

**EVOLUTIONARY ALGORITHMS FOR  
SYNTHESIS AND OPTIMIZATION OF  
SEQUENTIAL LOGIC CIRCUITS**

By

**Belgasem Ali**

**BSc, MSc, MIEE**

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

**Doctor of Philosophy**

**Napier University**

**Edinburgh**

May 2003

© Copyright by Belgasem Ali

## **Abstract**

Considerable progress has been made recently in the understanding of combinational logic optimization. Consequently a large number of university and industrial Electric Computing Aided Design (ECAD) programs are now available for optimal logic synthesis of combinational circuits. The progress with sequential logic synthesis and optimization, on the other hand, is considerably less mature.

In recent years, evolutionary algorithms have been found to be remarkably effective way of using computers for solving difficult problems. This thesis is, in large part, a concentrated effort to apply this philosophy to the synthesis and optimization of sequential circuits.

A state assignment based on the use of a Genetic Algorithm (GA) for the optimal synthesis of sequential circuits is presented. The state assignment determines the structure of the sequential circuit realizing the state machine and therefore its area and performances. The synthesis based on the GA approach produced designs with the smallest area to date. Test results on standard finite state machine (FSM) benchmarks show that the GA could generate state assignments, which required on average 15.44% fewer gates and 13.47% fewer literals compared with alternative techniques.

Hardware evolution is performed through a succession of changes/reconfigurations of elementary components, inter-connectivity and selection of the fittest configurations until the target functionality is reached.

The thesis presents new approaches, which combine both genetic algorithm for state assignment and extrinsic Evolvable Hardware (EHW) to design sequential logic circuits. The implemented evolutionary algorithms are able to design logic circuits with size and complexity, which have not been demonstrated in published work.

There are still plenty of opportunities to develop this new line of research for the synthesis, optimization and test of novel digital, analogue and mixed circuits. This should lead to a new generation of Electronic Design Automation tools.

## ACKNOWLEDGEMENTS

I would like to thank my supervisors Prof. A.E.A Almaini, Mr. M. MacCallum (Napier University) and Dr. T Kalganova, (University of Brunel), for their guidance, help, and constant support. I very much appreciate their immense patience and willingness to discuss my ideas about evolution, adaptation, electronic circuits design, and of course, science. I am grateful to Prof. T Muneer for always leaving his door open for stimulating discussions on science and many other problems that may occur in the world of Human beings. I am grateful to members our research group, and especially to Dr. Y. Xia, Dr. L. Wang, K. Farag and M. Yang for being colleague and a good friends.

Thanks also to those people who contributed anonymously by reviewing different parts of the thesis that later on have appeared as publications in journals and conference proceedings.

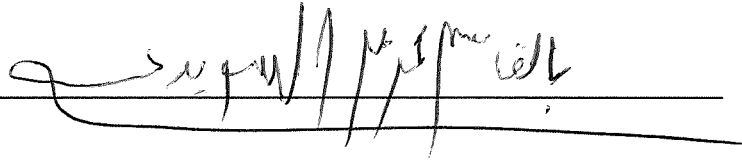
Finally, I would like to thank my parents for being patient with me, my brothers for their support, and my wife, for being always on my side in the struggle for survival.

I would like to say to those people that I am indebted to them for life and that the memories of the time spent with them at Napier University will always be with me, cherished forever and relived again and again.

## DECLARATION

I hereby declare that I have carried out all the work presented in this thesis.  
No part of this work has previously been submitted in support of a degree  
validated by this or any other University.

B. Ali

A handwritten signature in black ink, written over a horizontal line. The signature is stylized and appears to be 'B. Ali'.

## TABLE OF CONTENTS

List of Figures	VIII
List of Tables	XI
List of Abbreviations	XIII

### *Chapter 1*

#### **General Introduction**

1.1 Overview .....	1
1.1.1 Objectives .....	1
1.1.2 Generalities .....	2
1.1.3 Evolutionary Electronics .....	4
1.1.4 Electronic circuit design: Design and search .....	7
1.2 Scope of the Thesis .....	9
1.3 The position of this thesis within the field .....	10
1.4 Outline of the dissertation .....	11
1.5 Summary .....	14

### *Chapter 2*

#### **Review of the logic design process**

2.1 Introduction .....	15
2.2 Overview of optimal logic synthesis .....	16
2.3 Logic synthesis with ECAD tools .....	18
2.4 VLSI Design Flow .....	19
2.5 Probing the limits of Logic Synthesis .....	23
2.6 Sequential logic synthesis .....	25
2.6.1 Sequential circuits .....	25
2.6.2 Early work on sequential logic synthesis .....	29
2.6.3 Classic Synthesis Trajectory .....	30
2.6.3.1 State Minimization .....	31
2.6.3.2 State assignment of finite state machines .....	32
2.6.3.3 A Problem with existing state assignment approaches .....	43
2.6.4 Concurrent state minimization and assignment to improve area .....	38
2.7 CAD Algorithms and tools .....	39

