

Portland State University

**PDXScholar**

---

Dissertations and Theses

Dissertations and Theses

---

1-1-2011

# Simulation and emulation of massively parallel processor for solving constraint satisfaction problems based on oracles

Gunavant Dinkar Chaudhari  
*Portland State University*

Follow this and additional works at: [https://pdxscholar.library.pdx.edu/open\\_access\\_etds](https://pdxscholar.library.pdx.edu/open_access_etds)

**Let us know how access to this document benefits you.**

---

## Recommended Citation

Chaudhari, Gunavant Dinkar, "Simulation and emulation of massively parallel processor for solving constraint satisfaction problems based on oracles" (2011). *Dissertations and Theses*. Paper 11.  
<https://doi.org/10.15760/etd.11>

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: [pdxscholar@pdx.edu](mailto:pdxscholar@pdx.edu).

Simulation and Emulation of Massively Parallel  
Processor for Solving Constraint Satisfaction Problems Based on Oracles

by

Gunavant Dinkar Chaudhari

A thesis submitted in partial fulfillment of the  
requirements for the degree of

Master of Science  
in  
Electrical and Computer Engineering

Thesis Committee:  
Marek Perkowski, Chair  
Xiaoyu Song  
Douglas Hall

Portland State University  
©2011

## **Abstract**

Most part of my thesis is devoted to efficient automated logic synthesis of oracle processors. These Oracle Processors are of interest to several modern technologies, including Scheduling and Allocation, Image Processing and Robot Vision, Computer Aided Design, Games and Puzzles, and Cellular Automata, but so far the most important practical application is to build logic circuits to solve various practical Constraint Satisfaction Problems in Intelligent Robotics. For instance, robot path planning can be reduced to Satisfiability. In short, an oracle is a circuit that has some proposition of solution on the inputs and answers yes/no to this proposition. In other language, it is a predicate or a concept-checking machine. Oracles have many applications in AI and theoretical computer science but so far they were not used much in hardware architectures.

Systematic logic synthesis methodologies for oracle circuits were so far not a subject of a special research. It is not known how big advantages these processors will bring when compared to parallel processing with CUDA/GPU processors, or standard PC processing. My interest in this thesis is only in architectural and logic synthesis aspects and not in physical (technological) design aspects of these circuits. In future, these circuits will be realized using reversible, nano and some new technologies, but the interest in this thesis is not in the future realization technologies. We want just to answer the following question: Is there any speed advantage of the new oracle-based architectures, when compared with standard serial or parallel processors?

## **Acknowledgement**

I wish to express my sincere gratitude to Professor Dr. Marek Perkowski for having given me a wonderful research opportunity. I would like to thank him for his encouragement, support and guidance. I would like to thank Mentor Graphics Corporation for giving Veloce Emulator and hardware accelerator to Portland State University. I would also like to thank Professor Douglas Hall and Xiaoyu Song for their valuable feedback. I would also like to thank all of my friends for making my stay at Portland State University a wonderful one.

# Table of Contents

Abstract.....	i
Acknowledgement.....	ii
Chapter 1 Introduction, Goals and Background .....	1
1.1 Goals of This Thesis:.....	3
1.1.1 What this Thesis is about: Synthesis of Oracles for CSP problems.....	3
1.1.2 What this dissertation is NOT about?.....	4
1.2 Universality versus specialization of the methods that I proposed.....	5
1.3 Why do we need CAD methods for synthesis of oracle processors?.....	6
1.4 Previous work on synthesis of oracles.....	7
1.5 Innovative aspects of this thesis .....	7
1.6 Uniform presentation of methods .....	8
1.7 My approach.....	8
Chapter 2 The Concept of Oracle-Based Machines for Constraint Satisfaction Problems.....	10
2.1 My Goals .....	10
2.2 Why Logic Accelerators are superior to classical computers when solving certain classes of problems. ....	15
2.3 Towards combinatorial problem solving.....	16
2.3.1 NP problems and CSP problems .....	16
2.3.2 Building a circuit for a very simple oracle .....	18
2.3.3 Building oracles for CAD, CSP and robotics .....	24
2.4 Architectures .....	25
2.5 How to convert abstract problem to an architecture .....	27
2.6 The main motivation and principle for architecture .....	28
2.7 CSP architectures in practical robotics and digital CAD tools .....	30
2.8 Why logic puzzles are not only fun but are important.....	31
2.9 Historical Remark.....	32
2.10 Principles of problem-solving and design methodologies .....	33
2.10.1 Principle of reducing CSP problems to building Oracles for them.....	33
2.11 Solving Constraint Satisfaction problems Using Oracles.....	33
2.11.1 Problems with no information in robotics and CAD .....	35
2.12 Solving problems by reducing them to basic combinatorial search problems.....	36
2.12.1 Main problem representations and data structures in CSP problems.....	36
2.12.2 Universal and problem-specific methods or mixing them? .....	37
2.12.3 Properties of my problem-solving model .....	38
2.13 Combinatorial Problems in synthesis and optimization of circuits.....	39
2.13.1 Future CAD tools .....	39
2.13.2 Logic circuits other than AND/OR/NOT circuits.....	40
2.14 Types of oracles to solve combinational problem .....	40
2.14.1 Combinational oracle.....	40
2.14.2 Sequential oracle .....	41
2.14.3 Combinational oracles .....	41
2.14.4 Sequential oracles .....	43
2.15 Mechanisms of exercising oracles .....	44
2.15.1 Random number generation.....	44
2.15.2 Simulated annealing .....	44
2.15.3 Pseudo Boolean programming.....	44
2.15.4 Integer programming .....	45
2.15.5 Exhaustive search based on counting .....	45
2.15.6 Intelligent search methods.....	45
2.15.7 Genetic algorithms.....	45
2.15.8 Dynamic programming.....	45
2.16 Example of a sequential oracle .....	45

2.17	New General-Purpose Search Approaches for classes of combinatorial problems.	53
2.17.1	How to exercise oracles?	53
2.17.2	Types of exercising mechanism for CSP algorithms.	54
2.17.3	Exercising for complex data structure algorithms	55
2.18	Organization of the thesis	56
2.19	New concepts of CSP algorithms for particular structures	57
2.19.1	Use of heuristics versus universality of algorithms.	57
2.19.2	Universal algorithms	58
2.20	The role of additional knowledge and heuristics in creating algorithms	59
2.20.1	Machine Learning methods.	60
2.20.2	Summary of main concepts and ideas	65
Chapter 3	Emulators and Accelerators in CAD Industry	67
3.1	Verification Techniques.	67
3.1.1	Cycle Based Simulation	68
3.1.2	Hardware Based Verification Tools.	68
3.2	Transaction Based Verification	70
3.3	Co-Modeling	71
3.3.1	Transaction Interface Portal (TIP)	72
3.3.2	Veloce Emulator.	74
Chapter 4	The Veloce Emulator and How to Use it	80
Chapter 5	SAT: Search for Satisfiability, Petrick Function Minimization and Related Problems	93
5.1	Introduction	93
5.2	Solving the Satisfiability Class of Problems	98
5.2.1	Product of Sums SAT (POS SAT)	98
Chapter 6	Solving Problems in Artificial Intelligence And Robotics.	104
6.1	AND/OR DAGs.	104
Chapter 7	Solving the Unate and Binate Covering Problems	109
7.1	Solving the Unate Covering Problem.	109
7.2	Solving the Binate Covering Problem	115
Chapter 8	Specialized Processor for Man Wold Goat and Cabbage Constraint Satisfaction Sequential Problem	121
8.1	Introduction and goals	121
8.1	Checking Safe Condition at Both Bank.	123
8.2	Checking for invalid move	124
8.3	VHDL Code for Move Checking	129
8.4	VHDL code for Final Oracle	130
8.5	Test bench for testing Oracle and finding solution	133
8.6	The Output file contents	136
8.7	Emulator testbench	136
Chapter 9	Socialized Processor For Missionaries and Cannibals Constraint Sequential Problem	138
9.1	Problem Statement:	138
9.2	Checking Safe Condition at Both Bank.	140
9.3	Checking for invalid move	143
9.4	VHDL Code for Checking validity of state	146
9.5	VHDL Code for checking validity of Move	147
9.6	Final VHDL Code for Oracle	148
9.7	Testbench For Emulation	151
Chapter 10	Oracle for Graph Coloring	155
10.1	Proposed Architecture for Graph Coloring Problem using Grover's Algorithm	157
10.1.1	The C blocks:	160
10.1.2	The Sorter/Absorber:	161
10.2	Graph Coloring Oracle	161
10.3	Exercising the oracle using a counter	163

10.4	How does the sorter work? .....	163
10.5	Implementation of the oracle in Verilog.....	164
Chapter 11	Oracle for Longest Path .....	165
11.1	Introduction .....	165
11.2	Node Representation .....	169
Chapter 12	Conclusion to the thesis and Emulations on Oracle Based Processors .....	178
12.1	Wall clock speed-up comparison.....	179
12.2	Industry method for quantifying effect of emulation.....	180
12.3	Final Conclusion.....	183

## List of figures

Figure 2-1 Graph for coloring with five nodes.....	20
Figure 2-2 The inequality comparator used in Map Coloring and Graph Coloring problems.....	21
Figure 2-3 The inequality comparator.....	22
Figure 2-4 Encoding of colors for graph coloring oracle.....	22
Figure 2-5 Principle of graph coloring applied to a simple graph.....	23
Figure 2-6 Example of standard combinational oracle applied to a SAT problem.....	41
Figure 2-7 Oracle with cost function as a feedback.....	42
Figure 2-8 Oracle with direct feedback.....	43
Figure 2-9 Sequential Oracle implementation.....	47
Figure 3-1 Architecture of TIP.....	73
Figure 3-2 Veloce Family.....	75
Figure 3-3 Emulator Technology.....	76
Figure 3-4 Stand-alone Mode.....	77
Figure 3-5 HDL Link Mode.....	78
Figure 3-6 TBX Mode.....	79
Figure 5-1 Classical Oracle for POS satisfiability.....	98
Figure 5-2 Oracle for $f$ .....	99
Figure 5-3 Oracle for function $f_2$ .....	102
Figure 5-4 (a) Coloring Graph Problem (b) Maximum Independent set for graph coloring problem.....	102
Figure 6-1 AND/OR DAG.....	104
Figure 6-2 Example of the AND (a) the subgraph, (b) the logical transformation.....	105
Figure 6-3 Example of the OR node. (a) the subgraph, (b) the logical transformation.....	105
Figure 6-4 Oracle for function $f_2 = (\bar{a} + b + c)(\bar{b} + d + e)(\bar{c} + e \bullet f)(\bar{e} + g)(\bar{d} + h \bullet i)$ .....	106
Figure 7-1 Oracle for the function $f = \bar{a}\bar{c}\bar{d} + \bar{a}bc + acd + ab\bar{c} + bd$ .....	111
Figure 7-2 Solving the Petrick Function from the unate covering table 7-2.....	111
Figure 7-3 Oracle for the function $f = (AB + AD + CBF + CBE)$ . The circuit is not optimized.....	112
Figure 7-4 Implication graph for the binate covering problem.....	116
Figure 7-5 Oracle for binate covering problem.....	117
Figure 7-6 (a) Finite state machine (b) Triangular table indicating the compatibility for combining different states in the FSM.....	118
Figure 7-7 (a) Compatibility graph for the FSM (b) Incompatibility graph for the FSM.....	119
Figure 7-8 Minimized FSM using binate covering problem.....	120
Figure 9-1 One possible solution of the puzzle.....	139
Figure 9-2 Oracle for checking invalid state.....	143
Figure 9-3 Oracle for checking invalid moves.....	145
Figure 10-1 Maps of Europe.....	155
Figure 10-2. superposed quantum states with negative phase for all good colorings of a map.....	158
Figure 10-3 A simple graph coloring problem.....	159
Figure 10-4 Simplified schematic of my optimization Graph Coloring Oracle.....	160
Figure 10-5 Sorter/Absorber.....	161
Figure 10-6 Architecture of graph coloring oracle.....	162
Figure 10-7 Exercising the oracles using a counter.....	163
Figure 10-8 Sorter using butterfly pattern.....	164
Figure 11-1 Graph for the Longest Path.....	166
Figure 11-2 Example which shows how the node selection is done.....	167
Figure 11-3 Oracle for Longest Path.....	167
Figure 11-4 Example where the edge selection is not done properly.....	168
Figure 11-5 Oracle for 'n' edges.....	169
Figure 11-6 Representation for Node1.....	171
Figure 11-7 Representation for Node2.....	172
Figure 11-8 Representation for node3.....	173



Figure 11-9 Representation for Node 4 .....	174
Figure 11-10 Connection between Nodes .....	175
Figure 11-11 Circuit for path length Calculation .....	176
Figure 11-12 Final Multiplexer for longest path oracle.....	177
Figure 12-1 Speed-up on Veloce against input vectors traced .....	182

## List of Tables

Table 1.1 Goal of the thesis.....	5
Table 2.1 Assignment of bits to encoded colors of nodes for the graph from .....	20
Table 7.1 Finding graphically all prime implicants for minimal covering of a SOP circuit.....	109
Table 7.2 Covering table for function .....	110
Table 7.3 Example of an unite covering problem represented by a table.....	112
Table 7.4 Covering closure table for the implication graph .....	116
Table 7.5 Covering closure table for the FSM .....	119
Table 12.1 Wall clock comparison .....	179
Table 12.2 Results of emulations .....	181

# Chapter 1 Introduction, Goals and Background

This thesis is devoted to emulation and validation of digital and mixed systems using modern commercial hardware emulators. We take examples from the area of design of oracle-based hardware processors for constraint satisfaction problems.

The thesis is devoted to the efficient automated logic synthesis of oracle processors. These Oracle Processors are of interest to several modern technologies, including Scheduling and Allocation, Image Processing and Robot Vision, Computer Aided Design, Games and Puzzles, and Cellular Automata, but so far the most important practical application is to build logic circuits to solve various practical Constraint Satisfaction Problems in Intelligent Robotics. For instance, robot path planning can be reduced to Satisfiability. In short, an oracle is a circuit that has some proposition of solution on the inputs and answers yes/no to this proposition. In other language, it is a predicate or a concept-checking machine. Oracles have many applications in AI and theoretical computer science but so far they were not used much in hardware architectures. A typical oracle is a Satisfiability Processor that checks if a given Boolean function is satisfiable.

So far, the systematic logic synthesis methodologies for oracle circuits are not a subject of a special research. It is not known how big advantages these processors will bring when compared to parallel processing with CUDA/GPU processors, or standard PC processing. My interest in this thesis is only in architectural and logic synthesis aspects and not in physical (technological) design aspects of these circuits. In future, these

circuits will be realized using reversible, nano and some new technologies, but the interest in this thesis is not in the future realization technologies. We want just to answer the following question: “Is there any speed advantage of the new oracle-based architectures, when compared with standard serial processors?”

### **Comments and explanations**

1. Many decision and optimization problems can be reduced to solving the satisfiability problem, realized in hardware using oracles.
2. Oracle is an arbitrary circuit with one output that answers YES/NO and which has inputs that encode the elements of the space of all solutions to the given problem. Theoretically such circuits can be described by some type of truth tables. But this would be not practical. So we have to build oracles the same way as data path is built in standard processors; it is built from certain typical blocks such as comparators and adders and logic gates.
3. The main problem of synthesis of oracle circuits is how to encode the information on inputs and how to build and compose the individual blocks of the oracle.

The focus of this thesis is on synthesis of binary circuits for oracles and then on exercising these data-path oracles with sequences of solution-candidates – the binary vectors. The oracles built in this thesis are not related to any specific future technology, although the future of these types of processors is in new *nano-technologies*.

Many practical quantum algorithms, such as the Grover Algorithm, have oracles. These oracles are exercised in parallel based on quantum superposition. In my thesis, the oracles

are however exercised sequentially, as they are realized in standard digital technology, such as FPGAs or ASICs. Thus, the thesis, although not related directly to quantum technology, can be helpful to build quantum oracles. Any description of a classical oracle from classical binary logic circuits can be converted in a next stage to a reversible logic circuit next realized in some quantum technology to allow for superposition of states. This is however an area of future research.

In addition to quantum circuits that are reversible, it is expected by many researchers that reversible circuits will be built for many new nano-technologies other than quantum. There is very little published on the logic design aspects for these non-quantum reversible technologies. In this thesis, however, we discuss only classical circuits for oracles and we concentrate on block level design. Only in future the oracles presented here will be converted to reversible circuits, both quantum and non-quantum, such as for instance optical. Converting from non-reversible to reversible technology is however now a solved problem, so in this thesis I concentrate on the very issues of creating oracles and not how they are realized in final detail.

## **1.1 Goals of This Thesis:**

### **1.1.1 What this Thesis is about: Synthesis of Oracles for CSP problems**

I believe that Oracle-based computing is of high importance to future computing, and especially in Intelligent Robotics where many important problems can be characterized as Constraint Satisfaction Problems. Therefore as the background of this thesis I will

superficially introduce several CSP problems, in which special accelerated algorithms will be of high use. I will not go to the physical principles of designing such algorithms. Next I will present much improved and new methods for synthesizing circuits for such algorithms that will be highly practical for new nano-technologies, in which regularity of structure is fundamental. These methods will be superior to previous methods in the sense given below.

### **1.1.2 What this dissertation is NOT about?**

My interest is only in classical behaviors of oracle circuits. We are also not interested in the technological aspects of gate realizations. In this thesis I am not interested in what is inside a logical or arithmetical block of the oracle or the exercising controller. I just synthesize these blocks using the hardware description languages: Verilog or VHDL. Next, I use these blocks in logic synthesis of the entire processor. In classical logic synthesis, the combinational logic synthesis is a separate research subject from Finite State Machine (FSM) design, but designing a combinational circuit of excitation and output logics is used as a part of the so-called structural synthesis of FSMs. These topics are not described in this thesis, but the knowledge of the fact that combinational circuits are used to build sequential oracle circuits highlights the importance of developing systematic synthesis methods for them.

I will concentrate only on the logical structures of oracle circuits and algorithms. I will also consider the “realization costs” of circuits in these technologies (i.e., costs of circuit realization).

The goals of this thesis are specified in Table 1.1.

<i><b>Goal number</b></i>	<i><b>Goal formulation</b></i>	<i><b>Competitor to be measured against</b></i>
Goal 1	Being able to synthesize oracle-based processors and their circuits for various CSP problems.	Contemporary technologies such as CUDA, parallel processors, laptop PCs.
Goal 2	Being able to compare the speed of the oracle circuits emulated on VELOCE to other technologies.	Compare speed of emulated oracle on VELOCE with speed of software on PC.
Goal 3	Being able to compare simulation on VELOCE, simulation on a laptop, and emulation on VELOCE	No comparison has been done so far by other authors.
Goal 4	Propose a complete CAD system to design oracles.	No method, theory or CAD tool exists that can do this.

**Table 1.1 Goal of the thesis**

In coming chapters of my thesis, I will relate to these goals and show how these goals were satisfied by designs and software developed in this thesis. Note that improved processing speed is the main and only goal of the developed processors. The belief that some special-purpose architectures are able to solve very quickly many important CSP problems is a common opinion of several researchers in this area. It is also very important to understand current limitations for synthesis of large oracles from the algorithmic and synthesis points of view.

## **1.2 Universality versus specialization of the methods that I proposed**

In this thesis we will integrate and compare previous approaches to the synthesis of oracles. All processors and blocks proposed in this thesis can be tuned towards certain technologies by simply incorporating that technology's cost functions into the synthesis process for these algorithms. Because these technologies are not yet decided, we keep

classical technologies as the base of cost functions that evaluate the architectures and designs.

### **1.3 Why do we need CAD methods for synthesis of oracle processors?**

Automatic synthesis of oracle processors is an important component of the future design methodologies of specialized computers, simply because future CAD systems for such computers will use some software similar to software presented here. This task is important by analogy, as everybody appreciates the importance of logic synthesis CAD for classical binary computers. The methods developed in this thesis can find applications in “by-hand” synthesis. In future, however, these methods will find applications in automated synthesis of oracle processors for various new technologies such as quantum, optical, nano, DNA, carbon nanotube, quantum dots, and CMOS circuits. As these technologies become more mature, every year there is a need to develop efficient Computer Aided Design (CAD) methods for them. For example, a simplified but complete standard micro-processor was built for the (non-reversible) quantum dot technology at Notre Dame University. In other areas, such as reversible quantum dot technology, the design practice is also ahead of the current level of the synthesis tools. The thesis wants to partially bridge the gap by developing methods to build binary non-reversible oracles to be next (in the not shown research) converted to one of particular new technologies (reversible, quantum, quantum dot, optical, etc).



## **1.4 Previous work on synthesis of oracles**

In my thesis, I briefly present and analyze some important ideas from several already published papers about CSP problems. Many such papers have been published worldwide in the last 20 years. There are however no papers on systematic design of oracles. In general, the approach to synthesize processors based on oracles for CSP problems is not known for classical architectures, all research is only about quantum circuits based on Grover algorithm (Hossain, 2009). I have searched data bases such as Google Scholar, to find relevant papers, I have also (with the help of my advisor), familiarized myself with many algorithms that exist for synthesis of general digital circuits and with software to solve some important CSP problems. And I have studied different algorithms and search heuristics that are useful in my research. The research presented in this thesis combines ideas from various areas of research. My research is the second publication that applies the concept of oracles to solve a wide class of CSP problems – the first one comes from the PhD Dissertation of Sazzad Hossain(Hossain, 2009).

## **1.5 Innovative aspects of this thesis**

The next goal of this thesis is to show areas of oracle design for which no research has been done at all, and that are important to future design of reversible circuits and other nano-level systems.

## **1.6 Uniform presentation of methods**

I will also discuss in a uniform way several oracle and circuit synthesis methods that were already proposed in the domain of oracle based quantum computing. My dissertation will completely abstract from the underlying technologies, and concentrate only on mathematical, logical and algorithmic aspects of oracle system design. These are the core topics of my thesis.

In this thesis, I will present my original unified methods to synthesize binary oracle circuits from specification methods used in classical logic synthesis. My thesis is the first piece of research that uses these VHDL and Verilog specifications in the synthesis of non-quantum oracles.

## **1.7 My approach**

In this thesis I will discuss the experimental results of my software-realized hardware emulations based on some theories and algorithms outlined above. As the technology is ahead of CAD in the area of oracle synthesis, it is very important to develop new software that can be used to synthesize large functions and that will also have additional functionalities for speed, which is the main issue of the presented research. Such software was created during my research. Processors designed by my software will exceed in speed all results published in the literature. These processors, when realized in any existing and future technology will be able to solve new problems that remained unsolved as of year 2011 and achieve goals from Table 1.1 in section 1.1.2.

*Why are all these CSP problems so hard to solve? The reason is that they are all at least NP-hard or NP-complete in the complexity hierarchy.*

NP-hard are optimization equivalents of NP-complete problems. NP-complete problems require exponential complexity to find a solution (**Impagliazzo, 1998**) but only polynomial complexity to verify the solution. An example of NP complete problem is the well-known combinatorial problem to find a coloring of a graph (such that every two neighbor nodes of the graph have different colors) with less than **k** colors or prove that such coloring does not exist. This is a decision problem. The NP hard problem would be to find the coloring of the graph with the minimum number of colors such that every two neighbor nodes of the graph have different colors. This is an optimization problem.

## Chapter 2 The Concept of Oracle-Based Machines for Constraint Satisfaction Problems

### 2.1 My Goals

It is well-known that there are many problems in robotics, computational intelligence, artificial intelligence, computer vision, scheduling of industrial operations, planning of actions (for instance actions of an assembly line), computer aided design, computational biology, genetic engineering and other areas that require extremely fast processing speeds for specialized problems, assuming however a limited cost of the computing system – so that a supercomputer is not used for computations. These are basically logic, not arithmetic problems. Many of these problems can be formulated in a maximally simplified form as logic puzzles that you can find in high school textbooks or popular magazines. These processors would be especially useful in future intelligent robots and embedded systems.

Field Programmable Gate Arrays (FPGAs) are special electronic devices that their hardware can be programmed in arbitrary way by their user and thus they can replace standard computers in many applications (**Perkowski, 1993**). The user can program arbitrary logic gates, their structures and connections, including processors with high parallelism. You can find many FPGAs in your cars, refrigerators and daily life “embedded systems”. FPGAs are also used to build inexpensive supercomputers; specialized rather than general-purpose ones. Using FPGAs a student can design his or her “own computer” dedicated to some task or a general purpose processor with their

own list of instructions (**Moore, 1993**). Several FPGA-based hardware accelerators have been already built or proposed for specialized tasks that require high computing speeds (**Rubin, Omieljanowicz, & Petrovsky, 2007**). They allow accelerating algorithms by implementing them all or their crucial parts, in hardware. Such accelerators have for instance applications in robotics, multimedia and medical diagnostics. For example, an imaging system for medical application may use multiple ASICs and FPGAs (**Leong, Bento, Lousa, & Nobre, 2006**).

The main original thesis of the presented by me work is, that problems that massively use logic operations need a different type of hardware than the standard hardware used in contemporary standard computers, or in DSP computers. In these modern architectures the arithmetic operations such as multiplication are accelerated, and not Boolean operation or similar. My interest is in problems with massive parallelism but simple processors – such processors may have no arithmetic unit and multipliers at all! Such models of computing have been proposed by several new technologies such as nano, quantum or DNA computing (**Song, 2008**).

An example of a logic problem that needs hardware acceleration is the Satisfiability Problem (SAT). The thesis will show how popular logic puzzles or robot planners can be reduced to SAT.

SAT problem is formulated as follows:

Given is a Boolean equation (Boolean function  $F$ ) with many input variables and one output variable, Find if there exists an assignment of binary values to Boolean variables that function  $F$  is satisfied (has value 1) and if yes, find what is this assignment of Boolean values to variables. In other words, find a binary vector of all variable values (called a minterm) that satisfies the logic equation, if such vector exists. In similar formulation we may look for an assignment of not all, but a satisfactory number of variables.

Many important and very practical problems can be reduced to the SAT problem as formulated above. For instance in robotics, these problems are usually solved with advanced software programs. Special computers for this task were however also built for CAD applications by NEC, IBM, and other companies, because speeding up SAT would enable efficient solving of important problems that are reducible to SAT. There are thousands of such problems. There are also several important problems that are not identical but similar to SAT. For instance, Tsutomu Sasao from Kyushu Institute of Technology in Japan built a logic machine to solve the logic Tautology problem. Tautology problem is to check for a given function  $F$  if  $F = 1$  for all combinations of values of variables that are used in function  $F$  (we call them the input variables of  $F$ ). Many machines to solve Boolean problems, such as solving Boolean Equations, were built in Russia and Germany in 19<sup>th</sup> and 20<sup>th</sup> Centuries. We believe that future computers Constraint Satisfaction Problems (CSP) or robotic problems for CAD problems will use hardware accelerators of various kinds, including logic accelerators as those proposed in

this thesis. These will especially include special hardware accelerators to solve the kinds of logic-intensive problems that we are interested in here. This thesis is a speculation about the nature and technology of general purpose logic accelerators for a selected set of important problems. These problems have been never addressed in a unified way so far and as far as we know, they were never a subject of a thesis.

Hardware emulators such as Veloce are used to emulate/simulate new digital designs, usually general purpose processors, DSP and special embedded systems. They serve thus to verify if the hardware is doing what it was expected by its designer to do. Sometimes the emulators can also help to evaluate testability or speed of the designed prototypes. In this thesis we want to achieve these goals, and even more. We want to check if a hardware emulator can be used as a hardware accelerator in order to obtain a significant speed-up over a standard software implemented on a standard computer (such as a laptop). If yes, then Veloce will be used in the next phase to run, via internet, the real-time control software for mobile robots equipped with stereo-vision. Veloce box will become thus a “poor man specialized supercomputer” for PSU robotics projects.

A complete cycle of designing a logic accelerator processor will include therefore the stages of designing, simulating, emulating and verifying for correctness and analyzing the potential speed-up over standard software for a PC. Even if we will find out that the hardware emulator Veloce is not giving a sufficient speedup, these experiments will help us, for sure, to evaluate what would be a possible speedup of a VLSI-based (or FPGA-based) system that have been first prototyped using the Veloce tool. For this, we should approximately know only the delays of logic gates in the selected by us implementation

technology, such as one of Xilinx or Altera FPGA architectures. The emulated design will be thus redesigned with other tools, widely available free FPGA-based software tools such as logic synthesizers and physical design tools, but with a small risk that the design will have different speed or functionality, as the circuits found and emulated on VELOCE will be very accurate. Therefore, there is no danger that the methods developed in this thesis will be of no practical use for future oracle and architecture designers. The readers of this thesis, future PSU students, will learn about high-performance special processors, reconfigurable computing, and use of CAD tools and last but not least, many constraint satisfaction problems (such as logic puzzles or known mathematical problems) and how to solve them.

The general class of CSP-accelerated algorithms should have a wide applicability to logic problems; for automated logic circuit synthesis and optimization, machine learning, robotics, vision, and directed knowledge discovery. It is so because, as it will be shown in this thesis, all these research areas are closely related. Although some problems and their reductions are known from classical algorithms, they may be new to my reader, so all background material will be explained to make the thesis self-contained. The thesis assumes only the knowledge of basic logic design and machines, on the level of PSU ECE 271 class (second class in logic design). This material can be found in Wakerly's book, or equivalent.



## **2.2 Why Logic Accelerators are superior to classical computers when solving certain classes of problems.**

This thesis is devoted to some aspects of designing one type of future computers – computers with accelerated logic operations. We can call them the “logic accelerators”, or “logic computers” as they have been historically called. One may ask “Why such computers are of interest and why are they are expected to be more powerful than the standard computers of year 2011?”

First, such logic computer will operate on very long words (**Zhang, 2000**) and will be specialized to execute operations on Boolean, Multiple-Valued logic functions, rough sets, sets, partitions, morphological data and similar data types. A standard processor operates on numbers. Evolution of standard processor architectures and lists of instructions in last 20 years developed very few ideas how to speed-up logic operations. There was much more emphasis on arithmetic’s, communication and multimedia processing, as well as general DSP algorithms and especially graphic algorithms.

Second, logic computers will have high level of parallelism (**Becker & Schmidt, 1998**) because all the real estate of the available system’s hardware will be used to implement logic operations and not other operations. These other operations (for instance, floating-point arithmetic) are not important for logic algorithms (even very little of integer arithmetic is used in logic algorithms).

Third, logic computers may use probabilistic algorithms. Probabilistic algorithms are known to be superior to standard (deterministic) algorithms when solving many discrete optimization problems (**Grama, p. 1999**). Only limited forms of hardware for

probabilistic algorithms are built into standard processors hardware. There are pseudo-random number generators in some computers. In the oracle-based approaches the probabilistic hardware is built-in on a very low level of logic circuits and it will be parallelized and distributed (for instance by selecting randomly variable values in SAT). Fourth, these logic computers will be optimized for search algorithms, which are common to many Constraint Satisfaction Problems, such as SAT. Standard computers are not optimized for search; they are designed for general-purpose numerical algorithms. Systems developed by us can be thus called the “hardware search computers”.

## **2.3 Towards combinatorial problem solving**

### **2.3.1 NP problems and CSP problems**

It is popularly known, even among non-specialists, that modern computers and all integrated circuits are built using a computer equipped with Computer-Aided-Design software. Humans are just not able to deal with the enormous complexity of such designs without the use of computers in all stages of simulating, designing, optimizing, verifying, validating and testing modern digital systems. This is why CAD tools are developed for all kind of digital and analog design and will remain to be perfected and made more powerful and general.

It is however less well known that several basic problems in Computer Aided Design of standard logic circuits, parts of standard software packages, are “NP-hard”. NP stands for non-polynomially, relating to the complexity that is non-polynomial with respect to the

size of the problem. If for instance the size of the problem is  $n$  and the complexity of the algorithm is  $2^n$ , the problem is non-polynomial. The problem of complexity  $4n^3+3n^2+8n+99$  would be polynomial. Many familiar problems known to be hard to solve (like the travelling salesman problem) are NP problems. The word NP-hard means that these problems are optimization problems that are counterparts of the “NP-complete” decision problems. NP-complete problems are decision problems that allow verifying that  $S$  is a solution to problem  $P$  in polynomial time, but these problems need an exponential time to find the solution. The solution to an NP-complete problem is of Yes/No type. An example of such problem is the Satisfiability Problem in which we have a Boolean formula and we have to answer a question: *“does there exists an assignment of values to Boolean variables from the formula such that this formula is satisfied?”* Many CAD and robotics problems can be thus reduced to SAT (satisfiability) and few other similar basic combinatorial search problems. These are called the Constraint Satisfaction Problems or CSP problems for short. CSP problems are those that we have a set variables and a set of constraints on values that can be assigned to these variables. We look for a mapping of these variables to their binary values such that all constraints are satisfied. For instance, a problem of coloring a graph with the given specific number of colors  $K$ , such that every two adjacent nodes (nodes connected by an edge) have different colors -- is a CSP problem. It is an NP-complete problem. The minimum number of such colors for a given graph  $G$  is called the chromatic number of graph  $G$ . If one is asked to find the coloring with the number of colors equal to an unknown chromatic number, then it will be the NP-hard optimization problem. If the numerical value of the chromatic number is known, the

coloring problem will be NP-complete. Oracles can be built for with both types of problems; NP-complete and NP-hard.

### **2.3.2 Building a circuit for a very simple oracle**

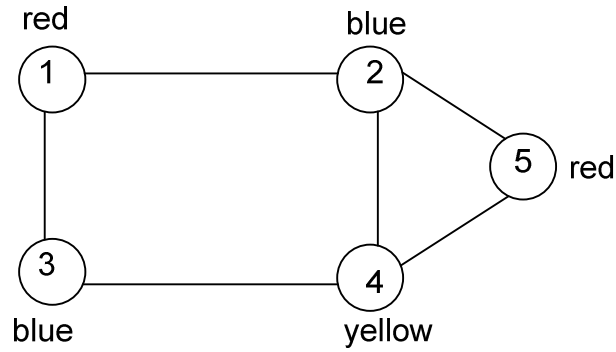
Building a SAT oracle is simply just to realize a circuit for the Boolean Function  $F$  from the SAT problem. This function in general can have arbitrary operators (logic gates) and arbitrary form (structure, number of logic levels). A commonly used form is the “Product of Sums of literals of variables form”. (A literal is a variable or a negated variable). An example of a SAT problem with Product of Sums (POS) formula of  $F$  is the following:

$F = (a' + b' + c)(a' + b' + c')(a + b')(b' + c)$ . Is there a set of literals that satisfies  $F$ ?

Oracle for this formula would have one AND gate with 4 inputs and four OR gates, each of OR gates would realize one of OR terms such as  $(a' + b' + c)$ ,  $(a' + b' + c')$ ,  $(a + b')$  and  $(b' + c)$ . Outputs of OR gates would be connected to inputs of the AND gate and the output of the AND gate would be  $F$ . We would generate all input variable combinations that go to OR gates in order to find a solution. We leave to the reader the task of drawing the circuit and simulating it with various input combinations of variables  $a$ ,  $b$  and  $c$ . The thesis will illustrate many important and practical problems, from robotics, CAD and other areas that can be reduced to SAT.

Now we will present one more example of building a simple oracle for a problem other than SAT, just to illustrate where we are heading for in my thesis.

The problem is this. We want to color nodes of the graph from Figure 1.1 below with as few colors as possible, so that any two nodes linked by an edge have different colors. Assuming that we have no any knowledge of the graph that is being colored other than that this graph has five nodes, we have to assume pessimistically that in the worst case the graph needs as many colors as there are nodes, which means five. Five numbers need at least 3 bits to encode them, it would be too bad to have this kind of a problem for a graph with 10,000,000 nodes which would be colorable with 2 colors, but let us make important point again that we have absolutely no information about the data (the graph properties that may help to evaluate its chromatic number) in this variant. We have only pure graph data, nothing else. However, if we would know some additional information, for instance that the graph is planar, we could solve this problem much more efficiently. For example, one can use the famous “Four Color Theorem” to know that only four colors are sufficient to color a planar graph, and thus encode the colors with only two bits. Thus, “knowledge is power” – any piece of general knowledge can help to solve the problem by designing a smarter search algorithm for it.



**Figure 2-1**Graph for coloring with five nodes.

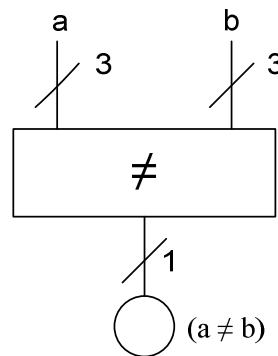
Figure 2-1 shows five nodes are colored with red, blue and yellow colors in such a way that every two neighbor nodes have different colors. The chromatic number of this graph is 3. Assuming no knowledge of the chromatic number of the graph the encoding requires three bits for each color and is shown as in Table from below. One particular example of encoding another simpler graph is shown in Figure 1.5.

Color	Bit
<i>red</i>	$a_1, a_2, a_3$
<i>blue</i>	$b_1, b_2, b_3$
<i>blue</i>	$c_1, c_2, c_3$
<i>yellow</i>	$d_1, d_2, d_3$
<i>red</i>	$e_1, e_2, e_3$

**Table 2.1**Assignment of bits to encoded colors of nodes for the graph from

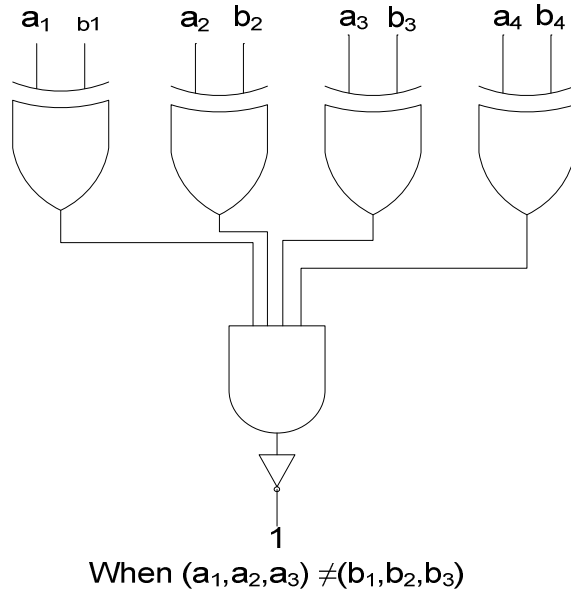
An inequality comparator circuit is used to compare two nodes of the graph, as shown in Figure 2.3 for nodes  $a$  and  $b$ . Such comparator is connected to encoding bits of any two nodes that are linked by an edge in the graph. If the colors of nodes  $a$  and  $b$  are the same then the output of the comparator will be zero. If the codes are different (which is good) then the output will be 1. Therefore, if oracle has such a comparator for every two nodes

of the graph linked by an edge and if a global AND gate of outputs of comparators is created, the output of this AND gate will be one for a good coloring and will be a zero for a bad coloring. Even if only for one pair of neighbor nodes of the graph the proper coloring is violated, the AND gate will produce a 0. See Figure 1.6 for the standard (combinational) oracle.



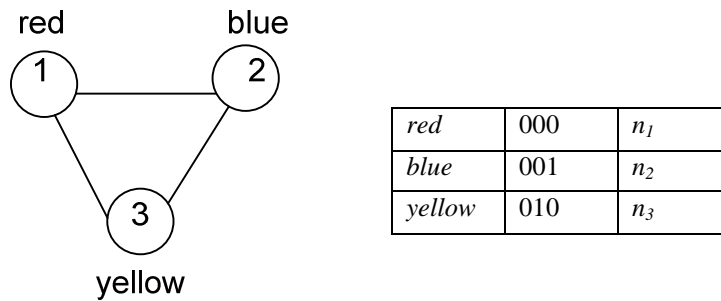
**Figure 2-2 The inequality comparator used in Map Coloring and Graph Coloring problems**

Figure 2-3 compares node *a* with node *b*. Observe that the size of this comparator depends significantly on the possible maximum number of colors. The comparator produces “1” at its output if the arguments *a* and *b* are different binary vectors of width 3. The binary signal  $(a \neq b)$  is also called a predicate.



