
256	245	16.612613	6.685848	.187441	Yes	88
256	244	1294 seconds	502	14.758701	Yes	88
256	243	2949	1119	32.134554	No	92
256	242	2183 seconds	886	24.889479	No	90
256	245	4.674909	1.824063	.051410	Yes	91
256	244	3748 seconds	1535	44.217833	No	85
256	243	3845 seconds	1218	34.311693	No	88
256	225	175.631178	72.568051	2.630510	No	66
256	200	34.138157	12.647959	.323884	No	105
256	100	.759341	0.315154	.006982	No	108
256	160	4.540354	1.795218	.042833	No	106

Table 4.5 (Contd.)

256	150	1.479390	0.602138	.014585	No	101
256	25	.706478	0.286341	.011974	Yes	59
256	24	.666915	0.259659	.009888	No	67
256	40	.365398	0.156231	.002860	No	127

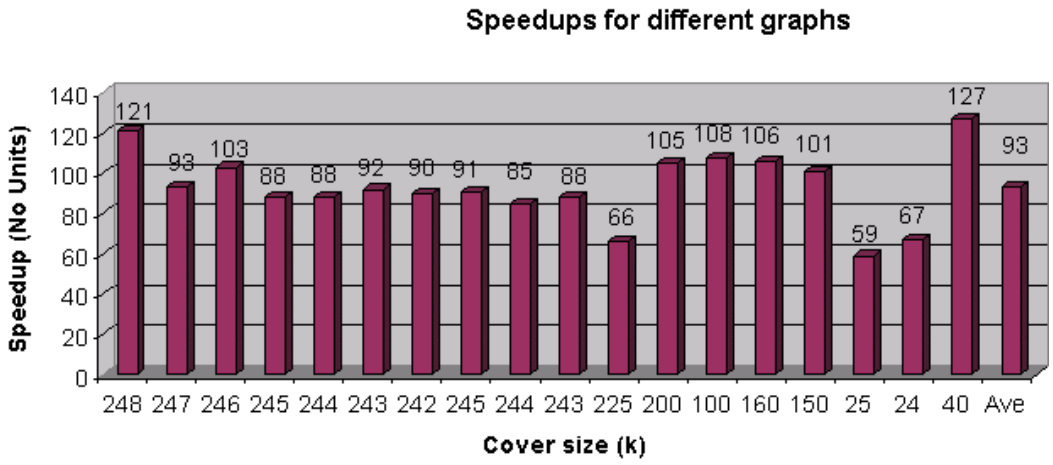

Figure 4.2 Speedup plot

Chapter 5

Future Work

What has been discussed and implemented in this work is just the tip of the iceberg. There is more to work on (as always). The Vertex Cover problem is just a prototype implementation that we have targeted as a part of an ongoing effort to target hard problems that require considerable amount of software computing power. Many CAD problems are NP-complete and hence we have at our disposal an entire suite of problems to tackle.

An immediate requirement for the vertex cover problem is to scale up to larger sized graphs. The maximum sized graph that has been implemented here is just 256, still a relatively small. What would be desirable is to interconnect the reconfigurable nodes with Netsolve. This way, any problem that takes more than a pre-determined amount of time to execute on hardware could be transferred to the reconfigurable platform.

There are several other issues to be dealt with too. The whole notion of developing a high performance reconfigurable network involves issues such as efficient load balancing, scheduling, modeling and analysis of high performance reconfigurable systems. The field of high performance reconfigurable systems is still a vastly unexplored area with ample scope for research. The final objective is to utilize the inherent

computing power of reconfigurable networks by building an array of efficient systems that permit the easy and efficient flow of information between hardware and software.

The work that has been shown here is merely a first step in this direction.

Bibliography

[1] Christian Plessl and Marco Platzner. Custom Computing Machines for the Set Covering Problem. *Proceedings of FPGAs for Custom Computing Machines (FCCM'02).* Napa, CA, USA, April 2002.

[2] Topics in Graph algorithms: structural results and algorithmic techniques, with applications, a dissertation presented for the doctor of philosophy degree, Faisal Nabih Abu-Khzam,University of Tennessee.

[3] ``Design Flow for Automatic Mapping of Graphical Programming Applications to Adaptive Computing Systems," *Workshop on High Performance Embedded Computing*, Boston, Massachusetts, September, 2000, S. Ong, N. Kerkiz, B. Srijanto, C. Tan, M.A. Langston, D. F. Newport and D. W. Bouldin.

[4] ``Automatic Mapping of Multiple Applications to Multiple Adaptive Computing Systems," *IEEE Symposium on Field-Programmable Custom Computing Machines*, Rohnert Park, California, April, 2001, S. Ong, N. Kerkiz, B. Srijanto, C. Tan, Langston M, D. F. Newport and D. W. Bouldin.

[5] ``On Special-Purpose Hardware Clusters for High-Performance Computational Grids," *International Conference on Parallel and Distributed Computing and Systems*, Cambridge, Massachusetts, November, 2002, J. M. Lehrter, F. N. Abu-Khzam, D. W. Bouldin , M. A. Langston and G. D. Peterson .

[6] J. L. Gustaffson, ``Reevaluating Amdahl's Law,", *Communicationas of ACM,* pp 532-533,1988

[7] J. Chen, I. Kanj, and W. Jia. Vertex cover: further observations and further improvements. *Journal of Algorithms,* 41:280–301, 2001.

[8] *Algorithms for VLSI design automation*- Sabih H. Gerez, Kluwer Publishers

[9] www.wikipedia.com

[10] Verification of intellectual property blocks using reconfigurable hardware, a thesis for the Master of Sceince degree, presented by Koay Teng Kuan , University of Tennesssee.

[11] M. R. Fellows and M. A. Langston. Nonconstructive advances in polynomial time complexity. *Information Processing Letters*, 26:157–162, 1987.

[12] M. R. Fellows and M. A. Langston. Nonconstructive tools for proving polynomial-time decidability. *Journal of the ACM*, 35:727–739, 1988.

[13] M. R. Fellows and M. A. Langston. On well-partial-order theory and its application to combinatorial problems of VLSI design. *SIAM Journal on Discrete Mathematics*, 5:117–126, 1992.

[14] M. R. Fellows and M. A. Langston. On search, decision and the efficiency of polynomial-time algorithms. *Journal of Computer and Systems Sciences*, 49:769–779, 1994.

[15] Christian Plessl and Marco Platzner. Instance-Specific Accelerators for Minimum Covering. *Proceedings of the 1st International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'01).* pages 85-91, Las Vegas, USA, June 2001.

[16] Marco Platzner. Reconfigurable Accelerators for Combinatorial Problems. *IEEE Computer*, pages 58-60, April 2000.

[17] Marco Platzner and Giovanni De Micheli. Acceleration of Satisfiability Algorithms by Reconfigurable Hardware. *Proceedings of the 8th international*

Workshop on Field Programmable Logic and Applications (FPL'98), pages 69-78, Tallinn, Estonia, 1998.

[18] Wong Hiu Yung, Yuen Wing Seung, Kin Hong Lee, and Philip Heng Wai Leong. A Runtime Reconfiguration Implementation of the GSAT Algorithm. *In Int'l Workshop on Field Programmable Logic and Applications*, pages 526–531. Springer, 1999.

[19] M. Abramovici and J. T. De Sousa. A SAT Solver Using Reconfigurable Hardware and Virtual Logic. *Journal of Automated Reasoning*, 24(1-2):5–36, 2000.

[20] M. Abramovici and D. Saab. Satisfiability on Reconfigurable Hardware. *In Int'l Workshop on Field-programmable Logic and Applications*, pages 448–456. Springer, 1997.

[21] J. Babb, M. Frank, and A. Agarwal. Solving graph problems with dynamic computation structures. *In SPIE: High-Speed Computing, Digital Signal Processing, and Filterung Using Reconfigurable Logic*, volume 2914, pages 225–236, 1996.

[22] A. Dandalis, A.Mei, and V. K. Prasanna. Domain Specific Mapping for Solving Graph Problems on Reconfigurable Devices. *In Reconfigurable Architecures Workshop*, 1999.

[23] Matlab.Documentation, "MATLAB-The Language of Technical Computing, Using Matlab version 6.0," August 2002 ed: COPYRIGHT 1984 - 2002 by The MathWorks, Inc., 2002.,

[24] *The Designers Guide to VHDL*, Peter J Ashenden, Morgan Kaufmann publishers.

[25] *VHDL Synthesis Primer*, Jeyaram Bhaskar, Morgan Kaufmannn publishers

[26] Y. Hamadi and D. Merceron. Reconfigurable Architectures: A New Vision for Optimization Problems. *In Int'l Conference on Principles and Practice of Constraint Programming*, pages 209–221. Springer, 1997.