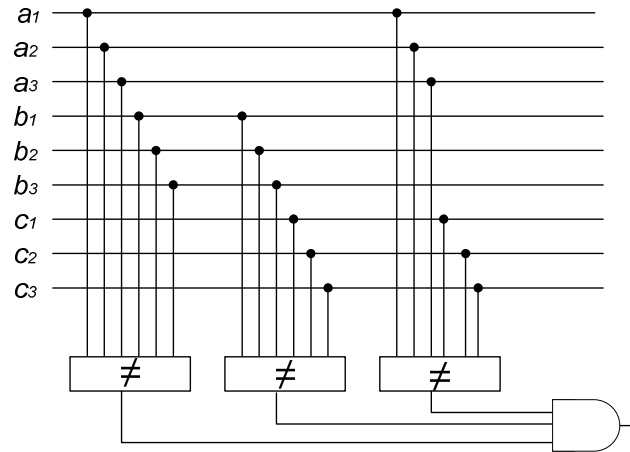
**Figure 2-3 The inequality comparator**

Figure 2-3 The inequality comparator from Figure 2-2 applied assuming five or more ( $\leq 8$ ) colors in the graph. This is a schematic for the inequality operator circuit, which is used in many oracles. The classical schematics of the comparator using EXOR, NOT and AND gates is shown in Figure 2-3



**Figure 2-4 Encoding of colors for graph coloring oracle**





**Figure 2-5 Principle of graph coloring applied to a simple graph**

Figure 2-5 shows a classical oracle. In this and previous graph coloring problem we are not checking for a minimal solution. We look here only for a coloring that satisfies the constraint of correct coloring. Thus every proper coloring that uses any 3 of 5 colors is good (this example is trivial, but we wanted to have a simple circuit for the explanation).

At this point my only goal was to explain the concept of an oracle for a constraints satisfaction problem different than SAT. Remember that many oracles can use various constraints similar to the constraint ( $A \neq B$ ) used here. In this example the oracle is very simple and can be designed by hand. In general, the oracle in the processor is a very complex circuit with many logic blocks; its design will require automation and the logic synthesis CAD should be used to build more complex oracles. Think about a graph coloring problem with 10,000 nodes. Think about some image processing system that uses many types of constraint operators other than the inequality constraint ( $A \neq B$ ) used above. Many such problems exist in real life. The need to design CAD tools for

hardware design of large and complex oracles is a one more reason for us to use the VELOCE emulator.

### **2.3.3 Building oracles for CAD, CSP and robotics**

In the previous section we showed that an oracle uses the global AND gate that combines partial results from partial predicates that verify (test) partial constraints of the entire constraint satisfaction problem. The intermediate logic blocks in the oracle correspond to some concepts. For instance, the blocks correspond to the concept that two adjacent (linked) nodes in a graph have different colors. In above case the partial constraints were inequality comparators on pairs of nodes corresponding to edges of the graph. This situation of checking many partial constraints can be found in most types of oracles. The blocks used in predicates are:

- the equality comparator,
- the order comparators,
- other comparators,
- the arithmetic blocks such as adder or multiplier,
- Logic gates.

Example of such partial constraint can be  $((A+10) < 2*(B-C))$  which includes an adder, a multiplier by 2, a subtractor and a “comparator of order” (less-than operator) blocks.

Concluding, the blocks in oracles that forward their outputs to the global AND gate can range from very simple (such as OR gates of literals) to very complex combinational processors (that check some mixed numeric-symbolic constraints for robot motion generation and include adders, comparators, arithmetic multipliers and complex function

such as trigonometric and exponential). The blocks can be also as complicated as the complete hardware simulators of some physical processes.

## 2.4 Architectures

Every combinatorial problem can be solved using at least four general architectures:

1. Architecture based on building an oracle and exercising this oracle using sequences of binary vectors. The problem-specifying data stay in place, inside the oracle. Easy examples of this approach are POS (Product of Sums) SAT and graph coloring, as presented above. The values of variables are created in the external exercising circuit (a generator) which creates input vectors to the oracle. The problem instance itself is described by the blocks inside the oracle and their connections. Thus the solution is found by collaborating circuits – the oracle and its exerciser. The oracle is usually a combinational circuit and the exerciser is a random number generator or a sequential circuit such as a counter. This is the approach taken by my thesis.
2. Architecture based on performing a tree search on data represented as the so-called “arrays of cubes”. The data in a form of “array of cubes” data stored in some memories and are processed by the “cube processors”. In this process the data flow from memory to memory. Often the iterative or recursive flow of data corresponds to a tree search. This tree search uses branching operators applied to functions represented by arrays of cubes. These branching operators can be

multiplying an array by some cube or a “cofactor operator” of substituting some variables for constants in all cubes from the array. These branching operators create two or more successor nodes of a parent node in a search tree. Each node of the tree is an array of cubes. At this point we can illustrate a simplified array of cubes as a vector (a set) of products of literals, which represents a SOP (Sum of Products) Boolean function. For instance a SOP,  $F = ab' + a'bc + abd'$  is represented by a set of cubes (products)  $\{ab', a'bc, abd'\}$ . A positive cofactor operator with respect to variable  $b$  on function  $F$ , denoted by  $F_b$ , creates a new array by substituting variable  $b$  to value 1. A negative cofactor operator with respect to variable  $b$  on function  $F$  substitutes variable  $b$  to value 0. A tree search algorithm is thus reduced to a data-flow architecture that uses the memories to store intermediate arrays (nodes) and the processors to execute cofactors and other operations on arrays. This data-flow architecture has certain similarities with image processing and Digital Signal Processing (DSP) Data Flow architectures, but we process arrays of cubes, and not 2-dimensional images or k-dimensional signals as in DSP architectures. This approach is not followed in my thesis but is a subject of work in VELOCE team at PSU.

3. Architecture based on performing algebraic operations on arrays of cubes that represent functions. For instance, Boolean operators can process Boolean functions. This is done for instance by Boolean multiplying the OR-terms in the POS SAT problem. Each OR-term is a cube, POS SAT is an array of cubes.

4. Architecture based on a “linear architecture” or “ring architecture”, in which solutions are pipelined from processors to processors. Each processor checks one constraint, for instance satisfaction of a single OR term in a POS formula for  $F$ . Pipelining is the fundamental concept of computing and a special case of systolic processing used in DSP. We will illustrate both concepts in the thesis. Pipelining moves data in one direction in a synchronized way, systolic processing may move data in parallel in more than one direction.

Thus the Cube Calculus methods, the oracle methods and other above approaches are interrelated. In this thesis we will analyze and compare only combinational oracles but we will give some examples of sequential oracles. The future goal of the research started in my thesis is to compare all four approaches, their variants and their mixtures, in order of creating a general methodology. But this would be not included in this thesis.

## **2.5 How to convert abstract problem to an architecture**

The question is: “how to proceed from a specification of a CSP problem to an oracle or other method for solving this problem?” We believe that in future certain new “high level languages” will be developed to specify input information to CAD tools. These innovative CAD tools will automatically design, adapt and reconfigure oracles. Thus the user will write programs in these languages, even without appreciating the complexity of the circuits created by these CAD tools. The future systems will be thus similar to the case of contemporary VHDL programming for ASICs or FPGAs. The nature of programming for these types of computers will then change. The presented thesis is only

the first attempt at creating such future software/hardware systems. I will show stages of transforming an initial description to a final system of low-level blocks, but I will not even attempt at automating my hand design procedures. Future systems will automatically convert the problem specification to VHDL or Verilog code and will automate emulation/acceleration. This thesis explains design stages as they are organized and executed by the designer that uses the existing CAD tools and not using future automatic tools. However, the developer of future tools can learn from my methodologies and design his future tools accordingly.

## **2.6 The main motivation and principle for architecture**

The reasons one may use non-standard processors in robotics are:

- (1) High speed of data processing,
- (2) low power,
- (3) small size of the chip,
- (4) other special requirements like integration with very fast sensors.

Nowadays, the most important reason is speed. In the research areas of Artificial Intelligence, Pattern Recognition and Robotics, the ability of a computer to solve high-dimensional combinatorial problems very quickly (in real time) is very important. Many problems in these areas can be reduced to few combinatorial search problems. These combinatorial problems, such as SAT, are very similar or essentially the same as in CAD. Another related area motivating my research in this thesis is “Evolutionary Hardware”, which is the research on realizing directly in hardware evolutionary algorithms - such as a

Genetic Algorithm. Genetic algorithm is one method of simulating an abstracted Darwinian Evolution in a man-made system.

Artificial Intelligence algorithms such as those based on CSP or search (depth first, breadth first) will be also more and more used in Intelligent Robotics and Evolutionary Hardware. New approaches to Computational Intelligence (CI) emphasize massive parallelism of applying deterministic or/and probabilistic rules combined with probabilistic evaluations of partial results. Although this thesis discusses only complete exercising of an oracle (exhaustive), in another methodology it is possible to use random number generators for this task, these algorithms have been not emulated by me, but this can be done as an easy extension of my methods.

Data Mining is another hot area of research based on finding some arbitrary “patterns” in huge data represented as data bases or text files. In contrast to statistical hypothesis testing, these patterns in Data Mining are to be found automatically and are not specified by humans only to be statistically confirmed by the system. Data mining methods also use SAT, Graph Coloring and other approaches discussed in this thesis. It is expected that Data Mining methods will be used in AI, CI, Intelligent Robotics and Evolvable Hardware. Perhaps, we will observe the synergism of research in all these approaches and their increased use in future generations of robots. Concluding on this point, the methods to solve problems presented here are very generic which means that they will find many applications in various areas; this gives me even higher motivation to work on these problems.

## **2.7 CSP architectures in practical robotics and digital CAD tools**

Theoretically, a robot can be controlled by an expensive supercomputer. But this is not very realistic with respect to high costs of supercomputers. Instead of controlling a robot with a supercomputer, we will use in this thesis a standard computer with accelerated specific operations or algorithms, just those that are needed for this robot, for instance the Robot Vision or Machine Learning operations or algorithms for a mobile robot navigating in a complex environment of full of obstacles rooms with a labyrinth of corridors among them.

Dr. Perkowski's plan for a future system at PSU is the Veloce accelerator connected to a local network to which several robots and intelligent sensor networks are connected. Now the team uses in the same way the CUDA parallel processing system based on Graphic Processing Unit (GPU). They solve CSP problems in a similar way by connecting my local network to a quantum computer at DWAVE, so my future robot vision/planning system should be a set of inter-networked powerful special purpose computer systems. There are thus many technologies to solve efficiently a CSP problem when the abstract algorithm for search can be formulated for it. This thesis explains only one of these technologies: emulation on VELOCE.

The system-level design, high-level design, logic synthesis, test and physical synthesis problems become even more difficult when the CAD system is being created to synthesize and minimize circuits for modern technologies which can include thousands of



processors in a “network on a chip”. The same will be true when accelerated algorithms to solve CSP problems will be developed for use in practical future robots. The thesis will characterize many of these robotics/vision and CAD problems.

We hope that my proposed accelerated computer will allow to efficiently solve at least some CSP problems for which the standard computer is inefficient. This thesis, among other new ideas, tries to answer the question – “what exactly are these problems?” The thesis will concentrate next on designing conceptual circuits, blocks, oracles and algorithms that will become useful to solve combinatorial problems of computational intelligence and robotics with the future introduction of practical accelerated computers.

## **2.8 Why logic puzzles are not only fun but are important.**

Everybody likes logic puzzles, but why? Perhaps the reason is that without much underlying theory to painstakingly learn, we can immediately use our brains for creative thinking. This is why logic puzzles are used to teach children logical thinking and to test intelligence of new employees in companies like Intel or Microsoft. On the other hand these puzzles started in 19<sup>th</sup> Century serious researches in many modern research areas; examples can be Euler Path or Hamilton Path problems that started as high society brain-teasers and conversation topics and ended up as fundamentals of graph theory and optimization theory. One can show many examples like these.

While most problems can be formulated in propositional logic, there are some interesting problems that require multiple-valued, modal or predicate logics. All of these problems

can be solved in software and in hardware. This thesis is the first one (to my knowledge) that attempts at creating unified methods to solve these problems in hardware. By this, the thesis teaches both about the logic itself, using logic to solve puzzles and combinatorial problems, and how to design specialized computers for them, using non-standard methods that cannot be found in textbooks on logic design or logic synthesis. The thesis emphasizes practically using the understanding of logic blocks to non-trivial and practical problems. Although we did not ultimately develop a complete methodology, we showed similarity of several such problems by practical Verilog and VHDL codes and reducing them to oracles.

## 2.9 Historical Remark

Many papers have been published by Dr. Perkowski and his teams at PSU since 1985 on the topics of building logic accelerators. They developed also logic accelerators based on Programmable Logic Devices (PLDs) and FPGAs. They used DEC PERLE accelerator board developed by Digital Equipment Corporation, as well as FPGA-based accelerators that used FPGAs. But technologically it was too early – they did not get sufficient speedups on practical problems. We now hope that a new technology like Veloce allows now to *practically* realize these ideas. If we will get a 10 times speedup on Veloce, my method would prove practical. If the speedup will be a small value like 2, then the ideas are still a theory only and we have to wait for better accelerators than Veloce. The previous speedup was 2.5 on some problems using the DEC PERLE board. They got similar speedup in past and it was not sufficient to have people interested in this approach (Marek Perkowski, 99).

## **2.10 Principles of problem-solving and design methodologies**

The new ideas presented in this thesis are based on three main principles:

### **2.10.1 Principle of reducing CSP problems to building Oracles for them**

Oracle is an arbitrary combinational circuit that answers yes/no to the proposed solution and checks solutions candidates in hardware with a high or even maximum parallelism. SAT circuit is thus the simplest possible oracle. In this approach solution candidate is a vector of values, often of binary values.

Principle of designing hardware controllers that exercise oracles with subsequent solution candidate vectors. These controller circuits have also massive parallelism and realize search algorithms in hardware. All principles used in software and parallel computing for SAT can be now reused in hardware (like learning from intermediate search results).

Principle of designing data-path architecture based on arrays of tuples and next hardware programming of controllers for this architecture. Previous machines at PSU operated on a restricted cube calculus (**Luis S. Kida, 92**) we use here standard controllers for accelerated data-path architecture on generalized tuples. Ideas of pipelining and systolic processing are used here.

## **2.11 Solving Constraint Satisfaction problems Using Oracles**

Algorithms to search the “unstructured data base” are very important and practical when there is no heuristic information to be used to solve the problem. “Unstructured data base” means that we want to find one item in the data base for which we can check some

function if this element is a solution. Do not be confused with name “data base”, we are rather having a data base created in hardware as some kind of function. The name “unstructured” means that there is no any structure in this “data base” that would allow us to search, or that would exclude some elements after finding other ones. This is a situation of no information at all. It is like having a “black box” with 99 black balls and one white ball. The box has a hole to insert a hand. If I want to find the white ball I can only keep reaching blindly to the box and removing balls one by one until I find the white ball. It will take me 100 attempts in the worst case and 50 attempts on the average. With no heuristic information nothing better can be invented and in many real life problems the software or hardware solving the problem is in this situation.

We are interested now in a set of problems in which the only thing that we can do is to test all elements one-by-one and check each if a given element is a solution. In worst case it will take to check all but one elements of the data base if we know that there is one solution to it. This is the “blind search problem” which can be speed-up only by fast mechanism of checking the items, including parallelism. Next, if some algorithmic or heuristic information about the problem is known, the search can be speed-up by some smart strategy (order) of selecting the candidate items. The strategy is reflected in the sequence of binary vectors (solution candidates) given to the oracle. The vectors with higher probability of being solutions are given first. These ideas cover two first principles listed above. These approaches are much interlinked and new methods can be created by combining the characteristic patterns to find solutions. These methods will be of my interest in the thesis.

### **2.11.1 Problems with no information in robotics and CAD**

As already alluded to, this thesis presents new approaches to solve several hard CSP problems with applications to a widely understood robotics and Computer Aided Design and particularly to logic synthesis. CAD algorithms are important for two reasons:

- Because they are used to construct hardware, a task important by itself,
- Because they can be used in AI/CI/Vision/Robotics reconfigurable/programmable systems, as part of an approach where the hardware algorithm is first constructed from high level specification to be next used in hardware to gain speed.

My approaches will be based on oracles and hardware search for standardized data-path architectures. Solving these categories of problems will be one of the most important applications of CSP-accelerated computers. My intention is to speed-up all NP problems. There are thousands of such problems, many of them of high practical use, especially in CAD of classical digital circuits, pattern recognition, robot vision, planning, scheduling and other areas of high practical importance.

We assume here a practical Veloce emulator as a model of the hypothetical, yet to be build CSP-accelerated computer. We analyze what would be its use in the area of solving CSP problems. We speculate on a real speedup based on my simulations. We will teach how Veloce can be used for these ideas and thus students knowledgeable in Veloce and these ideas will be able in future, when VLSI or other technology will allow it, to build

the accelerators proposed here to give a true advantage on some class of difficult and important problems.

## **2.12 Solving problems by reducing them to basic combinatorial search problems**

### **2.12.1 Main problem representations and data structures in CSP problems**

Many generic combinatorial problems are known in classical logic synthesis such as satisfiability, graph coloring, binate covering, spectral transforms and others (all these problems will be explained in full detail in my thesis). These problems use various parallel architectures such as inverse trees of processors, hypercubes of processors and parallel processors based on butterfly structures (see next chapters). Many of these problems are known as Constraint Satisfaction Problems where some solution must be found that satisfies a set of constraints, where constraints are equalities, orders and inequalities on natural numbers. The elementary data used in constraints can be binary vectors, products of variables, numbers, arbitrary symbols, or other data encoded as binary strings. In many problems the solution should additionally optimize certain cost function. The cost function may represent physical concepts such as energy or entropy (a measure of order in data). It may also reflect financial cost of creating the circuit. All kinds of cost functions can be calculated in my algorithms.

We will demonstrate in this thesis that these and other problems still remain a fundament for efficiently solving problems in robotics and CAD. How then to solve these problems

using a CSP-accelerated computer? How to create new motions of a robot based on spectral decompositions? How to find some shapes in images based on transforms? How to plan robot behaviors? How to learn concepts from positive and negative examples of concepts?

### **2.12.2 Universal and problem-specific methods or mixing them?**

Can we solve, in principle, all these various problems with few universal methods? The thesis promotes the belief that all these problems can be reduced to some finite set of problems for which a programmable logic accelerator will be build. This may be an FPGA-based reconfigurable hardware accelerator (FPGA stands for Field Programmable Gate Array) or an accelerator using some other technology.

To answer this question one has first to analyze what are the standard modern (binary logic) FPGAs? This is a relatively new technology (invented in 1986) in which the user can program not only the memory as in a standard computer, but can program also gates, blocks and their connections using special hardware design languages (such as Verilog or VHDL) and synthesis (CAD) software. This way the digital designer can practically “build his own computer” for any given task that can be programmed by him. FPGAs have truly revolutionized digital design since 1986 and are used in many practical products; from simple controllers to supercomputers. We introduce in this thesis a model of a FPGA-like, CSP-accelerated or logic-accelerated, computer. It is a company secret what is an FPGA or processor inside the Veloce hardware. At this point we have no

interest in this question. We are also not interested at this point in any particular FPGA devices to which my digital systems will be mapped after emulating them with VELOCE.

### **2.12.3 Properties of my problem-solving model**

Concluding, my model of an accelerated computer proposed in this thesis is a multi-purpose, parallel, programmable, reconfigurable, accelerator connected to a standard computer. My model is *multi-purpose* because it is not a universal computer to speed up arbitrary problem, but is specialized for only some classes of problems. These classes include however many practically important problems.

My model is *parallel* because we have available in my system not just a single computer but a collection of computers that allow decomposing large problems to structures of simple problems that share information between them. My model is *programmable* in an analogous way to the way that FPGAs are programmable in modern VLSI technology.

My model is *reconfigurable* in the same way as FPGA systems are reconfigurable in modern system design (it is even not known if Mentor uses an FPGA or a special VLSI chip in Veloce). This means that the top-level structure of the system can be reconfigured dynamically to become another system. Thus a vision processor can be modified to a DSP processor or a hardware sorter. When the basic structure of a reprogrammable hardware is created, it can be reprogrammed very quickly to an arbitrary given



application. The existing technology already allows for this re-configurability, but this technology is not yet scalable.

We call my model an *accelerator* -- to emphasize that only some problems are accelerated (a parallel computer would speed-up a wider class of problems). The accelerated problems are logic optimization problems and CSP problems.

## **2.13 Combinatorial Problems in synthesis and optimization of circuits**

### **2.13.1 Future CAD tools**

We hope that in future, the CSP-accelerated computers will be used to solve CSP problems in my area of interest. Special high level specification languages will be created. They will be inputs to high-level and system-level CAD tools. It will be the same way as the standard computers are used now to synthesize classical circuits from VHDL specifications (VHDL is Very High Level Design automation Language to specify hardware for VLSI and FPGAs). To aid in inventing the new types of hardware algorithms that will be used in such systems, a new generalized and unified approach is created and investigated in this thesis.

This new approach should be of interest to the robotics/Machine Learning community as well as to the CAD/logic synthesis community, because of its analogies and extensions to that of the classical Boolean logic (based on AND, OR and NOT gates), classical Reed-

Muller Logic (based on AND and EXOR gates), and other areas in which CSP problems are formulated.

### **2.13.2 Logic circuits other than AND/OR/NOT circuits**

Circuits based on AND, NOT and OR are well-known, so let us pay more attention here to the AND/EXOR logic that builds circuits with gates AND, NOT, EXOR and constant 1. In this thesis it will be shown how AND/EXOR forms and circuits are synthesized using CSP algorithms similar to those for AND/OR logic. We aim also that my synthesis methods for circuits will be good for practical data sizes that may appear in designing oracles to solve practical CSP problems. The presented synthesis and decision approaches will be good for AND/OR (classical), EXOR/AND types of logic. We selected these methods that are truly general. All listed here names and concepts will be illustrated in next chapters of this thesis.

## **2.14 Types of oracles to solve combinational problem**

There are mainly two types of oracles.

### **2.14.1 Combinational oracle**

These kinds of oracles use simple combinational logic blocks such as adder, subtractor, equality, order or equality comparators, count ones, and basic Boolean logic gates. The oracle has no memory elements such as RAM cells, latches or flip-flops in them. It is a purely combinational circuit (with no memory).

### 2.14.2 Sequential oracle

These kinds of oracles use for their realization a number of sequential blocks like: counters, shift registers, memories, registers, sequential adders and multipliers, counters circuits. Often they use register pipelining to some degree. Some of these oracles can be in theory represented as Finite State Machines, but in reality they are built from many state machines, memories and combinational components.

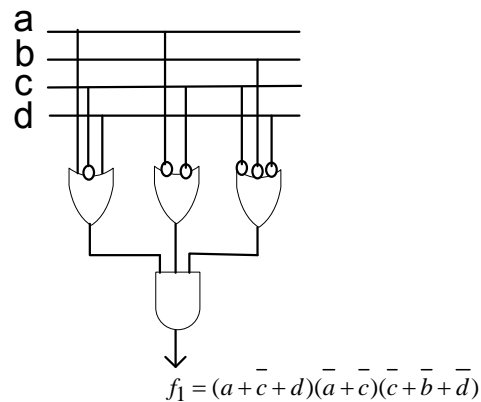
### 2.14.3 Combinational oracles

Combinational oracles are again sub-categorized into 3 types

#### 2.14.3.1 Simple Oracles

These oracles have inputs corresponding to variables and only one output corresponding to the decision value. The only information given by oracle to the controller is if the solution was found. The controller performs therefore a blind (unstructured) search.

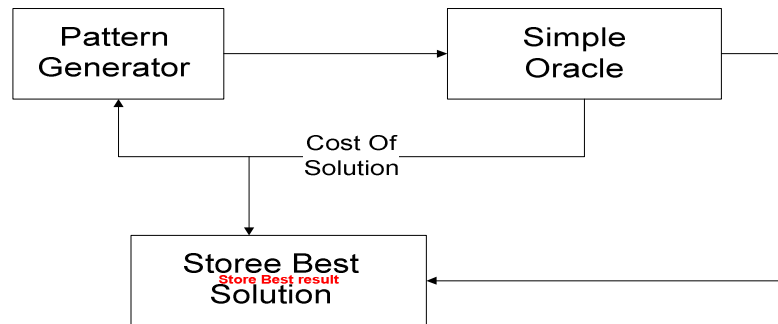
Example of such oracle is the POS SAT oracle in Figure 2-6



**Figure 2-6 Example of standard combinational oracle applied to a SAT problem**

### 2.14.3.2 Oracle with cost function as a feedback

These kinds of oracles include cost function value as a feedback to pattern generator.

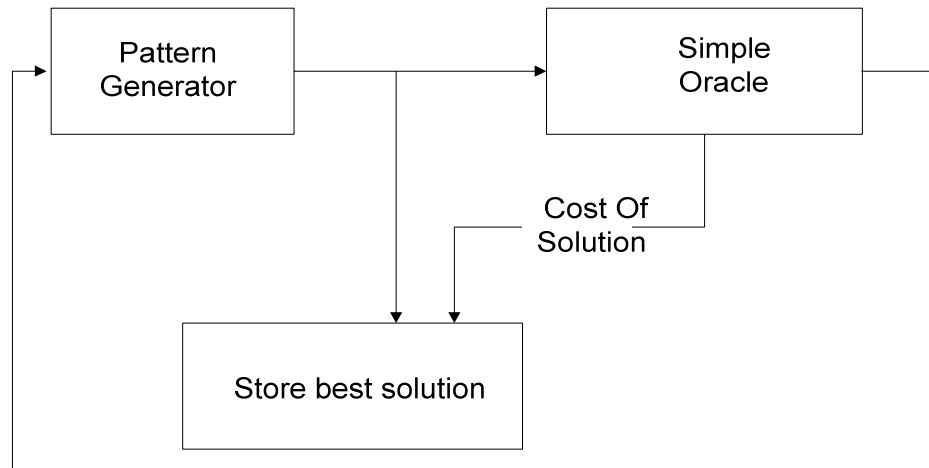


**Figure 2-7 Oracle with cost function as a feedback**

Figure 2-7 is a Pattern Generator is a sequential circuit that generates sequentially solution candidates that are checked by a combinational Simple Oracle. Cost of solutions is calculated in the oracle and compared with the cost of the best solution found so far. The best solution is stored for future comparisons. New cost found can affect the operation of the Pattern Generator by suppressing the generation of superfluous or worse solution candidates.

The feedback will tell the user how close he is to satisfy all constraints of the problem. Or it can suggest what the user should do next; e.g. Graph coloring oracle i.e. finding minimum number of colors to color the given graph.

### 2.14.3.3 Oracle with direct feedback



**Figure 2-8 Oracle with direct feedback**

These kinds of oracles will also have feedback but it will be direct. This means that the oracle itself will suggest the user some hint to find the next solution candidate or even a part of this solution. For instance it restricts the set of variable values for the given SAT formula. Or suggest directly some new assignments that are implied by previous solutions or their subsets.

### 2.14.4 Sequential oracles

Sequential oracles are again of three types:

#### 2.14.4.1 Controller type

These are architectures composed of simple data path and controller state machine. There is not much, if any parallelization. This is a standard architecture.

#### **2.14.4.2 Pipeline type**

These kinds of Oracles implement part of the oracles as a pipelined architecture. This will speed up the oracle but increase the complexity of the oracle.

#### **2.14.4.3 Systolic type**

The oracle has one or more systolic structures which transmit data in more than one dimension.

### **2.15 Mechanisms of exercising oracles**

There are number of ways by which given oracle can be exercised.

#### **2.15.1 Random number generation**

Generating random number pattern to test the oracle. This can be done with software or hardware such as LFSR.

#### **2.15.2 Simulated annealing**

Simulated annealing (SA) is a generic probabilistic model for creating test pattern locating a good approximation to the global optimum of a given function in a large search space. This need conscience description of configuration of system, random number generator, quantitative objective function that contains the constraints that have to meet.

#### **2.15.3 Pseudo Boolean programming**

The Pseudo-Boolean (0-1 integer programming) problem is a linear integer programming problem where all variables are restricted to take values of either 0 or 1.

#### **2.15.4 Integer programming**

In an integer programming test pattern creation, some or all of the variables are restricted to be integers.

#### **2.15.5 Exhaustive search based on counting**

In this case of pattern generation whole search space is scanned. Obviously it takes longer time.

#### **2.15.6 Intelligent search methods**

It is used when search space is huge, so we need artificial intelligent techniques specially dedicated to find near optimum solution.

#### **2.15.7 Genetic algorithms**

A **genetic algorithm (GA)** is a search technique used to find exact or approximate solutions to optimization and search problems. eg. Particle Swarm Optimization (PSO), Ant Algorithms, Bacteria Foraging search, Cultural Algorithms search, etc.

#### **2.15.8 Dynamic programming**

In this complex problems are solved by breaking them down into simpler steps. This takes very small time as compared to native method.

### **2.16 Example of a sequential oracle**

As an example of sequential oracle problem is SEND + MORE=MONEY which is explained more elaborately in the later chapters. Here only the basic idea is discussed.

	C4	C3	C2	C1	C0
		S	E	N	D
+		M	O	R	E
	M	O	N	E	Y

From the above puzzle, we can derive equations for each letter from the output. If we consider the ones place in the puzzle, the output obtained is Y and the inputs are D and E which are added. The carry obtained from adding up the numbers will be considered as C1. So, we will get an equation which is;

$$D + E = 10C1 + Y$$

In the same way the remaining output values are obtained as:

$$C1 + N + R = 10C2 + E$$

$$C2 + E + O = 10C3 + N$$

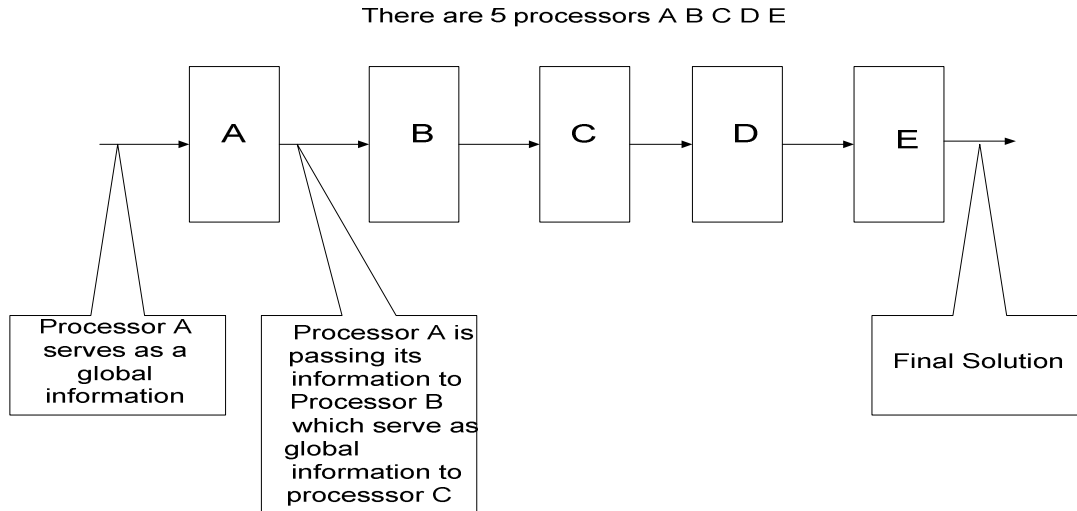
$$C3 + S + M = 10C4 + O$$

$$C4 = M$$

The equation is typically a basic operation of arithmetic, such as addition, multiplication.

This oracle can be implemented in a pipeline fashion, as shown in Figure 2-9



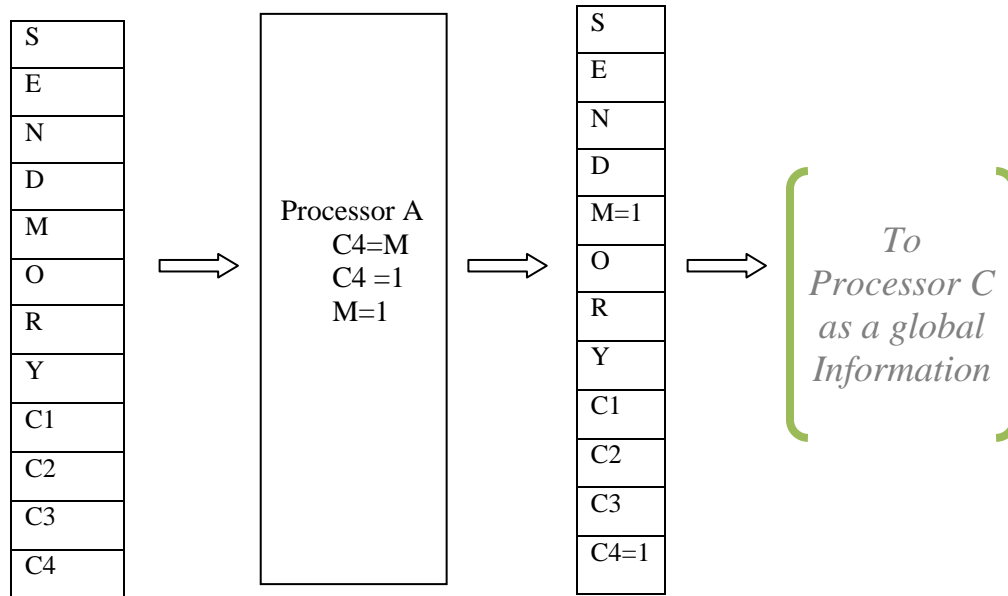


**Figure 2-9 Sequential Oracle implementation**

As shown in Figure 2-9, each processor has some local information and some global information from previous processor. Processor A serves as global information provide to processors B, C, D, E, that serve as filters. The solution candidate that passed all filters is a valid solution. The solutions are moved in pipelined way from left to right. The respective processor will take action based on its local and global information it has got from previous processor, and it will pass all information to next processor.

We need five processors to solve 5 equations. Each processor is responsible for some set of partial constraints. In this case, constraints for the equation.

## Operation of the 1<sup>st</sup> Processor

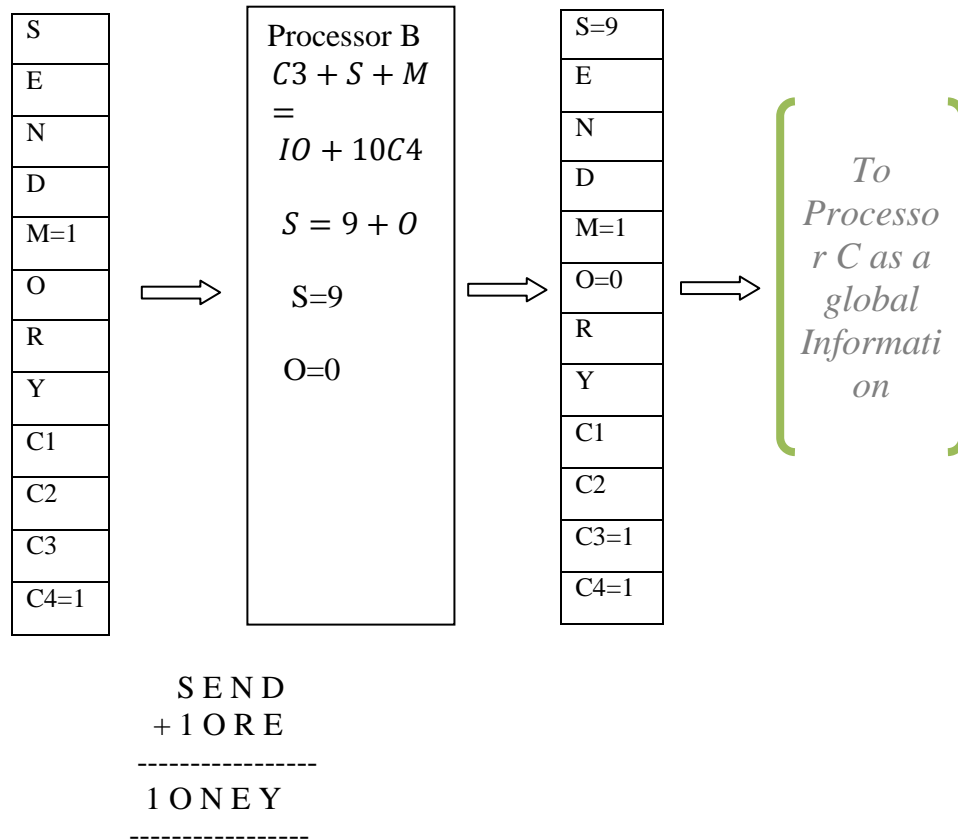


$$\begin{array}{r}
 \begin{array}{cccccc}
 & C4 & C3 & C2 & C1 & C0 \\
 \hline
 & S & E & N & D \\
 + & M & O & R & E \\
 \hline
 M & O & N & E & Y
 \end{array}
 \end{array}$$

**Figure 2.10. Explanation of stages of solving the SEND+MORE=MONEY puzzle in a pipelined oracle. The first processor.**

From the above puzzle, the M value is obtained by adding the digits at the thousands place. According to my assumption, when two 4- bit numbers are added the result will be either 0 or 1. Here there is no possibility to have 0 as the output. So, the value of M will be 1. And hence C4 will also be 1. So  $C4 = M = 1$  is local information for processor A.

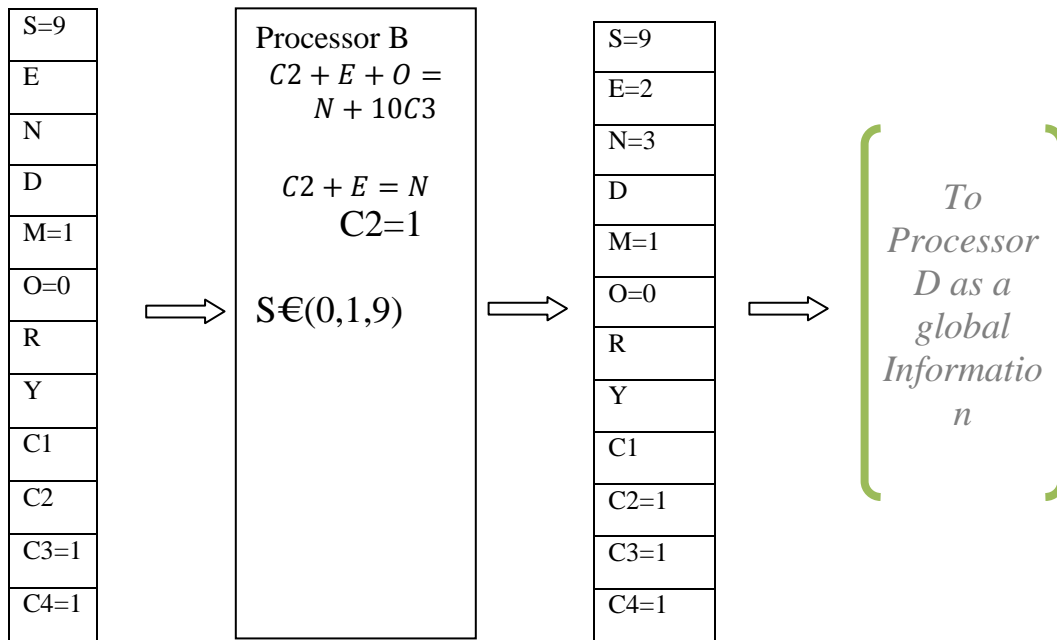
## Operation of the 2<sup>nd</sup> Processor



**Figure 2.11.** Explanation of stages of solving the *SEND+MORE=MONEY* puzzle in a pipelined oracle. The second processor.

The only possible value for S will be 9 since we are getting the carry as 1 and the one of the addend is 1. So, the value of S will automatically be considered as 9. In the same way the values of O will become 0 as 9 and 1 are added to form sum as 0 and the carry as 1. At this point the values at the thousands place are calculated.

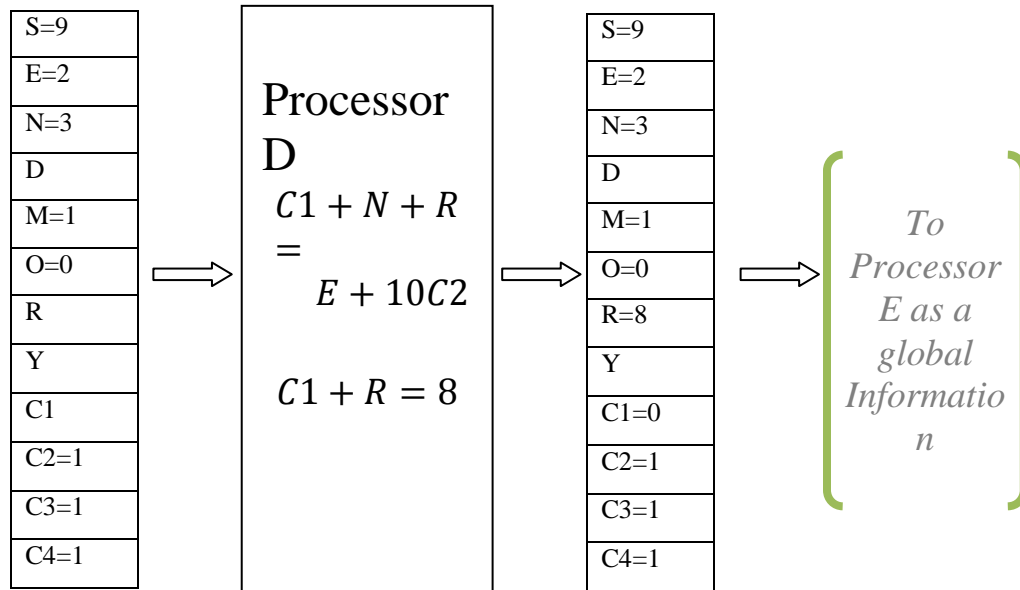
### Operation of the 3<sup>rd</sup> Processor



**Figure 2.12.** Explanation of stages of solving the SEND+MORE=MONEY puzzle in a pipelined oracle. The third processor.

Value of C2 can't be zero because then E=N which is not allowed so C2 has to be 1. So let's assume E=2 this will lead N=3.

### Operation of the 4<sup>th</sup> Processor



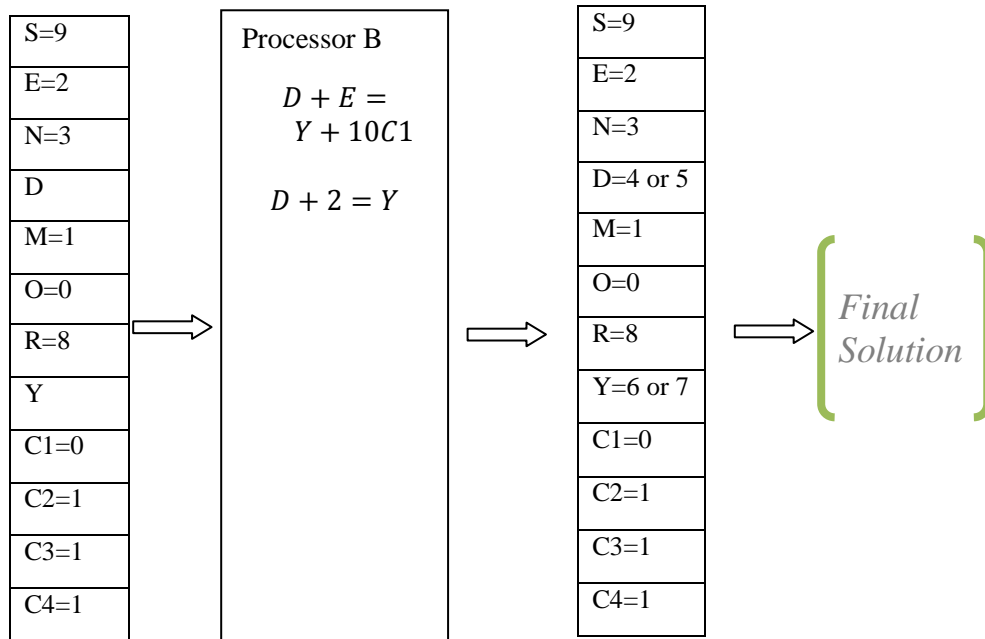
**Figure 2.13.** Explanation of stages of solving the *SEND+MORE=MONEY* puzzle in a pipelined oracle. The fourth processor.

Setting  $N=3$ ,  $E=2$ ,  $C2=1$

$C1 + N + R = E + 10C2$  will simplify to  $C1 + R = 8$ .

Let's assume that  $C1=0$ . Hence  $R=8$  and  $C1=0$ .

### Operation of the 5<sup>th</sup> Processor



**Figure 2.14. Explanation of stages of solving the SEND+MORE=MONEY puzzle in a pipelined oracle. The fifth processor.**

Setting  $E=9$  and  $C1=0$ . We have  $D+2 = Y$ . Now  $D$  can be 4 or 5 as other digits are used.

So  $Y$  Will be 6 or 7 respectively. This gives the final solution.

## **2.17 New General-Purpose Search Approaches for classes of combinatorial problems.**

### **2.17.1 How to exercise oracles?**

So far, we mostly concentrated on illustration of Oracles but design of generators to exercise oracles is of equal importance. In this thesis we will also present the development of general-purpose search meta-algorithms to be used in solving highly complex combinatorial problems, especially the Constraint Satisfaction Problems. It is a general constructive methodology, a kind of Super-Pattern Generator with many variants and strategies. It is applicable for CSP problems that exist in logic synthesis, logic minimization and similar problems. It is developed to create (binary) vectors that exercise oracles to solve design and optimization problems based on CSP model. My approach is very general. It includes classical AI algorithms: depth-first search, breadth-first search, and many other search methods as its - parameter-selected - special cases.

The method is based on the general concept of search in certain space of solutions and candidates for solutions (called also partial solutions in literature). The search can be done in serial controller or in a parallel processor. My hardware search approach is for both serial or parallel variants. The classical serial search is of course only a special case of a parallel search. Both the serial and parallel accelerated search variants presented here can be improved in future by other researchers. Why we believe this point? Looking to history of classical software search on standard computers one can observe that it was possible to find new better search methods long time after the concept of search itself and

the basic search algorithms were invented. For instance, the Iterative Deepening Search that we use in this thesis was invented in the area of AI many years after the classical depth-first search and the A\* search algorithms were originally created. These algorithms were meant for software and not for hardware and in general not much has been published about hardware realization of search. So there exists a potential to create new efficient hardware search algorithms for oracles.

### **2.17.2 Types of exercising mechanism for CSP algorithms.**

There are two main types of combinatorial algorithms: oracle-based CSP algorithms and data-flow algorithms. Most of my search ideas in this thesis are related to the combinational oracle model. Oracle function is a generalization of SAT function and thus we can borrow much from all kinds of SAT algorithms. There is a big library of papers on software and hardware SAT solvers. We can use much of these. However, we not only use the SAT solvers as they are, but we wrap them around in a more general search systems of parallel reconfigurable processors. My generalized systems use several basic concepts: parallelism of programming, parallelism of execution, heuristics, and reprogramming (as in FPGAs). They call Oracle Exercising Algorithm (OEA) for sub-problems, possibly with oracles that are adapted and modified. We call this the “dynamic approach to problem-solving based on Oracle Algorithm”. By solving some class of problems using Oracle Algorithm, we can learn certain parameters to improve the speedup of the next calls to the “Oracle Processor”. For instance, when one knows the chromatic number of a graph, the optimal coloring of this graph can be found more



efficiently by reducing the size of the oracle. Reducing the oracle's size leads to the reduction of the solution time of the Oracle Algorithm, as will be discussed. Any additional knowledge available to the system designer should be thus used in (parallel) oracle computing to improve the search efficiency. Observe that this is exactly the same problem-solving philosophy as the one used in standard contemporary parallel search (software) algorithms.

The thesis will present several applications of this meta-learning algorithm in detail: graph coloring, satisfiability, maximum cliques, DNF logic minimization (Sum-of-Products - SOP), and others.

### **2.17.3 Exercising for complex data structure algorithms**

Although most often problems are solved in propositional binary AND/OR/NOT logic, Cube Calculus and oracles allow us to use various data structures/representations such as representations based on other algebras such as: AND/EXOR, modal, and Galois Field algebras. One application of these data is the minimization of incompletely specified data with FPRM data structures, with applications in CAD and Machine Learning. Fixed polarity Reed-Muller forms (FPRM) are extensions of PPRM (Positive Polarity) Reed-Muller forms. In FPRM every variable has the same polarity, it means it is negated or not negated, but it must be negated or not negated consistently in the entire expression. Thus if variable  $d$  is not negated in product term  $def$ , it is also non-negated in every other product term in the EXOR. For instance,  $F = ab' \oplus acd \oplus b'd$  is an example of FPRM with variables  $a$ ,  $c$  and  $d$  in positive polarity and variable  $b$  in negative polarity.

Another application of my approach is the minimization of the GRM (Generalized Reed-Muller) forms (mixed polarities of variables). In GRM every variable can be both positive and negative, but for every subset of variables there is only one product in the EXOR. For instance, if product  $abc$  is included, product  $a'bc$  cannot be included as these terms both correspond to the same set of variables  $\{a,b,c\}$ . Expression  $ab \oplus bc \oplus a'b'c$  is an example of GRM and  $abc \oplus a'b'c'$  is an example of ESOP which is not a GRM. The FPRMs and GRMs are two most well-known types of the AND/EXOR expressions (Sasao93e). The algorithms for these forms are traditionally exhaustive search algorithms so they are good candidates for my methods. These are just the simplest of algebraic representation that can be used in my hardware approaches. Particularly spectacular are applications in image processing, digital signal processing and encryption. We hope that the users will be encouraged to design their own oracles and Cube Calculus machines for other algebras, for instance in cryptography, error detection and correction and DSP. We have no space in the thesis for them, but we include a list of possible problems with references to them.

## 2.18 Organization of the thesis

As already presented, this thesis introduces ideas in logic design of oracles, circuit structures and respective synthesis algorithms and also ideas in algorithm design and computer architecture. In a sense, “everything relates to everything” in this thesis; hardware, software, acceleration, emulation, simulation, verification, logic, CAD algorithms and general CSP. This multi-aspect core of the thesis makes its presentation

difficult. Therefore we organizationally separate the thesis to parts that are relatively less interconnected. We need also some small text repetitions to simplify the reading of the thesis.

The areas of logic design and algorithm design are respectively isolated, and they are linked by the fact that to build a practical oracle one has to be able to optimize it from logic gates. We will link these two ideas more in next chapters when all background will be already introduced.

## **2.19 New concepts of CSP algorithms for particular structures**

### **2.19.1 Use of heuristics versus universality of algorithms.**

It is desired that good partial heuristic methods be developed for any given specific CSP algorithm, or rather, for a class of such algorithms. This is thus the second task to be achieved in my thesis. For instance, let us assume that we use the Oracle algorithm to solve the problem of “graph coloring” where every two adjacent nodes of a non-directed graph should get different colors. We may constraint the number of colors and ask if there exists a coloring with  $k$  colors. Additionally, we may be asked to solve the graph coloring problem with the minimal number of colors. Being able to solve this particular problem efficiently, most of the important optimization and decision problems that appear in CAD algorithms could be solved. Graph coloring is a prototype constraints satisfaction problem. Other set of typical CSP problems is the class of path-finding problems in a graph, in which a problem is given in an explicit way as a data structure. The shortest path and the travelling salesman problems belong to this class. In another set

of problems the graph is given in an implicit, not explicit way, by some kind of rules that are expanded to create nodes and edges incrementally, thus saving space and time of processing the graph.

We will characterize basic typical CSP problems of various types, so the reader will be able to reduce his/her problem to one of them. This classification of problems and reducing them to my three CSP principles should be used in teaching design/verification classes.

### **2.19.2 Universal algorithms**

Universal algorithm is an algorithm that can solve a set of problems (in theory, all problems) rather than a single problem. Often, many problems are polynomially reducible to a standard problem such as SAT, and next solved using a SAT solver system. Instead of writing software for each problem separately, we just write software to reduce my problem to one of “universal problems” and we use “universal solvers”.

The mentioned above universality of optimization algorithms and the data structures used in them is the general promise of several areas of research:

**SAT solvers** – software, hardware or reconfigurable FPGA systems to solve the satisfiability problem,

**universal algorithms** to solve in software or hardware classes of problems such as integer programming, linear programming or CSP,

**resolution-based programming languages** such as Prolog; they use one automatic-theorem-proving mechanism, called *resolution* to solve problems reduced to predicate calculus,

**hardware accelerators** – hardware systems, for instance FPGA-based that solve some classes of problems.

Some of these general problem-solving approaches, like the universal (software) SAT solvers, are extensively used in modern research and CAD industry. Similarly in my case, the universal solvers will be used in the research towards the promise of future “CSP accelerators” that this thesis attempts to make a ground for.

Concluding on the “universality versus special domain” issue, this thesis develops hardware algorithms that are:

1. Universal,
2. Allow to create variants of the main algorithm,
3. Allow to incorporate problem-specific knowledge into the algorithm.

## **2.20 The role of additional knowledge and heuristics in creating algorithms**

It is well known that in classical algorithms, any additional knowledge about the problem, like for instance the upper bound to the chromatic number of the graph being colored, can help to create a more efficient algorithm. We will show that the same is true for hardware algorithms of certain type, including algorithms for various variants of graph coloring problems. Among the several possible approaches to create such a meta-

algorithm, the biologically motivated computations, such as evolutionary algorithms (Koza, 1992), were viewed as attractive because of their generality and flexibility. Thus, in the long research and development process to create hardware algorithms, the PSU group applied the biologically inspired, evolutionary processes of Genetic Algorithms and Genetic Programming. These algorithms were also sometimes combined with the, humanly designed, heuristic and search methods. This was done for instance in (Giesecke, 2006) and (Dill, 1998). The authors combined evolutionary, quantum-inspired and traditional Artificial Intelligence and integrate them in hardware structures and controls. Let us observe that Genetic Algorithm uses mutation and crossover to create a chromosome that is next evaluated by the fitness function. This can be realized in hardware oracle architecture in which the chromosome is the vector of variable values given to the oracle, the exercising mechanism creates these vectors using genetic and other operators and the oracle calculates the value of the fitness function. This is in general a generalization of the classical oracle concept as the oracle returns fitness function and not only binary data. My methods are exhaustive so far, but they can be modified to become Genetic Algorithm like hardware algorithms. These ideas have been not realized in my examples yet.

### **2.20.1 Machine Learning methods.**

An oracle can be created to find the best spectral transform in a family of such transforms (Li06). This paper can be explained as an application of an exhaustive search speed-up to create the best Fixed Polarity Reed-Muller Form (FPRM). For  $n$  variables there are  $2^n$  FPRM transforms in the family. The best transform is one with the minimal number of

non-zero spectral coefficients (product terms). This corresponds to learning, for a set of examples, a circuit which has the minimum complexity. This circuit is found for a given truth table of positive and negative examples. These “examples” (using the terminology of machine learning) are called “minterms” in the area of logic synthesis. Such FPRM form is a type of structured logic expression that should be as simple as possible and that should separate the truth from the false. Thus, for all minterm examples categorized as “false” (negative examples, zero-minterms) the value of the expression is false and for all minterm examples categorized as “true” (positive examples or “ones” in the truth table) the value of the expression is true. The oracle can be connected to the state-machine based hardware for exercising the oracle by giving binary vectors on its inputs.

Careful analysis of the approach from (Li06), reveals in addition that this idea can be applied with little modification to an incompletely specified function, thus becoming applicable in Data Mining and Machine Learning (Koza, Genetic Programming II: Automatic Discovery of Reusable Programs, 1994), (J. R. Koza, 1999), (K.M. Dill, 2001)

Let us explain the above idea on a simple example of inducing formula from a set of examples. A set of positive and negative examples is collected by observing successes or failures of various pairs of humans, related to their character, social position, physical properties, etc. Next an ideal life partner is induced from this set – it may be described by an expression “*(Beautiful and Smart)  $\oplus$  Rich*”. This formula means that the candidate person has to be either “*beautiful and smart*” or rich but not all positive properties at

once: beautiful smart and rich (somebody who is beautiful, smart and rich may drop his partner soon, which was reflected in the particular set of specific examples given to the learning tool). Denoting  $B = \textit{Beautiful}$ ,  $S = \textit{Smart}$ , and  $R = \textit{Rich}$ , the learned (Fixed Polarity) Reed-Muller expression is  $BS \oplus R$ . This is a Positive Polarity expansion formula (i.e. all variables are not negated). The equation  $BS \oplus R$  is realized with a single a Davio gate. Thus, a logic formula that generalizes the results from all examples is returned as the result of Machine Learning. Davio gates and multiplexers belong to set of useful gates in logic synthesis and Machine Learning that we will be using.

My conclusion of this generalization is very powerful – *every problem that can be solved using a “pure” Genetic Algorithm or a Constraint Satisfaction algorithm can be solved using a hardware search algorithm based on the HASPS.*

This observation applies to all algorithms from binary logic synthesis, Constraint Satisfaction based problem-solving, predicate calculus, integer programming and other combinatorial algorithms that appeared and continue to appear in literature.

My approach applies also to the data that are incomplete. Incomplete data include the so-called unknowns, which are “not shown” examples, i.e. combinations of input variable values that are not used as positive or negative examples in teaching). Thus we can construct or “build” a logical expression (in example above - the FPRM expression) to satisfy the behavioral criteria. As a new method of logic synthesis or Machine Learning, the hardware search, as this one, offers a unique approach to automated logic design or



"evolvable hardware". We can speculate that future evolvable hardware will be a hardware accelerator equipped with different "universal" components. These components will be in theory "universal" (such as SAT is universal in classical CSP) but practically they will be only "wide range application" components. These components will be developed for particular applications such as: algorithm for SAT, algorithm for all maximum cliques, SOP minimization, FPRM minimization, ESOP synthesis, Walsh and other spectral algorithms for image matching or spectral coefficients minimization for structured forms of learning, as well as other learning, problem-solving, pattern recognition and vision methods. Similarly as Fourier Transform represents a function as a sum of orthogonal cosines of various frequencies, other spectral transforms such as Walsh Transform use other orthogonal (base) functions, such as square functions. In PPRM transform of functions with 3 variables the base functions are: 1, a, b, c, ab, ac, bc, abc.

The "logic circuit" (equivalently, the "solution specification") is designed by evolutionary means and the process is entirely "hands-off" for the user. This is however not a biological "Darwinian evolution", but the evolution of the superposed states in a quantum-inspired computer. This computer is intentionally designed by humans to model evolution in order to solve certain class of problems. Like in FPGAs, each class of problems requires new computer hardware, in this case a new oracle. The beauty of the proposed here methods is that problems can be solved without explicit computer programming. We just use the general quantum-inspired search algorithm itself. While

the general search mechanism is universal, each specific problem is described by the user as a specific oracle.

Observe that in future problem-solving systems, the parameterized descriptions of many problems will be created by designers. This will be similar to how it is done now in the areas of “intellectual property design” and “circuit generators and hardware compilers”, in which the sophisticated blocks are designed in hardware languages such as Verilog and VHDL. These blocks are next reused and integrated in designs. Because the oracles for classes of problems are similar, in the further future certain “smart software generators” will be written to create parameterized descriptions, similarly as it is done now in software generators of VHDL or Verilog programs that are written in Matlab language or the languages of Artificial Intelligence (AI). There exists therefore a clear path from the FPGA fast prototyping methods that are at the forefront of CAD tools in year 2011 to the future CSP tools for accelerated CSP computers.

Further, let us observe that in theory, a single technique is applicable to solve all CSP problems (because all problems such as graph coloring or SOP minimization can be polynomially reduced to a Boolean Satisfiability formula - the SAT problem). Such approach is theoretically feasible in classical computers, but it is rarely practical. My hardware search has perhaps similar properties: although in theory we can reduce all problems to a SAT, it is better when the user of my system has at his disposal several types of reduction to hardware mechanisms rather than to SAT circuits only.

## **2.20.2 Summary of main concepts and ideas**

Concluding, there are several main ideas focused on in this thesis:

The concept of oracle as a generalization of SAT is introduced and several practical oracles are built. Oracles are fundamental concepts to solve puzzles and combinatorial problems. The reader should learn how to specify arbitrary oracles in VHDL or Verilog and test them on Veloce.

The combined search strategies originally developed in (Perkowski82) and extended in (Brown90), (Juling Liu), and (Dill) have been extended to parallel accelerated hardware computing, modified and implemented as the HASPS search algorithm with improved behavior. New strategies are added and they will be tested by students in VHDL, Verilog or System Verilog. The testing and verification results will be discussed in this thesis. The reader has to be able to design hardware realizations of various search strategies using stack, queue or sets of finite state machines, or other approaches.

Several new search algorithms for SOP, FPRM, and ESOP circuits and other logic synthesis problems have been already proposed. The algorithms are unaffected by the degree to which the problems were completely specified (i.e. a large or small number of “don’t cares” is unimportant). These algorithms can be realized with oracles or in data-path processors similar to our SEND+MORE=MONEY sequential oracle above. We will show realizations of only some of these algorithms but the reader should be able to design them in any of the presented here models.

We show Machine Learning algorithm based on oracle model. The goal is to demonstrate that my methods are applicable to many benchmarks, for the logic minimization/synthesis of binary logic hardware circuits, Data Mining, and Machine Learning. The hardware systems for oracles and data path architectures have been invented and explained. We simulated and emulated the hardware for Man-Wolf-Goat-Cabbage puzzle, Missionary and Cannibals puzzles; graph coloring, SAT, FPRM, SOP, ESOP and other circuit types. Only some of the simulated and emulated by us examples were presented in this thesis.

## Chapter 3 Emulators and Accelerators in CAD Industry

### 3.1 Verification Techniques

#### Event Based Simulation

The most common type of ASIC verification approach in use today is uniprocessor based event driven simulation. Changes in design inputs and internal states trigger a chain of signal changes at a particular time which are recorded in a time-ordered event queue. An event driven simulator maintains a list of events to simulate various time instances. As the event list corresponding to the current time is executed, it results in additions to event lists at later points in time. When the simulator completes execution of the current list, the event list for the next time point is executed. Contemporary event driven simulators are between three and four orders of magnitude faster than electrical simulators such as Spice. These simulators typically solve a set of partial differential equations at each time step and trace out analog voltages and currents. Event driven simulators are generally either interpreted or compiled-code. Interpreted simulators involve little compilation as each event is interpreted at run time. These type of simulators are very fast for small sized designs, but become time consuming for large size designs. Verilog-XL (**Cadence**) from Cadence is a well-known interpreted simulator. Compiled code simulators require a C compiler to convert the HDL source into C code (or some other programming language) and construct an executable file. This type of simulator requires time consuming compilation but achieves a faster runtime, especially for large designs. The Affirma NC Series (**Cadence,NCSIM**) of simulators from Cadence are examples of compiled code simulators.

### 3.1.1 Cycle Based Simulation

For large designs event driven simulation can take hours to complete. To improve verification speed, cycle-based simulators may be used. These simulators construct event lists only for each clock edge and are limited to functional verification rather than timing. By limiting simulation to clocked edges, the overhead of event management is reduced and an improvement in performance over event driven simulators can be obtained. The Cyclone simulator (**IKOS**) from Synopsys is an example of a cycle-based simulator.

### 3.1.2 Hardware Based Verification Tools

For large designs, even a cycle based simulation approach may be overly time consuming. Increased chip complexity and limited design times have resulted in the development of hardware based verification tools such as simulation accelerators and logic emulators. Simulation accelerators typically consist of a number of interconnected processors. A user's design is partitioned across the processors so that logic dependencies are minimized. All the processors operate in parallel offering a significant performance gain over uniprocessor simulation. NSIM (**IKOS**) from Icos is an example of such a system.

Logic emulators fully implement user designs in hardware. The emulator provides a complete functional implementation of the design that runs within an order of magnitude of real time speed. One limitation of emulation is that it retains only the functional behavior of the circuit, which means that validation of both performance and timing

features cannot be performed on a logic emulator. Two main approaches to logic emulation exist today -the custom processor approach and the field programmable gate array (FPGA) approach. CoBALT (Cobalt Datasheet) from Quickturn uses a custom processor approach. RTL code is partitioned and scheduled to execute on an array of customized concurrent processors. Celaro (**Celaro Datasheet**) from Mentor Graphics is another example of an emulation system that uses customized programmable devices.

The inherently reprogrammable nature of FPGAs make them an ideal choice for use in logic emulators. FPGAs are flexible and do not require the fabrication cost of custom processors. Both System Realizer (**SR Datasheet**) from Quickturn and Virtua-Logic/VStation (**V-logic Datasheet**) from Ikos are FPGA based systems. System Realizer contains crossbar interconnection devices to overcome inter-FPGA pin limitations. Ikos emulators use time division multiplexing of the FPGA interconnect called VirtualWires (**Tessier,1993**) instead of crossbars to overcome pin limitations.

Logic emulation shares many of the advantages and disadvantages of both prototyping and software simulation (**Hauck,1998**). Like a prototype, the design to be tested is implemented in hardware so that it can achieve high performance during testing. However, like software, the emulated design can be easily altered and observed to isolate bugs. Transforming a circuit description into a form that can be implemented onto an emulator can take many hours to perform. This task is usually done by emulation system software. The high performance of emulators can be attributed to the fact that they can

implement the complete circuit in parallel which software simulation and simulation accelerators cannot do.

For circuits that execute software programs, the emulator can be used to debug these programs much earlier in the design flow than a prototype. This is because an emulator can use a high level description of the circuit (such as a hardware description language) for implementation, while a prototype cannot be made until the complete circuit has been designed. Software simulation is too slow to run software in a feasible time frame. A circuit implemented on an emulator can be inserted into a target environment and the system can be evaluated in a more realistic setting. This helps both to debug the circuit and to test circuit interfaces. For example, an ASIC and the board that will contain it are often developed simultaneously. An emulated version of the ASIC can be inserted into the circuit board prototype for testing both ASIC and board functionality.

### **3.2 Transaction Based Verification**

Synchronization between different verification engines (netlist, RTL, or ISS simulators and emulators) plays a crucial role in determining the raw performance that can be achieved (**Selvidge,2001**). Event and cycle based synchronization are examples of fine grained synchronization in which the verification engines synchronize at every event and clock cycle respectively. Due to this tight coupling, the entire system proceeds at the rate of the slowest domain. As a result, verification performance is limited.



An alternative approach is to synchronize the engines only when necessary via transactions. A transaction can be defined as a multi-cycle communication sequence between two verification domains. Transactions contain both data and synchronization information. A single transaction results in multiple cycles of work being performed by a verification engine. A transaction can be as simple as a memory read or as complex as the transfer of an entire structured packet through a channel. This is accomplished by the transfer of a single message through one synchronization point. The increase in performance over cycle and event based synchronization is substantial.

### **3.3 Co-Modeling**

Co-modeling is a transaction based verification approach in which system behavior (modeled in a high level language such as C), interacts with an RTL design which is implemented on the emulator.

Co-modeling has the following advantages :

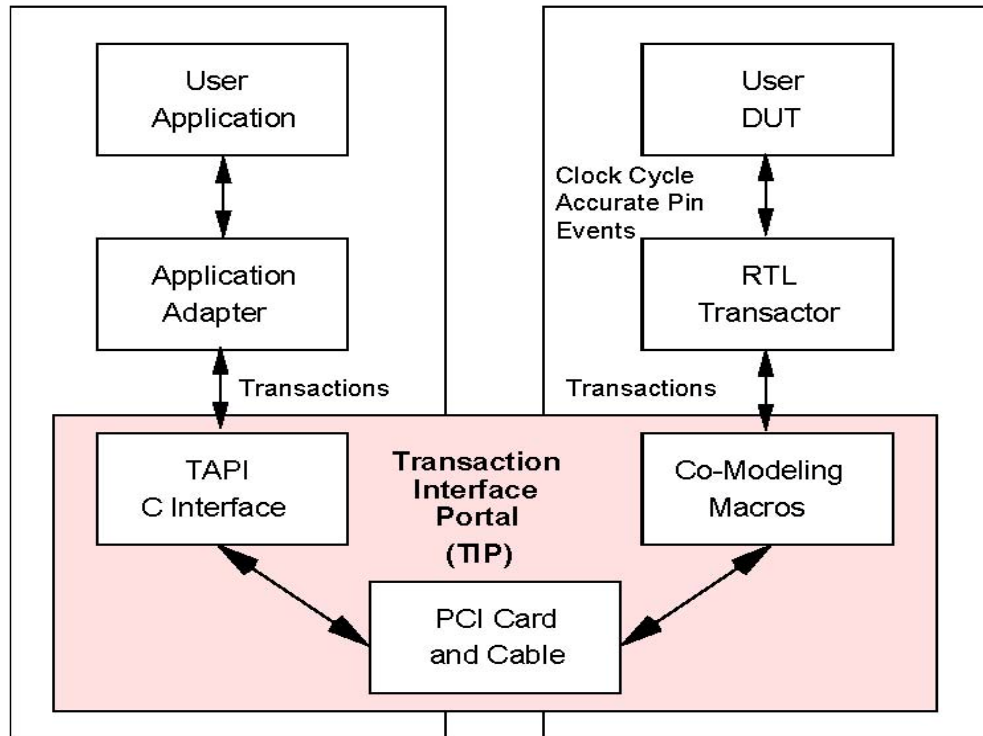
1. Co-modeling provides a system level verification solution that links system implementation (such as a gate level HDL model) with system behavior (such as a C/C++ based model). It makes use of the fact that the system designer has created a behavioral model of the system and reuses this behavioral model as part of the test environment for the entire system.
2. Co-modeling offers higher performance when compared to an event/cycle interface based on a Programming Language Interface (PLI) implementation. By using a co-modeling based methodology, we can build interfaces that are standard and

re-usable. For example, if an Ethernet transaction model is built, it can be re-used for future designs that might have an Ethernet port built in.

Co-modeling can be used in two ways -data streaming and reactive co-modeling. This thesis focuses on data streaming. In data streaming, transactions are independent of each other and are sent continuously from the user application to the DUT and vice versa. The data passed between the models is intended to correspond to specific pins of the DUT. A transactor located on the emulator does not modify or process the data in any way, but provides the handshaking signals which enable data exchange between the software model and the DUT also located on the emulator. This kind of operation provides maximum throughput. In reactive co-modeling, the transaction sent by the user application depends on the previous transaction. The user application has to wait for the DUT to process the current transaction before it can send a new one. The channel is not utilized effectively in this way since the user application and DUT may be idle awaiting new transactions.

### **3.3.1 Transaction Interface Portal (TIP)**

Transaction Interface Portal(**IKOS-TIP,200**)(**IKOS-TIP-Co-modeling**)(**TIP User's guide**) is a transaction based co-modeling interface that provides a communication channel between a host workstation and an emulator.



**Figure 3-1 Architecture of TIP**

Figure 3-1 illustrates the layered architecture of TIP for an Ikos emulation system. It consists of a design environment (DE) running on a host workstation, and a DUT running on the emulator. These two domains are connected via a communication channel which provides a mechanism for transporting data and synchronization between the DUT and DE. One main component of the DE is the User Application. The User Application may contain a complex C model of a system component, or it may contain a test environment that provides test vectors for the DUT. It utilizes an Application Programming Interface (API) called the Transaction API (TAPI) to communicate with the DUT. This API is provided by the Application Adapter. The DUT is comprised of the User's Netlist, an RTL Transactor, and Co-Modeling Macros.

The User Netlist is a gate level model of the design to be verified with the C code. The transactor acts as an interface between the User's Netlist and the underlying comodeling macros. The co-modeling macros are responsible for data transfer between the transactor and the PCI card. This card is installed on the host workstation and performs data transfer between the emulator and the workstation.

### **3.3.2 Veloce Emulator**

Veloce Emulator is Hardware accelerator as well as emulator from Mentor corporation. It offers an option for accelerating the process of embedded software verification with ICE (In Circuit Emulation).

#### **3.3.2.1 Architecture of Veloce**

The Veloce emulator is available in 3 versions among them Veloce Solo (Figure 1) which is located in the Department of Electronics at Polytech'Nice-Sophia. This version is the single user mode which involves a specific organization for labs. The emulator Veloce Solo allows emulating up to 16 MGates (with 2 cards), a storage capacity of 1GB memory and a clock rate up to 1.5 MHz.

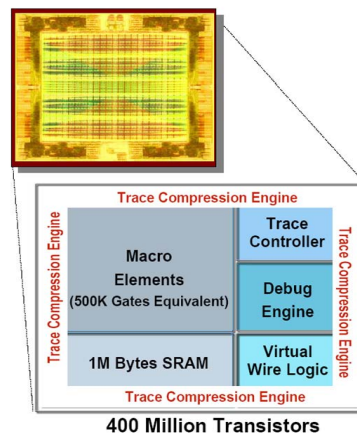
	Veloce Trio-24/48	Veloce Solo	Veloce Quattro
			
Use Model	Simulation Acceleration	Simulation Acceleration, ICE	Simulation Acceleration, ICE
# of Users	3 Users	1 User	Up to 4 Users
Capacity	Up to 8MG or 16MG per user	Up to 16 MG	Up to 128MG
Performance	Up to 1.5 MHz	Up to 1.5 MHz	Up to 1.5 MHz
Memory	Up to 1 GB per User	Up to 1 GB	Up to 8 GB
Dimensions W x D x H	35cm x 80cm x 68cm	45cm x 75cm x 75cm	133cm x 89cm x 115cm

**Figure 3-2 Veloce Family**

The emulation technology (Figure 3-2) is based on a custom SoC, 90nm, 8 layers of copper, 500KGates and including a module VirtualWires integrated for the communications between Custom SoC. A 1MB of memory is built into each "Custom SoC" for the design. In addition, a debug module and a controller track is included and allow capturing signals.

### 3.3.2.2 The environment around Veloce

For an easy use of Veloce, the emulator is connected to a host machine behind a Linux server. The host is a machine PowerEdge 2900 (Quadri Intel Xeon processor at 2.3 GHz) with 16GB of memory and using operating system Red Hat Linux Enterprise 4.



**Figure 3-3 Emulator Technology**

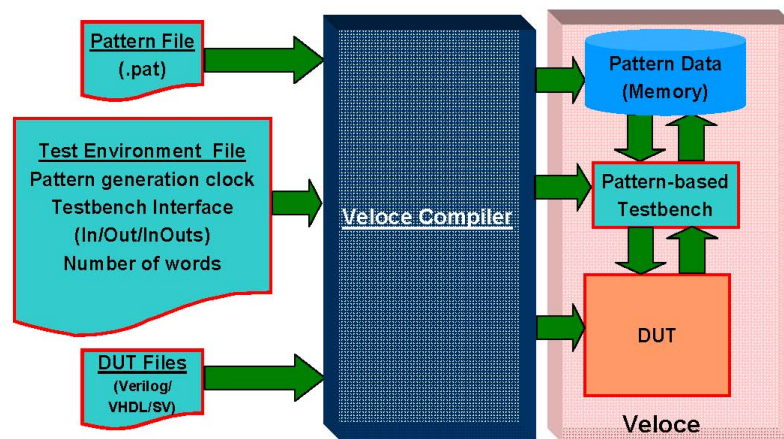
The software runs on the host machine. The input languages are all HDL languages: VHDL, Verilog for the design of IP and SystemVerilog, SystemC for testbench. The first part of the flow is carried out off-line and the rest of the flow online (only one user). The main steps of the flow are:

- Offline: Preparation of the HDL modules or IP (Intellectual Property) and its environment,
- Offline: Analysis of the design, compilation of the code (RTL) and synthesis of the code (velsyn)
- Offline: Preparation of the IP for emulation.
- Online: Connection to the emulator,

- Online: Debug of IP,
- Offline or Online: Analysis of trace

### 3.3.2.3 Standalone Mode

The objective is to discover the general flow of the emulator. To do this, we propose to use the standalone mode. The principle is to verify an IP described in VHDL at RTL level (Register Transfer Level) and using test vectors embedded in the emulator.

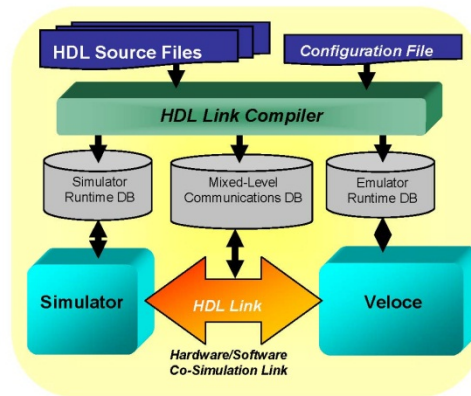


**Figure 3-4 Stand-alone Mode**

For a right illustration of Veloce, the IP includes a control part (registers, counters) and a memory.

### 3.3.2.4 HDL Link Mode

In this case, the test bench is used and runs on the host machine with ModelSim (VHDL/Verilog simulator) while the IP is still emulated. This configuration resembles a classic ModelSim simulation and is called the HDL Link mode. The disadvantage of this mode is the amount of information (VHDL transaction, delta time), which runs between the host and the emulator. The immediate consequence is the slowdown of global emulation.



**Figure 3-5 HDL Link Mode**



### 3.3.2.5 TBX Mode

This mode is useful because we have the possibility to describe the test bench in a highest level language as SystemVerilog or SystemC. The objective is to accelerate emulation passing only non-time transactions between the emulator and the host machine. The principle is to write transactors described in SystemVerilog which translate untimed transactions sent by the host signals to the level RTL and vice-versa. The IP and the transactors are emulated on Veloce (Figure 3-6). This emulation mode is called TBX.

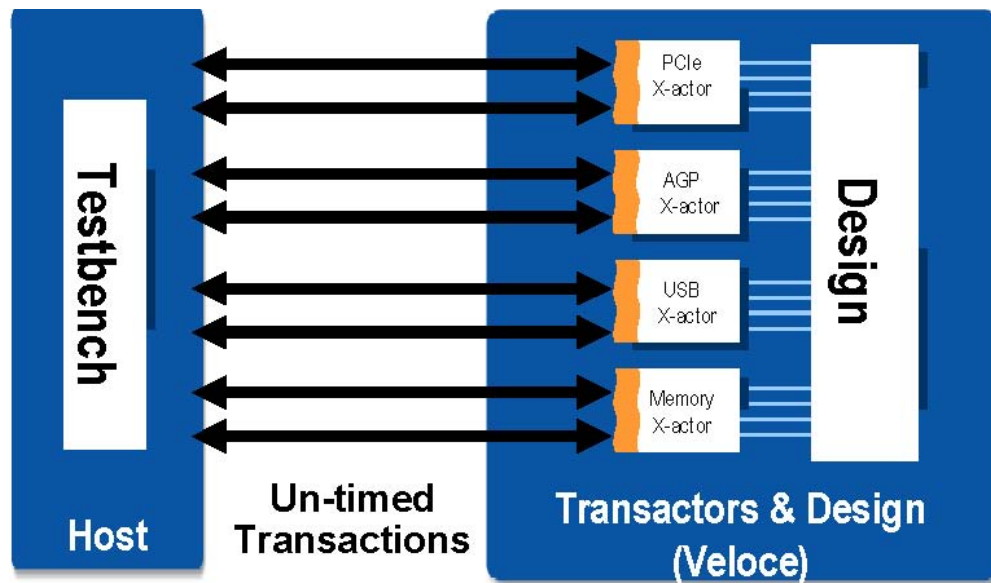


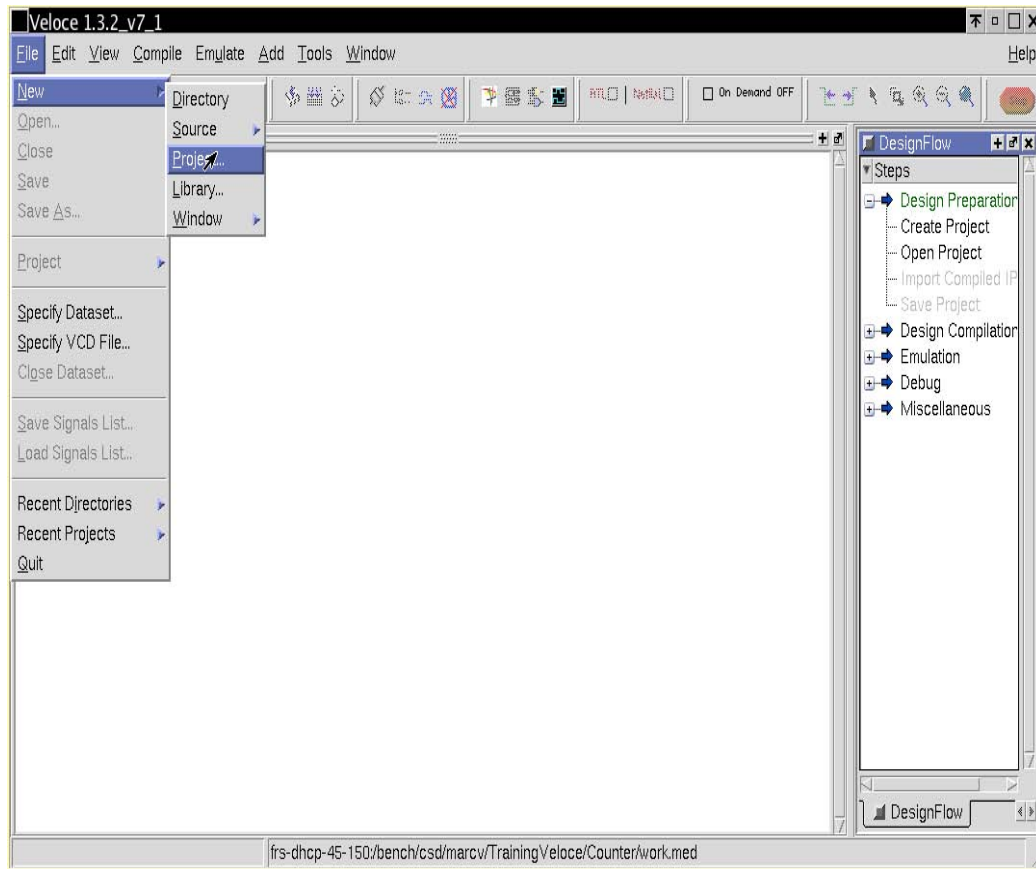
Figure 3-6 TBX Mode

## Chapter 4 The Veloce Emulator and How to Use it

This tutorial explains the Mentor Graphics veloce emulator flow.