[27] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued Minimization for PLA Optimization. *IEEE Transactions on CAD/ICAS, 6(5):727–750, 1987*.

[28] T. Suyama, M. Yokoo, and A. Nagoya. Solving satisfiability problems on FPGAs using experimental unit propagation. *In Int'l Conference on Principles and Practice of Constraint Programming*, volume 1713, pages 434–445. Springer, 1999.

[29] E. E. Swartzlander JR. Parallel counters. *IEEE Transactions on Computers,* C-22(11):1021–1024, November 1973.

[30] P. Zhong, P. Ashar, S. Malik, and M. Martonosi. Using reconfigurable computing techniques to accelerate problems in the CAD domain: a case study with Boolean satisfiability. *In Design Automation Conference,* pages 194–199. IEEE, 1998.

[31] P. Zhong, M. Martonosi, P. Ashar, and S. Malik. Using configurable computing to accelerate Boolean satisfiability. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 18(6):861–868, June 1999.

[32] P. Leong, C. Sham, W.Wong, H. Wong, W. Yuen, M. Leong. A Bitstream Reconfigurable FPGA implementation of the WSAT Algorithm. *IEEE Transactions on Very Large Scale Integration Systems,* VOL.9,No.1,Febraury 2001.

[33] T. Suyama, M. Yokoo, H.Sawada . Solving satisfiability problems using field programmable gate arrays: first results. *In Second International conference on Principles and Practice of constraint programming*,pages 497-509,1996.

[34] T. Suyama, M. Yokoo, H.Sawada and A. Nagoya. Solving satisfiability problems using reconfigurable computing. *IEEE Transactions on Very Large Scale Integration Systems,* Vol. 9,No.1, Febraury 2001.

[35] Eduardo Sanchez, Moshe Sipper, Jacques-Olivier Haenni, Jean-Luc Beuchat, Andre¬ Stauffer, and Andres Perez-Uribe. Static and Dynamic Configurable Systems. *IEEE transactions on computers*, Vol. 48, No. 6, June 1999.

[36] Brad L. Hutchings and Michael J. Wirthlin. Implementation Approaches for Reconfigurable Logic Applications. *In International Workshop on Field-Programmable Logic and Applications (FPL)*, pages 419-428,1995

[37] Rodney G. Downey, Micheal R. Fellows, and Ulrike Stege. Parameterized Complexity: A Framework for Systematically Confronting Computational Intractability. Springer-Verlag,1999

[38] Faisal N. Abu-Khzam, Michael A. Langston and Pushkar Shanbhag. Scalable Parallel Algorithms for Difficult Combinatorial Problems: A Case Study in Optimization, *IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, Marina Del Rey, USA, November 2003.

[39] Peterson, G. D. and Smith, M. C., "Programming High Performance Reconfigurable Computers," *SSGRR 2001*, 2001, Rome, Italy.

[40] Pilchard User Reference (v0.2) . K.H.Tsoi, Chinese University of Hong Kong, July 2003

[41] P. H. W. Leong, M. P. Leong, O. Y. H. Cheung, T. Tung, C. M. Kwok, M. Y. Wong, and K. H. Lee, "Pilchard - A Recongurable Computing Platform with Memory Slot Interface," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

[42] Synopsys Designware Foundation Library Databook Volume 1

[43] Xilinx Coregen – Documentation and product information manuals from www.xilinx.com