- 1) Run vncserver to connect to velocesolo
- 2) Login to velocesolo using your MCECS login
- 3) Set your directory to the one in which you put the Verilog files in step 1:
- 4) Invoke the Veloce GUI from the command line as follows:  
  
    % velocegui &
- 5) Create a “**New Project** “
  - a. On the Veloce GUI menu-bar, **select File □New□Project**
  - b. Give the project a name (Ex: ‘missionary cannibals’).
  - c. Select appropriate Project location.

**Click OK.**

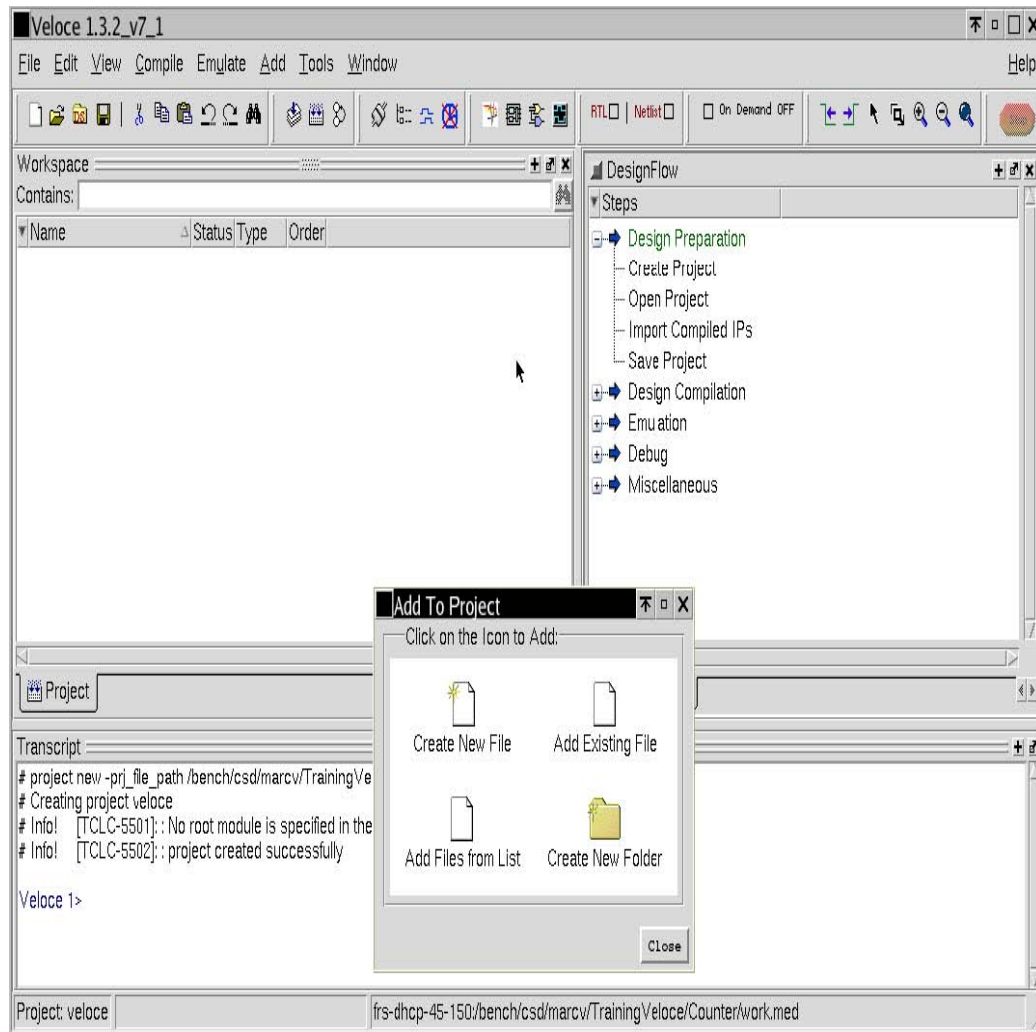


6) Create New Files:

a. From the “Add to Project” popup window, select the “Create new File” option (icon)

Give appropriate file name. Select File type such as (VHDL, Verilog, System verilog)

**Click OK.**



**Select** design source files and **click OPEN and OK. Close** window and **Save** the project.

b. Or select “Project Tab” in “Workspace” window, **click right** mouse button, select “Addto Project” ☐ “Existing file”

**Select** design source files and **click OPEN and OK. Close** window and **Save** the project.

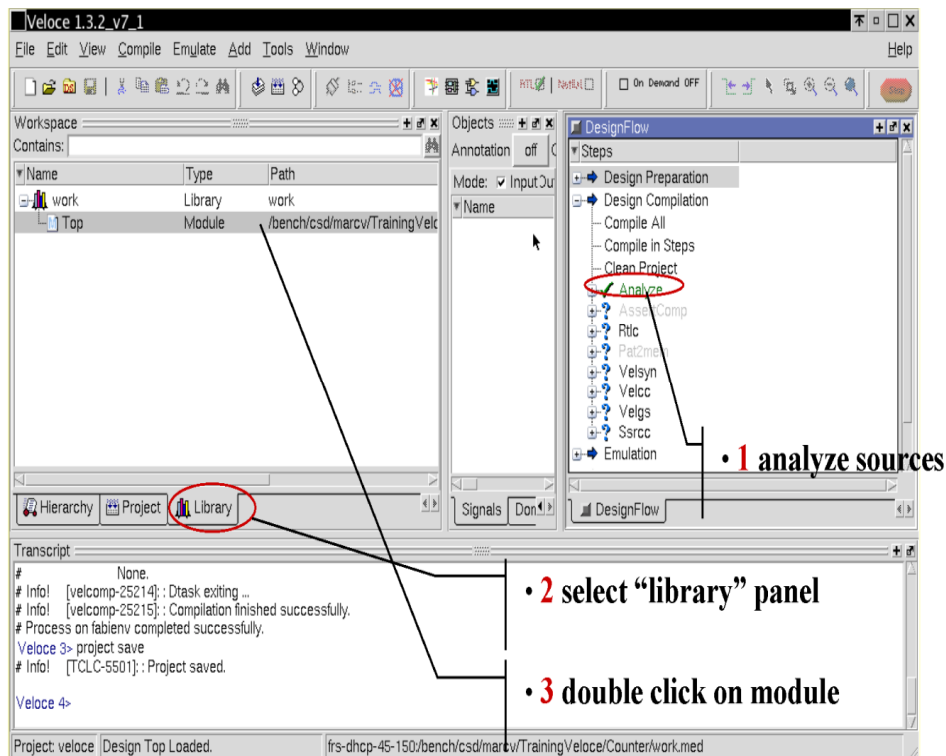
- 7) Design Compilation: In the “**DesignFlow**” window, select “**Design Compilation**” and click“(+)”.

<Compile in Steps>

- a. To run Analyze Step: **Double click “Analyze”**
- b. To run Rtlc Step: Need to define the top module.

---

## Same Selection of Top Module as in Modelsim



In “**Workspace**” window, **right click** in the top module (the testbench) of the “**Library**” pane, select “**set as top**”. **Double click** “**Rtlc**” in the “**DesignFlow**” window

c. To run Velsyn Step: Need to **Set**specific Options &**Setup** Parameters

**Click** “**(+)**” to the left of “**Velsyn**” in the “**DesignFlow**” window then you can see “**Option**” and “**Setup**” menu under “**velsyn**”.

**Double click** “**Option**” and **select** as:

- Target ☐ Solo(Comodel (D1SC))
- Board # ☐ 1

**Click OK.**

ii. **Click** “**(+)**” to the left of “**Setup**” and **Double click** “**Timing Setup**”.

An error message will pop up. **Click OK.** (We haven’t specified the clock so there will be an error message.) Now we are going to create the ‘vmw.clk’ file.

**Double click** “**Timing Setup**” again.

**Adding** clock &**setting** Data I/O information as:

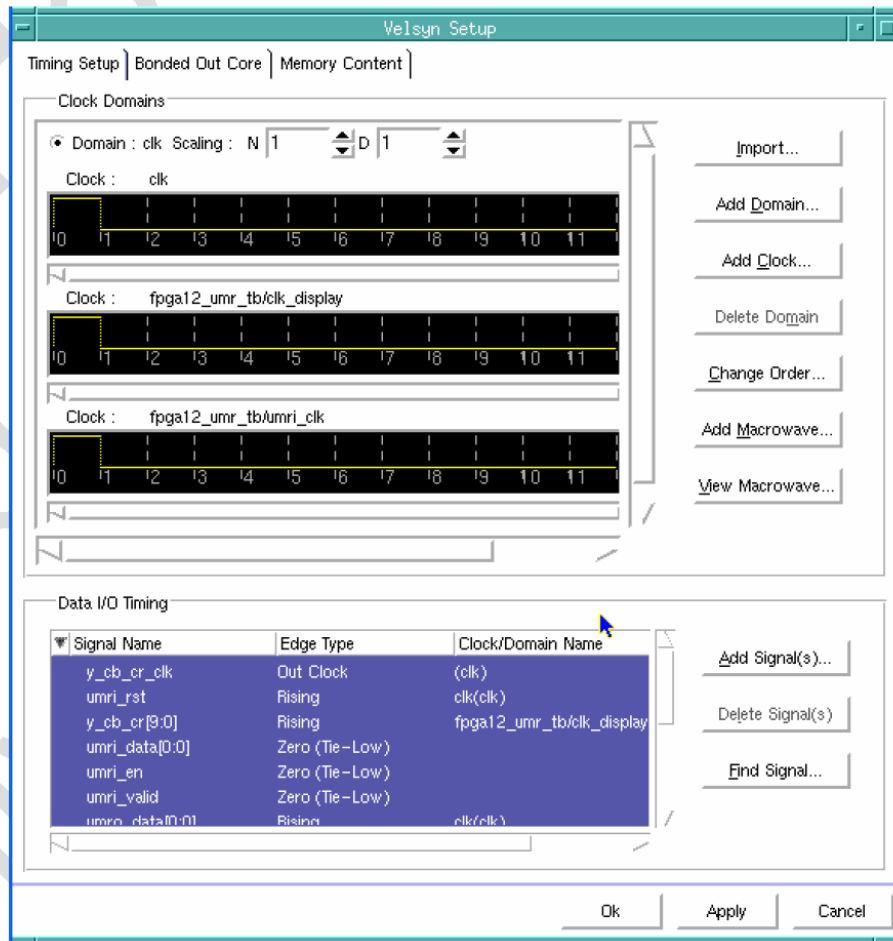
d. Clock domains:

**Click** “**Add Domain**”, give a domain name (‘domain1’), and **click OK.**

**Add** a clock by **clicking** “**Add Clock**”, **select** a clock signal (‘*clk*’ signal), and **click OK.**

After adding the clocks, waveforms can be edited by clicking on the waveform region ( ).

(A single clock cycle needs to be set.)



e. Data I/O Timing:

Click “Add Signal(s)”, select data signals (*‘reset’*, *‘enable’*), and click OK. Define ‘Signal Type’ for the data signals (*‘reset’* & *‘enable’*) by selecting the signal, clicking right mouse button while in the Signal Type column, and select “Register”. Click OK.

f Double click “Velsyn”

g To run Velcc Step: Double click “Velcc”

h To run Velgs Step: Double click “Velgs”

i. To run Ssrcc Step: Double click “Ssrcc”

- 8) Perform Emulation:
  - a. Setup the Emulation:

In the “**DesignFlow**” window, select “**Emulation**” and click “**(+)**”.

**Double click “Emulation setup”, type Emulator name as “velocesolo1”, and click OK.**

- b. Check to see if another user is connected to the emulator:

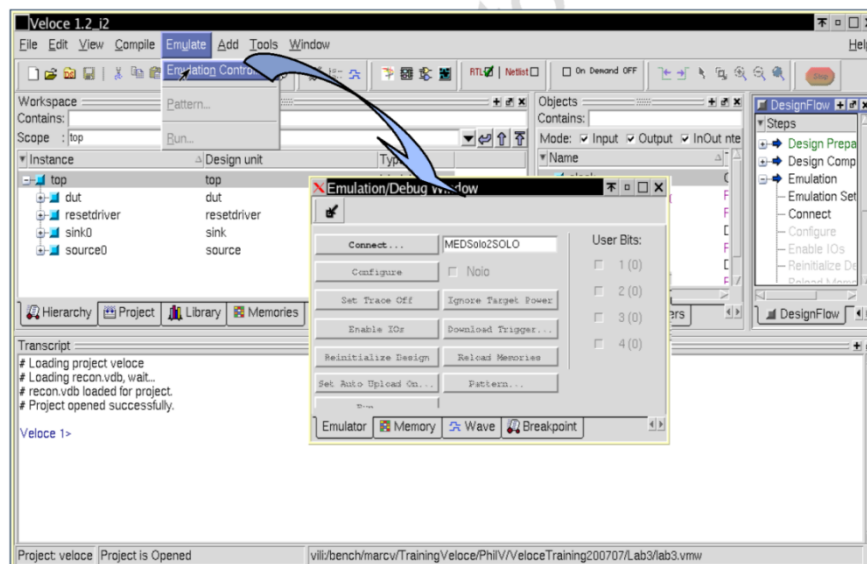
% whoison –emul velocesolo1

If no one is connected to the Emulator, you can connect to the machine. If someone is using the Emulator, you have to wait until the user gets disconnected.

Click on **Emulate** then **Emulation control**

---

## Emulation – control Panel





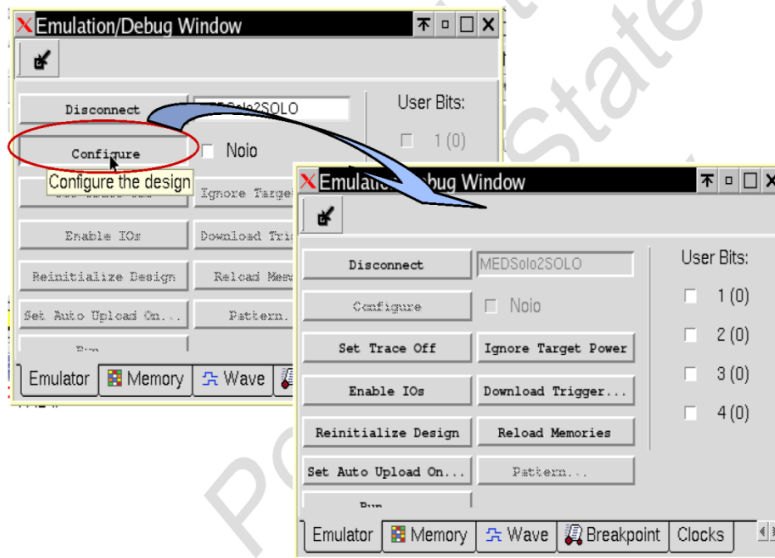
Type Emulator name as “**velocesolo1**”,( Picture shows incorrect name)

- c. Connect to Emulator: **Double click “Connect”**
- d. Load Design(Configure): **Double click “Configure”**

---

## Emulation - Load design

- ◆ Tcl command: configure

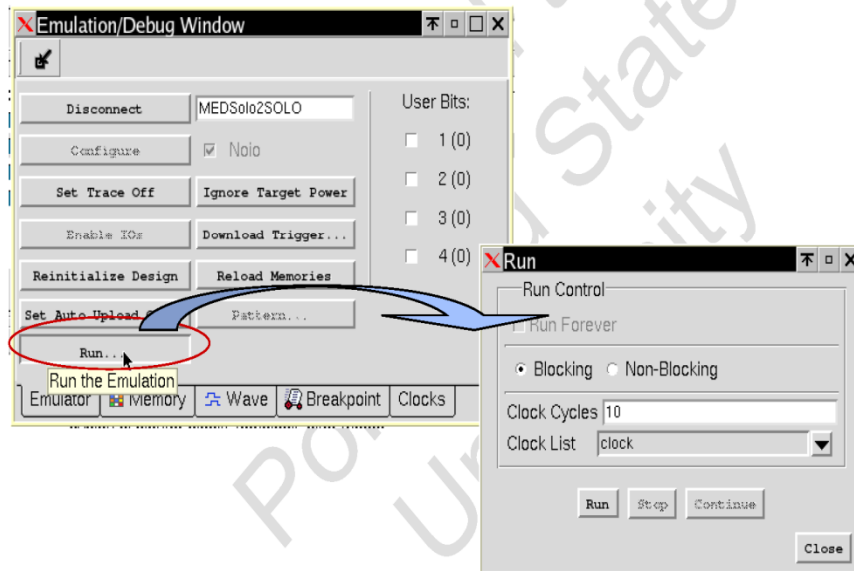


- e. Run the Emulator:

---

## Emulation - run

◆ Tcl command: `run 10 -clk clock`



*Alternatively*

*You can try run 50000 .....This will run 50000 cycles of the emulation clock.*


- f. Upload Waveform: On the Veloce GUI menu-bar, select **Emulate** □ **Emulator Control** select “Wave” Tab, click “Upload Waveform”, give a new waveform name next to “Trace Directory name”(ex: waveform\_1), and click “Upload”. If a file was successfully produced, a green check appears to the right of ‘waveform\_1’ file.
- g. Disconnect from Emulator: **Double click** “Disconnect” from “Design Flow” window.

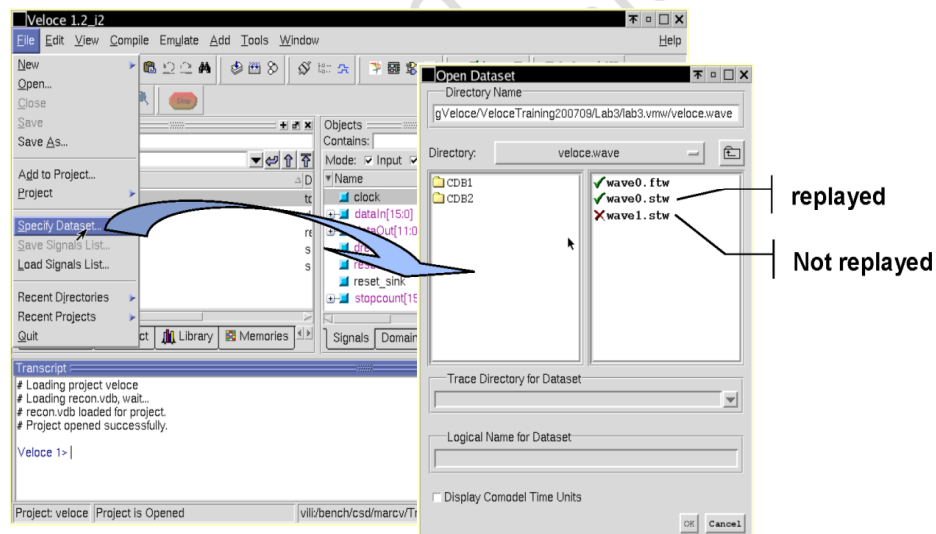
(**IMPORTANT:** After generating a “waveform” file, each user must “Disconnect” from the emulator so other users can use the emulator.)

- h. Invoke “Waveform Viewer” Window:
- i. On the Veloce GUI menu-bar, select **View** ☐ **Debug Windows** ☐ **Wave**
- ii. On the “Wave” window, select **File** ☐ **Specify Dataset**, select waveform file(‘waveform\_1.stw’), and **click OK**.

---

### Veloce GUI Waveform viewer – Specify Dataset

- ◆ **Tcl command:** `dataset open <tracedir> <db name>`
- ◆ **Button** 



- iii. **Add** all /selected signals in the “**Objects**” window(located at main Veloce GUI)to “**Wave**” window as:

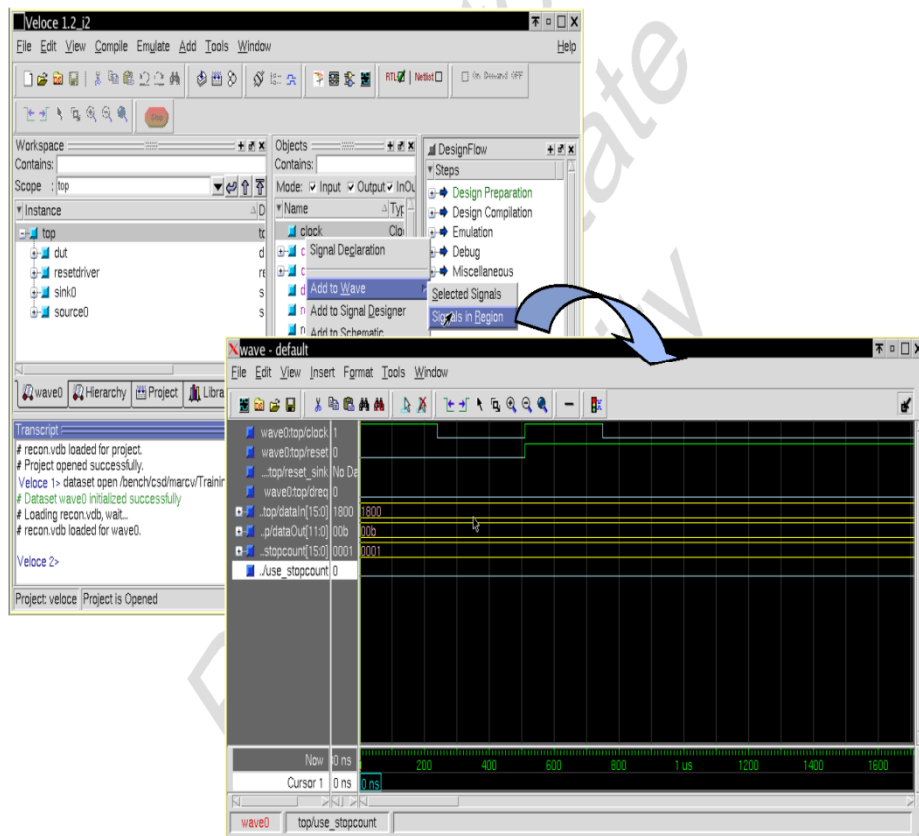
Select all/selected signals from “Signal” Tab in “Objects” window.

Click **RIGHT** mouse button, select “Add to Wave” ”selected Signals”.

---

## Velocite GUI Waveform Viewer – add signal

◆ Tcl command: `wave add <signal name>`



9) Close the Velocite GUI (this will disconnect you from the emulator)

- a. Save the project: On the Veloce GUI menu-bar, **select File** ☐ **Save**.
  - b. Close the project: On the Veloce GUI menu-bar, **select File** ☐ **Project** ☐ **Close**.
  - c. Close the GUI: On the Veloce GUI menu-bar, **select File** ☐ **Quit**, **click YES**".
- 10) Logout from velocesolo.

# Chapter 5 SAT: Search for Satisfiability, Petrick Function Minimization and Related Problems

## 5.1 Introduction

In this chapter we discuss how to construct binary and multi-valued input SAT models for hardware-realized algorithms. These include reconfigurable, data-path, tree search, multiplication, oracle-like and other hardware algorithms. I analyzed several algorithms to solve SAT and some close combinatorial problems and found important similarities. These similarities were next used to construct general concepts of algorithms based on SAT and CSP models. These algorithms cover a wide class of problems. They can be realized in hardware and can model binary logic problems as well as multi-valued logic problems. There is also a didactic value in comparing these algorithms and building oracles and other hardware for them. Ultimately, my goal is to create a hardware system for prototyping algorithms which has elements of evolutionary, tree search and other classical software and hardware algorithms.

In this chapter I show a class of problems that can be reduced to oracle-based algorithms:

1. Genetic algorithms and similar,
2. Tree search algorithms,
3. SAT/CSP algorithms for the Oracle model.

I hope that my analysis shown in this chapter will allow create and analyze various classes of algorithms quickly and with little effort. All these applications are practical. They are all based on creating SAT or CSP models or equivalently on creating oracles for Oracle

algorithm (Hossain09). The first task is then to be able to design oracles systematically and for any problem.

This chapter shows four methods for solving SAT and similar problems:

1. Using simple oracle, oracle with information - how many terms are satisfied and more advanced oracles.
2. Multiplying terms in POS SAT and simplifying expressions using Boolean Logic.
3. Tree search based on positive and negative cofactors.
4. Sequential oracle based on linear architecture.

My first main task in this chapter is to explain the SAT model of solving problems. We will start from the simplest of these problems and continue towards explanation of a large class of problems. Working in a systematic way on designing SAT models for many problems, we realized that there are certain categories of problems for which the models are very similar. Therefore, we tried to classify all these models to certain types of models. Such characterization will allow us to design hardware architectures with less effort, systematically and by reusing typical sub-methods.

Based on the solved problems, there exist the following types of models:

**Satisfiability models:** These models are based on creating a single-output satisfiability formula. The formula can use various gate types and logic structures, depending on the problem.

**Constraint satisfaction models:** These types of models are for constraint satisfaction problems such as graph coloring, image matching or cryptographic puzzles. These



models use logical, arithmetical and relational blocks and have often the decision oracle and the optimization oracle as their components. The decision model is a global AND of several partial decision sub-models. The Constraint Satisfaction Models can be treated as generalizations of Satisfiability Model. This understanding helped us to build Constraint Satisfaction models and we believe that it should be always used when new SAT/CSP models for CS problems are being built.

**Path problems:** Path problems are to find certain path in a graph, for instance an Euler path or Hamiltonian path. Many games and puzzles such as “Man, wolf, Goat and Cabbage” belong to this category. The graph is given explicitly or implicitly. The SAT/CSP models for path problems include decision sub-models for each move (edge) in the graph of the problem (game). The path problems can be also treated as constraint satisfaction problems in which constraint variables are repeated for successive units of time. When we know how many time units are needed in a sequence of moves to solve the given problem or the given problem instance, then this problem can be reduced to the constraint satisfaction problem based on oracle.

**Problems related to spectral transforms:** Walsh, Reed-Muller, Haar, Fourier, etc. For instance; (1) finding the best transform in certain transform family for a given input data, one that optimizes certain cost function, (2) finding the completion of incomplete function to a complete function that has the minimum number non-zero coefficients in its spectral transform (for instance Walsh).

The mapping problems, including their special classes;

1. the subset selection problems,
2. the permutation problem,
3. the combination problem. These problems are also constraint satisfaction problems, but they have special properties which makes them easier to solve based on analogies certain known algorithms for similar problems.

The Satisfiability oracles include the following:

1. POS satisfiability,
2. Solving theunate covering problem by Petrick Function,
3. Solving binate covering problem,
4. Solving various multi-level SAT formulas, especially the generalized SAT of the form  $\prod \sum \prod x_i$
5. Solving the even-odd covering problem for ESOP, PPRM, FPRM and similar logic minimization problems,
6. Solving the AND-OR Directed Acyclic Graphs (DAGs) from robotics and Artificial intelligence.

The constraint satisfiability oracles include:

1. Proper graph coloring
2. Compatible graph coloring
3. Graph coloring problems with non-standard cost functions
4. Waltz algorithm for image matching
5. Cryptarithmic puzzles such as SEND + MORE = MONEY

The Mapping oracles include:

1. Maximum cliques (used in Maghoute algorithm for graph coloring),
2. Maximum independent set,
3. Finding prime implicants of a Boolean Function.

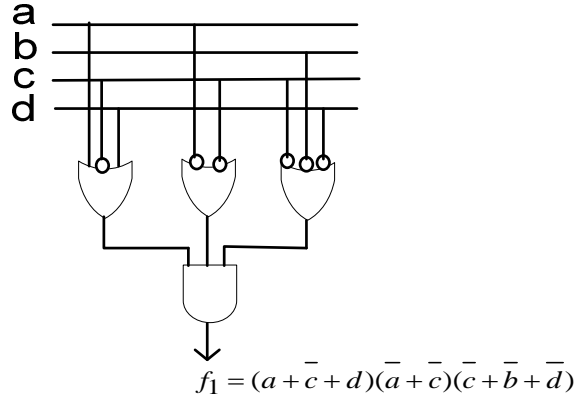
Path oracles include:

1. Euler path,
2. Hamiltonian path,
3. Shortest path,
4. Longest path,
5. Traveling salesman path,
6. Missionaires and cannibals logic puzzle,
7. Man, Wolf, Goat and Cabbage logic puzzle.

Exhaustive solving of equations includes:  $a_n + b_n = c_n$

## 5.2 Solving the Satisfiability Class of Problems

### 5.2.1 Product of Sums SAT (POS SAT)



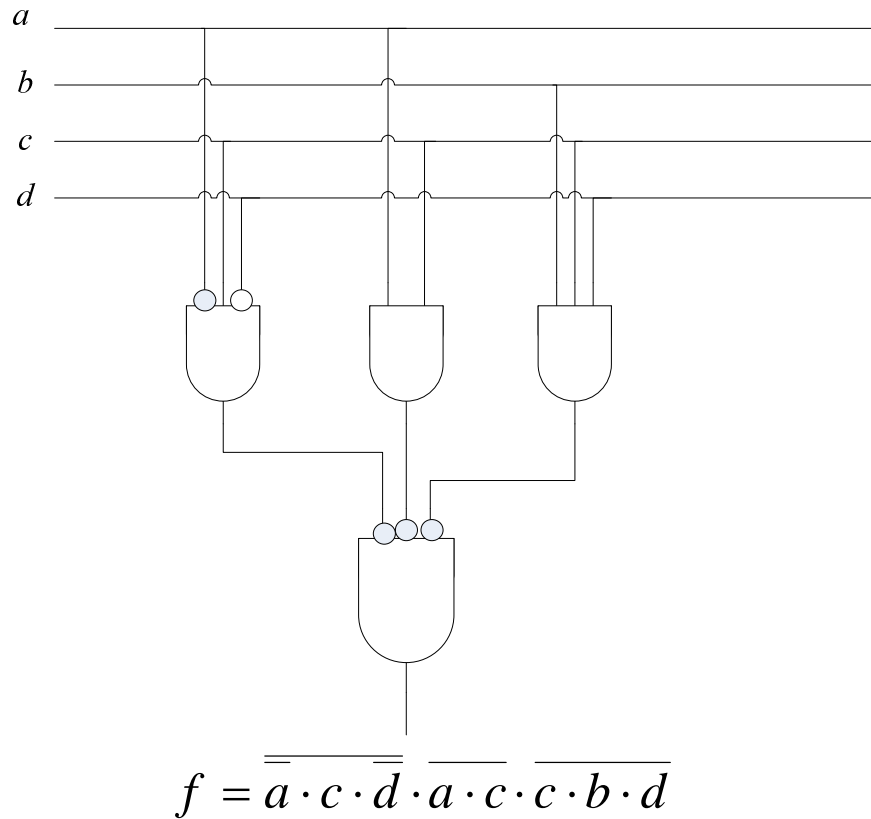
**Figure 5-1 Classical Oracle for POS satisfiability**

The fundamental role of satisfiability to computer science, algorithm design, CAD and complexity theory is well-known.

In this section we will present examples of building oracles for various satisfiability problems. Let us build first the oracle for function  $f_1 = (a + \bar{c} + d)(\bar{a} + \bar{c})(\bar{c} + \bar{b} + \bar{d})$ . The standard oracle is presented in Figure 5.1. The formula can be transformed as follows, using De Morgan rules:

$$\overline{\overline{(a + \bar{c} + d)}} \bullet \overline{\overline{(\bar{a} + \bar{c})}} \bullet \overline{\overline{(\bar{c} + \bar{b} + \bar{d})}} = \overline{\overline{acd}} \bullet \overline{\overline{ac}} \bullet \overline{\overline{cbd}} \quad \text{Equation 5.1}$$

Using Equation 5.1 the oracle can be built as shown in figure 5.2



**Figure 5-2 Oracle for f**

```
//-----
//-----File name: pos.v-----
//Description: Verilog code for Product of sum
//-----
module pos(f,a,b,c,d,clk,reset);
input clk,reset;
input a,b,c,d;
output f;
reg f;

always@(posedge clk)
begin
if(reset)
begin
f<=0;
end
else
f <= ((a|(~c)|d)&((~a)|(~c))&((~b)|(~c)|(~d)));
end
end
```

```

endmodule

//-----
//-----File Name:pos_tb.v-----
//Description: Testbench for produt of sums
//-----

module pos_tb(f,clk,reset);
input clk,reset;
output f;
reg (3:0)A;//temperory variable

//Instatiation
pos pos_inst(.f(f),.a(A(3)),.b(A(2)),.c(A(1)),.d(A(0)),.clk(clk),.reset(reset));

always@(posedge clk)
begin
if(reset)
begin
A <= 4'b0;
end
else
begin
if(A == 4'b1111)
A <= 0;
else
A <= A+4'b0001;
end
end//always

endmodule
-----File Name:pos_tb.v-----
//Description: Testbench for produt of sums
//-----

module pos_tb(f,clk,reset);
input clk,reset;
output f;
reg (3:0)A;//temperory variable

//Instatiation
pos pos_inst(.f(f),.a(A(3)),.b(A(2)),.c(A(1)),.d(A(0)),.clk(clk),.reset(reset));

always@(posedge clk)
begin
if(reset)
begin
A <= 4'b0;
end
else
begin
if(A == 4'b1111)
A <= 0;
else

```

### 5.2.2. Generalized SAT.

In some problems the satisfiability formula is not in POS form. It can be either converted to a POS form, which is often very inefficient, or it can be designed as a multilevel circuit of other structure than that from Figure 5.3.

For example, given is a SAT formula:

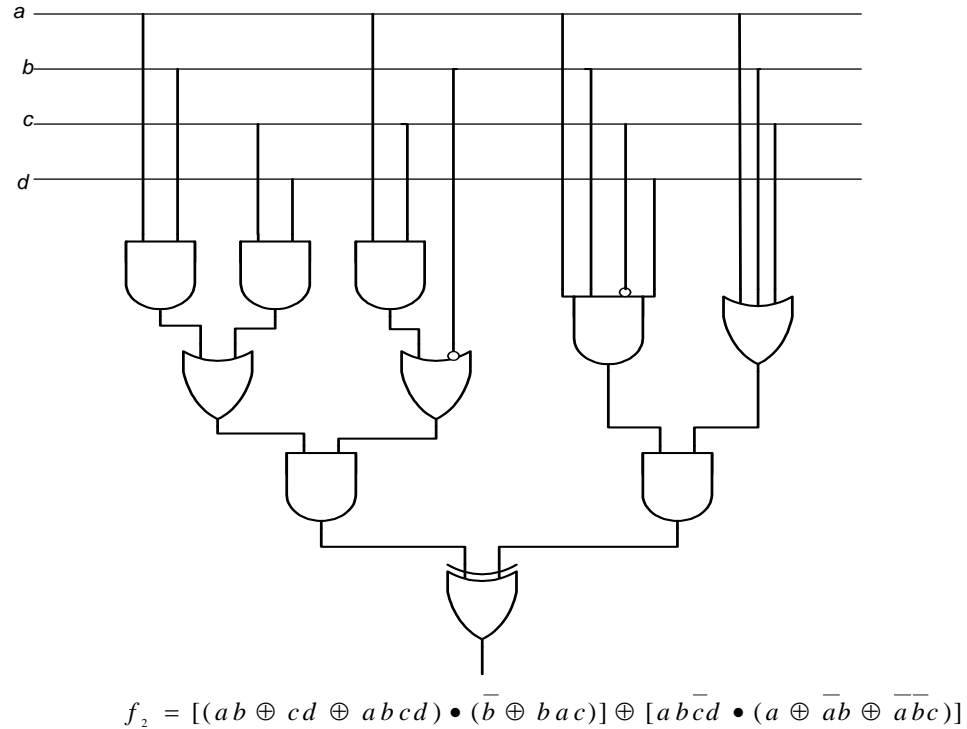
$$[(ab + cd) \bullet (ac + \bar{b})] \oplus [(a\bar{b}\bar{c}d) \bullet (a + b + c)] \quad \text{Equation 5.2}$$

The formula is transformed to the following form

$$f_2 = [(ab \oplus cd \oplus abcd) \bullet (\bar{b} \oplus bac)] \oplus [a\bar{b}\bar{c}d \bullet (a \oplus \bar{a}\bar{b} \oplus \bar{a}\bar{b}\bar{c})]$$

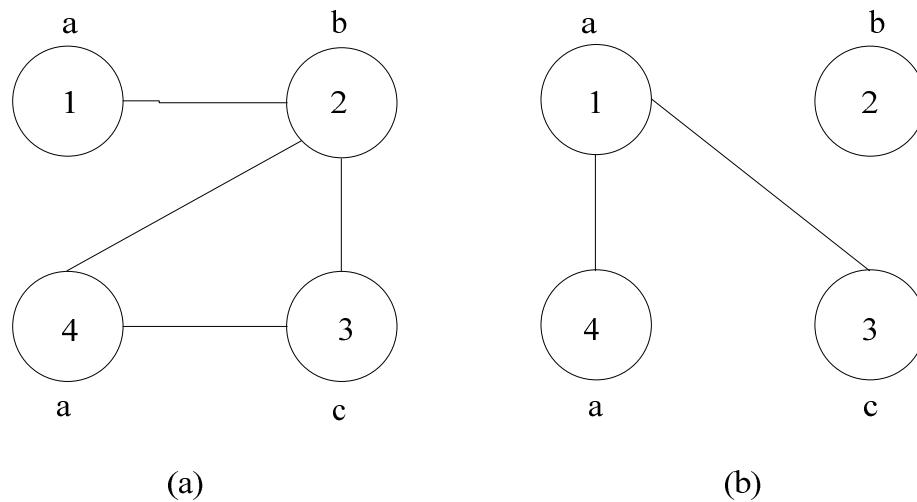
and realized as in Figure 2.1.1.3. Another way to realize  $(a + b + c)$  would be to use

$$\overline{\overline{a \bullet b \bullet c}}.$$



**Figure 5-3 Oracle for function  $f_2$**

The circuit is not minimized.



**Figure 5-4 (a) Coloring Graph Problem (b) Maximum Independent set for graph coloring problem**

Graph Coloring can be reduced to covering of nodes with Maximum Independent Sets.

Covering is SAT problem and Maximum Independent set is SAT problem.



```

//-----
//File Name:gen_sat.v
//Description:Verilog code for an generalized SAT example
//-----
module gen_sat(f2,a,b,c,d,clk,reset);

input a,b,c,d;
input clk,reset;
output f2;
reg f2;

always@(posedge clk)
begin
if(reset)
begin
f2 <= 0;
end
else
begin
f2 <= (((a&b)|(c&d))&((a&c)|(~b)))^((a&b&(~c)&d)&(a|b|(~c)));
end
end//always

endmodule
//-----
//File Name:gen_sat_test.v
//Description: Testbench for Generalized SAT problem
//-----
module gen_sat_tb(f2,clk,reset);
input clk,reset;
output f2;
reg (3:0)A;//temperory variable

//Instatiation
gen_sat gen_sat_inst(.f2(f2),.a(A(3)),.b(A(2)),.c(A(1)),.d(A(0)),.clk(clk),.reset(reset));

always@(posedge clk)
begin
if(reset)
begin
A <= 4'b0;
end
else
begin
if(A == 4'b1111)
A <= 0;
else
A <= A+4'b0001;
end
end//always

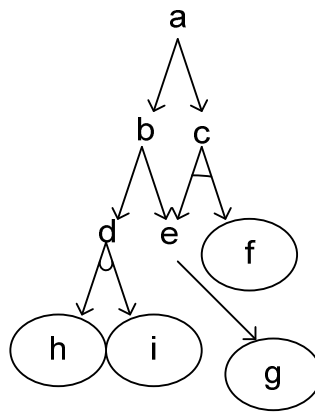
endmodule

```

## Chapter 6 Solving Problems in Artificial Intelligence And Robotics

### 6.1 AND/OR DAGs.

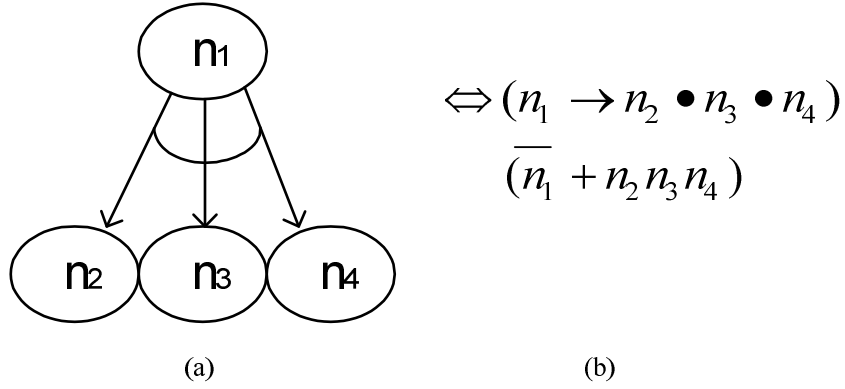
There are several problems in Artificial intelligence, CAD, planning and scheduling that can be represented by trees or DAGs (directed acyclic graphs). These structures can be converted to satisfiability formulas in classical logic which are next converted to arrays.



**Figure 6-1AND/OR DAG**

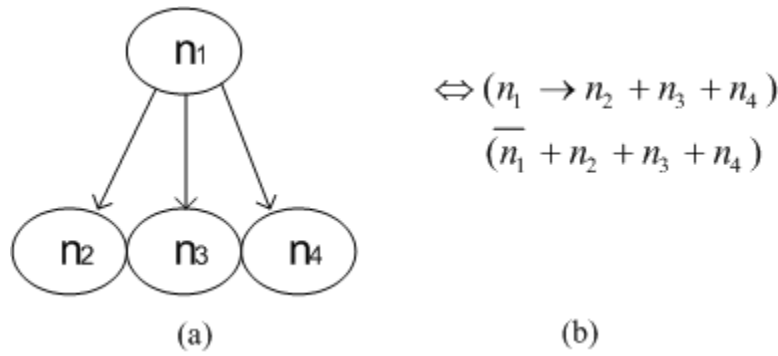
DAG shown in Figure 6-1 is for certain Artificial Intelligence Task (such as robot planning). Nodes represent tasks. Leafs represent trivial actions. Arrows represent task dependence. Nodes c and d are AND-nodes. Others are OR-nodes. Node e is implication node and nodes h, i, g and f are terminal nodes (leaves).

Given is for instance a DAG from Figure 6-1AND/OR DAG, called the AND-OR graph—the data structure used in AI. There are two types of nodes in Figure 2.1.3.1 – the AND nodes, denoted by an angle symbol between outgoing edges. This means that to satisfy the parent node, all its children nodes must be satisfied, see Figure 2.1.3.2.



**Figure 6-2**Example of the AND (a) the subgraph, (b) the logical transformation.

The other type of nodes are OR nodes. They mean that to satisfy the parent node, any of its children should be satisfied, see Figure 6-3



**Figure 6-3**Example of the OR node. (a) the subgraph, (b) the logical transformation.

From the graph from Figure 6-1 the following logic equation is written:

$$(a \rightarrow b + c)(b \rightarrow d + e)(c \rightarrow e \bullet f)(e \rightarrow g)(d \rightarrow h \bullet i) \quad \text{Equation 6.1}$$

By using the logic transformation rule

$$(A \rightarrow B) \Leftrightarrow (\overline{A} + B) \quad \text{Equation 6.2}$$

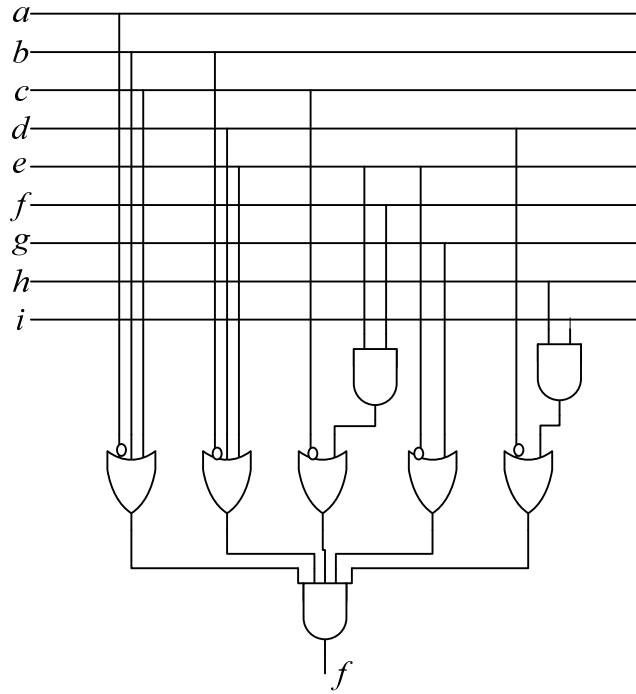
Equation 6.1 is converted to Equation 6.3 below

$$\overline{a} + b + c)(\overline{b} + d + e)(\overline{c} + e \bullet f)(\overline{e} + g)(\overline{d} + h \bullet i) \quad \text{Equation 6.3}$$

Applying the OR-to-EXOR transformation  $(A + B) \Leftrightarrow (A \oplus \overline{A}B)$  the following Equation 6.4 is created:

$$(\overline{a} \oplus ab \oplus \overline{a}bc) \bullet (\overline{b} \oplus bd \oplus \overline{b}de)(\overline{c} \oplus cef) \bullet (\overline{e} \oplus eg)(\overline{d} \oplus dhi) \quad \text{Equation 6.4}$$

It is now easy to create an oracle for the function from Equation 2.1.3.4, using in general the methods already outlined in this thesis; including mirrors and factorizations, and possibly, ancilla bits.



**Figure 6-4 Oracle for function**  $f_2 = (\overline{a} + b + c)(\overline{b} + d + e)(\overline{c} + e \bullet f)(\overline{e} + g)(\overline{d} + h \bullet i)$

*The circuit is not optimized.*

//-----  
//File Name:dag.v

```

//Description:Verilog code for an Directed Acyclic Graph
//-----
module dag(f1,a,b,c,d,e,f,g,h,i,clk,reset);

input a,b,c,d,e,f,g,h,i;
input clk,reset;
output f1;
reg f1;

always@(posedge clk)
begin
if(reset)
begin
f1 <= 0;
end
else
begin
f1
(((~a)^(a&b)^(a&(~b)&c))&((~b)^(b&d)^(b&(~d)&e))&((~c)^(c&e&f))&((~e)^(e&g))&((~d)^(d&h&i)));
end
end//always

endmodule

-----
//File Name:dag.v
//Description:Verilog code for an Directed Acyclic Graph
//-----
module dag(f1,a,b,c,d,e,f,g,h,i,clk,reset);

input a,b,c,d,e,f,g,h,i;
input clk,reset;
output f1;
reg f1;

always@(posedge clk)
begin
if(reset)
begin
f1 <= 0;
end
else
begin
f1
(((~a)^(a&b)^(a&(~b)&c))&((~b)^(b&d)^(b&(~d)&e))&((~c)^(c&e&f))&((~e)^(e&g))&((~d)^(d&h&i)));
end
end//

//-----
//File Name:dag_test.v
//Description: Testbench for Directed acyclic graph problem
//-----

```

```

module dag_tb(f1,clk,reset);
input clk,reset;
output f1;
reg (8:0)A;//temporary variable

//Instantiation
dag
dag_inst(.f1(f1),.a(A(8)),.b(A(7)),.c(A(6)),.d(A(5)),.e(A(4)),.f(A(3)),.g(A(2)),.h(A(1)),.i(A(0)),.clk(clk),.reset(reset));

always@(posedge clk)
begin
if(reset)
begin
A <= 9'b0;
end
else
begin
if(A == 9'b1)
A <= 0;
else
A <= A+1;
end
end//always

endmodule
for Directed acyclic graph problem
//-----
module dag_tb(f1,clk,reset);
input clk,reset;
output f1;
reg (8:0)A;//temporary variable

//Instantiation
dag
dag_inst(.f1(f1),.a(A(8)),.b(A(7)),.c(A(6)),.d(A(5)),.e(A(4)),.f(A(3)),.g(A(2)),.h(A(1)),.i(A(0)),.clk(clk),.reset(reset));

always@(posedge clk)
begin
if(reset)
begin
A <= 9'b0;

```

## Chapter 7 Solving the Unate and Binate Covering Problems

Unate covering problem finds applications in test generation, minimization of Boolean functions and many others. Binate covering problems are used in many aspects of logic circuit minimization and in state machine minimization problem.

### 7.1 Solving the Unate Covering Problem

Given is a function from Figure 2.2.1. All its prime implicants are marked as ovals (loops). Using the minterm compatibility graph  $G$  all primes are found as maximum cliques. They can be also found as maximum independent sets of graph  $\bar{G}$  ( $G$  complement). Based on KMap and primes we can create the covering table from.

ab \ cd	00	01	11	10
00	0	1	0	0
01	0	1	1	1
11	1	1	1	0
10	0	0	1	0

**Table 7.1 Finding graphically all prime implicants for minimal covering of a SOP circuit**

		0001	0101	0111	0110	1100	1101	1111	1011
A	$\bar{a}\bar{c}\bar{d}$	X	X						
B	$\bar{a}bc$			X	X				
C	$acd$							X	X
D	$ab\bar{c}$					X	X		
E	$bd$		X	X			X	X	

**Table 7.2 Covering table for function**

From the table, denoting rows A, B, C, D, E we compile the Petrick function in a standard way:

$$1 = A \cdot (A + E) \cdot (B + E) \cdot B \cdot D(D + E) \cdot (C + E) \cdot C$$

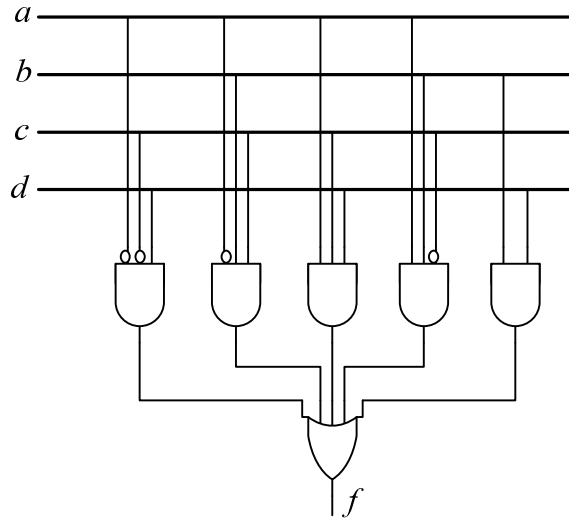
This function can be simplified using the Boolean law as follows.

$$A \cdot (A + E) = A \cdot A + A \cdot E = A(1 + E) = A$$

Therefore,

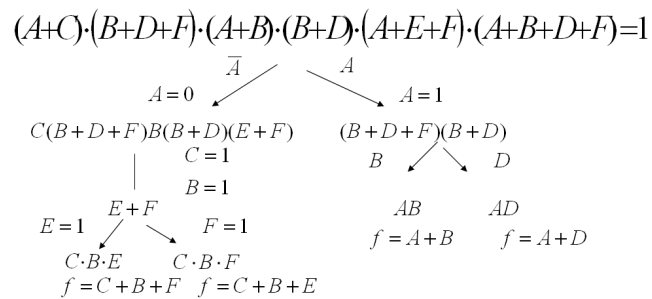
$$f = A + B + C + D = \bar{a}\bar{c}\bar{d} + \bar{a}bc + acd + ab\bar{c} + bd$$





**Figure 7-1 Oracle for the function**  $f = \bar{a}\bar{c}\bar{d} + \bar{a}bc + acd + ab\bar{c} + bd$

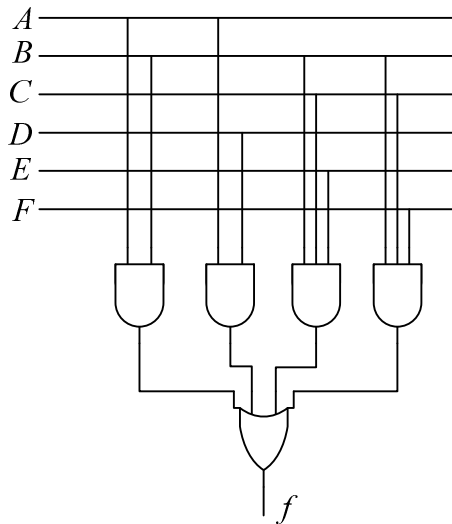
Another search method for (another)unate covering table from figure 7.1 is illustrated in Figure 7-2. It shows the branching tree for the unate covering problem from. All leafs are solutions, as showed in Figure 7-2. Both these methods can be use to build search oracles, as well as hybrid parallel searches.



**Figure 7-2 Solving the Petrick Function from the unate covering table 7-2.**

A	X		X			X
B		X	X	X		X
C	X					
D		X		X		X
E					X	
F		X			X	X

**Table 7.3 Example of an unate covering problem represented by a table**



**Figure 7-3 Oracle for the function  $f=(AB+AD+CBF+CBE)$ . The circuit is not optimized.**

```
//-----
//File name: unate1.v
//Description: Verilog code for Unate covering problem1
//-----
module unate1(f,a,b,c,d,clk,reset);
input clk,reset;
input a,b,c,d;
output f;
reg f;

always@(posedge clk)
begin
if(reset)
begin
f<=0;
end
else
f <= (((~a)&(~c)&(~d))|((~a)&b&c)|(a&c&d)|(a&b&(~c))|(b&d));
end
end
```

```

endmodule

//-----
//File Name:unate1_tb.v
//Description: Testbench for Unate covering problem1
//-----
module unate1_tb(f,clk,reset);
input clk,reset;
output f;
reg (3:0)A;//temperory variable

//Instatiation
unate1 unate1_inst(.f(f),.a(A(3)),.b(A(2)),.c(A(1)),.d(A(0)),.clk(clk),.reset(reset));

always@(posedge clk)
begin
if(reset)
begin
A <= 4'b0;
end
else
begin
if(A == 4'b1111)
A <= 0;
else
A <= A+4'b0001;
end
end//always

endmodule
e Name:unate1_tb.v
//Description: Testbench for Unate covering problem1
//-----
module unate1_tb(f,clk,reset);
input clk,reset;
output f;
reg (3:0)A;//temperory variable

//Instatiation
unate1 unate1_inst(.f(f),.a(A(3)),.b(A(2)),.c(A(1)),.d(A(0)),.clk(clk),.reset(reset));

always@(posedge clk)
begin
if(reset)
begin
A <= 4'b0;
end
else
begin
if(A == 4'b1111)
A <= 0;
els

```

```
//-----
//File name: unate2.v
//Description: Verilog code for Unate covering problem2
//-----
module unate2(f1,a,b,c,d,e,f,clk,reset);
input clk,reset;
input a,b,c,d,e,f;
output f1;
reg f1;

always@(posedge clk)
begin
if(reset)
begin
f1 <=0;
end
else
f1 <= ((a|c)&(b|d|f)&(a|b)&(b|d)&(a|e|f)&(a|b|d|f));
end

endmodule

-----
//-----
//File Name:unate2_tb.v
//Description: Testbench for Unate covering problem2
//-----
module unate2_tb(f1,clk,reset);
input clk,reset;
output f1;
reg (5:0)A;//temporary variable

//Instantiation
unate2 unate2_inst(.f1(f1),.a(A(5)),.b(A(4)),.c(A(3)),.d(A(2)),.e(A(1)),.f(A(0)),.clk(clk),.reset(reset));

always@(posedge clk)
begin
if(reset)
begin
A <= 6'b0;
end
else
begin
if(A == 6'b1)
A <= 0;
else
A <= A+1;
end
end//always

endmodule
```

```

_tb.v
//Description: Testbench for Unate covering problem2
//-----
module unate2_tb(f1,clk,reset);
input clk,reset;
output f1;
reg (5:0)A;//temporary variable

//Instantiation
unate2 unate2_inst(.f1(f1),.a(A(5)),.b(A(4)),.c(A(3)),.d(A(2)),.e(A(1)),.f(A(0)),.clk(clk),.reset(reset));

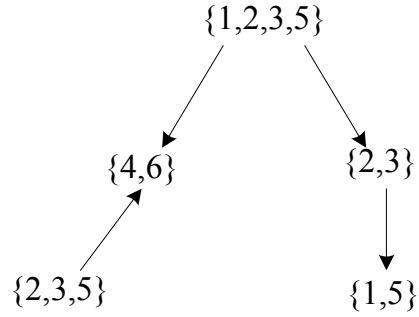
always@(posedge clk)
begin
if(reset)
begin
A <= 6'b0;
end
else
begin
if(A == 6'b1)

```

## 7.2 Solving the Binate Covering Problem

Binate covering problem was also known as “Covering with closures”. This is because in binate covering problem, variables in the function appears both in positive form and negative form. Unate covering always has a solution whereas binate covering may or may not have a solution. Binate covering problem is used in artificial intelligence, state machine minimization, technology mapping and Boolean relations.

One example of binate covering problem is in Figure 7-4 Implication graph for the binate covering problem. The covering-closure table for the implication graph from Figure 7-4 is shown in Table 7.2



**Figure 7-4 Implication graph for the binate covering problem**

	Covering						Closure				
	1	2	3	4	5	6	A {1,2,3,5}	B {4,6}	C {2,3}	D {1,5}	E {2,3,5}
A {1,2,3,5}	X	X	X		X						
B {4,6}				X		X	•				•
C {2,3}		X	X				•				
D {1,5}	X				X				•		
E {2,3,5}		X	X		X						

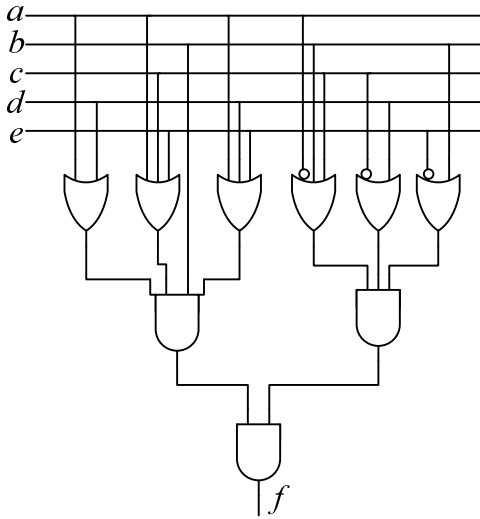
**Table 7.4 Covering closure table for the implication graph**

The equation for the binate covering problem is evaluated as

$$\underbrace{(A + D)(A + C + E)B(A + D + E)}_{\text{Covering}} \underbrace{(A \rightarrow B + C)(C \rightarrow D)(E \rightarrow B)}_{\text{Closure}} = 1$$

The function can be simplified using the logic transformation rule  $(A \rightarrow B) \Leftrightarrow (\bar{A} + B)$  as

$$f = (A + D)(A + C + E)B(A + D + E)(\bar{A} + B + C)(\bar{C} + D)(\bar{E} + B) = 1$$



**Figure 7-5 Oracle for binate covering problem**

$$f = (A + D)(A + C + E)B(A + D + E)(\bar{A} + B + C)(\bar{C} + D)(\bar{E} + B).$$

Given is a finite state machine in figure 7.6(a) A triangular table in 7.6(b) was generated covering all possible cases to minimize the number of states in the state machine. The ‘X’ symbol in the table indicates there is no possibility for grouping the corresponding states, ‘V’ indicates that the states can be combined without any problem and variables in the table indicates that states can be grouped only if the states mentioned in the block can be combined without any problem. Based on the triangular table, a compatibility graph for the state machine can be generated shown in

	00	01	11	10
A	A/0	B/0	C/0	D/1
B	A/1	B/1	C/0	D/-
C	C/0	-	C/0	-
D	-	A/0	-	D/-
E	D/-	B/0	-	-
F	-	-	F/0	D/-

(a)

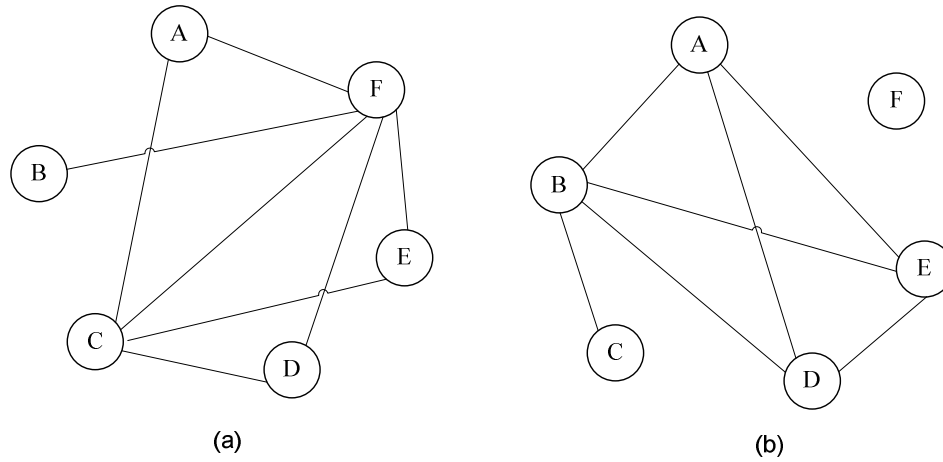
B					
C	A,C				
D	A,B		V		
E	A,D		C,D	A,B	
F	C,F	C,F	C,F	V	V
	A	B	C	D	E

(b)

**Figure 7-6 (a) Finite state machine (b) Triangular table indicating the compatibility for combining different states in the FSM**

To minimize the state machine, a covering-closure table shown in Figure 7.6 is created by considering the maximum cliques and all its subsets as rows and all the states in the machine and the implications in the compatibility graph as columns.





**Figure 7-7 (a) Compatibility graph for the FSM (b) Incompatibility graph for the FSM**

	Covering						Closure		
	A	B	C	D	E	F	V {A,C}	Q {C,F}	R {C,D}
X {A,C,F}	X		X			X	●	●	
Y {B,F}		X				X		●	
Z {C,D,F}			X	X		X		●	
U {C,E,F}			X		X	X		●	●
V {A,C}	X		X				●		
P {A,F}	X					X		●	
Q {C,F}			X			X		●	
R {C,D}			X	X					
S {D,F}				X		X			
T {C,E}			X		X				●
W {E,F}					X	X			

**Table 7.5 Covering closure table for the FSM**

From the table, binate covering problem can be specified using the equation

$$\begin{aligned}
 &(X + V + P)Y(X + Z + U + V + Q + R + T)(Z + R + S)(U + T + W) \\
 &(X + Y + Z + U + P + Q + S + W)(X \rightarrow V.Q)(Y \rightarrow Q)(Z \rightarrow Q) \\
 &(U \rightarrow Q.R)(V \rightarrow V)(P \rightarrow Q)(Q \rightarrow Q)(T \rightarrow R) = 1
 \end{aligned}$$

The function can be simplified using the Boolean laws  $(A \rightarrow B = \bar{A} + B)$  and  $A.(A+E) = A.A + A.E = A(I+E) = A$ .

The minimized state machine is shown in Figure 7-8

	00	01	11	10
X {A,C}	X/0	Y/0	Z,V, X/0	Z/1
Y {B,F}	X/1	Y/1	Z,V, X/0	Z/-
Z {C,D}	Z,V, X/0	X/0	Z,V, X/0	Z/-
V {C,E,F}	Z/0	Y/0	V/0	Z/-

**Figure 7-8 Minimized FSM using binate covering problem.**

## Chapter 8 Specialized Processor for Man Wold Goat and Cabbage Constraint Satisfaction Sequential Problem

### 8.1 Introduction and goals

A man has a wolf, a goat, and a cabbage. He must cross a river with the two animals and the cabbage. There is a small rowing-boat, in which he can take only one thing with him at a time. If, however, the wolf and the goat are left alone, the wolf will eat the goat. If the goat and the cabbage are left alone, the goat will eat the cabbage.

How can the man get across the river with the two animals and the cabbage?

Rules for the transport

1. Man has to present in every move.
2. Not more than 2 objects are can move in any move.

There are two solutions to this. These two solutions can be seen in flow graph as follows.

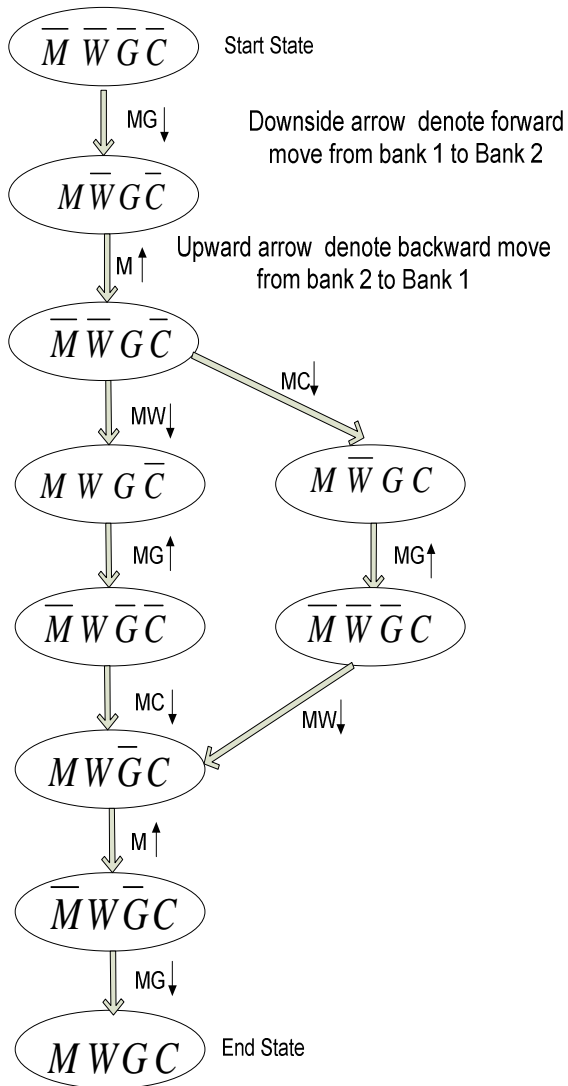
#### **Solution 1:**

1. Man takes the goat across the river, from bank 1 to bank 2,
2. Leaving the wolf and the cabbage at 1.
3. Man returns empty to pick up the wolf from bank 1 and takes it to 2, where he leaves it and picks up the goat and bring it back to bank 1.
4. Man takes Cabbage to bank 2 and return empty handed back to bank 1 leaving wolf and cabbage to bank 2.
5. Now Man takes goat from bank 1 to bank 2 and all now man get across the river with 2 animals and cabbage.

**Solution 2:**

1. Man takes the goat across the river, from bank 1 to bank 2,
2. Leaving the wolf and the cabbage at 1 and returns empty to pick up the cabbage from bank 1 and takes it to 2, where he leaves it and picks up the goat and bring it back to
3. bank 1.
4. Man takes wolf from bank 1 to bank 2 and and return empty handed back to bank 1 leaving wolf and cabbage to bank 2.
5. Now Man takes goat from bank 1 to bank 2 and all now man get across the river with 2 animals and cabbage.

## Flow Chart For taking man , 2 animals and cabbage to other Bank



Negated literal denote object is at bank 1

Possitive literal denote object is at bank 2

### 8.1 Checking Safe Condition at Both Bank

We can find the safe condition from unsafe condition. Because state expression for unsafe condition is simple and we can simply negate this expression in order to get safe state expression.

Unsafe condition at bank 1 is when Man is not there but wolf and goat are there  $M \bar{W} \bar{G}$

Or Man is not there and goat and cabbage are there  $M \bar{G} \bar{C}$

So Unsafe condition at bank 1 is  $M \bar{W} \bar{G} + M \bar{G} \bar{C}$

Similarly Un-safe condition at bank 2  $\bar{M} W G + \bar{M} G C$

Unsafe condition at bank 1 and 2 is  $\bar{M} G (W + C) + M \bar{G} (\bar{W} + \bar{C})$

Safe state equation will be negation of above expression.

Safe condition at both banks

$$\overline{\bar{M} G (W + C) + M \bar{G} (\bar{W} + \bar{C})} \dots\dots\dots \text{Eq. 1}$$

## 8.2 Checking for invalid move

Cheating or invalid move will be the move with

A move in which man does not move at all

A move in which man but more than 2 objects also moves with him.

Let's take a first case,

When man doesn't move ex-or of two variable representing man in two successive stage is 0

If  $M_n$  is variable representing man in stage n and  $M_{n+1}$  is variable representing man in stage n+1

then

$$M_n \oplus M_{n+1} = 0$$

Here subscript n denote states

So in order to check whether man is moving between successive move or not , the above equation should be negated and should be checked for the value 1

That means check for

$$\overline{M_n \oplus M_{n+1}} = 1$$

When  $M_n$  denote man variable in nth state and  $M_{n+1}$  represent man variable in n+1 state.

Final equation for checking invalid move of man does not moving will be Oring operation of all moves between successive stages.

Now for the 2<sup>nd</sup> condition of invalid move in which man moves but more than 2 object moves with him.

When object moves between successive states, the ex-or operation between the literal representing that objet is 1.

This means when goat is moving

$$G_n \oplus G_{n+1} = 1$$

When Cabbage is moving

$$C_n \oplus C_{n+1} = 1$$

When wolf is moving then

$$W_n \oplus W_{n+1} = 1$$

When man is moving then

$$M_n \oplus M_{n+1} = 1$$

$$\text{Let } Z1 = M_n \oplus M_{n+1}$$

$$\text{Let } Z2 = W_n \oplus W_{n+1}$$

$$\text{Let } Z3 = C_n \oplus C_{n+1}$$

$$\text{Let } Z4 = G_n \oplus G_{n+1}$$

So when man moves but wolf and cabbage also moves with him then

$$Z1 \bullet Z2 \bullet Z3 = 1$$

Similarly when wolf and goat moves with man then  $Z1 \bullet Z2 \bullet Z4 = 1$ .

and when goat and cabbage moves with man then  $Z1 \bullet Z2 \bullet Z3 = 1$

so final equation for bad or invalid move is

$$Z1 \bullet Z2 \bullet Z3 + Z1 \bullet Z2 \bullet Z4 + Z1 \bullet Z3 \bullet Z4 = 1$$

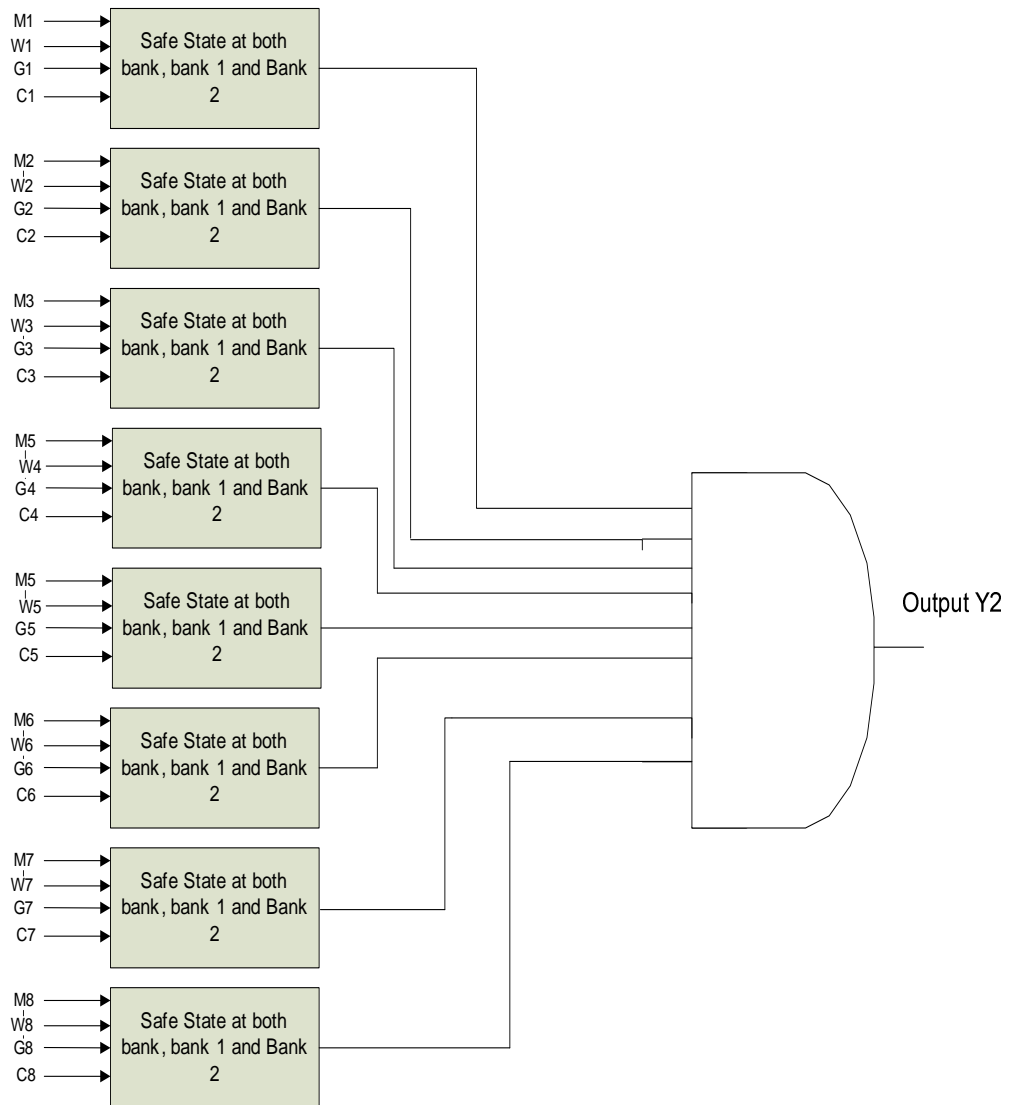
in summary, For any move to be invalid,

$$\overline{M_n \oplus M_{n+1}} + Z1 \bullet Z2 \bullet Z3 + Z1 \bullet Z2 \bullet Z4 + Z1 \bullet Z3 \bullet Z4 = 1$$

.....Eq 2

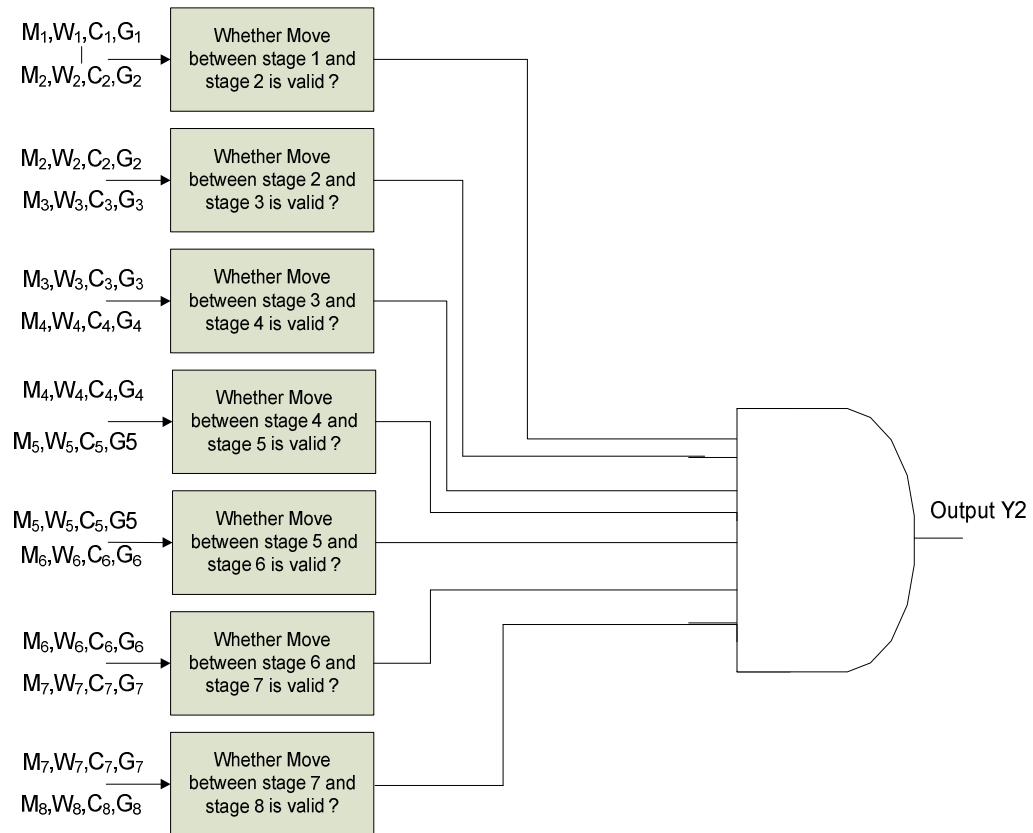
The Final oracle will be implementation of Eq 1 and 2





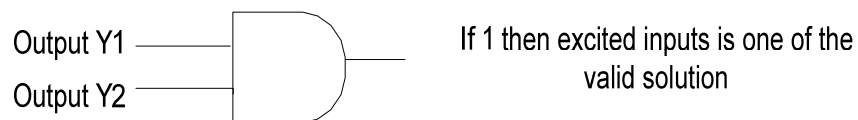
Oracle for Checking invalid state

**Figure 8-1 Oracles for checking invalid state**



**Figure 8-2 Oracle for checking invalid move**

Final oracle will be AND operation between Output Y1 and Output Y2.