Appendix

PILCHARD.VHD

```
library ieee;
use ieee.std_logic_1164.all;

entity pilchard is
port
	(
	PADS_exchecker_reset: in std_logic;
	PADS_dimm_ck: in std_logic;
	PADS_dimm_cke: in std_logic_vector(1 downto 0);
	PADS_dimm_ras: in std_logic;
	PADS_dimm_cas: in std_logic;
	PADS_dimm_we: in std_logic;
	PADS_dimm_s: std_logic_vector(3 downto 0);
	PADS_dimm_a: in std_logic_vector(13 downto 0);
	PADS_dimm_ba: in std_logic_vector(1 downto 0);
	PADS_dimm_rege: in std_logic;
	PADS_dimm_d: inout std_logic_vector(63 downto 0);
	PADS_dimm_cb: inout std_logic_vector(7 downto 0);
	PADS_dimm_dqmb: in std_logic_vector(7 downto 0);
	PADS_dimm_scl: in std_logic;
	PADS_dimm_sda: inout std_logic;
	PADS_dimm_sa: in std_logic_vector(2 downto 0);
	PADS_dimm_wp: in std_logic;
	PADS_io_conn: inout std_logic_vector(27 downto 0)
	);
end pilchard;

architecture syn of pilchard is

component INV
port
	(
	O: out std_logic;
	I: in std_logic
	);
end component;

component BUF
port
	(
	I: in std_logic;
	O: out std_logic
	);
```

```
end component;

component BUFG
port
	(
	I: in std_logic;
	O: out std_logic
	);
end component;

component CLKDLLHF is
port
	(
	CLKIN: in std_logic;
	CLKFB: in std_logic;
	RST: in std_logic;
	CLK0: out std_logic;
	CLK180: out std_logic;
	CLKDV: out std_logic;
	LOCKED: out std_logic
	);
end component;

component FDC is
port
	(
	C: in std_logic;
	CLR: in std_logic;
	D: in std_logic;
	Q: out std_logic
	);
end component;

component IBUF
port
	(
	I: in std_logic;
	O: out std_logic
	);
end component;

component IBUFG
port
```

```
        (
        I: in std_logic;
        O: out std_logic
        );
end component;

component IOB_FDC is
port
        (
        C: in std_logic;
        CLR: in std_logic;
        D: in std_logic;
        Q: out std_logic
        );
end component;

component IOBUF
port
        (
        I: in std_logic;
        O: out std_logic;
        T: in std_logic;
        IO: inout std_logic
        );
end component;

component OBUF
port
        (
        I: in std_logic;
        O: out std_logic
        );
end component;

component STARTUP_VIRTEX
port
        (
        GSR: in std_logic;
        GTS: in std_logic;
        CLK: in std_logic
        );
end component;

component pcore
port
```

```
        (
        clk: in std_logic;
        clkdiv: in std_logic;
        rst: in std_logic;
        read: in std_logic;
        write: in std_logic;
        addr: in std_logic_vector(13 downto 0);
        din: in std_logic_vector(63 downto 0);
        dout: out std_logic_vector(63 downto 0);
        dmask: in std_logic_vector(63 downto 0);
        extin: in std_logic_vector(25 downto 0);
        extout: out std_logic_vector(25 downto 0);
        extctrl: out std_logic_vector(25 downto 0)
        );
end component;

signal clkdllhf_clk0: std_logic;
signal clkdllhf_clkdiv: std_logic;
signal dimm_ck_bufg: std_logic;
signal dimm_s_ibuf: std_logic;
signal dimm_ras_ibuf: std_logic;
signal dimm_cas_ibuf: std_logic;
signal dimm_we_ibuf: std_logic;
signal dimm_s_ibuf_d: std_logic;
signal dimm_ras_ibuf_d: std_logic;
signal dimm_cas_ibuf_d: std_logic;
signal dimm_we_ibuf_d: std_logic;
signal dimm_d_iobuf_i: std_logic_vector(63 downto 0);
signal dimm_d_iobuf_o: std_logic_vector(63 downto 0);
signal dimm_d_iobuf_t: std_logic_vector(63 downto 0);
signal dimm_a_ibuf: std_logic_vector(14 downto 0);
signal dimm_dqmb_ibuf: std_logic_vector(7 downto 0);
signal io_conn_iobuf_i: std_logic_vector(27 downto 0);
signal io_conn_iobuf_o: std_logic_vector(27 downto 0);
signal io_conn_iobuf_t: std_logic_vector(27 downto 0);
signal s,ras,cas,we : std_logic;
signal VDD: std_logic;
signal GND: std_logic;
signal CLK: std_logic;
signal CLKDIV: std_logic;
signal RESET: std_logic;
signal READ: std_logic;
signal WRITE: std_logic;
signal READ_p: std_logic;
signal WRITE_p: std_logic;
```

```
signal READ_n: std_logic;
signal READ_buf: std_logic;
signal WRITE_buf: std_logic;
signal READ_d: std_logic;
signal WRITE_d: std_logic;
signal READ_d_n: std_logic;
signal READ_d_n_buf: std_logic;
signal pcore_addr_raw: std_logic_vector(13 downto 0);
signal pcore_addr: std_logic_vector(13 downto 0);
signal pcore_din: std_logic_vector(63 downto 0);
signal pcore_dout: std_logic_vector(63 downto 0);
signal pcore_dmask: std_logic_vector(63 downto 0);
signal pcore_extin: std_logic_vector(25 downto 0);
signal pcore_extout: std_logic_vector(25 downto 0);
signal pcore_extctrl: std_logic_vector(25 downto 0);
signal pcore_dqmb: std_logic_vector(7 downto 0);

-- CLKDIV frequency control, default is 2
-- uncomment the following lines so as to redefined the clock rate
-- given by clkdiv
--attribute CLKDV_DIVIDE: string;
--attribute CLKDV_DIVIDE of U_clkdllhf: label is "3"; -- 1.5, 2, 2.5, 3, 4, 5, 8, or 16 ----
--(default value is 2)

begin

VDD <= '1';
GND <= '0';

U_ck_bufg: IBUFG port map
        (
        I => PADS_dimm_ck,
        O => dimm_ck_bufg
        );

U_reset_ibuf: IBUF port map
        (
        I => PADS_exchecker_reset,
        O => RESET
        );

U_clkdllhf: CLKDLLHF port map
        (
        CLKIN => dimm_ck_bufg,
        CLKFB => CLK,
```

```
        RST => RESET,
        CLK0 => clkdllhf_clk0,
        CLK180 => open,
        CLKDV => clkdllhf_clkdiv,
        LOCKED => open
        );

U_clkdllhf_clk0_bufg: BUFG port map
        (
        I => clkdllhf_clk0,
        O => CLK
        );

U_clkdllhf_clkdiv_bufg: BUFG port map
        (
        I => clkdllhf_clkdiv,
        O => CLKDIV
        );

U_startup: STARTUP_VIRTEX port map
        (
        GSR => RESET,
        GTS => GND,
        CLK => CLK
        );

U_dimm_s_ibuf: IBUF port map
        (
        I => PADS_dimm_s(0),
        O => dimm_s_ibuf
        );

U_dimm_ras_ibuf: IBUF port map
        (
        I => PADS_dimm_ras,
        O => dimm_ras_ibuf
        );

U_dimm_cas_ibuf: IBUF port map
        (
        I => PADS_dimm_cas,
        O => dimm_cas_ibuf
        );

U_dimm_we_ibuf: IBUF port map
```

```
        (
        I => PADS_dimm_we,
        O => dimm_we_ibuf
        );

G_dimm_d: for i in integer range 0 to 63 generate

        U_dimm_d_iobuf: IOBUF port map
                (
                I => dimm_d_iobuf_i(i),
                O => dimm_d_iobuf_o(i),
                T => dimm_d_iobuf_t(i),
                IO => PADS_dimm_d(i)
                );

        U_dimm_d_iobuf_o: IOB_FDC port map
                (
                C => CLK,
                CLR => RESET,
                D => dimm_d_iobuf_o(i),
                Q => pcore_din(i)
                );

        U_dimm_d_iobuf_i: IOB_FDC port map
                (
                C => CLK,
                CLR => RESET,
                D => pcore_dout(i),
                Q => dimm_d_iobuf_i(i)
                );

        U_dimm_d_iobuf_t: IOB_FDC port map
                (
                C => CLK,
                CLR => RESET,
                D => READ_d_n_buf,
                Q => dimm_d_iobuf_t(i)
                );
end generate;

G_dimm_a: for i in integer range 0 to 13 generate

U_dimm_a_ibuf: IBUF port map
```

```
        (
        I => PADS_dimm_a(i),
        O => dimm_a_ibuf(i)
        );

U_dimm_a_ibuf_o: IOB_FDC port map
        (
        C => CLK,
        CLR => RESET,
        D => dimm_a_ibuf(i),
        Q => pcore_addr_raw(i)
        );
end generate;

pcore_addr(3 downto 0) <= pcore_addr_raw(3 downto 0);

addr_correct: for i in integer range 4 to 7 generate
ADDR_INV: INV port map (
O => pcore_addr(i),
I => pcore_addr_raw(i) );
end generate;
pcore_addr(13 downto 8) <= pcore_addr_raw(13 downto 8);

G_dimm_dqmb: for i in integer range 0 to 7 generate

U_dimm_dqmb_ibuf: IBUF port map (
I => PADS_dimm_dqmb(i),
O => dimm_dqmb_ibuf(i) );

U_dimm_dqmb_ibuf_o: IOB_FDC port map (
C => CLK,
CLR => RESET,
D => dimm_dqmb_ibuf(i),
Q => pcore_dqmb(i) );

end generate;

pcore_dmask(7 downto 0) <= (others => (not pcore_dqmb(0)));
pcore_dmask(15 downto 8) <= (others => (not pcore_dqmb(1)));
pcore_dmask(23 downto 16) <= (others => (not pcore_dqmb(2)));
pcore_dmask(31 downto 24) <= (others => (not pcore_dqmb(3)));
pcore_dmask(39 downto 32) <= (others => (not pcore_dqmb(4)));
pcore_dmask(47 downto 40) <= (others => (not pcore_dqmb(5)));
pcore_dmask(55 downto 48) <= (others => (not pcore_dqmb(6)));
pcore_dmask(63 downto 56) <= (others => (not pcore_dqmb(7)));
```

```
G_io_conn: for i in integer range 2 to 27 generate

U_io_conn_iobuf: IOBUF port map (
I => io_conn_iobuf_i(i),
O => io_conn_iobuf_o(i),
T => io_conn_iobuf_t(i),
IO => PADS_io_conn(i) );

U_io_conn_iobuf_o: IOB_FDC port map (
C => CLK,
CLR => RESET,
D => io_conn_iobuf_o(i),
Q => pcore_extin(i - 2) );

U_io_conn_iobuf_i: IOB_FDC port map (
C => CLK,
CLR => RESET,
D => pcore_extout(i - 2),
Q => io_conn_iobuf_i(i) );

U_io_conn_iobuf_t: IOB_FDC port map (
C => CLK,
CLR => RESET,
D => pcore_extctrl(i - 2),
Q => io_conn_iobuf_t(i) );

end generate;

U_io_conn_0_iobuf: IOBUF port map (
I => dimm_ck_bufg,
O => open,
T => GND,
IO => PADS_io_conn(0) );

U_io_conn_1_iobuf: IOBUF port map (
I => GND,
O => open,
T => VDD,
IO => PADS_io_conn(1) );

READ_p <=
(not dimm_s_ibuf) and
(dimm_ras_ibuf) and
(not dimm_cas_ibuf) and
```

```
(dimm_we_ibuf);

U_read: FDC port map (
C => CLK,
CLR => RESET,
D => READ_p,
Q => READ );

U_buf_read: BUF port map (
I => READ,
O => READ_buf );

U_read_d: FDC port map (
C => CLK,
CLR => RESET,
D => READ,
Q => READ_d );

WRITE_p <=
(not dimm_s_ibuf) and
(dimm_ras_ibuf) and
(not dimm_cas_ibuf) and
(not dimm_we_ibuf);

U_write: FDC port map (
C => CLK,
CLR => RESET,
D => WRITE_p,
Q => WRITE );

U_buf_write: BUF port map (
I => WRITE,
O => WRITE_buf );

U_write_d: FDC port map (
C => CLK,
CLR => RESET,
D => WRITE,
Q => WRITE_d );

READ_n <= not READ;

U_read_d_n: FDC port map (
C => CLK,
CLR => RESET,
```

```
D => READ_n,
Q => READ_d_n );

U_buf_read_d_n: BUF port map (
I => READ_d_n,
O => READ_d_n_buf );

-- User logic should be placed inside pcore
U_pcore: pcore port map (
clk => CLK,
clkdiv => CLKDIV,
rst => RESET,
read => READ,
write => WRITE,
addr => pcore_addr,
din => pcore_din,
dout => pcore_dout,
dmask => pcore_dmask,
extin => pcore_extin,
extout => pcore_extout,
extctrl => pcore_extctrl );

end syn;
```