**Figure 8-3 Final Oracle**

#### VHDL Code for State Checking

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

entity check_state is
port(Man,Wolf, Goat, Cabbage :in std_logic;
      Valid_state : out std_logic);
end check_state;
```

```

architecture Behavioural of check_state is
    signal Man_bar,Wolf_bar,Cabbage_bar,Goat_bar,temp1,temp2,temp3,temp4,invalid_state : std_logic;

begin
    --- Implementing equation for state checking
    Man_bar <= not Man;
    Wolf_bar <= not Wolf;
    Cabbage_bar <= not Cabbage;
    Goat_bar <= not Goat;

    temp1 <= Man_bar and Goat;
    temp2 <= Wolf or Cabbage;
    temp3 <= Man and Goat_bar;
    temp4 <= Wolf_bar or Goat_bar;

    invalid_state <= (temp1 and temp2) or (temp3 and temp4);
    Valid_state <= not invalid_state;

end behavioural;

```

### 8.3 VHDL Code for Move Checking

```

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

entity check_move is
port(Man_ps,Wolf_ps,Cabbage_ps,Goat_ps :in std_logic;
     Man_ns,Wolf_ns,Cabbage_ns,Goat_ns :in std_logic;
     Valid_move : out std_logic);
end check_move;

```

```

architecture Behavioural of check_move is

signal Z1,Z2,Z3,Z4,invalid_move : std_logic;

begin

```

```

    -- Checkin validity of move 1

    Z1 <= Man_ps xor Man_ns; -----MAN MOVES
    Z2 <= Wolf_ps xor Wolf_ns;
    Z3 <= Cabbage_ps xor Cabbage_ns;
    Z4 <= Goat_ps xor Goat_ns;

```

```

invalid_move <= (not Z1) or (Z1 and Z2 and Z3) or ( Z1 and Z2 and Z4) or (Z1 and Z3 and Z4);--- tells
wheather move from present state to next state is valid or not
Valid_move <= invalid_move;

end Behavioural;

```

## 8.4 VHDL code for Final Oracle

```

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

entity oracle is
port(M1_state1,M1_state2,M1_state3,M1_state4,M1_state5,M1_state6,M1_state7,M1_state8,M1_state9,M
1_state10,M1_state11,M1_state12 :in std_logic;

M2_state1,M2_state2,M2_state3,M2_state4,M2_state5,M2_state6,M2_state7,M2_state8,M2_state9,M2_st
ate10,M2_state11,M2_state12 :in std_logic;

M3_state1,M3_state2,M3_state3,M3_state4,M3_state5,M3_state6,M3_state7,M3_state8,M3_state9,M3_st
ate10,M3_state11,M3_state12 :in std_logic;

C1_state1,C1_state2,C1_state3,C1_state4,C1_state5,C1_state6,C1_state7,C1_state8,C1_state9,C1_state10,
C1_state11,C1_state12 :in std_logic;

C2_state1,C2_state2,C2_state3,C2_state4,C2_state5,C2_state6,C2_state7,C2_state8,C2_state9,C2_state10,
C2_state11,C2_state12 :in std_logic;

C3_state1,C3_state2,C3_state3,C3_state4,C3_state5,C3_state6,C3_state7,C3_state8,C3_state9,C3_state10,
C3_state11,C3_state12 :in std_logic;
Valid_state : out std_logic);
end oracle;

architecture structural of oracle is

component check_state
port(M1,M2,M3,C1,C2,C3:in std_logic;
Valid_state : out std_logic);
end component;

component check_move
port(M1_ps,M2_ps,M3_ps,C1_ps,C2_ps,C3_ps :in std_logic;
M1_ns,M2_ns,M3_ns,C1_ns,C2_ns,C3_ns :in std_logic;
valid_move : out std_logic);
end component;

begin

```

```

State_1                :                check_state                port
map(M1=>M1_state1,M2=>M2_state1,M3=>M3_state1,C1=>C1_state1,C2=>C2_state1,C3=>C3_state1);
State_2                :                check_state                port
map(M1=>M1_state2,M2=>M2_state2,M3=>M3_state2,C1=>C1_state2,C2=>C2_state2,C3=>C3_state2);
State_3                :                check_state                port
map(M1=>M1_state3,M2=>M2_state3,M3=>M3_state3,C1=>C1_state3,C2=>C2_state3,C3=>C3_state3);
State_4                :                check_state                port
map(M1=>M1_state4,M2=>M2_state4,M3=>M3_state4,C1=>C1_state4,C2=>C2_state4,C3=>C3_state4);
State_5                :                check_state                port
map(M1=>M1_state5,M2=>M2_state5,M3=>M3_state5,C1=>C1_state5,C2=>C2_state5,C3=>C3_state5);
State_6                :                check_state                port
map(M1=>M1_state6,M2=>M2_state6,M3=>M3_state6,C1=>C1_state6,C2=>C2_state6,C3=>C3_state6);
State_7                :                check_state                port
map(M1=>M1_state7,M2=>M2_state7,M3=>M3_state7,C1=>C1_state7,C2=>C2_state7,C3=>C3_state7);
State_8                :                check_state                port
map(M1=>M1_state8,M2=>M2_state8,M3=>M3_state8,C1=>C1_state8,C2=>C2_state8,C3=>C3_state8);
State_9                :                check_state                port
map(M1=>M1_state9,M2=>M2_state9,M3=>M3_state9,C1=>C1_state9,C2=>C2_state9,C3=>C3_state9);
State_10               :                check_state                port
map(M1=>M1_state10,M2=>M2_state10,M3=>M3_state10,C1=>C1_state10,C2=>C2_state10,C3=>C3_s
tate10);
State_11               :                check_state                port
map(M1=>M1_state11,M2=>M2_state11,M3=>M3_state11,C1=>C1_state11,C2=>C2_state11,C3=>C3_s
tate11);
State_12               :                check_state                port
map(M1=>M1_state12,M2=>M2_state12,M3=>M3_state12,C1=>C1_state12,C2=>C2_state12,C3=>C3_s
tate12);

Move_1                 :                check_move                port                map
(M1_ps=>M1_state1,M2_ps=>M2_state1,M3_ps=>M3_state1,C1_ps=>C1_state1,C2_ps=>C2_state1,C3_
ps=>C3_state1,

M1_ns=>M1_state2,M2_ns=>M2_state2,M3_ns=>M3_state2,C1_ns=>C1_state2,C2_ns=>C2_state2,C3_n
s=>C3_state2);

Move_2                 :                check_move                port                map
(M1_ps=>M1_state2,M2_ps=>M2_state2,M3_ps=>M3_state2,C1_ps=>C1_state2,C2_ps=>C2_state2,C3_
ps=>C3_state2,

M1_ns=>M1_state3,M2_ns=>M2_state3,M3_ns=>M3_state3,C1_ns=>C1_state3,C2_ns=>C2_state3,C3_n
s=>C3_state3);

Move_3                 :                check_move                port                map
(M1_ps=>M1_state3,M2_ps=>M2_state3,M3_ps=>M3_state3,C1_ps=>C1_state3,C2_ps=>C2_state3,C3_
ps=>C3_state3,

M1_ns=>M1_state4,M2_ns=>M2_state4,M3_ns=>M3_state4,C1_ns=>C1_state4,C2_ns=>C2_state4,C3_n
s=>C3_state4);

Move_4                 :                check_move                port                map
(M1_ps=>M1_state4,M2_ps=>M2_state4,M3_ps=>M3_state4,C1_ps=>C1_state4,C2_ps=>C2_state4,C3_
ps=>C3_state4,

```

```
M1_ns=>M1_state5,M2_ns=>M2_state5,M3_ns=>M3_state5,C1_ns=>C1_state5,C2_ns=>C2_state5,C3_ns=>C3_state5);
```

```
Move_5 : check_move port map
(M1_ps=>M1_state5,M2_ps=>M2_state5,M3_ps=>M3_state5,C1_ps=>C1_state5,C2_ps=>C2_state5,C3_ps=>C3_state5,
```

```
M1_ns=>M1_state6,M2_ns=>M2_state6,M3_ns=>M3_state6,C1_ns=>C1_state6,C2_ns=>C2_state6,C3_ns=>C3_state6);
```

```
Move_6 : check_move port map
(M1_ps=>M1_state6,M2_ps=>M2_state6,M3_ps=>M3_state6,C1_ps=>C1_state6,C2_ps=>C2_state6,C3_ps=>C3_state6,
```

```
M1_ns=>M1_state7,M2_ns=>M2_state7,M3_ns=>M3_state7,C1_ns=>C1_state7,C2_ns=>C2_state7,C3_ns=>C3_state7);
```

```
Move_7 : check_move port map
(M1_ps=>M1_state7,M2_ps=>M2_state7,M3_ps=>M3_state7,C1_ps=>C1_state7,C2_ps=>C2_state7,C3_ps=>C3_state7,
```

```
M1_ns=>M1_state8,M2_ns=>M2_state8,M3_ns=>M3_state8,C1_ns=>C1_state8,C2_ns=>C2_state8,C3_ns=>C3_state8);
```

```
Move_8 : check_move port map
(M1_ps=>M1_state8,M2_ps=>M2_state8,M3_ps=>M3_state8,C1_ps=>C1_state8,C2_ps=>C2_state8,C3_ps=>C3_state8,
```

```
M1_ns=>M1_state9,M2_ns=>M2_state9,M3_ns=>M3_state9,C1_ns=>C1_state9,C2_ns=>C2_state9,C3_ns=>C3_state9);
```

```
Move_9 : check_move port map
(M1_ps=>M1_state9,M2_ps=>M2_state9,M3_ps=>M3_state9,C1_ps=>C1_state9,C2_ps=>C2_state9,C3_ps=>C3_state9,
```

```
M1_ns=>M1_state10,M2_ns=>M2_state10,M3_ns=>M3_state10,C1_ns=>C1_state10,C2_ns=>C2_state10,C3_ns=>C3_state10);
```

```
Move_10 : check_move port map
(M1_ps=>M1_state10,M2_ps=>M2_state10,M3_ps=>M3_state10,C1_ps=>C1_state10,C2_ps=>C2_state10,C3_ps=>C3_state10,
```

```
M1_ns=>M1_state11,M2_ns=>M2_state11,M3_ns=>M3_state11,C1_ns=>C1_state11,C2_ns=>C2_state11,C3_ns=>C3_state11);
```

```
Move_11 : check_move port map
(M1_ps=>M1_state11,M2_ps=>M2_state11,M3_ps=>M3_state11,C1_ps=>C1_state11,C2_ps=>C2_state11,C3_ps=>C3_state11,
```

```
M1_ns=>M1_state12,M2_ns=>M2_state12,M3_ns=>M3_state12,C1_ns=>C1_state12,C2_ns=>C2_state12,C3_ns=>C3_state12);
```

```
end structural;
```

## 8.5 Test bench for testing Oracle and finding solution

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_textio.all;
use std.textio.all;

entity oracle_test is
end oracle_test;

architecture beha of oracle_test is -- Sequential Modelling Style

component final_oracle
port(M1,M2,M3,M4,M5,M6,M7,M8 :in std_logic;
     W1,W2,W3,W4,W5,W6,W7,W8 :in std_logic;
     G1,G2,G3,G4,G5,G6,G7,G8 :in std_logic;
     C1,C2,C3,C4,C5,C6,C7,C8 :in std_logic;
     Valid_solution:out std_logic);
end component;

signal M1,M2,M3,M4,M5,M6,M7,M8 :std_logic;
signal W1,W2,W3,W4,W5,W6,W7,W8 :std_logic;
signal G1,G2,G3,G4,G5,G6,G7,G8 :std_logic;
signal C1,C2,C3,C4,C5,C6,C7,C8 :std_logic;
signal temp_1,temp_2,temp_3,temp_4 : std_logic_vector(7 downto 0):=(others=>'0');
signal Valid_solution :std_logic;

file outfile : text open WRITE_MODE is "//khensu/Home04/gunavant/Desktop/output.txt";

procedure write_result is
variable buff : line;
begin
write(buff,string("Solution found to be "));
writeline (outfile, buff);

write(buff,string("M1 = "));
write(buff,M1);
write(buff,string("M2 = "));
write(buff,M2);
write(buff,string("M3 = "));
write(buff,M3);
write(buff,string("M4 = "));
write(buff,M4);
write(buff,string("M5 = "));
write(buff,M5);
write(buff,string("M6 = "));
```

```

write(buff,M6);
write(buff,string("M7 = "));
write(buff,M7);
write(buff,string("M8 = "));
write(buff,M8);
writeline (outfile, buff);

```

```

write(buff,string("W1 = "));
write(buff,W1);
write(buff,string("W2 = "));
write(buff,W2);
write(buff,string("W3 = "));
write(buff,W3);
write(buff,string("W4 = "));
write(buff,W4);
write(buff,string("W5 = "));
write(buff,W5);
write(buff,string("W6 = "));
write(buff,W6);
write(buff,string("W7 = "));
write(buff,W7);
write(buff,string("W8 = "));
write(buff,W8);
writeline (outfile, buff);

```

```

write(buff,string("C1 = "));
write(buff,C1);
write(buff,string("C2 = "));
write(buff,C2);
write(buff,string("C3 = "));
write(buff,C3);
write(buff,string("C4 = "));
write(buff,C4);
write(buff,string("C5 = "));
write(buff,C5);
write(buff,string("C6 = "));
write(buff,C6);
write(buff,string("C7 = "));
write(buff,C7);
write(buff,string("C8 = "));
write(buff,C8);
writeline (outfile, buff);

```

```

write(buff,string("G1 = "));
write(buff,G1);
write(buff,string("G2 = "));
write(buff,G2);
write(buff,string("G3 = "));
write(buff,G3);
write(buff,string("G4 = "));
write(buff,G4);
write(buff,string("G5 = "));

```



```

write(buff,G5);
write(buff,string("G6 = "));
write(buff,G6);
write(buff,string("G7 = "));
write(buff,G7);
write(buff,string("G8 = "));
write(buff,G8);
writeline (outfile, buff);
end procedure write_result;

```

```

begin

```

```

u                                     :               final_oracle                                     port
map(M1=>M1,M2=>M2,M3=>M3,M4=>M4,M5=>M5,M6=>M6,M7=>M7,M8=>M8,
    G1=>G1,G2=>G2,G3=>G3,G4=>G4,G5=>G5,G6=>G6,G7=>G7,G8=>G8,
    C1=>C1,C2=>C2,C3=>C3,C4=>C4,C5=>C5,C6=>C6,C7=>C7,C8=>C8,
    W1=>W1,W2=>W2,W3=>W3,W4=>W4,W5=>W5,W6=>W6,W7=>W7,W8=>W8,
    Valid_solution =>Valid_solution);

```

```

P1 : process

```

```

file outfile : text open WRITE_MODE is "//khensu/Home04/gunavant/Desktop/output.txt";
variable buff : line;

```

```

begin

```

```

L1: for k in 0 to 255 loop
    temp_1 <= temp_1+ "00000001";

```

```

    L2: for m in 0 to 255 loop
        temp_2 <= temp_2+ "00000001";

```

```

        L3: for n in 0 to 255 loop
            temp_3 <= temp_3+ "00000001";

```

```

            L4: for p in 0 to 255 loop
                temp_4 <= temp_4+ "00000001";

```

```

                W1<=temp_2(0);   W2<=temp_2(1);   W3<=temp_2(2);   W4<=temp_2(3);   W5<=temp_2(4);
W6<=temp_2(5); W7<=temp_2(6); W8<=temp_2(7);
                M1<=temp_1(0);   M2<=temp_1(1);   M3<=temp_1(2);   M4<=temp_1(3);   M5<=temp_1(4);
M6<=temp_1(5); M7<=temp_1(6); M8<=temp_1(7);
                G1<=temp_3(0);   G2<=temp_3(1);   G3<=temp_3(2);   G4<=temp_3(3);   G5<=temp_3(4);
G6<=temp_3(5); G7<=temp_3(6); G8<=temp_3(7);
                C1<=temp_4(0);   C2<=temp_4(1);   C3<=temp_4(2);   C4<=temp_4(3);   C5<=temp_4(4);
C6<=temp_4(5); C7<=temp_4(6); C8<=temp_4(7);

```

```

wait for 5 ns;

```

```

if(Valid_solution= '1') then
    write_result;

```

```

end if;          ----- This is procedural call to write results in file

end loop L4;
end loop L3;
end loop L2;
end loop L1;
wait;
end process;
end beha;

```

## 8.6 The Output file contents

Solution found to be:

```

M1 = 0 M2 = 1 M3 = 0 M4 = 1 M5 = 0 M6 = 1 M7 =0 M8 = 1
W1 = 0 W2 = 0 W3 = 0 W4 = 1 W5 = 1 W6 = 1 W7 =1 W8 = 1
C1 = 0 C2 = 1 C3 = 1 C4 = 1 C5 = 0 C6 = 0 C7 =0 C8 = 1
G1 = 0 G2 = 0 G3 = 0 G4 = 0 G5 = 0 G6 = 1 G7 =1 G8 = 1

```

## 8.7 Emulator testbench

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_textio.all;
use std.textio.all;

entity oracle_test is
port (clock,enable,reset:in std_logic);
end oracle_test;

architecture beha of oracle_test is -- Sequential Modelling Style

component final_oracle
port(M1,M2,M3,M4,M5,M6,M7,M8 :in std_logic;
     W1,W2,W3,W4,W5,W6,W7,W8 :in std_logic;
     G1,G2,G3,G4,G5,G6,G7,G8 :in std_logic;
     C1,C2,C3,C4,C5,C6,C7,C8 :in std_logic;
     Valid_solution:out std_logic);
end component;

signal M1,M2,M3,M4,M5,M6,M7,M8 :std_logic;
signal W1,W2,W3,W4,W5,W6,W7,W8 :std_logic;
signal G1,G2,G3,G4,G5,G6,G7,G8 :std_logic;
signal C1,C2,C3,C4,C5,C6,C7,C8 :std_logic;
signal counter : std_logic_vector(31 downto 0):=(others=>'0');
signal Valid_solution ,test_vector_enable:std_logic;

```

```

begin

u                                     :                               final_oracle                               port
map(M1=>M1,M2=>M2,M3=>M3,M4=>M4,M5=>M5,M6=>M6,M7=>M7,M8=>M8,
    G1=>G1,G2=>G2,G3=>G3,G4=>G4,G5=>G5,G6=>G6,G7=>G7,G8=>G8,
    C1=>C1,C2=>C2,C3=>C3,C4=>C4,C5=>C5,C6=>C6,C7=>C7,C8=>C8,
    W1=>W1,W2=>W2,W3=>W3,W4=>W4,W5=>W5,W6=>W6,W7=>W7,W8=>W8,
    Valid_solution =>Valid_solution);

P1 : process (CLOCK)
begin
if( reset ='1') then

counter<="00000000000000000000000000000000";

else

counter<= counter+"00000000000000000000000000000001";
end if;

end process;

W1<=counter(0);  W2<=counter(1);  W3<=counter(2);  W4<=counter(3);  W5<=counter(4);
W6<=counter(5); W7<=counter(6); W8<=counter(7);
M1<=counter(8);  M2<=counter(9);  M3<=counter(10);  M4<=counter(11);  M5<=counter(12);
M6<=counter(13); M7<=counter(14); M8<=counter(15);
G1<=counter(16); G2<=counter(17); G3<=counter(18); G4<=counter(19); G5<=counter(20);
G6<=counter(21); G7<=counter(22); G8<=counter(23);
C1<=counter(24); C2<=counter(25); C3<=counter(26); C4<=counter(27); C5<=counter(28);
C6<=counter(29); C7<=counter(30); C8<=counter(31);
end beha;

```

## Chapter 9 Socialized Processor For Missionaries and Cannibals Constraint Sequential Problem

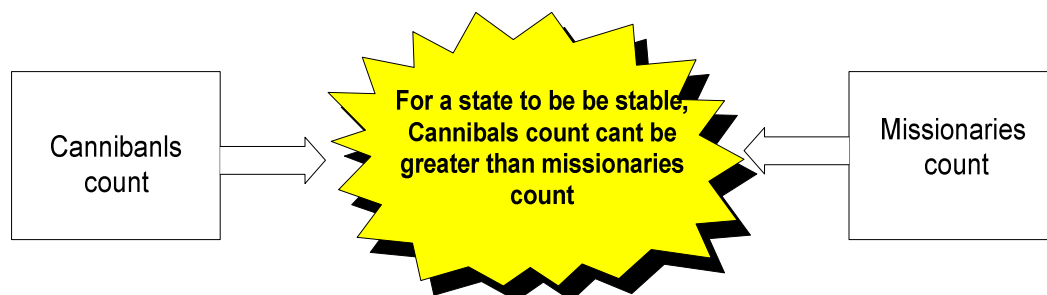
### 9.1 Problem Statement:

Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. The final goal is to get everyone to the other side, without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place.

Rules of Transport:

1. At the max only 2 object can sail through the boat.
2. Any one of the 6 objects or any combination out of 6 objects can sail through the boat.

Main satisfiability condition here is count of missionaries and cannibals. For a move to be valid, the resulting states at both bank should be valid. i.e. the number of cannibals can't be greater than number of missionaries.



Only exception to this is when missionary count is zero....  
that means all the cannibals are on one bank and all the missionareis are on another bank

So we need to have some kind of computational circuit which will tell whether the achieved state at both bank is valid or not. We also need other combinational circuits to check for the validity of move.

Let the Positive literal here means respective missionary or cannibals is on bank 1 and negated literal means respective missionary or cannibals is on bank 2.

One solution to this problem can be as shown in flowchart

Flow Chart For taking man , 2 animals and cabbage to other Bank

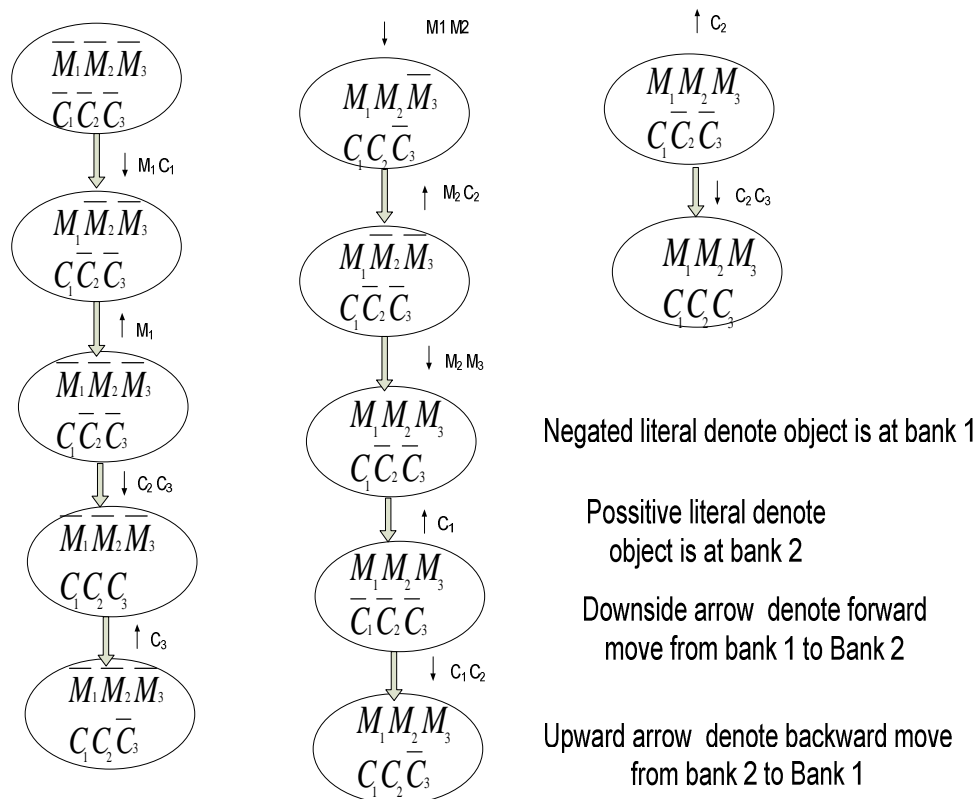


Figure 9-1 One possible solution of the puzzle

## 9.2 Checking Safe Condition at Both Bank

Lets find the logical sets of equations for safe condition at both banks

Lets have following notation for 3 cannibals and 3 missionaries

$C1$  Denotes 1<sup>st</sup> Cannibals at bank 2.

$\overline{C1}$  Denotes 1<sup>st</sup> Cannibals at bank 1.

$C2$  denotes 2<sup>nd</sup> Cannibals at bank 2

$\overline{C2}$  Denotes 2<sup>nd</sup> Cannibals at bank 1

$C3$  denotes 3<sup>rd</sup> Cannibals at bank 2

$\overline{C3}$  Denotes 3<sup>rd</sup> Cannibals at bank 1

$M1$  Denotes 1<sup>st</sup> Missionary at bank 2.

$\overline{M1}$  Denotes 1<sup>st</sup> Missionary at bank 1.

$M2$  denotes 2<sup>nd</sup> Missionary at bank 2

$\overline{M2}$  Denotes 2<sup>nd</sup> Missionary at bank 1

$M3$  denotes 3<sup>rd</sup> Missionary at bank 2

$\overline{M3}$  Denotes 3<sup>rd</sup> Missionary at bank 1

Lets us derive Valid state equation from invalid state equation. We can just negate the expression for invalid state to get valid state expression

Situation at Bank will be invalid when

This comprises of three cases

1 ) 3 Cannibals and 2 missionaries.

2 ) 3 Cannibals and 1 missionary

3) 2 Cannibals and 1 missionary

Lets take first Case of 3 Cannibals and 2 missionaries

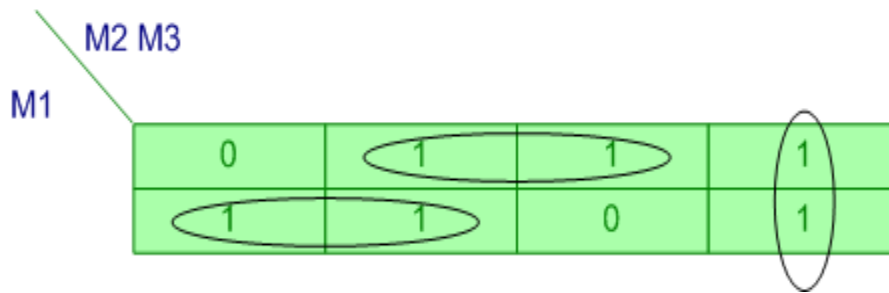
The equation for 3 Cannibals and 2 missionaries at bank 1 is

$$\overline{C_1 C_2 C_3} (\overline{M_1 M_2 M_3} + M_1 \overline{M_2 M_3} + \overline{M_1 M_2} \overline{M_3} + \overline{M_1 M_2} M_3 + M_1 \overline{M_2} M_3 + M_1 M_2 \overline{M_3})$$

Let us simplify expression

$$\overline{M_1 M_2 M_3} + M_1 \overline{M_2 M_3} + \overline{M_1 M_2} \overline{M_3} + \overline{M_1 M_2} M_3 + M_1 \overline{M_2} M_3 + M_1 M_2 \overline{M_3}$$

The k-map for this expression is



$$(\overline{M_1} M_3 + M_2 \overline{M_3} + M_1 \overline{M_2})$$

The final equation for Safe condition at bank 1 is

$$\overline{C_1 C_2 C_3} (\overline{M_1} M_3 + M_2 \overline{M_3} + M_1 \overline{M_2}) \dots \dots \dots \text{Eq A}$$

Similarly safe condition at bank 2 will be simply negation of literal in above equation

$$C_1 C_2 C_3 (M_1 \overline{M_3} + \overline{M_2} M_3 + \overline{M_1} M_2) \dots \dots \dots \text{Eq B}$$

So Un-safe condition at both bank with 3 cannibals and 2 missionaries is Eq A + Eq B

$$\overline{C_1 C_2 C_3} (\overline{M_1} M_3 + M_2 \overline{M_3} + M_1 \overline{M_2}) + C_1 C_2 C_3 (M_1 \overline{M_3} + \overline{M_2} M_3 + \overline{M_1} M_2) \dots \dots \dots \text{Eq C}$$

Lets take the 2<sup>nd</sup> case of 3 Cannibals and 1 missionary at bank 1 is

$$\overline{C_1} \overline{C_2} \overline{C_3} (\overline{M_1} M_2 M_3 + M_1 \overline{M_2} M_3 + M_1 M_2 \overline{M_3}) \dots \dots \dots \text{Eq. D}$$

Similarly the 2<sup>nd</sup> case of 3 Cannibals and 1 missionary at bank 2 is

$$C_1 C_2 C_3 (\overline{M_1} \overline{M_2} M_3 + M_1 \overline{M_2} \overline{M_3} + \overline{M_1} M_2 \overline{M_3}) \dots \dots \dots \text{Eq. E}$$

And last take 3<sup>rd</sup> case of Similarly Equation for 2 cannibals and 1 missionary at bank 1 is

$$\begin{aligned} & \overline{C_1} \overline{C_2} C_3 \overline{M_1} M_2 M_3 + \overline{C_1} \overline{C_2} C_3 M_1 \overline{M_2} M_3 + \overline{C_1} \overline{C_2} C_3 M_1 M_2 \overline{M_3} \\ & \dots \text{Eq.F} \\ & + C_1 \overline{C_2} \overline{C_3} \overline{M_1} M_2 M_3 + C_1 \overline{C_2} \overline{C_3} M_1 \overline{M_2} M_3 + C_1 \overline{C_2} \overline{C_3} M_1 M_2 \overline{M_3} \\ & + \overline{C_1} C_2 \overline{C_3} \overline{M_1} M_2 M_3 + \overline{C_1} C_2 \overline{C_3} M_1 \overline{M_2} M_3 + \overline{C_1} C_2 \overline{C_3} M_1 M_2 \overline{M_3} \end{aligned}$$

Similarly we can find invalid condition for 2 cannibals and 1 missionary at bank 2 is

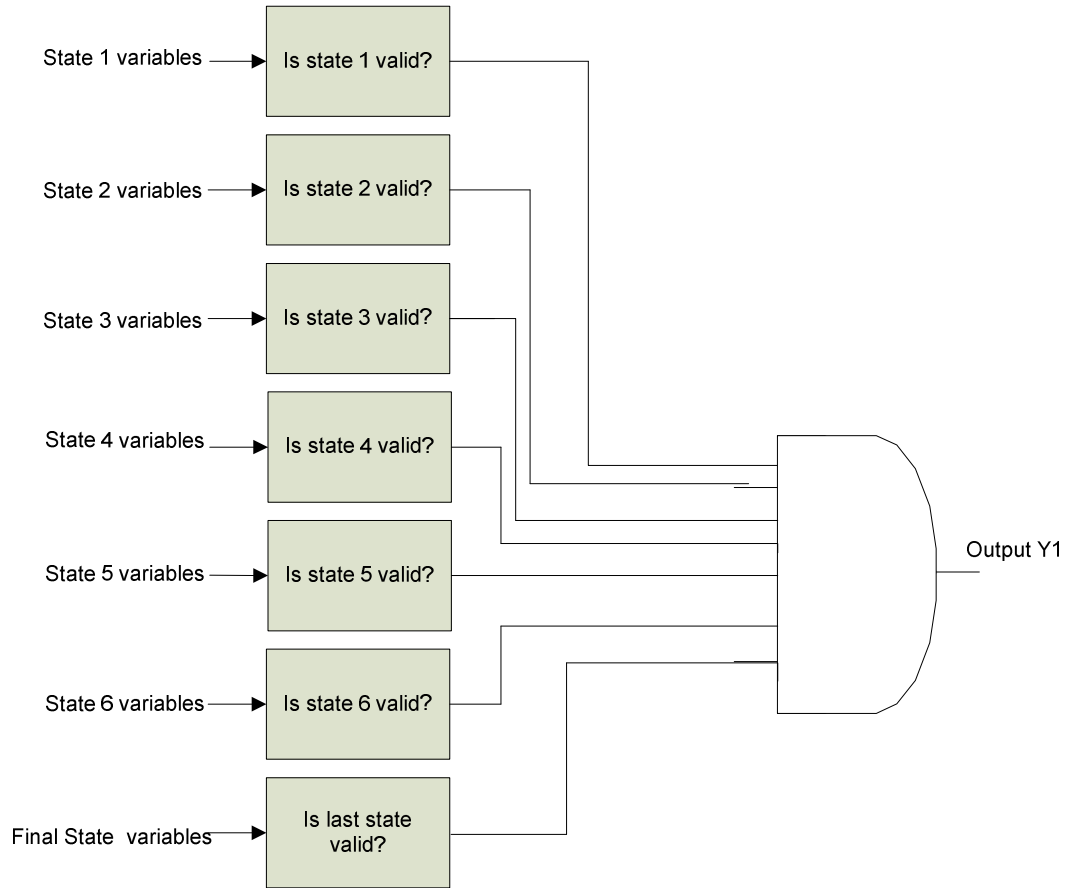
$$\begin{aligned} & \overline{C_1} C_2 C_3 \overline{M_1} \overline{M_2} M_3 + \overline{C_1} C_2 C_3 \overline{M_1} M_2 \overline{M_3} + \overline{C_1} C_2 C_3 M_1 \overline{M_2} \overline{M_3} \\ & \dots \text{Eq.G} \\ & + C_1 \overline{C_2} \overline{C_3} \overline{M_1} \overline{M_2} M_3 + C_1 \overline{C_2} \overline{C_3} \overline{M_1} M_2 \overline{M_3} + C_1 \overline{C_2} \overline{C_3} M_1 \overline{M_2} \overline{M_3} \\ & + C_1 C_2 \overline{C_3} \overline{M_1} \overline{M_2} M_3 + C_1 C_2 \overline{C_3} \overline{M_1} M_2 \overline{M_3} + C_1 C_2 \overline{C_3} M_1 \overline{M_2} \overline{M_3} \end{aligned}$$

So finally expression for Invalid condition at both banks including all possibilities will be

$$\text{Eq.C} + \text{Eq.D} + \text{Eq.E} + \text{Eq.F} + \text{Eq.G}$$

The expression for invalid condition will be negation of above expression.





**Figure 9-2 Oracle for checking invalid state**

### 9.3 Checking for invalid move

Cheating moves is when more than 2 object moves.

To simplify the equation for cheating move we can check for valid move.

A Move is valid when one only one object moves between banks or maximum two objects moves between banks.

Let's find expression for these conditions separately and later we can just logically OR them.

If  $M_1^n$  represent 1st missionary in state n when  $M_1^{n+1}$  represent 1<sup>st</sup> missionary in state n+1.

When M1 moves between two successive states then  $M_1^n \oplus M_1^{n+1} = 1$ .

When M2 moves between two successive states then  $M_2^n \oplus M_2^{n+1} = 1$ .

When M3 moves between two successive states then  $M_3^n \oplus M_3^{n+1} = 1$ .

When C1 moves between two successive states then  $C_1^n \oplus C_1^{n+1} = 1$ .

When C2 moves between two successive states then  $C_2^n \oplus C_2^{n+1} = 1$ .

When C3 moves between two successive states then  $C_3^n \oplus C_3^{n+1} = 1$ .

Let  $M_1^n \oplus M_1^{n+1} = Z1$

Let  $M_2^n \oplus M_2^{n+1} = Z2$

Let  $M_3^n \oplus M_3^{n+1} = Z3$

Let  $C_1^n \oplus C_1^{n+1} = Z4$

Let  $C_2^n \oplus C_2^{n+1} = Z5$

Let  $C_3^n \oplus C_3^{n+1} = Z6$

For every move to be correct

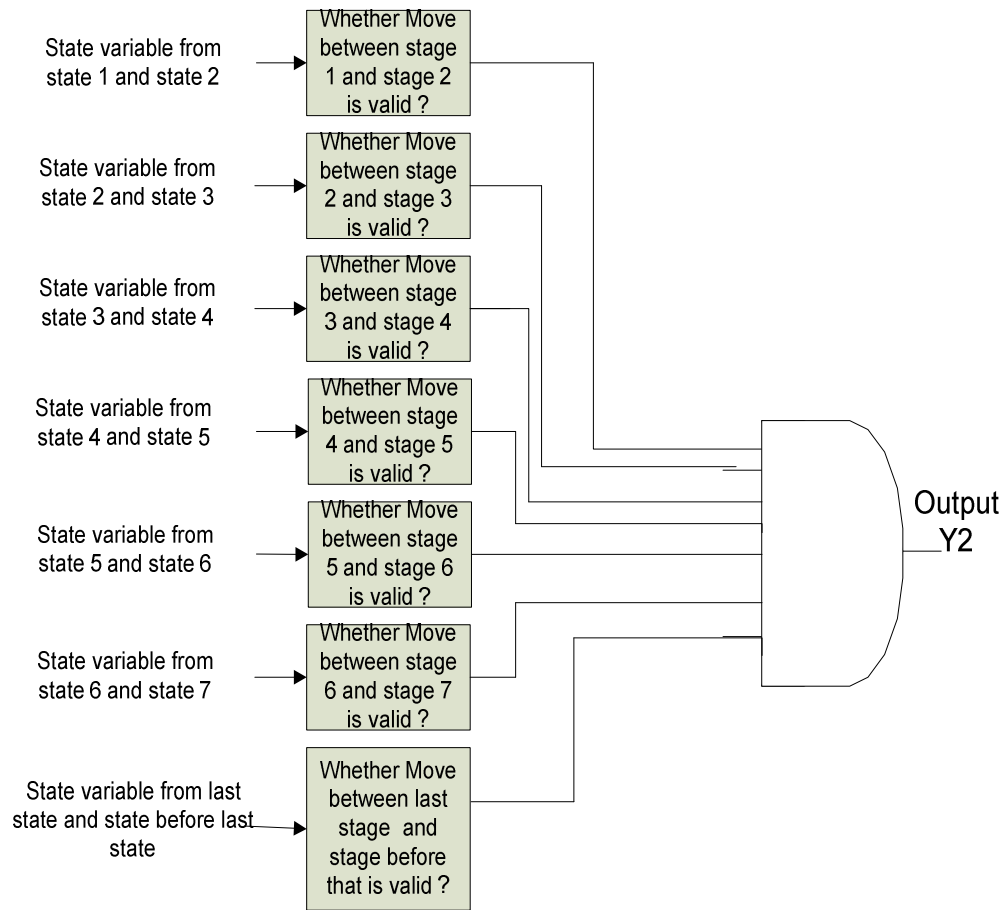
When only 1 object moves

$$Z1 \overline{Z2Z3Z4Z5Z6} + \overline{Z1} Z2 \overline{Z3Z4Z5Z6} + \overline{Z1Z2Z3} \overline{Z4Z5Z6} + \overline{Z1Z2Z3} Z4 \overline{Z5Z6} + \overline{Z1Z2Z3Z4} Z5 \overline{Z6} + \overline{Z1Z2Z3Z4Z5} Z6 \dots\dots\dots\text{Eq A}$$

Now the second case of valid move in which maximum 2 objects can move between two banks is

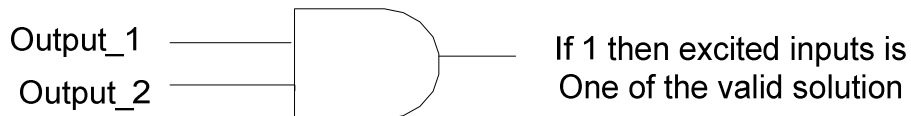
$$Z1 Z2 \overline{Z3Z4Z5Z6} + Z1 \overline{Z2} Z3 \overline{Z4Z5Z6} + Z1 \overline{Z2Z3} Z4 \overline{Z5Z6} + Z1 \overline{Z2Z3Z4} Z5 \overline{Z6} + Z1 \overline{Z2Z3Z4Z5} Z6 + \overline{Z1} Z2 Z3 \overline{Z4Z5Z6} + \overline{Z1} Z2 Z3 Z4 \overline{Z5Z6} + \overline{Z1} Z2 \overline{Z3Z4} Z5 \overline{Z6} + \overline{Z1} Z2 \overline{Z3Z4Z5} Z6 + \overline{Z1Z2} Z3 Z4 \overline{Z5Z6} + \overline{Z1Z2} Z3 \overline{Z4Z5} Z6 + \overline{Z1Z2} Z3 \overline{Z4Z5} Z6 + \overline{Z1Z2Z3} Z4 Z5 \overline{Z6} + \overline{Z1Z2Z3} Z4 \overline{Z5} Z6 + \overline{Z1Z2Z3Z4} Z5 Z6 \dots\dots\dots\text{Eq B}$$

So for any move to be correct  $\text{Eq A} + \text{Eq B}$  should evaluate to one.



**Figure 9-3 Oracle for checking invalid moves**

Final Oracle will look like



## 9.4 VHDL Code for Checking validity of state

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

entity check_state is
port(M1,M2,M3,C1,C2,C3:in std_logic;
     Valid_state : out std_logic);
end check_state;

architecture Behavioural of check_state is

signal M1_BAR,M2_BAR, M3_BAR,C1_BAR,C2_BAR,C3_BAR:std_logic;
signal
temp1,temp2,temp3,temp4,temp5,temp6,temp7,temp8,temp9,temp10,temp11,temp12,temp13,temp14,temp
15,temp16,temp17,temp18:std_logic;
signal
invalid_state_temp1,invalid_state_temp2,invalid_state_temp3,invalid_state_temp4,invalid_state:std_logic;

begin
    M1_BAR <= not M1;   M2_BAR <= not M2;   M3_BAR <= not M3;
    C1_BAR <= not C1;   C2_BAR <= not C2;   C3_BAR <= not C3;

    -- Checking validity of state with 3 cannibals and 1 or 2 missionaries on any bank

    temp1 <= C1_BAR and C2_BAR and C3_BAR;

    temp2 <= C1 and C2 and C3;

    temp3 <= ((M1_BAR and M3) or ( M2 and M3_BAR) or (M1 and M2_BAR));

    temp4 <= ((M1 and M3_BAR) or ( M2_BAR and M3) or ( M1_BAR and M2));

    ---Checking validity of state 2 cannibals and 1 missionary is

    invalid_state_temp1 <= (temp1 and temp3) or ( temp2 and temp4);

    temp5 <= (M1_BAR and M2 and M3)or (M1 and M2_BAR and M3) or (M1 and M2 and M3_BAR);

    temp6 <= (M1_BAR and M2_BAR and M3)or (M1 and M2_BAR and M3_BAR) or (M1_BAR and M2
and M3_BAR);

    invalid_state_temp2 <= (temp1 and temp5) or (temp2 and temp6);

    temp7 <= C1_BAR and C2_BAR and C3;
    temp8 <= C1 and C2_BAR and C3_BAR;
    temp9 <= C1_BAR and C2 and C3_BAR;
    temp10<= M1_BAR and M2 and M3;
```

```

temp10<= M1 and M2_BAR and M3;
temp10<= M1 and M2 and M3_BAR;
invalid_state_temp3 <= (temp7 and temp10) or (temp7 and temp11) or(temp7 and temp12) or (temp8 and
temp10) or (temp8 and temp11) or(temp8 and temp12)
or (temp9 and temp10) or (temp9 and temp11) or(temp9 and temp12);

temp13 <= C1_BAR and C2 and C3;
temp14 <= C1 and C2_BAR and C3;
temp15 <= C1 and C2 and C3_BAR;

temp16 <= M1_BAR and M2_BAR and M3;
temp17 <= M1_BAR and M2 and M3_BAR;
temp18 <= M1 and M2_BAR and M3_BAR;

invalid_state_temp4 <= (temp13 and temp16) or (temp13 and temp17) or(temp13 and temp18) or (temp14
and temp16) or (temp14 and temp17) or(temp14 and temp18)
or (temp15 and temp16) or (temp15 and temp17) or(temp15 and temp18);

invalid_state <= invalid_state_temp1 or invalid_state_temp2 or invalid_state_temp3 or
invalid_state_temp4;

valid_state <= not invalid_state;

end Behavioural;

```

## 9.5 VHDL Code for checking validity of Move

```

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

entity check_move is
port(M1_ps,M2_ps,M3_ps,C1_ps,C2_ps,C3_ps :in std_logic;
     M1_ns,M2_ns,M3_ns,C1_ns,C2_ns,C3_ns :in std_logic;
     valid_move : out std_logic);
end check_move;

architecture Behavioural of check_move is

signal Z1,Z2,Z3,Z4,Z5,Z6:std_logic;
signal
temp1,temp2,temp3,temp4,temp5,temp6,temp7,temp8,temp9,temp10,temp11,temp12,temp13:std_logic;
signal temp14,temp15,temp16,temp17,temp18,temp19,temp20,temp21:std_logic;
signal valid_move_temp1,valid_move_temp2:std_logic;

begin

    Z1 <= C1_ps xor C1_ns; --C1 MOVES
    Z2 <= C2_ps xor C2_ns;