PCORE.VHD

```
-- pcore interface
-- author: Mahesh Dorai

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity pcore is

port

        (
        clk: in std_logic;
        clkdiv: in std_logic;
        rst: in std_logic;
        read: in std_logic;
        write: in std_logic;
        addr: in std_logic_vector(13 downto 0);
        din: in std_logic_vector(63 downto 0);
        dout: out std_logic_vector(63 downto 0);
        dmask: in std_logic_vector(63 downto 0);
        extin: in std_logic_vector(25 downto 0);
        extout: out std_logic_vector(25 downto 0);
        extctrl: out std_logic_vector(25 downto 0)
        );

end pcore;


architecture syn of pcore is

COMPONENT dpram2100_32
port
        (
        addra: IN std_logic_VECTOR(11 downto 0);
        addrb: IN std_logic_VECTOR(11 downto 0);
        clka: IN std_logic;
        clkb: IN std_logic;
        dina: IN std_logic_VECTOR(31 downto 0);
        dinb: IN std_logic_VECTOR(31 downto 0);
        douta: OUT std_logic_VECTOR(31 downto 0);
        doutb: OUT std_logic_VECTOR(31 downto 0);
```

```
        wea: IN std_logic;
        web: IN std_logic
        );
end COMPONENT;
COMPONENT dpram512_256
port

        (
        addra: IN std_logic_VECTOR(8 downto 0);
        addrb: IN std_logic_VECTOR(8 downto 0);
        clka: IN std_logic;
        clkb: IN std_logic;
        dina: IN std_logic_VECTOR(255 downto 0);
        dinb: IN std_logic_VECTOR(255 downto 0);
        douta: OUT std_logic_VECTOR(255 downto 0);
        doutb: OUT std_logic_VECTOR(255 downto 0);
        wea: IN std_logic;
        web: IN std_logic
        );
end COMPONENT;

component dpram16_64
port
        (
        addra: IN std_logic_VECTOR(4 downto 0);
        addrb: IN std_logic_VECTOR(4 downto 0);
        clka: IN std_logic;
        clkb: IN std_logic;
        dina: IN std_logic_VECTOR(63 downto 0);
        dinb: IN std_logic_VECTOR(63 downto 0);
        douta: OUT std_logic_VECTOR(63 downto 0);
        doutb: OUT std_logic_VECTOR(63 downto 0);
        wea: IN std_logic;
        web: IN std_logic
        );
END component;

component ram_load
port

        (
        clk : in std_logic;
        rst : in std_logic;
        row_cont: in std_logic_vector(31 downto 0);
        start_ini : in std_logic;
```

```
        addr_a : out std_logic_vector(11 downto 0);
        addr_b : out std_logic_vector(8 downto 0);
        concat_out: out std_logic_vector(255 downto 0);
        finish_load : out std_logic;
        we_2 : out std_logic
        );
end component;

component ram_cntl
port
        (
        clk : in std_logic;
        rst : in std_logic;
        adj_list: in std_logic_vector(255 downto 0);
        start_gen : in std_logic;
        addr : out std_logic_vector(8 downto 0);
        mask : out std_logic_vector(255 downto 0);
        finish : out std_logic
        );
end component;


--************ SIGNAL DECLARATIONS START HERE**********************
signal clkb : std_logic;
signal doutb_1 : std_logic_vector(31 downto 0);
signal start_debug: std_logic;
signal addr_1 : std_logic_vector(11 downto 0);
signal addr_2 : std_logic_vector(8 downto 0);
signal fin_out : std_logic_vector(255 downto 0);
signal finish : std_logic;
signal finish_load : std_logic;
signal tmp_finish_load : std_logic;
signal web_2 : std_logic;
signal bram_dout: std_logic_vector(31 downto 0);
signal dinb_2 : std_logic_vector(31 downto 0);
signal web_1 : std_logic;
signal douta_2 : std_logic_vector(255 downto 0);
signal addrb : std_logic_vector(8 downto 0);
signal dinb : std_logic_vector(255 downto 0);
signal tmp_doutb: std_logic_vector(255 downto 0);
signal web : std_logic;
signal start : std_logic; -- From pcore to the Processing Core
signal out_dina : std_logic_vector(63 downto 0);
signal out_douta: std_logic_vector(63 downto 0);
signal out_wea : std_logic;
```

```
signal out_addrb: std_logic_vector(4 downto 0);
signal out_dinb : std_logic_vector(63 downto 0);
signal out_doutb: std_logic_vector(63 downto 0);
signal out_web : std_logic;
signal state_write : std_logic_vector(2 downto 0);
signal mask : std_logic_vector(255 downto 0);
signal tmp_start_debug: std_logic;

--************ SIGNAL DECLARATIONS END HERE *********************

--******** PORT MAPPING OF ALL COMPONENTS START HERE **********
begin

dpram2100_32_1 : dpram2100_32
port map
        (
addra => din(11 downto 0),
clka => clk,
dina => din(63 downto 32),
douta => bram_dout,
wea => write,
addrb => addr_1,
clkb => clkb,
dinb => dinb_2,
doutb => doutb_1,
web => web_1
);


dpram512_256_1 : dpram512_256
port map
        (
addra => addr_2,
clka => clkb,
dina => fin_out,
douta => douta_2,
wea => web_2,
addrb => addrb,
clkb => clkb,
dinb => dinb,
doutb => tmp_doutb,
web => web
);
```

```
dpram16_64_1 : dpram16_64
port map
        (
addra => addr(4 downto 0),
clka => clk,
dina => out_dina,
douta => out_douta,
wea => out_wea,
addrb => out_addrb,
clkb => clkb,
dinb => out_dinb,
doutb => out_doutb,
web => out_web
);

ram_load1 : ram_load
port map
        (
clk => clkb,
rst => rst,
row_cont => doutb_1,
start_ini => start,
addr_a => addr_1,
addr_b => addr_2,
concat_out => fin_out,
finish_load => finish_load,
we_2 => web_2
);

ram_cntl1 : ram_cntl
port map
        (
        clk => clkb,
        rst => rst,
        adj_list => tmp_doutb,
        start_gen => tmp_finish_load,
        addr => addrb,
        mask => mask,
        finish => finish
        );

--****** PORT MAPPING OF ALL COMPONENTS ENDS HERE **********
process(clk,rst)
```

```
variable ini_counter : integer range 0 to 7;
begin

if (rst = '1') then
        start <= '0';
        web <= '0';
        out_wea <= '0';
        ini_counter := 0;

elsif (clk'event and clk ='1') then
        if write ='1' and addr(7 downto 0)="11111111" and start='0' then
                start <='1';
                ini_counter :=0;

        elsif start='1' and ini_counter/=7 then
                ini_counter:= ini_counter+1;
        else
                start <='0';
                ini_counter :=0;
        end if;
end if;
end process;


process(clkb,rst)
begin

if (rst = '1') then

        state_write <= (others => '0');
        out_dinb <= (others => '0');
        out_web <= '0';
        out_addrb <= "00001";

elsif (clkb'event and clkb ='1') then

        if (finish = '1' and state_write = "000") then

                out_addrb <= "00001";
                out_web <= '1';
                out_dinb <= mask(63 downto 0);
                state_write <= "001";

        elsif (state_write = "001") then
```

```
		out_addrb <= "00010";
		out_web <= '1';
		out_dinb <= mask(127 downto 64);
		state_write <= "010";

	elsif (state_write = "010") then

		out_addrb <= "00011";
		out_web <= '1';
		out_dinb <= mask(191 downto 128);
		state_write <= "011";

	elsif (state_write = "011") then

		out_addrb <= "00100";
		out_web <= '1';
		out_dinb <= mask(255 downto 192);
		state_write <= "100";

	elsif (state_write = "100") then

		out_addrb <= "00101";
		out_web <= '1';
		out_dinb <= mask(63 downto 0);
		state_write <= "101";

	elsif (state_write = "101") then
		out_web <= '0';
		if addr(7 downto 0)="11111110" then
			state_write <= "110";
		else
			state_write <= state_write;
		end if;

	elsif (state_write = "110") then
		out_addrb <= "00001";
		out_web <= '1';
		out_dinb <= (others => '0');
		state_write <= "111";

	elsif (state_write = "111") then
		out_addrb <= (others => '1');
		out_web <= '0';
		state_write <= "000";
```

```
        else
                out_web <= '0';
        end if;
end if;

end process;

dout <= out_douta ;
tmp_finish_load <= '1' when (finish_load = '1') else '0';
--define the core clock
clkb <= clkdiv;
dinb_2 <= (others => '0');
dinb <= (others => '0');
out_dina <= (others => '0');
web_1 <= '0';


end syn;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity ram_load is
port
        (
clk : in std_logic;
rst : in std_logic;
row_cont: in std_logic_vector(31 downto 0);
start_ini : in std_logic;
addr_a : out std_logic_vector(11 downto 0);
addr_b : out std_logic_vector(8 downto 0);
concat_out: out std_logic_vector(255 downto 0);
finish_load : out std_logic;
we_2 : out std_logic
);
end ram_load;

architecture rtl_a of ram_load is

signal state : std_logic_vector(4 downto 0);
signal tmp_dina : std_logic_vector(255 downto 0);
signal tmp_finish: std_logic;
signal tmp_we_2: std_logic;
signal addr_count: integer range 0 to 258;
signal idx_a : std_logic_vector(11 downto 0);
signal idx_b : std_logic_vector(8 downto 0);

begin

process(clk,rst)
variable load_counter : integer range 0 to 7;
for several clock cycles
begin

if (rst = '1') then

        state <= (others => '0');
        idx_a <= (others => '0');
        idx_b <= (others => '0');
        tmp_dina <= (others => '0');
```

```
            addr_count <= 0;
            tmp_finish <= '0';
            tmp_we_2 <= '0';
            load_counter := 0;

elsif (clk = '1' and clk' event) then

if (start_ini = '1' and state = "00000") then
            idx_a <= (others => '0');
            idx_b <= (others => '0');
            tmp_we_2 <= '1';
            tmp_finish <= '0';
            state <= "00001";
            load_counter := 0;

elsif (state = "00001") then
            tmp_we_2 <= '1';
            state <= "00010";

elsif (state = "00010") then
            tmp_dina(31 downto 0) <= row_cont;
            idx_a <= idx_a + "000000000001";
            state <= "00011";

elsif (state = "00011") then
            tmp_we_2 <= '1';
            state <= "00100";

elsif (state = "00100") then

tmp_dina(63 downto 32) <= row_cont;
idx_a <= idx_a + "000000000001";
state <= "00101";

elsif (state = "00101") then

tmp_we_2 <= '1';
state <= "00110";

elsif (state = "00110") then
            tmp_dina(95 downto 64) <= row_cont;
            idx_a <= idx_a + "000000000001";
            state <= "00111";

elsif (state = "00111") then
```

```
            tmp_we_2 <= '1';
            state <= "01000";

    elsif (state = "01000") then
            tmp_dina(127 downto 96) <= row_cont;
            idx_a <= idx_a + "000000000001";
            state <= "01001";

    elsif (state = "01001") then
            tmp_we_2 <= '1';
            state <= "01010";

    elsif (state = "01010") then
            tmp_dina(159 downto 128) <= row_cont;
            idx_a <= idx_a + "000000000001";
            state <= "01011";

    elsif (state = "01011") then
            tmp_we_2 <= '1';
            state <= "01100";

    elsif (state = "01100") then
            tmp_dina(191 downto 160) <= row_cont;
            idx_a <= idx_a + "000000000001";
            state <= "01101";

    elsif (state = "01101") then
            tmp_we_2 <= '1';
            state <= "01110";

    elsif (state = "01110") then
            tmp_dina(223 downto 192) <= row_cont;
            idx_a <= idx_a + "000000000001";
            state <= "01111";

    elsif (state = "01111") then
            tmp_we_2 <= '1';
            state <= "10000";

    elsif (state = "10000") then
            tmp_dina(255 downto 224) <= row_cont;
            addr_count <= addr_count + 1;
            state <= "10001";

    elsif (state = "10001") then
```

```
        if (addr_count = 257) then
                state <= "10010";
                idx_a <= "000000000010";
                tmp_finish <= '1';
                tmp_we_2 <= '0';
        else
                state <= "00001";
                idx_a <= idx_a + "000000000001";
                idx_b <= idx_b + "00000001";
        end if;

elsif (state = "10010") then
        if tmp_finish = '1' and load_counter/=7 then
                load_counter:= load_counter+1;
                state <= state;
        else
                tmp_finish <='0';
                load_counter :=0;
                state <= (others => '0');
        end if;

else
        tmp_finish <= '0';
end if;
end if;

end process;

addr_a <= idx_a;
addr_b <= idx_b;
concat_out <= tmp_dina;
finish_load <= tmp_finish;
we_2 <= tmp_we_2;
end rtl_a;
```

```
library ieee;
use ieee.std_logic_1164.all;
package memory is
        type INT_ARR is array(0 to 255) of integer range 0 to 255;
end memory;


library ieee;
use ieee.std_logic_1164.all;
use work.memory.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity ram_cntl is
port (
clk : in std_logic;
rst : in std_logic;
adj_list: in std_logic_vector(255 downto 0);
start_gen : in std_logic;
addr : out std_logic_vector(8 downto 0);
mask : out std_logic_vector(255 downto 0);
finish : out std_logic
--we : out std_logic
);
end ram_cntl;

architecture rtl of ram_cntl is

signal k : std_logic_vector(255 downto 0);
signal k_int : integer range 0 to 255;
signal graph_size : integer range 1 to 255;
signal idx : std_logic_vector(8 downto 0);
signal state : std_logic_vector(4 downto 0);
signal state_edge : std_logic_vector(2 downto 0);
signal state_select : std_logic_vector(3 downto 0);
signal cover : std_logic_vector(255 downto 0);
signal new_vect : std_logic_vector(255 downto 0);
signal i,j,l,m : integer range 0 to 255;
signal stack_addra : std_logic_VECTOR(7 downto 0);
signal stack_addrb : std_logic_VECTOR(7 downto 0);
signal tmp_dinb : std_logic_VECTOR(255 downto 0);
signal stack_douta : std_logic_VECTOR(255 downto 0);
signal tmp_doutb : std_logic_VECTOR(255 downto 0);
signal stack_wea : std_logic;
signal tmp_web : std_logic;
```

```
signal k_new : integer range 0 to 255;
signal k_edit : integer range 0 to 255;
signal status : integer range 0 to 255;
signal tmp_status : integer range 0 to 255;
signal edge_addr_count : integer range 0 to 255;
signal select_addr_count: integer range 0 to 255;
signal base : std_logic_vector(8 downto 0);
signal tmp_selected : integer range 0 to 255;
signal order_vec : INT_ARR;
signal tmp_finish : std_logic;
signal ones_ct_1 : std_logic_vector(4 downto 0);
signal ones_ct_2 : std_logic_vector(4 downto 0);
signal ones_ct_3 : std_logic_vector(4 downto 0);
signal ones_ct_4 : std_logic_vector(4 downto 0);
signal ones_ct_5 : std_logic_vector(4 downto 0);
signal ones_ct_6 : std_logic_vector(4 downto 0);
signal ones_ct_7 : std_logic_vector(4 downto 0);
signal ones_ct_8 : std_logic_vector(4 downto 0);
signal ones_ct_9 : std_logic_vector(4 downto 0);
signal ones_ct_10 : std_logic_vector(4 downto 0);
signal ones_ct_11 : std_logic_vector(4 downto 0);
signal ones_ct_12 : std_logic_vector(4 downto 0);
signal ones_ct_13 : std_logic_vector(4 downto 0);
signal ones_ct_14 : std_logic_vector(4 downto 0);
signal ones_ct_15 : std_logic_vector(4 downto 0);
signal ones_ct_16 : std_logic_vector(4 downto 0);
signal tmp_ones_ct_1 : std_logic_vector(5 downto 0);
signal tmp_ones_ct_2 : std_logic_vector(5 downto 0);
signal tmp_ones_ct_3 : std_logic_vector(5 downto 0);
signal tmp_ones_ct_4 : std_logic_vector(5 downto 0);
signal tmp_ones_ct_5 : std_logic_vector(5 downto 0);
signal tmp_ones_ct_6 : std_logic_vector(5 downto 0);
signal tmp_ones_ct_7 : std_logic_vector(5 downto 0);
signal tmp_ones_ct_8 : std_logic_vector(5 downto 0);
signal tmp_ones_ct_9 : std_logic_vector(6 downto 0);
signal tmp_ones_ct_10 : std_logic_vector(6 downto 0);
signal tmp_ones_ct_11 : std_logic_vector(6 downto 0);
signal tmp_ones_ct_12 : std_logic_vector(6 downto 0);
signal tmp_ones_ct_13 : std_logic_vector(7 downto 0);
signal tmp_ones_ct_14 : std_logic_vector(7 downto 0);
signal tmp_ones_ct_15 : std_logic_vector(8 downto 0);
signal cover_status : std_logic;
signal mulx_cover : std_logic_vector(255 downto 0);
signal mix : std_logic_vector(1 downto 0);
signal mix_vector : std_logic_vector(255 downto 0);
```

```
signal tmp_cover : std_logic_vector(255 downto 0);
signal stack_ind : std_logic_vector(255 downto 0);
signal scan_left : std_logic_vector(255 downto 0);
signal pre_tmp_status : integer range 0 to 255;
signal tmp_stack_addra : std_logic_vector(7 downto 0);

component adder_sum1
port (
bit_vector_1 : in std_logic_vector(15 downto 0);
god_sum : out std_logic_vector(4 downto 0)
);
end component;

component stage_mix
port (
mix_st : in std_logic_vector(1 downto 0);