```

```

Z3 <= C3_ps xor C3_ns;
Z4 <= M1_ps xor M1_ns;
Z5 <= M2_ps xor M2_ns;
Z6 <= M3_ps xor M3_ns;

temp1<= Z1 and (not Z2) and (not Z3) and (not Z4) and (not Z5) and (not Z6);
temp2<= (not Z1) and Z2 and (not Z3) and (not Z4) and (not Z5) and (not Z6);
temp3<= (not Z1) and (not Z2) and Z3 and (not Z4) and (not Z5) and (not Z6);
temp4<= (not Z1) and (not Z2) and (not Z3) and Z4 and (not Z5) and (not Z6);
temp5<= (not Z1) and (not Z2) and (not Z3) and (not Z4) and Z5 and (not Z6);
temp6<= (not Z1) and (not Z2) and (not Z3) and (not Z4) and (not Z5) and Z6;

valid_move_temp1 <= temp1 or temp2 or temp3 or temp4 or temp5 or temp6;

temp7<= Z1 and Z2 and (not Z3) and (not Z4) and (not Z5) and (not Z6);
temp8<= Z1 and (not Z2) and Z3 and (not Z4) and (not Z5) and (not Z6);
temp9<= Z1 and (not Z2) and (not Z3) and Z4 and (not Z5) and (not Z6);
temp10<= Z1 and (not Z2) and (not Z3) and (not Z4) and Z5 and (not Z6);
temp11<= Z1 and (not Z2) and (not Z3) and (not Z4) and (not Z5) and Z6;

temp12<= (not Z1) and Z2 and Z3 and (not Z4) and (not Z5) and (not Z6);
temp13<= (not Z1) and Z2 and (not Z3) and Z4 and (not Z5) and (not Z6);
temp14<= (not Z1) and Z2 and (not Z3) and (not Z4) and Z5 and (not Z6);
temp15<= (not Z1) and Z2 and (not Z3) and (not Z4) and (not Z5) and Z6;

temp16<= (not Z1) and (not Z2) and Z3 and Z4 and (not Z5) and (not Z6);
temp17<= (not Z1) and (not Z2) and Z3 and (not Z4) and Z5 and (not Z6);
temp18<= (not Z1) and (not Z2) and Z3 and (not Z4) and (not Z5) and Z6;

temp19<= (not Z1) and (not Z2) and (not Z3) and Z4 and Z5 and (not Z6);
temp20<= (not Z1) and (not Z2) and (not Z3) and Z4 and (not Z5) and Z6;

temp21<= (not Z1) and (not Z2) and (not Z3) and (not Z4) and Z5 and Z6;

valid_move_temp2 <= temp7 or temp8 or temp9 or temp10 or temp11 or temp12
or temp13 or temp14 or temp15 or temp16 or temp17 or temp18 or temp19 or temp20 or temp21;

valid_move <= valid_move_temp1 or valid_move_temp2;

end Behavioural;

```

## 9.6 Final VHDL Code for Oracle

```

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

```

```

entity oracle is
port(M1_state1,M1_state2,M1_state3,M1_state4,M1_state5,M1_state6,M1_state7,M1_state8,M1_state9,M1_state10,M1_state11,M1_state12 :in std_logic;

M2_state1,M2_state2,M2_state3,M2_state4,M2_state5,M2_state6,M2_state7,M2_state8,M2_state9,M2_state10,M2_state11,M2_state12 :in std_logic;

M3_state1,M3_state2,M3_state3,M3_state4,M3_state5,M3_state6,M3_state7,M3_state8,M3_state9,M3_state10,M3_state11,M3_state12 :in std_logic;

C1_state1,C1_state2,C1_state3,C1_state4,C1_state5,C1_state6,C1_state7,C1_state8,C1_state9,C1_state10,C1_state11,C1_state12 :in std_logic;

C2_state1,C2_state2,C2_state3,C2_state4,C2_state5,C2_state6,C2_state7,C2_state8,C2_state9,C2_state10,C2_state11,C2_state12 :in std_logic;

C3_state1,C3_state2,C3_state3,C3_state4,C3_state5,C3_state6,C3_state7,C3_state8,C3_state9,C3_state10,C3_state11,C3_state12 :in std_logic;
Valid_state : out std_logic);
end oracle;

```

architecture structural of oracle is

```

component check_state
port(M1,M2,M3,C1,C2,C3:in std_logic;
Valid_state : out std_logic);
end component;

```

```

component check_move
port(M1_ps,M2_ps,M3_ps,C1_ps,C2_ps,C3_ps :in std_logic;
M1_ns,M2_ns,M3_ns,C1_ns,C2_ns,C3_ns :in std_logic;
valid_move : out std_logic);
end component;

```

begin

```

State_1 : check_state port
map(M1=>M1_state1,M2=>M2_state1,M3=>M3_state1,C1=>C1_state1,C2=>C2_state1,C3=>C3_state1);
State_2 : check_state port
map(M1=>M1_state2,M2=>M2_state2,M3=>M3_state2,C1=>C1_state2,C2=>C2_state2,C3=>C3_state2);
State_3 : check_state port
map(M1=>M1_state3,M2=>M2_state3,M3=>M3_state3,C1=>C1_state3,C2=>C2_state3,C3=>C3_state3);
State_4 : check_state port
map(M1=>M1_state4,M2=>M2_state4,M3=>M3_state4,C1=>C1_state4,C2=>C2_state4,C3=>C3_state4);
State_5 : check_state port
map(M1=>M1_state5,M2=>M2_state5,M3=>M3_state5,C1=>C1_state5,C2=>C2_state5,C3=>C3_state5);
State_6 : check_state port
map(M1=>M1_state6,M2=>M2_state6,M3=>M3_state6,C1=>C1_state6,C2=>C2_state6,C3=>C3_state6);
State_7 : check_state port
map(M1=>M1_state7,M2=>M2_state7,M3=>M3_state7,C1=>C1_state7,C2=>C2_state7,C3=>C3_state7);
State_8 : check_state port
map(M1=>M1_state8,M2=>M2_state8,M3=>M3_state8,C1=>C1_state8,C2=>C2_state8,C3=>C3_state8);

```

```

State_9                                :                check_state                port
map(M1=>M1_state9,M2=>M2_state9,M3=>M3_state9,C1=>C1_state9,C2=>C2_state9,C3=>C3_state9);
State_10                               :                check_state                port
map(M1=>M1_state10,M2=>M2_state10,M3=>M3_state10,C1=>C1_state10,C2=>C2_state10,C3=>C3_s
tate10);
State_11:                             check_state                             port
map(M1=>M1_state11,M2=>M2_state11,M3=>M3_state11,C1=>C1_state11,C2=>C2_state11,C3=>C3_s
tate11);
State_12                               :                check_state                port
map(M1=>M1_state12,M2=>M2_state12,M3=>M3_state12,C1=>C1_state12,C2=>C2_state12,C3=>C3_s
tate12);

Move_1                                 :                check_move                port                map
(M1_ps=>M1_state1,M2_ps=>M2_state1,M3_ps=>M3_state1,C1_ps=>C1_state1,C2_ps=>C2_state1,C3_
ps=>C3_state1,

M1_ns=>M1_state2,M2_ns=>M2_state2,M3_ns=>M3_state2,C1_ns=>C1_state2,C2_ns=>C2_state2,C3_n
s=>C3_state2);

Move_2                                 :                check_move                port                map
(M1_ps=>M1_state2,M2_ps=>M2_state2,M3_ps=>M3_state2,C1_ps=>C1_state2,C2_ps=>C2_state2,C3_
ps=>C3_state2,

M1_ns=>M1_state3,M2_ns=>M2_state3,M3_ns=>M3_state3,C1_ns=>C1_state3,C2_ns=>C2_state3,C3_n
s=>C3_state3);

Move_3                                 :                check_move                port                map
(M1_ps=>M1_state3,M2_ps=>M2_state3,M3_ps=>M3_state3,C1_ps=>C1_state3,C2_ps=>C2_state3,C3_
ps=>C3_state3,

M1_ns=>M1_state4,M2_ns=>M2_state4,M3_ns=>M3_state4,C1_ns=>C1_state4,C2_ns=>C2_state4,C3_n
s=>C3_state4);

Move_4                                 :                check_move                port                map
(M1_ps=>M1_state4,M2_ps=>M2_state4,M3_ps=>M3_state4,C1_ps=>C1_state4,C2_ps=>C2_state4,C3_
ps=>C3_state4,

M1_ns=>M1_state5,M2_ns=>M2_state5,M3_ns=>M3_state5,C1_ns=>C1_state5,C2_ns=>C2_state5,C3_n
s=>C3_state5);

Move_5                                 :                check_move                port                map
(M1_ps=>M1_state5,M2_ps=>M2_state5,M3_ps=>M3_state5,C1_ps=>C1_state5,C2_ps=>C2_state5,C3_
ps=>C3_state5,

M1_ns=>M1_state6,M2_ns=>M2_state6,M3_ns=>M3_state6,C1_ns=>C1_state6,C2_ns=>C2_state6,C3_n
s=>C3_state6);

Move_6                                 :                check_move                port                map
(M1_ps=>M1_state6,M2_ps=>M2_state6,M3_ps=>M3_state6,C1_ps=>C1_state6,C2_ps=>C2_state6,C3_
ps=>C3_state6,

M1_ns=>M1_state7,M2_ns=>M2_state7,M3_ns=>M3_state7,C1_ns=>C1_state7,C2_ns=>C2_state7,C3_n
s=>C3_state7);

```



```

Move_7                                :      check_move      port      map
(M1_ps=>M1_state7,M2_ps=>M2_state7,M3_ps=>M3_state7,C1_ps=>C1_state7,C2_ps=>C2_state7,C3_
ps=>C3_state7,

M1_ns=>M1_state8,M2_ns=>M2_state8,M3_ns=>M3_state8,C1_ns=>C1_state8,C2_ns=>C2_state8,C3_n
s=>C3_state8);

Move_8                                :      check_move      port      map
(M1_ps=>M1_state8,M2_ps=>M2_state8,M3_ps=>M3_state8,C1_ps=>C1_state8,C2_ps=>C2_state8,C3_
ps=>C3_state8,

M1_ns=>M1_state9,M2_ns=>M2_state9,M3_ns=>M3_state9,C1_ns=>C1_state9,C2_ns=>C2_state9,C3_n
s=>C3_state9);

Move_9                                :      check_move      port      map
(M1_ps=>M1_state9,M2_ps=>M2_state9,M3_ps=>M3_state9,C1_ps=>C1_state9,C2_ps=>C2_state9,C3_
ps=>C3_state9,

M1_ns=>M1_state10,M2_ns=>M2_state10,M3_ns=>M3_state10,C1_ns=>C1_state10,C2_ns=>C2_state1
0,C3_ns=>C3_state10);

Move_10                               :      check_move      port      map
(M1_ps=>M1_state10,M2_ps=>M2_state10,M3_ps=>M3_state10,C1_ps=>C1_state10,C2_ps=>C2_state1
0,C3_ps=>C3_state10,

M1_ns=>M1_state11,M2_ns=>M2_state11,M3_ns=>M3_state11,C1_ns=>C1_state11,C2_ns=>C2_state1
1,C3_ns=>C3_state11);

Move_11                               :      check_move      port      map
(M1_ps=>M1_state11,M2_ps=>M2_state11,M3_ps=>M3_state11,C1_ps=>C1_state11,C2_ps=>C2_state1
1,C3_ps=>C3_state11,

M1_ns=>M1_state12,M2_ns=>M2_state12,M3_ns=>M3_state12,C1_ns=>C1_state12,C2_ns=>C2_state1
2,C3_ns=>C3_state12);

end structural;

```

## 9.7 Testbench For Emulation

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_textio.all;
use std.textio.all;

entity oracle_test is
port (clock,enable,reset:in std_logic);
end oracle_test;

```

architecture beha of oracle\_test is -- Sequential Modelling Style

component oracle

port(M1\_state1,M1\_state2,M1\_state3,M1\_state4,M1\_state5,M1\_state6,M1\_state7,M1\_state8,M1\_state9,M1\_state10,M1\_state11,M1\_state12 :in std\_logic;

M2\_state1,M2\_state2,M2\_state3,M2\_state4,M2\_state5,M2\_state6,M2\_state7,M2\_state8,M2\_state9,M2\_state10,M2\_state11,M2\_state12 :in std\_logic;

M3\_state1,M3\_state2,M3\_state3,M3\_state4,M3\_state5,M3\_state6,M3\_state7,M3\_state8,M3\_state9,M3\_state10,M3\_state11,M3\_state12 :in std\_logic;

C1\_state1,C1\_state2,C1\_state3,C1\_state4,C1\_state5,C1\_state6,C1\_state7,C1\_state8,C1\_state9,C1\_state10,C1\_state11,C1\_state12 :in std\_logic;

C2\_state1,C2\_state2,C2\_state3,C2\_state4,C2\_state5,C2\_state6,C2\_state7,C2\_state8,C2\_state9,C2\_state10,C2\_state11,C2\_state12 :in std\_logic;

C3\_state1,C3\_state2,C3\_state3,C3\_state4,C3\_state5,C3\_state6,C3\_state7,C3\_state8,C3\_state9,C3\_state10,C3\_state11,C3\_state12 :in std\_logic;

Valid\_state : out std\_logic);

end component;

signal

M1\_state1,M1\_state2,M1\_state3,M1\_state4,M1\_state5,M1\_state6,M1\_state7,M1\_state8,M1\_state9,M1\_state10,M1\_state11,M1\_state12 : std\_logic;

signal

M2\_state1,M2\_state2,M2\_state3,M2\_state4,M2\_state5,M2\_state6,M2\_state7,M2\_state8,M2\_state9,M2\_state10,M2\_state11,M2\_state12 : std\_logic;

signal

M3\_state1,M3\_state2,M3\_state3,M3\_state4,M3\_state5,M3\_state6,M3\_state7,M3\_state8,M3\_state9,M3\_state10,M3\_state11,M3\_state12 : std\_logic;

signal

C1\_state1,C1\_state2,C1\_state3,C1\_state4,C1\_state5,C1\_state6,C1\_state7,C1\_state8,C1\_state9,C1\_state10,C1\_state11,C1\_state12 : std\_logic;

signal

C2\_state1,C2\_state2,C2\_state3,C2\_state4,C2\_state5,C2\_state6,C2\_state7,C2\_state8,C2\_state9,C2\_state10,C2\_state11,C2\_state12 : std\_logic;

signal

C3\_state1,C3\_state2,C3\_state3,C3\_state4,C3\_state5,C3\_state6,C3\_state7,C3\_state8,C3\_state9,C3\_state10,C3\_state11,C3\_state12 : std\_logic;

signal Valid\_State:std\_logic;

signal M1,M2,M3,M4,M5,M6,M7,M8 :std\_logic;

signal W1,W2,W3,W4,W5,W6,W7,W8 :std\_logic;

signal G1,G2,G3,G4,G5,G6,G7,G8 :std\_logic;

signal C1,C2,C3,C4,C5,C6,C7,C8 :std\_logic;

signal counter : std\_logic\_vector(71 downto 0):=(others=>'0');

signal Valid\_solution ,test\_vector\_enable:std\_logic;

```

begin

u                                     :                                     oracle                                     port
map(M1_state1=>M1_state1,M1_state2=>M1_state2,M1_state3=>M1_state3,M1_state4=>M1_state4,M1
_state5=>M1_state5,M1_state6=>M1_state6,

M1_state7=>M1_state7,M1_state8=>M1_state8,M1_state9=>M1_state9,M1_state10=>M1_state10,M1_st
ate11=>M1_state11,M1_state12=>M1_state12,

M2_state1=>M2_state1,M2_state2=>M2_state2,M2_state3=>M2_state3,M2_state4=>M2_state4,M2_state
5=>M2_state5,M2_state6=>M2_state6,

M2_state7=>M2_state7,M2_state8=>M2_state8,M2_state9=>M2_state9,M2_state10=>M2_state10,M2_st
ate11=>M2_state11,M2_state12=>M2_state12,

M3_state1=>M3_state1,M3_state2=>M3_state2,M3_state3=>M3_state3,M3_state4=>M3_state4,M3_state
5=>M3_state5,M3_state6=>M3_state6,

M3_state7=>M3_state7,M3_state8=>M3_state8,M3_state9=>M3_state9,M3_state10=>M3_state10,M3_st
ate11=>M3_state11,M3_state12=>M3_state12,

C1_state1=>C1_state1,C1_state2=>C1_state2,C1_state3=>C1_state3,C1_state4=>C1_state4,C1_state5=>
C1_state5,C1_state6=>C1_state6,

C1_state7=>C1_state7,C1_state8=>C1_state8,C1_state9=>C1_state9,C1_state10=>C1_state10,C1_state11
=>C1_state11,C1_state12=>C1_state12,

C2_state1=>C2_state1,C2_state2=>C2_state2,C2_state3=>C2_state3,C2_state4=>C2_state4,C2_state5=>
C2_state5,C2_state6=>C2_state6,

C2_state7=>C2_state7,C2_state8=>C2_state8,C2_state9=>C2_state9,C2_state10=>C2_state10,C2_state11
=>C2_state11,C2_state12=>C2_state12,

C3_state1=>C3_state1,C3_state2=>C3_state2,C3_state3=>C3_state3,C3_state4=>C3_state4,C3_state5=>
C3_state5,C3_state6=>C3_state6,

C3_state7=>C3_state7,C3_state8=>C3_state8,C3_state9=>C3_state9,C3_state10=>C3_state10,C3_state11
=>C3_state11,C3_state12=>C3_state12);

P1 : process (clock)
begin
if( reset ='1') then

counter<="0000000000000000000000000000000000000000000000000000000000000000";

else

counter<= counter+"0000000000000000000000000000000000000000000000000000000000000001";
end if;

```

end process;

```
    M1_state1<=counter(0);    M1_state2<=counter(1);M1_state3<=counter(2);M1_state4<=counter(3);
M1_state5<=counter(4);M1_state6<=counter(5);
    M1_state7<=counter(6);    M1_state8<=counter(7);M1_state9<=counter(8);M1_state10<=counter(9);
M1_state11<=counter(10);M1_state12<=counter(11);
    M2_state1<=counter(12);
M2_state2<=counter(13);M2_state3<=counter(14);M2_state4<=counter(15);
M2_state5<=counter(16);M2_state6<=counter(17);
    M2_state7<=counter(18);
M2_state8<=counter(19);M2_state9<=counter(20);M2_state10<=counter(21);
M2_state11<=counter(22);M2_state12<=counter(23);
    M3_state1<=counter(24);
M3_state2<=counter(25);M3_state3<=counter(26);M3_state4<=counter(27);
M3_state5<=counter(28);M3_state6<=counter(29);
    M3_state7<=counter(30);
M3_state8<=counter(31);M3_state9<=counter(32);M3_state10<=counter(33);
M3_state11<=counter(34);M3_state12<=counter(35);
```