mix_status : in std_logic;
mix_adj_list : in std_logic_vector(255 downto 0);
mix_cover : in std_logic_vector(255 downto 0);
mix_vector : out std_logic_vector(255 downto 0)
);
end component;

component stack
port (
addra: IN std_logic_VECTOR(7 downto 0);
addrb: IN std_logic_VECTOR(7 downto 0);
clka: IN std_logic;
clkb: IN std_logic;
dina: IN std_logic_VECTOR(255 downto 0);
dinb: IN std_logic_VECTOR(255 downto 0);
douta: OUT std_logic_VECTOR(255 downto 0);
doutb: OUT std_logic_VECTOR(255 downto 0);
wea: IN std_logic;
web: IN std_logic);
end component;

begin
process(clk,rst)
variable curr_state : std_logic_vector(4 downto 0);
variable ram_counter : integer range 0 to 31; --counter to key start high for several clock -
--cycles
begin

if (rst = '1') then
```

```
state <= "11010";
curr_state := (others => '0');
state_edge <= (others => '0');
state_select <= (others => '0');
idx <= (others => '1');
            cover <= (others => '0');
new_vect <= (others => '0');
order_vec <= (others => 0);
k_new <= 0;
k_edit <= 0;
graph_size <= 0;
status <= 0;
tmp_status <= 0;
edge_addr_count <= 0;
tmp_finish <= '0';
select_addr_count <= 0;
edge_addr_count <= 0;
base <= (others => '0');
tmp_selected <= 0;
i <= 0;
j <= 0;
k <= (others => '0');
cover_status <= '0';
mulx_cover <= (others => '0');
mix <= (others => '0');
tmp_cover <= (others => '0');
stack_ind <= (others => '1');
scan_left <= (others => '1');
k_int <= 0;
l <= 0;
pre_tmp_status <= 0;
stack_addra <= (others => '0');
stack_addrb <= (others => '1');
tmp_dinb <= (others => '0');
stack_wea <= '0';
tmp_web <= '0';
tmp_stack_addra <= (others => '0');
tmp_ones_ct_1 <= (others => '0');
tmp_ones_ct_2 <= (others => '0');
tmp_ones_ct_3 <= (others => '0');
tmp_ones_ct_4 <= (others => '0');
tmp_ones_ct_5 <= (others => '0');
tmp_ones_ct_6 <= (others => '0');
tmp_ones_ct_7 <= (others => '0');
```

```
tmp_ones_ct_8 <= (others => '0');
tmp_ones_ct_9 <= (others => '0');
tmp_ones_ct_10 <= (others => '0');
tmp_ones_ct_11 <= (others => '0');
tmp_ones_ct_12 <= (others => '0');
tmp_ones_ct_13 <= (others => '0');
tmp_ones_ct_14 <= (others => '0');
tmp_ones_ct_15 <= (others => '0');
m <= 1;
ram_counter := 0;

elsif (clk = '1' and clk' event) then

if (state = "11010") then

state <= (others => '0');
curr_state := (others => '0');
state_edge <= (others => '0');
state_select <= (others => '0');
idx <= (others => '1');
cover <= (others => '0');
new_vect <= (others => '0');
order_vec <= (others => 0);
k_new <= 0;
k_edit <= 0;
graph_size <= 0;
status <= 0;
tmp_status <= 0;
edge_addr_count <= 0;
tmp_finish <= '0';
select_addr_count <= 0;
edge_addr_count <= 0;
base <= (others => '0');
tmp_selected <= 0;
--hit <= '0';
i <= 0;
j <= 0;
k <= (others => '0');
cover_status <= '0';
mulx_cover <= (others => '0');
mix <= (others => '0');
tmp_cover <= (others => '0');
stack_ind <= (others => '1');
scan_left <= (others => '1');
k_int <= 0;
```

```
l <= 0;
pre_tmp_status <= 0;
stack_addra <= (others => '0');
stack_addrb <= (others => '1');
tmp_dinb <= (others => '0');
stack_wea <= '0';
tmp_web <= '0';
tmp_stack_addra <= (others => '0');
tmp_ones_ct_1 <= (others => '0');
tmp_ones_ct_2 <= (others => '0');
tmp_ones_ct_3 <= (others => '0');
tmp_ones_ct_4 <= (others => '0');
tmp_ones_ct_5 <= (others => '0');
tmp_ones_ct_6 <= (others => '0');
tmp_ones_ct_7 <= (others => '0');
tmp_ones_ct_8 <= (others => '0');
tmp_ones_ct_9 <= (others => '0');
tmp_ones_ct_10 <= (others => '0');
tmp_ones_ct_11 <= (others => '0');
tmp_ones_ct_12 <= (others => '0');
tmp_ones_ct_13 <= (others => '0');
tmp_ones_ct_14 <= (others => '0');
tmp_ones_ct_15 <= (others => '0');
m <= 1;
ram_counter := 0;

        elsif (start_gen = '1' and state = "00000") then
        idx <= (others => '0');
        state <= "00001";

elsif (state = "00001") then
        k <= adj_list;
        state <= "00010";

elsif (state = "00010") then
        k <= adj_list;
        k_int <= conv_integer(adj_list(7 downto 0)); --This almost gave me a scare
        k_edit <= conv_integer(adj_list(7 downto 0));
        graph_size <= conv_integer(adj_list(15 downto 8));
        state <= "00011";

elsif (state = "00011") then
        if(i = k_edit) then
                state <= "00111";
                select_addr_count <= 0;
```

```
                mix <= "00";
                i <= 1;
                pre_tmp_status <= tmp_status;
                stack_wea <= '0';
        else
                i <= i + 1;
                mix <= "01";
                state <= "11011";
                state_select <= "0001";
                curr_state := state;
                --state <= "11000";
                mulx_cover <= cover;
                idx <= "000000001";
                base <= (others => '0');
                select_addr_count <= 0;
                tmp_selected <= 0;
                cover_status <= cover(select_addr_count);
                stack_addra <=
                conv_std_logic_vector(status,8);
                stack_wea <= '1';
        end if;

elsif (state = "00100") then
        cover(tmp_selected) <= '1';
select_addr_count <= 0;
        order_vec(status) <= tmp_selected;
        stack_ind(status) <= '0';
        status <= status + 1;
        tmp_status <= status;
        pre_tmp_status <= status;
        state <= "00101";

elsif (state = "00101") then
        mix <= "10";
        state_edge <= "001";
        curr_state := state;
        state <= "11011";
        mulx_cover <= cover;
        idx <= "000000001";
        edge_addr_count <= 0;
        cover_status <= cover(edge_addr_count);

elsif (state = "00110") then
        state <= "00011";
```

```
elsif (state = "00111") then

        if (stack_ind(k_int-i) = '0') then

                tmp_status <= k_int-i;
                stack_addra <=
                conv_std_logic_vector((k_int-i),8);
                pre_tmp_status <= k_int-i;
                state <= "01000";
                i <= 1;
        else

                i <= i + 1;
                state <= "00111";
        end if;

elsif (state = "01000") then
        tmp_stack_addra <= stack_addra;
        l <= k_int - tmp_status;
        if (tmp_status < pre_tmp_status) then

                pre_tmp_status <= tmp_status;
                stack_ind <= (others => '0');
        else
                pre_tmp_status <= pre_tmp_status;
        end if;
        k_new <= k_int - tmp_status;
        k_edit <= k_int - tmp_status;
        state <= "01001";
        idx <=conv_std_logic_vector(order_vec(tmp_status),9) + 1;

elsif (state = "01001") then
        tmp_cover <= stack_douta;
        state <= "01010";

elsif (state = "01010") then

        if (m = l) then
                state <= "01011";
                stack_ind(tmp_status+m) <= '0';
                m <= 1;
                l <= 0;
                tmp_cover(order_vec(tmp_status)) <='0';
                order_vec(tmp_status) <= 0;
        else
```

```
                stack_ind(tmp_status+m) <= '0';
                m <= m + 1;
        end if;

elsif (state = "01011") then
        stack_ind(k_int) <= '1';
         mulx_cover <= tmp_cover;
        mix <= "11";
        state <= "01100";

elsif (state = "01100") then
        cover <= tmp_cover or mix_vector;
tmp_ones_ct_1 <= ('0' & ones_ct_1) + ('0' &ones_ct_2);
tmp_ones_ct_2 <= ('0' & ones_ct_3) + ('0' &ones_ct_4);
tmp_ones_ct_3 <= ('0' & ones_ct_5) + ('0' &ones_ct_6);
tmp_ones_ct_4 <= ('0' & ones_ct_7) + ('0' &ones_ct_8);
tmp_ones_ct_5 <= ('0' & ones_ct_9) + ('0' &ones_ct_10);
tmp_ones_ct_6 <= ('0' & ones_ct_11) + ('0' &ones_ct_12);
tmp_ones_ct_7 <= ('0' & ones_ct_13) + ('0' &ones_ct_14);
tmp_ones_ct_8 <= ('0' & ones_ct_15) + ('0' &ones_ct_16);
stack_wea <= '1';
state <= "01101";
elsif (state = "01101") then

                        tmp_ones_ct_9 <= ('0' & tmp_ones_ct_1) + ('0' &tmp_ones_ct_2);
                        tmp_ones_ct_10  <=  ('0'  &  tmp_ones_ct_3)  +  ('0'
                        &tmp_ones_ct_4);
                        tmp_ones_ct_11  <=  ('0'  &  tmp_ones_ct_5)  +  ('0'
                        &tmp_ones_ct_6);
                        tmp_ones_ct_12  <=  ('0'  &  tmp_ones_ct_7)  +  ('0'
                        &tmp_ones_ct_8);
                        state <= "01110";

elsif (state = "01110") then
        tmp_ones_ct_13 <= ('0' & tmp_ones_ct_9) + ('0' &tmp_ones_ct_10);
        tmp_ones_ct_14 <= ('0' & tmp_ones_ct_11) + ('0' &tmp_ones_ct_12);
        state <= "11000";

elsif (state = "11000") then

        tmp_ones_ct_15  <=  ('0'  &  tmp_ones_ct_13)  +  ('0'  &tmp_ones_ct_14);
        state <= "01111";
```

```
    elsif (state = "01111") then


        if ((l = tmp_ones_ct_15) or (l =k_int-tmp_status)) then
            state <= "10000";
            stack_wea <= '0';
            l <= 0;
        else
            stack_ind(tmp_status+l) <= '1';
            stack_addra <= tmp_stack_addra +
            conv_std_logic_vector(l,7);
            l <= l + 1;
            state <= state;
        end if;

elsif (state = "10000") then

        if (tmp_ones_ct_15 > k_new) then
            state <= "10001";
        elsif (tmp_ones_ct_15 = k_new) then
            state <= "11011";
            state_edge <= "001";
            curr_state := state;
            mix <= "10";
            mulx_cover <= cover;
            idx <= "000000001";
            edge_addr_count <= 0;
            cover_status <= cover(edge_addr_count);
        elsif (tmp_ones_ct_15 < k_new) then
            k_edit <= k_edit -conv_integer(tmp_ones_ct_15);
            state <= "10010";
            j <= 0;
            status <=tmp_status+conv_integer(tmp_ones_ct_15);
        end if;

elsif (state = "10001") then

        if (stack_ind =
        "11111111111111111111111111111111111111111111111111111111111111111
        11111111111111111111111111111111111111111111111111111111111111111
        11111111111111111111111111111111111111111111111111111111111111111
        11111111111111111111111111111111111111111111111111111111111111111") then

            tmp_finish <= '1';
```

```
                state <= "11001";
                mulx_cover <= (others => '1');
                else
                state <= "00111";
                end if;

elsif (state = "10010") then

                        mix                             <=                      "10";
                        state_edge                          <=                  "001";
                        state                           <=                  "11011";
                        curr_state                          :=                  state;
                        mulx_cover                          <=                  cover;
                        idx                         <=                  "000000001";
                        edge_addr_count                             <=          0;
                        cover_status <= cover(edge_addr_count);

elsif (state = "10011") then
        if(j = k_edit) then
                state <= "10111";
                select_addr_count <= 0;
                mix <= "00";
                stack_wea <= '0';
        else
                j <= j + 1;
                state_select <= "0001";
                state <= "11011";-- Temporary escape plan
                curr_state := state;
                mix <= "01";
                mulx_cover <= cover;
                idx <= "000000001";
                base <= (others => '0');
                select_addr_count <= 0;
                tmp_selected <= 0;
                cover_status <= cover(select_addr_count);
                stack_addra <=conv_std_logic_vector(status,8);
                stack_wea <= '1';
        end if;

elsif (state = "10100") then
        cover(tmp_selected) <= '1';
base <= (others => '0');
select_addr_count <= 0;
order_vec(status) <= tmp_selected;
status <= status + 1;
```

```
tmp_status <= status;
state <= "10101";

elsif (state = "10101") then
                        mix                                 <=                          "10";
                        state_edge                          <=                          "001";
                        state                               <=                          "11011";
                        curr_state                          :=                          state;
                        mulx_cover                          <=                          cover;
                        idx                         <=                          "000000001";
                        edge_addr_count                             <=                  0;
                        cover_status <= cover(edge_addr_count);

elsif (state = "10110") then
        state <= "10011";

elsif (state = "10111") then
        state <= "00111";

elsif (state = "11001") then
        if tmp_finish ='1' and ram_counter/=31 then
                ram_counter:= ram_counter + 1;
        else
                tmp_finish <= '0';
                ram_counter := 0;
                state <= "11010";
        end if;

--/*/*/*/*/*/*/*/*/Edgeless function check starts here /*/*/*/*/*/*/*/*/
elsif (state_edge = "001") then
        edge_addr_count <= 0;
        state_edge <= "010";

elsif (state_edge = "010") then

if (edge_addr_count = graph_size) then

state_edge <= "101";
edge_addr_count <= 0;

else

state_edge <= "011";

end if;
```

```
elsif (state_edge = "011") then

if (mix_vector /=
"11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111") then
        state_edge <= "101";
else
        idx <= idx + 1;
        edge_addr_count <= edge_addr_count + 1;
        state_edge <= "100";

end if;

elsif (state_edge = "100") then
        cover_status <= mulx_cover(edge_addr_count);
        state_edge <= "010";

elsif (state_edge = "101") then

        if (mix_vector =
        "11111111111111111111111111111111111111111111111111111111111111111111111111111111
        11111111111111111111111111111111111111111111111111111111111111111111111111111111
        11111111111111111111111111111111111111111111111111111111111111111111111111111111
        11111111111111111111111111111111111111111111111111111111111111111111111111111111") then
                tmp_finish <= '1';
                state_edge <= (others => '0');
                state <= "11001";-- Temporary escape plan
                mix <= "00";

        else
                state <= curr_state + "00001";
                state_edge <= (others => '0');
                mix <= "00";
                edge_addr_count <= 0;
         end if;
--/*/*/*//*/*/*/*/*/*Edgeless function check ends here/*/*/*/*/*/*/*/*/*/*/*/*/*/*/


--/*/*/*/*/*SELECT vertices function starts here/*/*/*/*/*/**/*/*/*/*/*/*/*/*/*/

elsif (state_select = "0001") then
```

```
        base <= (others => '0');
        state_select <= "0010";

elsif ( state_select = "0010") then

        tmp_ones_ct_1 <= ('0' & ones_ct_1) + ('0' &ones_ct_2);
        tmp_ones_ct_2 <= ('0' & ones_ct_3) + ('0' &ones_ct_4);
        tmp_ones_ct_3 <= ('0' & ones_ct_5) + ('0' &ones_ct_6);
        tmp_ones_ct_4 <= ('0' & ones_ct_7) + ('0' &ones_ct_8);
        tmp_ones_ct_5 <= ('0' & ones_ct_9) + ('0' &ones_ct_10);
        tmp_ones_ct_6 <= ('0' & ones_ct_11) + ('0' &ones_ct_12);
        tmp_ones_ct_7 <= ('0' & ones_ct_13) + ('0' &ones_ct_14);
        tmp_ones_ct_8 <= ('0' & ones_ct_15) + ('0' &ones_ct_16);
        state_select <= "0011";

elsif (state_select = "0011") then
                tmp_ones_ct_9 <= ('0' & tmp_ones_ct_1) + ('0' &tmp_ones_ct_2);
                tmp_ones_ct_10 <= ('0' & tmp_ones_ct_3) + ('0'
                &tmp_ones_ct_4);
                tmp_ones_ct_11 <= ('0' & tmp_ones_ct_5) + ('0'
                &tmp_ones_ct_6);
                tmp_ones_ct_12 <= ('0' & tmp_ones_ct_7) + ('0'
                &tmp_ones_ct_8);
                state_select <= "0100";

elsif (state_select = "0100") then
        tmp_ones_ct_13 <= ('0' & tmp_ones_ct_9) + ('0' &tmp_ones_ct_10);
tmp_ones_ct_14 <= ('0' & tmp_ones_ct_11) + ('0' &tmp_ones_ct_12);
state_select <= "0101";