```
    C1_state1<=counter(36);    C1_state2<=counter(37);C1_state3<=counter(38);C1_state4<=counter(39);
C1_state5<=counter(40);C1_state6<=counter(41);
    C1_state7<=counter(41);    C1_state8<=counter(43);C1_state9<=counter(44);C1_state10<=counter(45);
C1_state11<=counter(46);C1_state12<=counter(47);
    C2_state1<=counter(48);    C2_state2<=counter(49);C2_state3<=counter(50);C2_state4<=counter(51);
C2_state5<=counter(52);C2_state6<=counter(53);
    C2_state7<=counter(54);    C2_state8<=counter(55);C2_state9<=counter(56);C2_state10<=counter(57);
C2_state11<=counter(58);C2_state12<=counter(59);
    C3_state1<=counter(60);    C3_state2<=counter(61);C3_state3<=counter(62);C3_state4<=counter(63);
C3_state5<=counter(64);C3_state6<=counter(65);
    C3_state7<=counter(66);    C3_state8<=counter(67);C3_state9<=counter(68);C3_state10<=counter(69);
C3_state11<=counter(70);C3_state12<=counter(71);
```

end beha;

## Chapter 10 Oracle for Graph Coloring

In graph theory, graph coloring is a special case of graph labeling; it is an assignment of labels traditionally called "colors" to elements of a graph subject to certain constraints. In its simplest form, it is a way of coloring the vertices of a graph such that no two adjacent vertices share the same color; this is called a vertex coloring. Similarly, an edge coloring assigns a color to each edge so that no two adjacent edges share the same color, and a face coloring of a planar graph assigns a color to each face or region so that no two faces that share a boundary have the same color.

The Graph Coloring Problem.



**Figure 10-1 Maps of Europe**

Graph coloring is a relatively easy problem to formulate in principle, but large amounts of nodes in the graph would result in an extremely large amount of combinations making the problem extremely difficult to solve exactly on a standard computer. Thus this problem is a great candidate for quantum computing. The problem of using a quantum

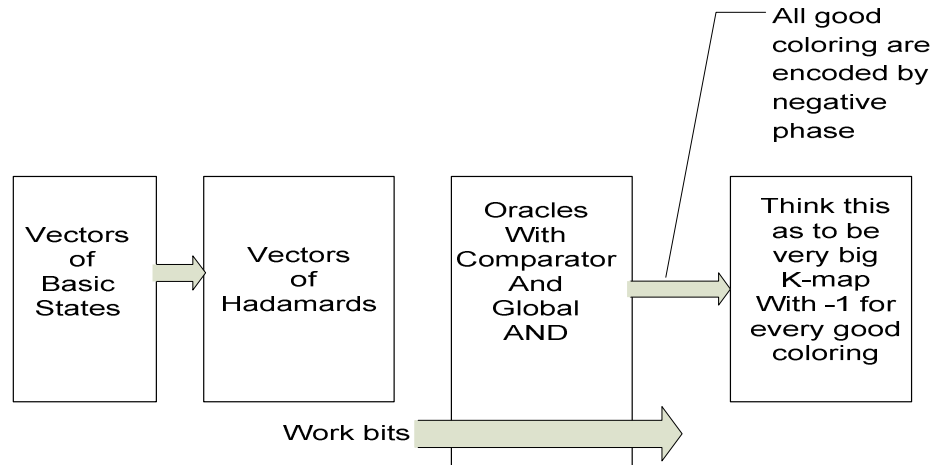
computer to find the minimum solution to the graph coloring problem is of interest because there is no literature on this subject, although much is known about graph coloring and related problems on standard computers. This gave us the idea to maybe adapting standard graph-coloring approaches in quantum computing. The main result to be expected from Grover was that the optimal (exact) solution could be found in a number of steps proportional to the square root of  $N$ , where  $N$  is  $2^n \cdot \log c$ , where  $n$  is the number of nodes and  $c$  is the upper bound of the number of colors. The classical algorithm would require an order of  $N$  amount of steps. Thus, while a classical computer would take 10,000 hours to solve a complex graph coloring problem, the quantum device would take 100 hours. The relative speedup of quantum computing is only quadratic in this case, but in any case it is dramatic in real-life situations like military image recognition. For instance; South Korea installed robots soldiers on its frontier with North Korea that are equipped with image recognition abilities. In this case the quadratic speedup is very important, 5 seconds versus 25 seconds may make a life-or-death difference. There are many other problem instances like this.

In the simplest formulation of graph coloring, a graph is denoted as a standard graph (not a multi-graph) with a certain number of edges and nodes. Every node is connected to at least one other node, by means of an edge. Every node may also obtain a color, which is represented as a bit string. A solution to a graph coloring problem consists of having no uncolored nodes, and having no edges connecting 2 nodes of the same color. We want also to minimize the number of colors used (this leads to finding the chromatic number of the graph). A rather popular branch of graph coloring is called “Map Coloring”. Maps,

for easy distinction between countries in them, tend to have different, adjacent countries colored differently. For those whose eyesight is not perfect, the distinction between 2 shapes of different colors is far more easily recognized than a thin black borderline. In graph coloring, each country is represented as a node, and borders are represented as graph edges, (see Figure 9.1). The interest in Map Coloring was started by Francis Guthrie, who in the 1850's formulated a problem involving coloring a map with only 4 colors. The problem remained unsolved until 1976, when after hundreds of computer-hours of calculation, Kenneth Appel and Wolfgang Haken proposed a solution that, as of yet, has not been disproven and mathematicians agree that the solution is correct. Map coloring is thus the first and easy variant of graph coloring and constraint satisfaction problems that we explain and simulate in this thesis. Since it was proved that every map can be colored with 4 colors, my oracle is greatly simplified, especially if one would try to apply it to a very big map.

### **10.1 Proposed Architecture for Graph Coloring Problem using Grover's Algorithm**

In this section, we introduce the proposed architecture for finding the minimum coloring that takes ideas from the Quantum Grover Algorithm.

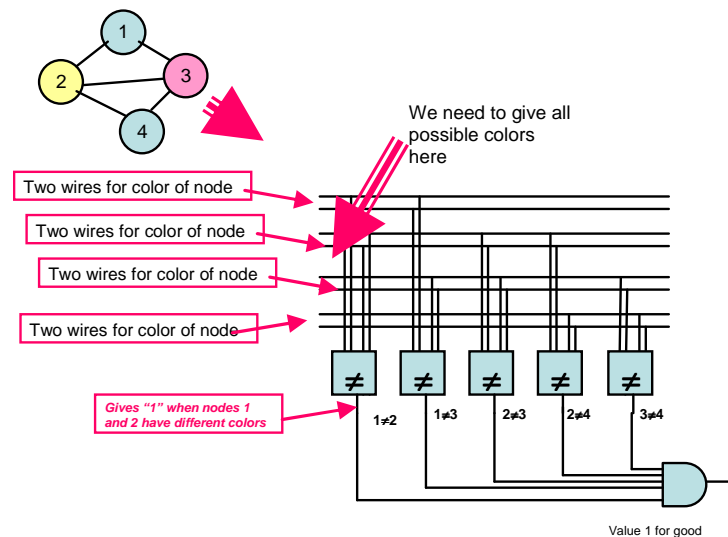


**Figure 10-2. superposed quantum states with negative phase for all good colorings of a map**

Figure 10-2 shows Block Diagram of creating superposed quantum states with negative phase for all good colorings of a map. Observe that information if a given coloring is good is seen by the output of AND in oracle, but the argument for which the oracle is satisfied is shown in negative phase of the respective minterm of the color encoding variables. It gives the idea of using Grover for graph coloring. Nodes(countries) are represented as groups of neighbor input variables. Coloring of a node is represented as a binary encoding of the set of qubits corresponding to this node. All possible colorings are created at the oracle's inputs by the vector of Hadamard gates on each input. As always, they are all initialized to state  $|0\rangle$ . Let us observe that this idea can be realized in standard, non-quantum hardware but when the superposition in quantum hardware allows for high parallelism, there is no superposition in standard hardware so the variants in the oracle must be realized sequentially, thus making it much slower.



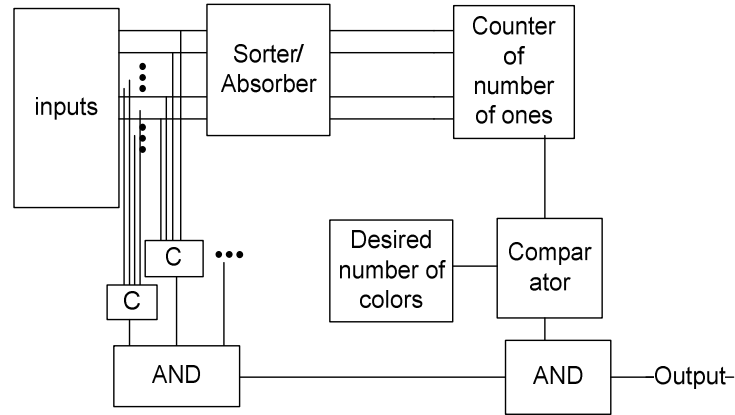
Figure 10-3 gives the example of coloring a particular map (left top corner) with inequality comparators and a global AND. The global AND produces a logic one when all neighbor nodes have different nodes. Observe that although the graph is 3-colorable, a coloring with 4 colors is given here as a good coloring because this simple oracle is not trying to minimize the number of colors used for the coloring i.e., (this is a Decision Oracle, not an Optimization Oracle). The first solution out of many can terminate if the standard Grover algorithm is run. This figure shows also that all primary inputs are repeated to the outputs and forwarded to the next stages together with the output bit(yes/no) of the oracle. The details of a sequential oracle without Hadamard gates are presented in Figure 10-3.



**Figure 10-3 A simple graph coloring problem.**

In figure 10.4 the color comparators correspond to the borders of the countries or the edges of the graph. Observe that this oracle can be used not only in quantum but also in reversible and classical technologies, but in such cases it would require sequential inputs and not parallel superposed inputs as created by Hadamard gates in quantum oracles. The

blocks for the complete Oracle for Graph Coloring and how they are connected together are illustrated in Figure 10-4.



**Figure 10-4**Simplified schematic of my optimization Graph Coloring Oracle

The rough explanation of blocks from Figure 10-4 is as follows.

### 10.1.1 The C blocks:

These are the Inequality Comparators. As we know, they act upon sets of two inputs. Those two inputs are representative of connected nodes' color encoding. If these two inputs binary strings are the same, then they violated coloring rule and output of the C block will be "0". The quantum oracle is to run through every possible color configuration of inputs (see Figure 10-3 and Figure 10-4); only a few are solutions. In order to determine whether it is a solution, we run the representative inputs through the comparators. The C comparators outputs are then forwarded into an AND gate at the bottom left to determine whether the configuration is a solution.

### 10.1.2 The Sorter/Absorber:

Here, the inputted color encodings are sorted. If two inputs are the same for different nodes (same color used more than once), then only one will be outputted and all other same input will be “absorbed” (removed). This circuit sorts and absorbs colors such that all inputs will be sorted from the “smallest” to the “largest” and each color only has one output. We can observe that this is a general circuit to convert a list of items with repetitions to a set with no repeated elements.

The entire circuit is very big and it is difficult to put it on paper. Here we give only some of the blocks and we do not show the complete layout that includes CNOT gates for copying and SWAP gates to be able to combine all blocks together.

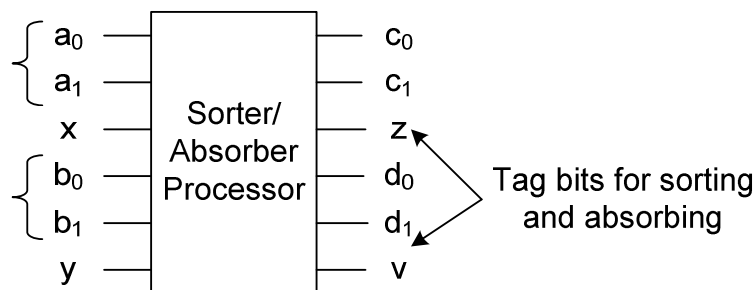
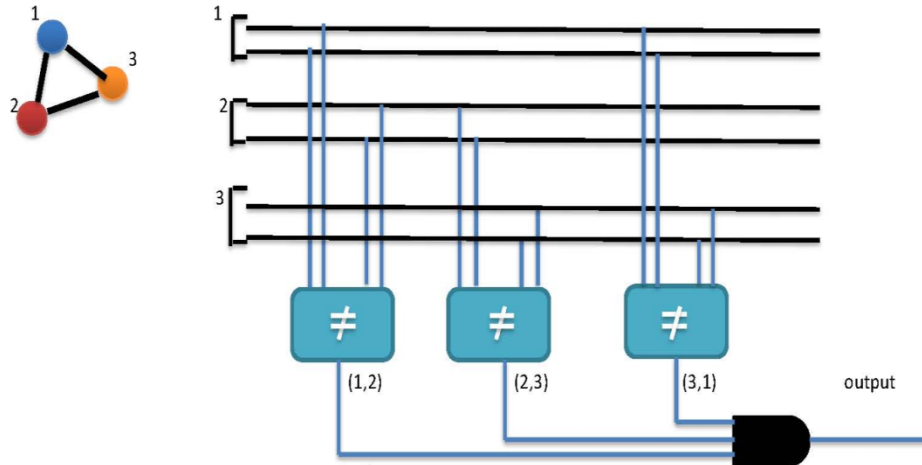


Figure 10-5 Sorter/Absorber

## 10.2 Graph Coloring Oracle

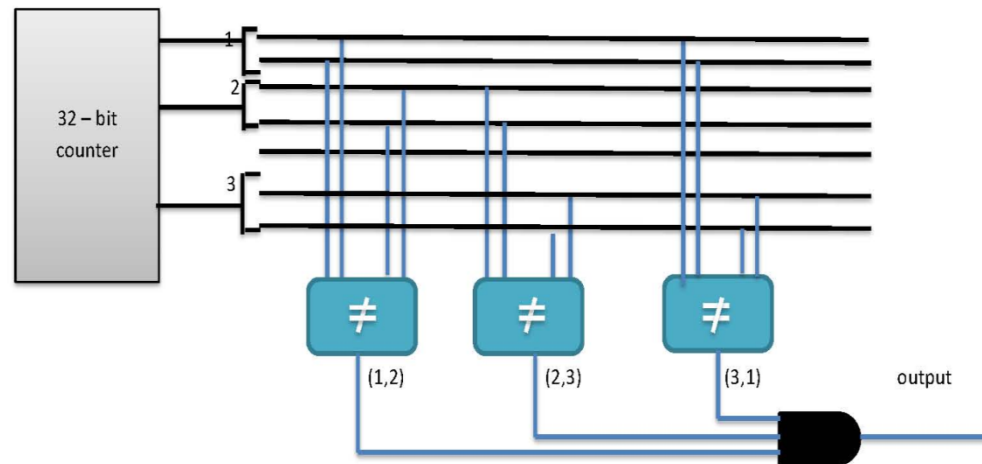
The graph coloring oracle defines the minimum number of colors needed to color the edges of a graph. The graph we are using is a planar graph. The proposed architecture for the graph coloring oracle is as follows.



**Figure 10-6 Architecture of graph coloring oracle**

Two wires have been assigned for each edge and the comparator compares the colors of two edges at a time and finally all the outputs are given to a AND gate to check if the result is good or bad.

### 10.3 Exercising the oracle using a counter

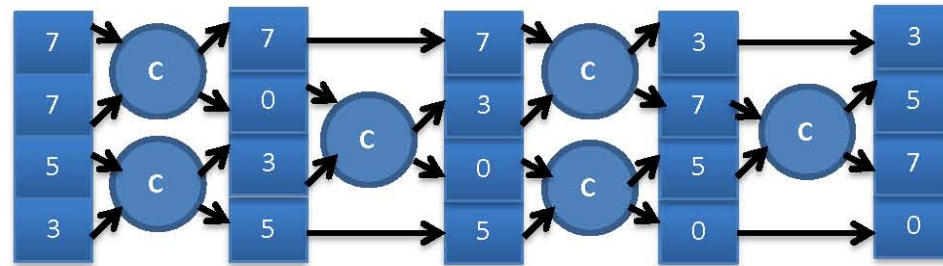


**Figure 10-7 Exercising the oracles using a counter**

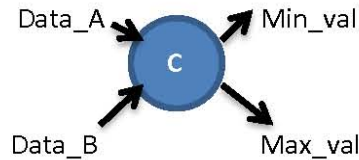
As shown in figure Figure 10-7 a 32 – bit counter has been used in order to generate the colors for the edges. From this 32-bit value 16 bits are taken for one edge and 16 bits for another edge.

### 10.4 How does the sorter work?

The sorting is implemented by recursively running the above process of comparison for number of times by using the following butterfly pattern.



*C represents comparator and 0 indicates the null value*



**Figure 10-8 Sorter using butterfly pattern**

Observe in Figure 10-8 Sorter using butterfly pattern that the continuous recursion of the comparator logic finally gave a sorted result and also the number of times the pattern must be repeated is equal to number of colors that are present in the graph.

### 10.5 Implementation of the oracle in Verilog

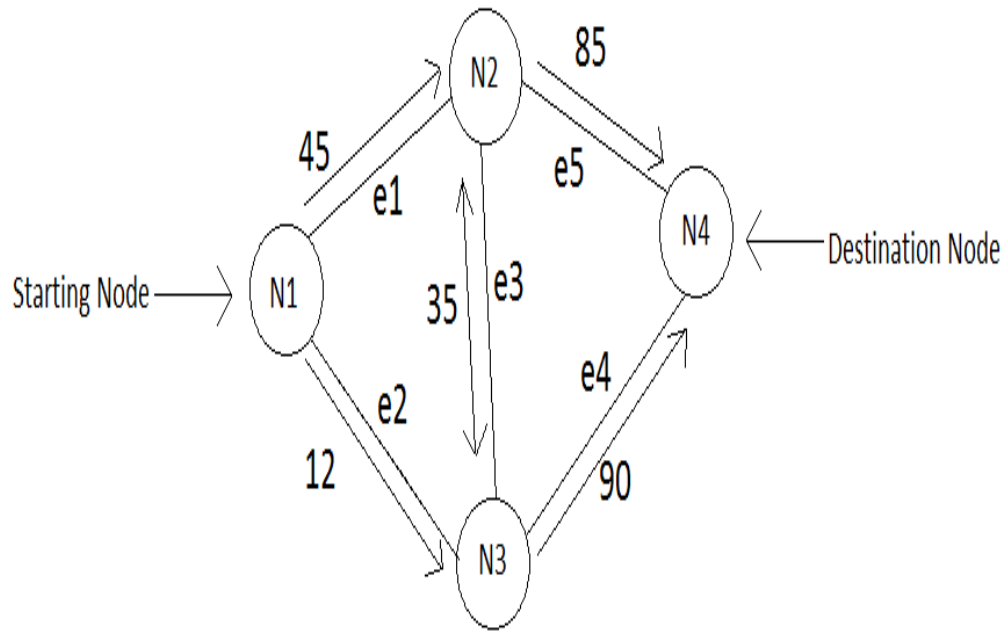
Since we need to find the amount of time it takes for the hardware to run this process and a simulator to do the same work, we have increased the number of comparisons as the process speed depends only these comparisons. So a large number of random values are generated and are fed to the comparator and allowed to run for a large number of clock cycles.

## Chapter 11 Oracle for Longest Path

### 11.1 Introduction

The longest path problem determines whether a given path is longer than a path traversed by the user. We formulate the problem as a graph and traverse along a path of the graph. The graph is encoded as edges. The edges are encoded in binary. The distances for the edges are encoded as integer.

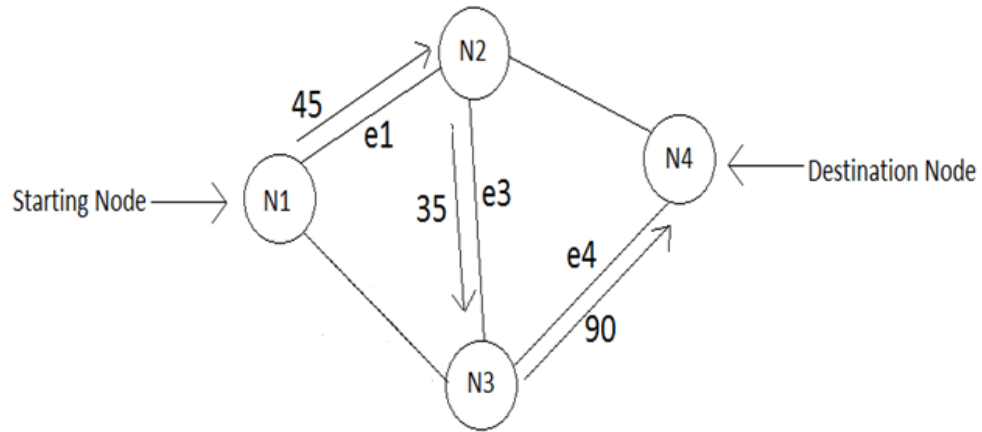
The longest path problem can be explained by illustrating a problem. In the graph given below, we have the initial node as N1 and the final destination node as N4. We have the edges between nodes as e1, e2, e3, e4, e5. As there are 5 edges we can encode it using 3 bits. This method can be extended for 'n' edges. The distances for particular edges are also given.



**Figure 11-1 Graph for the Longest Path**

The user can only traverse in the direction given in the figure above. The user just needs to give the edges to be traversed and the length with which he wants it to be compared. If the path to be traversed is e1, e3, e4 and the length to be compared with is 200. Hence, they would be traversed in the path shown in figure3.

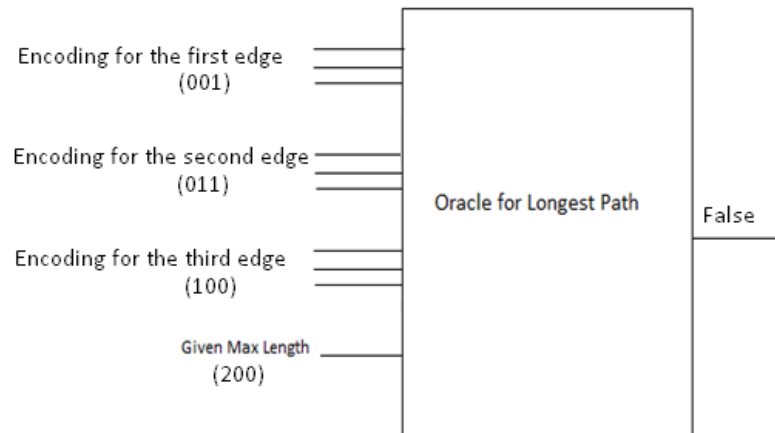




**Figure 11-2 Example which shows how the node selection is done.**

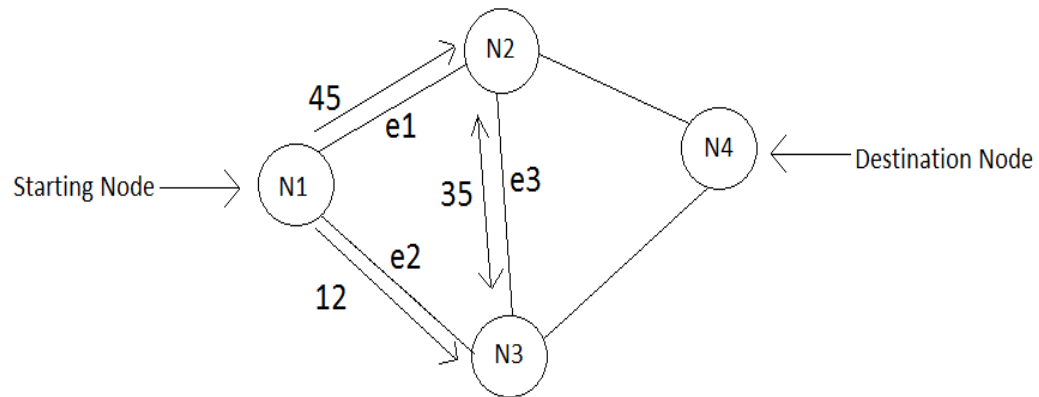
As the path traversed is in the right direction, the path lengths of the edges are added as  $45 + 35 + 90 = 170$ , which is less than 200. Hence the output of this would be False as it is less than the given path length.

Black box for the oracle can be shown below.



**Figure 11-3 Oracle for Longest Path**

If the edges selected are e1, e2, e3 and the path length to be calculated is 300. The oracle will check whether the direction is proper.

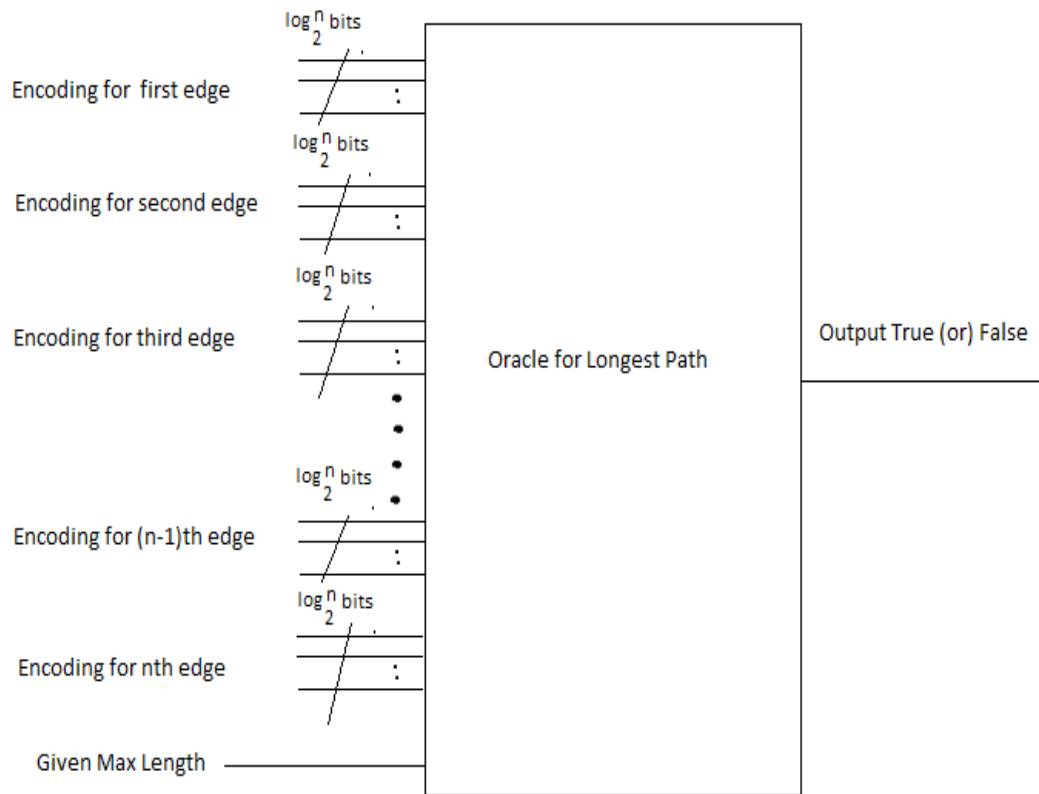


**Figure 11-4 Example where the edge selection is not done properly**

The oracle does not even calculate the distance, because the direction traversed is not proper and does not reach the destination node. The output is 'false' in this case.

This problem for longest path can be generalized to any number of nodes. The same method can be used for finding the shortest path, equal path, by changing the comparator in longest path.

For any graph the generalized oracle would be as shown in the figure given below.



**Figure 11-5 Oracle for ‘n’ edges**

## 11.2 Node Representation

The nodes are first encoded in the code. We have the following constraints for the nodes:

The path starts only at the first node and ends at the destination node.

We have to enter and leave a node only once.

The nodes are encoded in the following way:

Basing on the constraints given above the graph given is coded in the following way:

There are four nodes in total. Hence covering all nodes at maximum we have 3 edges to cover.

We can encode the edges using a binary number. We can have edge1 – 001, edge2 – 010, edge3 – 011, edge4 – 100, edge5- 101.

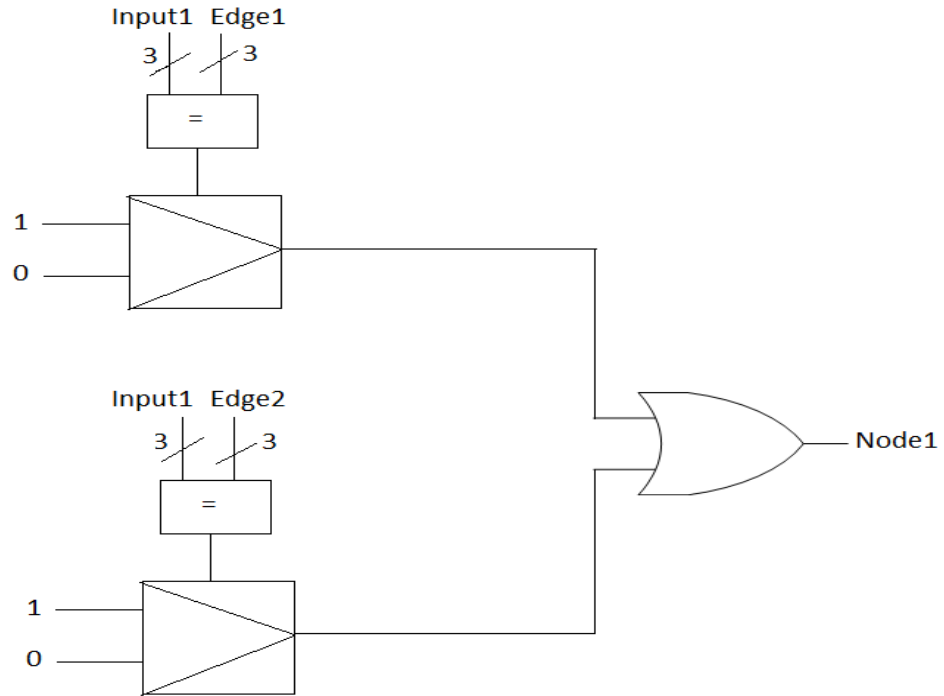
The first input can be either edge1 or edge2. The second input can be 000 or edge3 , the third input can be either edge4 or edge5. The path lengths are initially given by the user and we add them once the edges are selected.

I have used a multiplexer for selecting the edges and adding the distances. A multiplexer is used to select the edges and if the edges are selected properly we add up the distances. The select signal would be from the nodes. The input will be distances.

By using these rules we can encode the nodes in the oracle. These rules are applicable to any number of nodes. I have illustrated it for the graph given in the figure1.

The nodes can be represented in the following way:

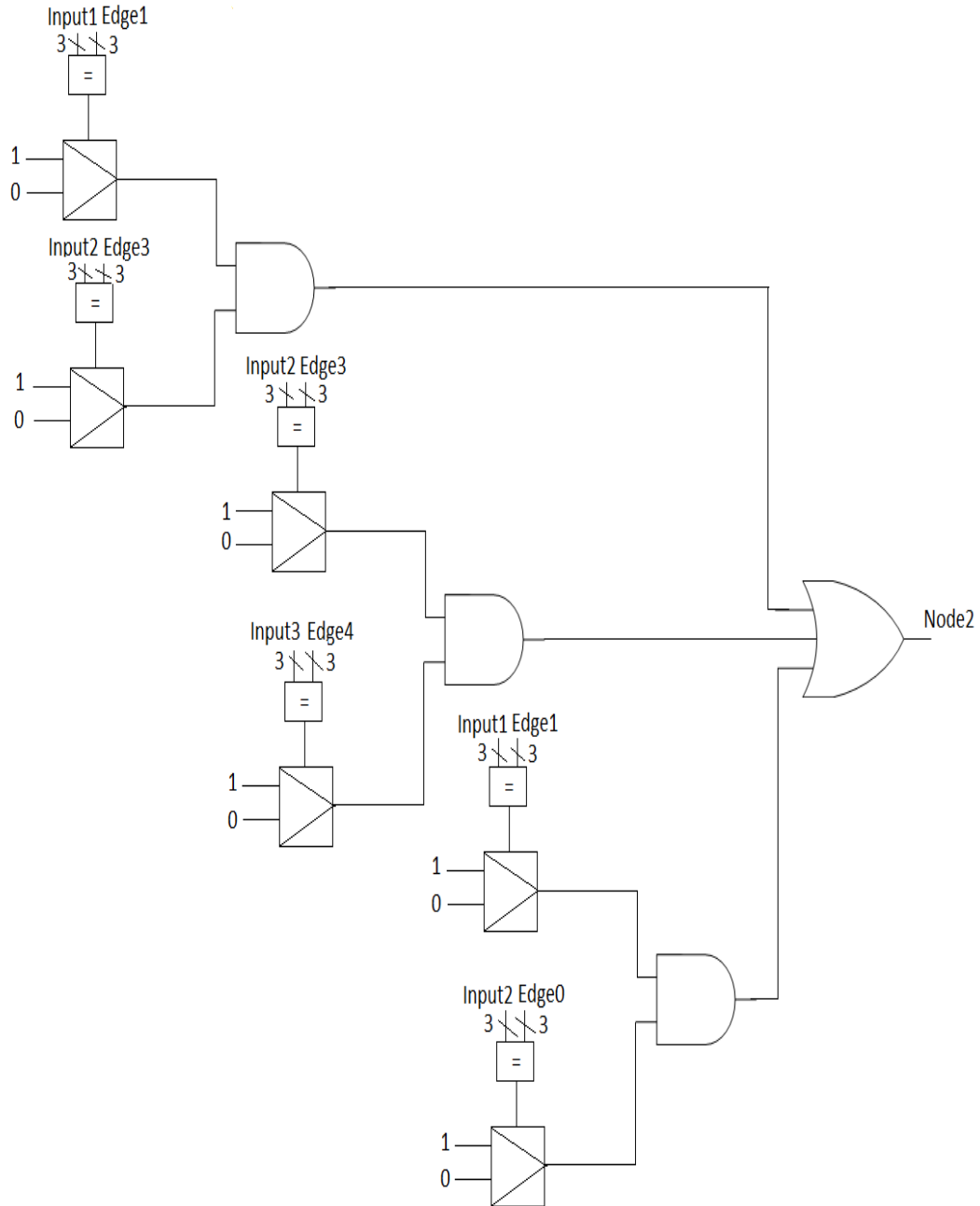
The equation for node1 can be represented as  $e1 + e2 = \text{node1}$ . The node1 can start either from edge1 or edge2. And it can be only 1<sup>st</sup> input as it is the starting node.



**Figure 11-6Representation for Node1**

The node2 is represented as  $(\text{edge1.edge3}) + (\text{edge3.edg4}) + (\text{edge1.edge0}) = \text{Node2}$ .

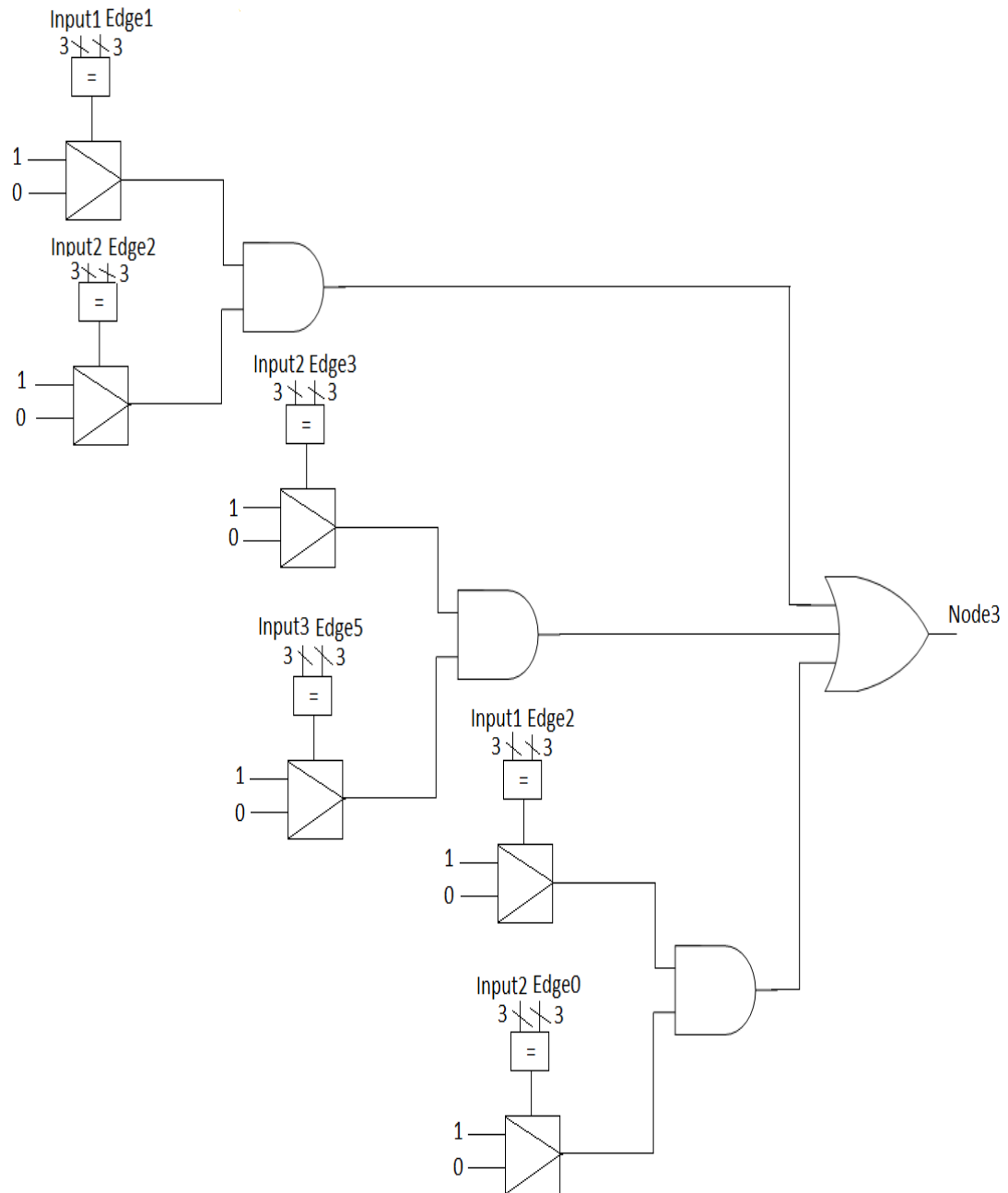
For the node2 we have three edges. So it can start either from one edge and move on to the other edge. Hence, we have three different entry and exit points.



**Figure 11-7Representation for Node2**

The node3 can be represented as  $\text{edge1}.\text{edge2} + \text{edge3}.\text{edge5} + \text{edge2}.\text{edge0} = \text{Node3}$ .

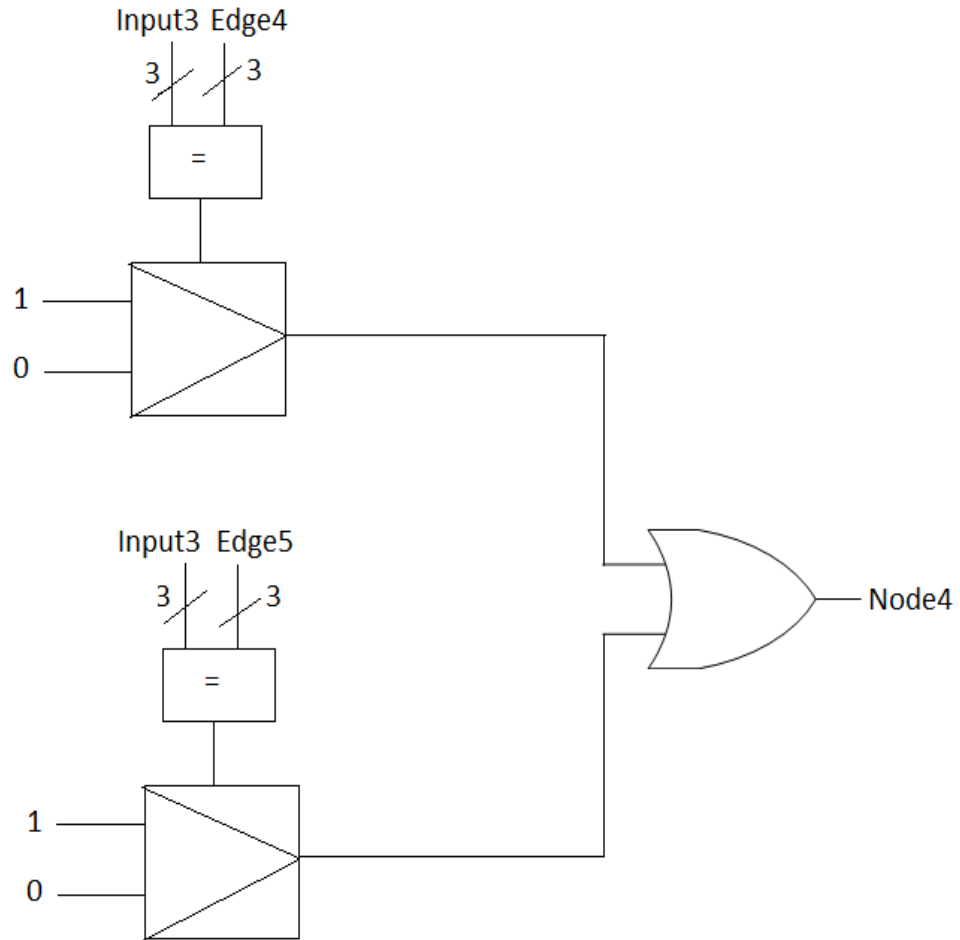
For the node3 we have three edges. So it can start either from one edge and move on to the other edge. Hence, we have three different entry and exit points.



**Figure 11-8Representation for node3**

The node4 can be represented as  $\text{edge4} + \text{edge5} = \text{node4}$ .

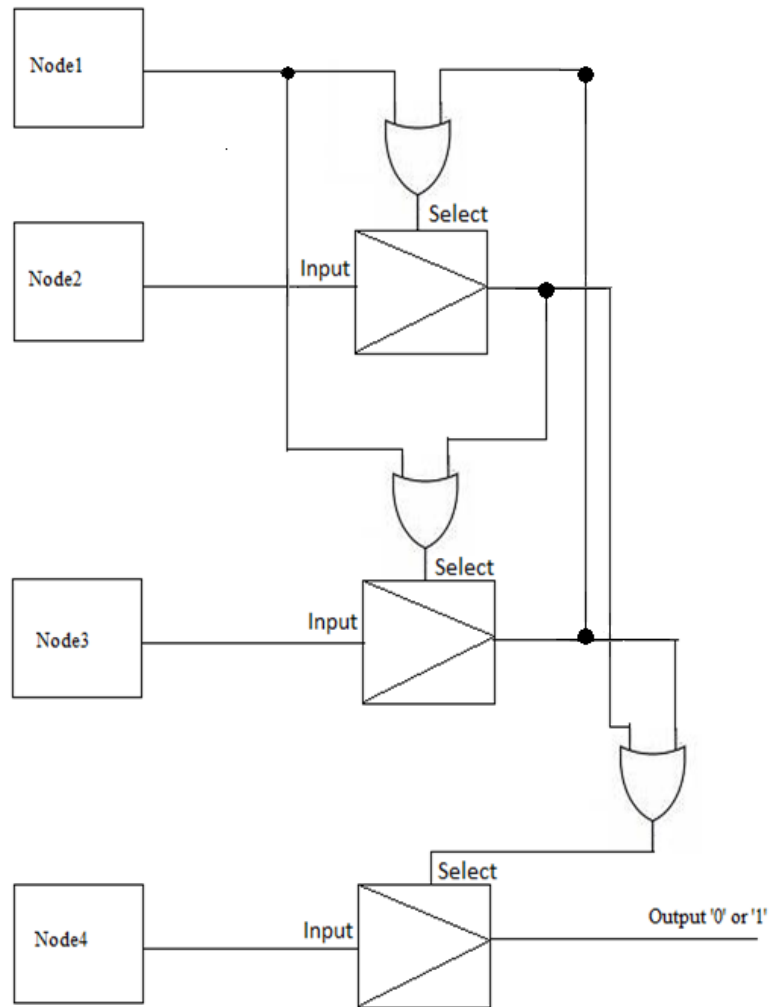
This is the destination point for the graph given. This would be the final input for the node calculation.



**Figure 11-9 Representation for Node 4**

Nodes are connected using multiplexers based on the connections between nodes. So, once an edge is selected, we have the output of that mux as '1' and connections between the nodes are checked in this way.





**Figure 11-10 Connection between Nodes**

### Path Length Calculation

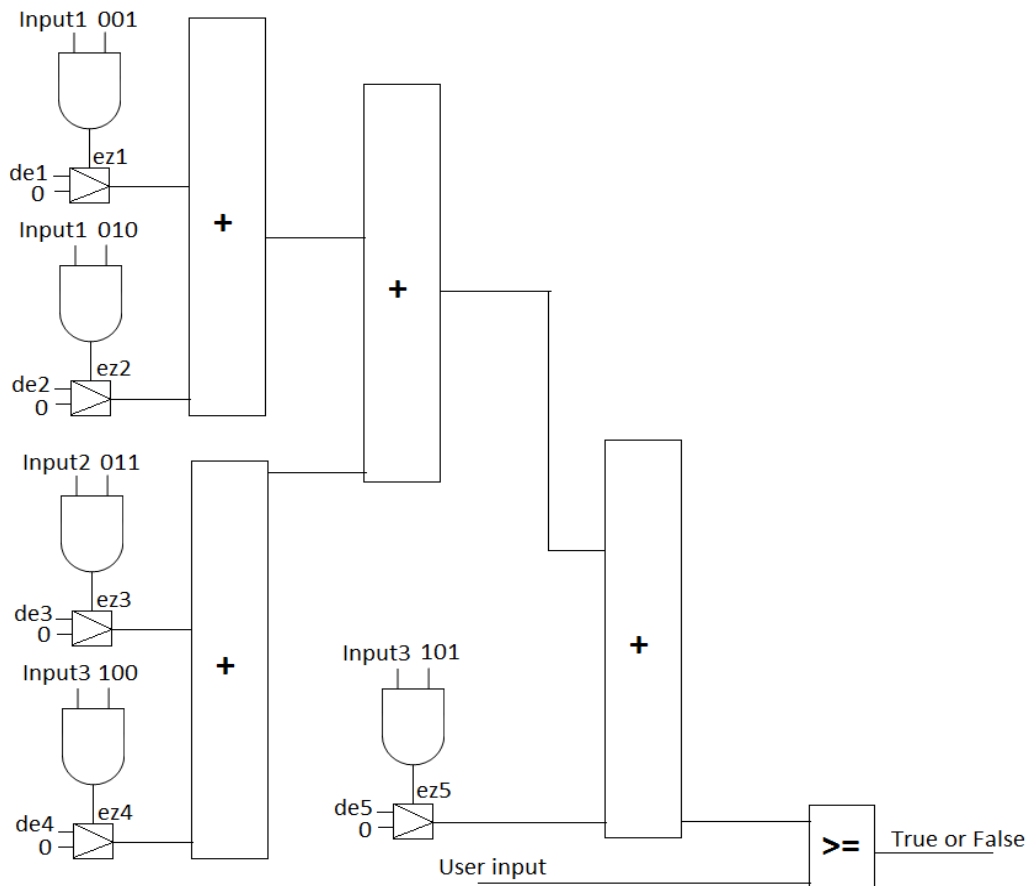
The algorithm for path length calculation is as follows:

- The path lengths and edges are already encoded.
- The user inputs edges through which we have to traverse the path.
- If the edges are traversed based on the constraints given above we have an output of '1' from the node selection design.

Hence if the input1 = “001”, input2 = “011” and input = “100”. Then we have an output of ‘1’ from the node selection design.

If the input1 = “100”, input2 = “101” and input3 = “111”, we have an output of ‘0’ from the circuit. This input acts as a select signal for the distance calculation. If the nodes are selected properly we pass the total distance oracle output.

The distance is simply calculated based on the edges given by the user.

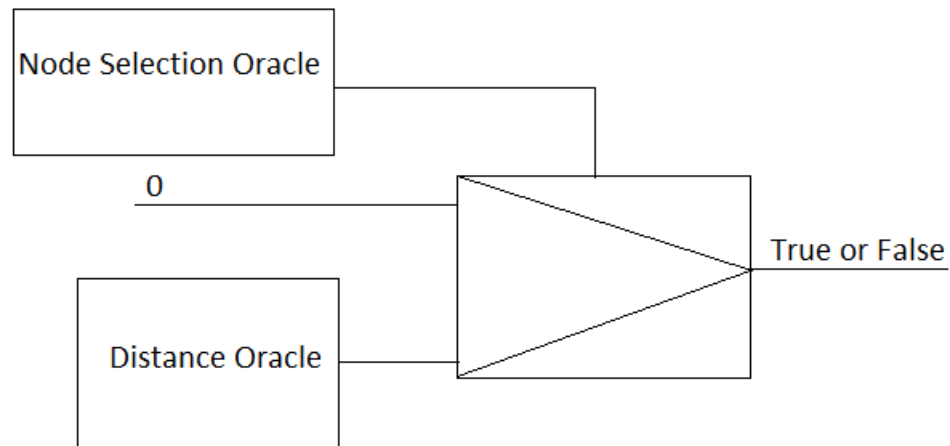


**Figure 11-11 Circuit for path length Calculation**

ez1, ez2, ez3, ez4, ez5 are the outputs from the 'and' gate of input and the edge. So if input2 is given as '011' then we have the ez3 as '1' and the particular length of the edge which is represented as de3 is selected and added using the adder. The distances of edges are represented as de1, de2, de3, de4, de5. Depending on the edge selected we add the distances.

### **Final Multiplexer:**

A multiplexer is used finally to represent the output. If the select signal is one, then the output form Longest Path oracle is transferred. If it is zero then a zero which represents the edges are not properly selected.



**Figure 11-12 Final Multiplexer for longest path oracle**

## Chapter 12 Conclusion to the thesis and Emulations on Oracle Based Processors

Constraint satisfaction problems offer an interesting paradigm in a way that solution may not exist for a given sets of constraints. Classical software running on general purpose; yet very powerful processor would still end up spending hours and hours on trying to satisfy these constraints. I proposed the idea of taking the aid of hardware configurable hardware for solving the constraint satisfaction problem. I very much believe specially tailored hardware should reduce the time required to solve CSP by thousand folds.

The CSP problems allow accelerating algorithms by implementing them (all or their crucial parts), in hardware. Such accelerators have for instance applications in robotics, multimedia and medical diagnostics.

I compared the speedup using two methods: (1) the clock time (Table 12.1) and the (2) speed-up formula from **(Haufe, 1998)**. The real time clock comparison was done and the results showed that solving constraint satisfaction problem using veloce save time by many folds. The details description of experiment is shown in table below.

## 12.1 Wall clock speed-up comparison

ORACLE	INPUT VECTORS traced	Modelsim simulation time in seconds (Running on nemo server)	Veloce Emulator (HDL Link mode) emulation time in seconds	Speed up of Veloce emulation with respect to nemo server)
Missionary and Cannibals	$2^{48}$	167018	1369	122
Man-Wolf-Goat-Cabbage	$2^{32}$	5322.30	147.83	36.00
SEND+MORE=MONEY	$2^{32}$	4683.11	117.085	39.97
Longest Path	$2^{17}$	26.565	0.483	55
Graph coloring	$2^{32}$	7865	182.11	42.95
TWO+TWO=FOUR	$2^{24}$	2322.16	61.86	37.53

**Table 12.1 Wall clock comparison**

As shown in Table 12.1, Speed-up I got in wall clock was typical 40. For large design I got wall clock speed-up to be 122 which was expected as at higher complexity typical digital simulator tends to get saturated resulting in lowered performance in terms of time consumed. This is the point where emulator holds edge over typical digital simulator. At higher design complexity emulator performance tends to be better in terms of time taken.

## 12.2 Industry method for quantifying effect of emulation

There is technical process to quantify the effect of emulations used in industry

For speed-up estimation, some parameters are required:

$n_{event}$  number of simulation events without using hardware acceleration

$m_{event}$  number of simulation events using hardware acceleration ( $m_{event} < n_{event}$ )

$a_{hard}$  number of hardware activations

$t_{event}$  average time to execute one simulation event including time required for component activation and component computation

$t_{hard}$  average time to process the logical protocol

$t_{trans}$  average time to transfer input and output vectors between simulator and hardware during a hardware activation cycle

$t_{clock}$  hardware clock period

$$F = (n_{event} - m_{event} + a_{hard}) * t_{event} / (n_{event} * t_{event})$$

$$N = (n_{event} - m_{event} + a_{hard}) * t_{event} / (a_{hard} * (t_{event} + t_{hard}))$$

$$t_{hard} = t_{trans} + t_{clock}$$

The maximum value for Speedup can be achieved on the condition  $N \rightarrow \infty$ . In practical applications, one can assume  $a_{hard} \ll n_{event}$  so that in the limit the maximum simulation speed-up is:

$$MaxSpeedup = \frac{1}{1 - F} = \frac{n_{event}}{m_{event}}$$

In case of pure hardware emulation ( $m_{event} = 0$ ), the maximum emulation speed-up is

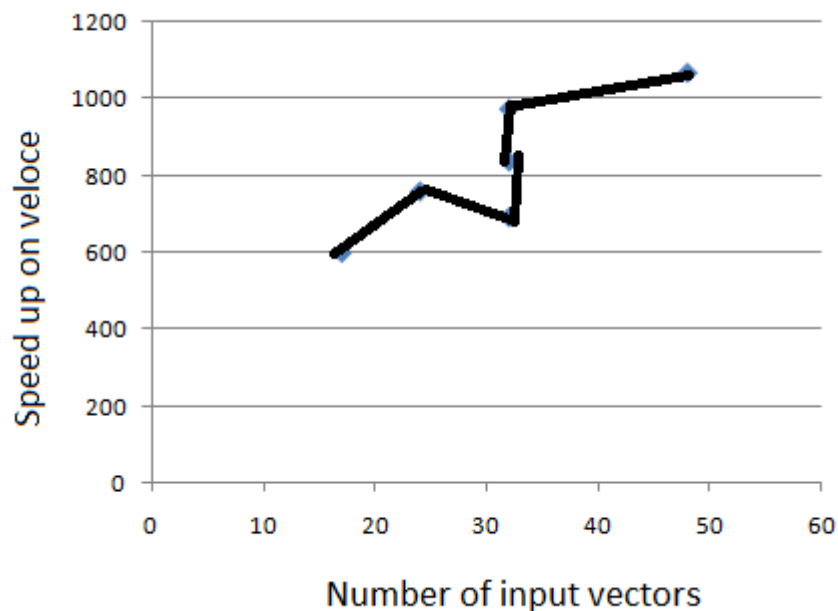
$$EmulationSpeedup = N = \frac{n_{event} * t_{event}}{a_{hard} * t_{clock}}$$

TheTable 12.2 summarizes results of my emulations based on the speed-up formula.

ORACLE	INPUT T Vector s Trace d	$n_{event}$	$t_{event}$ (us)	$a_{hard}$	$t_{clock}$ (us)	Speed up of Veloce emulation
Missionary and Cannibals	$2^{48}$	37279	76	3213	0.714	1069
Man-Wolf-Goat-Cabbage	$2^{32}$	41232	48	2743	0.714	975
SEND+MORE=MONKEY	$2^{32}$	27365	43	1963	0.714	839
Longest Path	$2^{17}$	21526	43	2153	0.714	602
Graph coloring	$2^{32}$	31284	51	3214	0.714	694
TWO+TWO=FOUR	$2^{24}$	27365	39	1963	0.714	761

**Table 12.2 Results of emulations**

As clearly seen from Table12.1, the speed up on veloce is around 600 to 1100 times. Our veloce solo is runs on speed of 1.4 MHz system clock but there are some emulators which runs upto 50MHz system bus. Such emulator will further improve the speedup. I believe such emulator will solve CSP problems in a real quick time. Commercial emulator usually runs much faster but it's a very expensive approach. A very high speed-up can be obtained by FPGA and processor based ASIC emulators. Speed-up values of more than 100,000 are reported (Rowson,1994) (Tamme,1994) (Turner,1997) (Kumar,1997). FPGA based prototyping can run at much higher speed of 10MHz-80MHz (Gallagher, 2006). Lastly current ASIC runs on GHz speed, so if current ASIC chips running on 3 GHz would be potentially many thousands times faster.



**Figure 12-1 Speed-up on Veloce against input vectors traced**

We can draw important conclusion from graph above, that as size of input vector increases we gets more speed-up. This is evident because as size of system (hence input vector increases) conventional simulator tends to get saturated in performance and this is point where emulator beats simulator.

I have exercised all my oracles using the counter approach. My test bench counter keeps on creating very many variable assignments for which the solution doesn't exist at all (this is typically seen in Man Wolf problem in which all bad beginnings are still generated). This technique ends up wasting extra time in finding solution to CSP due to lack of intelligence in test bench. So there lies some intelligent algorithm which beforehand can predict some direction where solution for given CSP may exist. One of such intelligent algorithm can be back tracking.



I now hope that new technology like Veloce allows to *practically* realize idea using specialized hardware for solving constraint satisfaction problem. If we can get bigger speed up of (>1000 times) on next generation emulator, my idea of solving constraint satisfaction problem using hardware would prove practical.

### 12.3 Final Conclusion

My final conclusions are the following:

1. The speedups are already quite high and the oracles solve the particular problems such as SEND+MORE=MONEY much faster than humans are able to solve them.
2. It is easier to create the emulation for many problems similar to those shown here than to write C++ software based on tree search, predicate calculus or some other advanced AI Methods.
3. My general approach is still too slow with the emulator used as an accelerator to be applied in practical problems of the size shown in thesis for practical robotics problems that are reduced to SAT or graph coloring and real-time problem solving and planning is expected.
4. Speedup grows when the size of the problem is larger and has a higher parallelism. This gives us the hope that the method can be still applicable as an accelerator for larger oracles.
5. More tests should be done on very large benchmarks and the exhaustive “counter approach” developed in this thesis.

6. More intelligent counters are necessary to exercise oracle, they can correspond to depth-first-search or breadth-first-search tree algorithms, for instance, or another methods of obtaining a feedback from the oracle, but this should be another thesis.

## References

- **(Dill, K. 1998)** Applying Genetic and Evolutionary Programming to Optimize Multi-Valued Networks in Arbitrary Finite Galois Fields. Ph.D thesis .
- **(Pasquino,2005)** T. Pasquino, "Tolapai: Kodiak Island Emulation System Hardware Design Verification/Test/Bring-up Plan," 2005.
- **(Giesecke,2004)** Ternary Quantum Logic. Physical Fundaments. Gates. Evolutionary and Exhaustive Search methods. Discussion of other possible approaches to minimization.
- **(Kida,1992)** The Cube Calculus Machine: a ring of asynchronous automata to process multiple-valued Boolean functions
- **(Koza,1999)** Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann Publishers .
- **(Dill,2001)** Baldwinian Learning utilizing Genetic and Heuristic for Logic Synthesis and Minimization of Incompletely specified data with Generalized Reed-Muller (AND-EXOR) forms. Journal of System Architecture 47 , pp. 477-489.
- **(Koza,1994)**Genetic Programming II: Automatic Discovery of Reusable Programs. (Cambridge, Massachusetts: The MIT Press, .
- **(Koza,1992)**Genetic Programming: On the Programming of Computers by Means of Natural Selection.
- **(Kida,1992)**Luis S. Kida, M. A. (92). THE CUBE CALCULUS MACHINE:A RING OF ASYNCHRONOUS AUTOMATA TO PROCESS MULTIPLE-VALUED BOOLEAN FUNCTIONS. IEEE International Symposium on .
- **(Perkowski,1999)** Learning in Hardware: Architecture and Implementation of an FPGA-Based Rough Set Machine.
- **(Rowson,1994)**J.A. Rowson: Hardware/Software Co-Simulation. Proc. 31st ACM/IEEE DAC, 1994, pp. 439-440.
- **(Tamme,1994)**S. Tamme: Rapid Prototyping for Circuits using High Level Design Tools. Proc. EuroDAC, pp. 682-687,Grenoble 1994

- **(Turner,1997)**R. Turner: In-circuit route leads to fast verification. EE-Times, Issue 935, Jan., 1997.
- **(Kumar,1997)**J. Kumar: Prototyping the M68060 for Concurrent Verification. IEEE Design & Test of Computers. January-March 1997, pp. 34 - 41
- **(Becker,1998)** Automatic parallelism exploitation for FPL-based accelerators.
- **(CHATTERJEE,2003)** Special Issue on Device Integration Technology for Mixed-Signal SOC.
- **(Daglio,2006)** A Complete and Fully Qualified Design Flow for Verification of Mixed-Signal SoC with Embedded Flash Memories .
- **(Gallagher,2006)** Verification Techniques: Going Beyond Simulation. Chip.
- **(Grama,1999)** State of the Art in Parallel Search Techniques for Discrete Optimization Problems.
- **(Hossain,2009)**Classical Search and Quantum Search Algorithms for Synthesis of Quantum Circuits and Optimization of Quantum Oracles.
- **(Impagliazzo,1998)**Which Problems Have Strongly Exponential Complexity?
- **(Leong,2006)** Design and test issues of an FPGA based data acquisition system for medical imaging using PEM.
- **(Moore,1993)**International Workshop on Field Programmable Logic and Applications.
- **(Okamoto,2003)** A fully integrated 0.13- $\mu$ m CMOS mixed-signal SoC for DVD player applications .
- **(Pan,2003)** Mixed signal validation of the Intel® Pentium® 4 microprocessor power-up sequence .
- **(Perkowski,1993)** FPGA COMPUTER ARCHITECTURES.
- **(Rubin,2007)**Reconfigurable FPGA-Based Hardware Accelerator for Embedded DSP.
- **(Sharma,2008)** Predicting the Correlation between Analog Behavioral Models and SPICE Circuits for robust SoC Verification.

- **(Song,2008)** The design of reversible gate and reversible sequential circuit based on DNA computing .
- **(Walters,1991)** "Computer-aided Prototyping for ASIC-based Systems," IEEE Design and Test of Computers, vol. 8, pp. 4-10, 1991 June.
- **(Zhang,2000)** Design of a VLIW compute accelerator on the Transmogriifier-2.
- **(Cadence)** Cadence Design Systems Inc., <http://www.cadence.com>, Verilog-XL Datasheet.
- **(Haufe, 1998)** Accelerated Logic Simulation by Using Prototype Boards
- **(Cadence NCSIM)** Cadence Design Systems Inc., NC-Sim Datasheet, <http://www.cadence.com>.
- **(IKOS)** Ikos Systems Inc., <http://www.ikos.com>, NSIM Datasheet.
- **(Cobalt Datasheet)** Quickturn Inc., <http://www.quickturn.com>, Cobalt Datasheet.
- **(Celaro Datasheet)** Mentor Graphics Corp., <http://www.mentor.com>, Celaro Datasheet.
- **(SR Datasheet)** Quickturn Inc., <http://www.quickturn.com>, System Realizer Datasheet.
- **(V-logic datasheet)** Ikos Systems Inc., <http://www.ikos.com>, VirtuaLogic Asic Emulation Datasheet.
- **(Hauck,1998)** Scott Hauck. The Roles of FPGAs in Reprogrammable Systems. In Proceedings of the IEEE, pp. 615-638, Apr. 1998.
- **(Tessier, 1993)** R. Tessier, J. Babb and A. Agarwal. "Virtual wires: Overcoming Pin Limitations in FPGA-Based Logic Emulators," in Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, April 1993, pp. 142–151.
- **(Selvidge, 2001)** C. Selvidge, M. Kudlugi, S. Hassoun and D. Pryor. "A Transaction Based Unified Simulation/Emulation Architecture for Functional Verification," in Proceedings of the 38th Design Automation Conference, June 2001.
- **(Ikos TIP, 2000)** Ikos Systems Inc., Architectual Notes : TIP, January 2000.

- **(Ikos TIP Co-Modeling)** Ikos Systems Inc., TIP Co-Modeling Environment SRS, February 2000.
- **(TIP-User's guide)** Ikos Systems Inc., TIP User's Guide, March 2001.