        elsif (state_select = "0101") then

                tmp_ones_ct_15 <= ('0' & tmp_ones_ct_13) + ('0' &
                tmp_ones_ct_14);
                state_select <= "0110";

elsif (state_select = "0110") then

        if (select_addr_count = graph_size) then

                state_select <= "1000";
        else

                if (tmp_ones_ct_15 > base) then
```

```
                    tmp_selected <= select_addr_count;
                    base <= tmp_ones_ct_15;
                else
                    tmp_selected <= tmp_selected;
                end if;
                state_select <= "0111";
                idx <= idx + 1;
                select_addr_count <= select_addr_count +1;
        end if;

elsif (state_select = "0111") then
        cover_status <= mulx_cover(select_addr_count);
        state_select <= "0010";

elsif (state_select = "1000") then

        if (tmp_ones_ct_15 > base) then
                tmp_selected <= select_addr_count;
                base <= tmp_ones_ct_15;
        else
                tmp_selected <= tmp_selected;
        end if;
        state <= curr_state + "00001";
        state_select <= (others => '0');


--/*/*/*/*/*/*/*SELECT vertices function ends here/*/*/*/*/*/*/*/*/


        end if;

end if;

end process;

-- PORT MAPPING FOR THE INDIVIDUAL COMPONENTS STARTS HERE

UUT_MIX : stage_mix port map (mix_st => mix, mix_status =>cover_status,
mix_adj_list  => adj_list,mix_cover => mulx_cover,mix_vector => mix_vector);
UUT_SUM1: adder_sum1 port map (bit_vector_1 => mix_vector(15 downto 0),god_sum
=> ones_ct_1);
UUT_SUM2: adder_sum1 port map (bit_vector_1 => mix_vector(31 downto 16),
god_sum => ones_ct_2);
UUT_SUM3: adder_sum1 port map (bit_vector_1 => mix_vector(47 downto 32),
```

```
god_sum => ones_ct_3);
UUT_SUM4: adder_sum1 port map (bit_vector_1 => mix_vector(63 downto 48),
god_sum => ones_ct_4);
UUT_SUM5: adder_sum1 port map (bit_vector_1 => mix_vector(79 downto 64),
god_sum => ones_ct_5);
UUT_SUM6: adder_sum1 port map (bit_vector_1 => mix_vector(95 downto 80),
god_sum => ones_ct_6);
UUT_SUM7: adder_sum1 port map (bit_vector_1 => mix_vector(111 downto 96),
god_sum => ones_ct_7);
UUT_SUM8: adder_sum1 port map (bit_vector_1 => mix_vector(127 downto 112),
god_sum => ones_ct_8);
UUT_SUM9: adder_sum1 port map (bit_vector_1 => mix_vector(143 downto 128),
god_sum => ones_ct_9);
UUT_SUM10: adder_sum1 port map (bit_vector_1 => mix_vector(159 downto 144),
god_sum => ones_ct_10);
UUT_SUM11: adder_sum1 port map (bit_vector_1 => mix_vector(175 downto 160),
god_sum => ones_ct_11);
UUT_SUM12: adder_sum1 port map (bit_vector_1 => mix_vector(191 downto 176),
god_sum => ones_ct_12);
UUT_SUM13: adder_sum1 port map (bit_vector_1 => mix_vector(207 downto 192),
god_sum => ones_ct_13);
UUT_SUM14: adder_sum1 port map (bit_vector_1 => mix_vector(223 downto 208),
god_sum => ones_ct_14);
UUT_SUM15: adder_sum1 port map (bit_vector_1 => mix_vector(239 downto 224),
god_sum => ones_ct_15);
UUT_SUM16: adder_sum1 port map (bit_vector_1 => mix_vector(255 downto 240),
god_sum => ones_ct_16);

UUT_STACK: stack port map
(addra=>stack_addra,addrb=>stack_addrb,clka=>clk,clkb=>clk,dina=>cover,dinb=>tmp
_dinb,douta=>stack_douta,doutb=>tmp_doutb,wea=>stack_wea,web=>tmp_web);

-- PORT MAPPING FOR THE INDIVIDUAL COMPONENTS ENDS HERE

addr <= idx;
finish <= tmp_finish;
mask <= mulx_cover;

end rtl;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity stage_mix is
port
        (
mix_st : in std_logic_vector(1 downto 0);
mix_status : in std_logic;
mix_adj_list : in std_logic_vector(255 downto 0);
mix_cover : in std_logic_vector(255 downto 0);
mix_vector : out std_logic_vector(255 downto 0)
);
        end stage_mix;

architecture stage_mix_a of stage_mix is
begin
process(mix_st,mix_status,mix_adj_list,mix_cover)
begin
case mix_st is

when "01" => -- Select vertex
        for i in 0 to 255 loop
                        if (mix_status = '0') then
                                if((mix_adj_list(i) = '1') and(mix_cover(i) = '0')) then
                                        mix_vector(i) <= '1';
                        else
                                        mix_vector(i) <= '0';
                                end if;
        else
                        mix_vector(i) <= '0';
                end if;
        end loop;


when "10" => -- Edgeless
for i in 0 to 255 loop
        if (mix_status = '0') then
                        if((mix_adj_list(i) = '1') and(mix_cover(i) = '0')) then
                                mix_vector(i) <= '0';
        else
```

```
                        mix_vector(i) <= '1';
                end if;

        else
                        mix_vector(i) <= '1';
                end if;
        end loop;

when "11" => -- Neighbour count
for i in 0 to 255 loop
                        if((mix_adj_list(i) = '1') and (mix_cover(i) ='0')) then
                                mix_vector(i) <= '1';
                        else
                        mix_vector(i) <= '0';
                end if;
        end loop;

when others =>
                        mix_vector <= (others => '0');

        end case;
end process;
end stage_mix_a;
```

```
	library ieee,synopsys,dware,DW01;
use ieee.std_logic_1164.all;
use synopsys.attributes.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use DWARE.DWpackages.all;
use DW01.DW01_components.all;
entity adder_sum1 is
port
	(
	bit_vector_1 : in std_logic_vector(15 downto 0);
	god_sum : out std_logic_vector(4 downto 0)
	);
end adder_sum1;

architecture adder_sum1_a of adder_sum1 is
signal tmp_2,tmp_5,tmp_8,tmp_11,tmp_12,tmp_13,tmp_14,tmp_15: std_logic;
signal tmp_0, tmp_1, tmp_3, tmp_4, tmp_6, tmp_7, tmp_9, tmp_10, sum_1,
sum_2,sum_3,sum_4: std_logic_vector(1 downto 0);
signal tmp_sum_1,tmp_sum_2,tmp_sum_3,tmp_sum_4,sum_5,sum_6
: std_logic_vector(2 downto 0);
signal tmp_sum_5,tmp_sum_6,sum_7 : std_logic_vector(3 downto 0);
signal tmp_sum_7,tmp : std_logic_vector(4 downto 0);

begin
U1: DW01_add
generic map (width => 2)
port map ( A => tmp_0, B => tmp_1,CI => tmp_2, SUM =>sum_1);

U2: DW01_add
generic map (width => 2)
port map ( A => tmp_3, B => tmp_4,CI => tmp_5, SUM =>sum_2);

U3: DW01_add
generic map (width => 2)
port map ( A => tmp_6, B => tmp_7,CI => tmp_8, SUM =>sum_3);

U4: DW01_add
generic map (width => 2)
port map ( A => tmp_9, B => tmp_10,CI => tmp_11, SUM =>sum_4);
```

```
U5: DW01_add
generic map (width => 3)
port map ( A => tmp_sum_1, B => tmp_sum_2, CI =>tmp_12,SUM => sum_5);

U6: DW01_add
generic map (width => 3)
port map ( A => tmp_sum_3, B => tmp_sum_4, CI =>tmp_13,SUM => sum_6);

U7: DW01_add
generic map (width => 4)
port map ( A => tmp_sum_5, B => tmp_sum_6, CI =>tmp_14,SUM => sum_7);

U8: DW01_add
generic map (width => 5)
port map ( A => tmp_sum_7, B => tmp,CI => tmp_15, SUM =>god_sum);

process(bit_vector_1)
begin
end process;
tmp_0 <='0' & bit_vector_1(0);
tmp_1 <='0' & bit_vector_1(1);
tmp_3 <='0' & bit_vector_1(3);
tmp_4 <='0' & bit_vector_1(4);
tmp_6 <='0' & bit_vector_1(6);
tmp_7 <='0' & bit_vector_1(7);
tmp_9 <='0' & bit_vector_1(9);
tmp_10 <='0' & bit_vector_1(10);
tmp_2 <= bit_vector_1(2);
tmp_5 <= bit_vector_1(5);
tmp_8 <= bit_vector_1(8);
tmp_11 <= bit_vector_1(11);
tmp_12 <= bit_vector_1(12);
tmp_13 <= bit_vector_1(13);
tmp_14 <= bit_vector_1(14);
tmp_15 <= bit_vector_1(15);
tmp_sum_1 <= '0' & sum_1;
tmp_sum_2 <= '0' & sum_2;
tmp_sum_3 <= '0' & sum_3;
tmp_sum_4 <= '0' & sum_4;
tmp_sum_5 <= '0' & sum_5;
tmp_sum_6 <= '0' & sum_6;
tmp_sum_7 <= '0' & sum_7;
tmp <= (others => '0');
end adder_sum1_a;
```

Vita

Mahesh Dorai was born in Madras, India. He received his Bachelor of Engineering in Electrical and Electronics engineering from Anna University, Madras, India. He joined the University of Tennessee, Knoxville as a Graduate student (Masters program) in August 2001. Subsequently he has been doing his research under the guidance of Prof. Gregory D. Peterson. He plans to graduate with a Master's degree in Electrical engineering in May 2004.