2.8 Other approaches to FSM design and optimaztion.....	40
2.9 Summary .....	41

## Chapter 3

### Evolutionary Algorithms: A tool for Digital design

3.1 Introduction.....	42
3.2 Outline of Evolutionary Algorithms.....	43
3.2.1 Genetic Algorithms.....	45
3.2.1.1 Usage Requirements of Genetic Algorithms.....	46
3.2.1.2 Genetic Operators.....	52
3.2.1.3 Elitist Selection and Fitness Scaling .....	61
3.2.2 Genetic Programming.....	55
3.2.2.1 GP Crossover Operator .....	56
3.2.2.2 GP Mutation .....	57
3.2.3 Evolutionary Programming.....	57
3.2.4 Evolution Strategies .....	58
3.2.5 Phenotypic versus Genotype Evolution.....	59
3.3 Evolutionary designs of digital circuits.....	61
3.4 Evolvable Hardware .....	62
3.4.1 Introduction.....	62
3.4.2 Taxonomy of EHW.....	65
3.4.2.1 Evaluation process .....	65
3.4.2.1.1 Extrinsic EHW .....	65
3.4.2.1.2 Intrinsic EHW .....	67
3.4.2.2 Evolving platform.....	68
3.4.2.3 Evolution process .....	71
3.5 Differences from traditional methods .....	72
3.6. Evolvable hardware implementation .....	73
3.6.1 Advantages of evolutionary design.....	75
3.6.2 Disadvantage of evolutionary design .....	75
3.7 Summary .....	77

## Chapter 4

### Optimal State Assignment of Finite State Machine

4.1 Introduction .....	78
4.2 Complexity of the state assignment problem.....	79
4.2.1 Problem Formulation.....	81
4.2.2 Motivation example.....	82
4.3 Automated state assignment .....	86
4.4 Mechanical state assignment .....	86
4.5 Previous Application of Genetic Algorithm to State Assignment.....	87
4.6 Genetic Algorithms for states assignment problem .....	90

4.6.1 Chromosome representation for GA State assignment problem .....	90
4.6.2 Fitness function .....	92
4.7 Genetic operators .....	94
4.8 Experimental results .....	97
4.8.1 Parameter Effects .....	98
4.8.2 Parameter space .....	99
4.8.3 Crossover and mutation .....	100
4.8.4 Population Size .....	105
4.8.5 GA Generation .....	108
4.9 Comparison of results .....	112
4.10 Comparison of GA algorithm with different approaches .....	118
4.11 Discussion of Results .....	121
4.12 Summary .....	122

## **Chapter 5**

### **Evolutionary Design for logic Circuits**

5.1 Introduction .....	123
5.2 The space of all representations .....	125
5.2.1 Representation of Boolean function .....	126
5.2.2 Reed-Muller function .....	129
5.2.3 Mixed function .....	130
5.3 Backgrounds and Motivation .....	130
5.4 Basic idea of the proposed approach .....	136
5.5 Problem Modelling .....	137
5.6 Encoding of evolution .....	139
5.6.1 Chromosome representation and connectivity .....	140
5.6.2 Genetic operators .....	142
5.6.3 Connection repair algorithm .....	146
5.6.4 Criteria used in an extrinsic EHW .....	147
5.6.4.1 Fitness function .....	148
5.6.4.2 Test vectors .....	149
5.6.5 Definition of subcircuits for combinational logic circuits .....	150
5.7 Experimental results .....	153
5.7.1 Example 1: Serial Adder .....	154
5.7.2 Example 2: Module-4 counter .....	157
5.7.3 Example 3: sequence detector .....	159
5.7.4 Example 4: 1010 detector .....	161
5.8 Analysis of the results .....	163
5.9 Summary .....	168

## ***Chapter 6***

### **Extrinsic Evolution of Finite State Machine**

6.1 Introduction .....	169
6.2 Evolution of MCNC FSMs .....	169
6.3 EHW to design the combinational part of the circuit .....	172
6.4 Motivating Example .....	173
6.5 Experimental results .....	181
6.6 Summary .....	186

## ***Chapter 7***

### **Conclusions and further work**

7.1 Conclusions .....	187
7.2 Future work .....	192

<b>Pubilcations</b> .....	194
---------------------------	-----

<b>References and Bibliography</b> .....	195
--	-----

### **Appendixes**

Appendix A Benchmark Kiss file .....	A-1
Appendix B SIS for FSM .....	B-1
Appendix C Extrinsic EHW tools .....	C-1
Appendix D CD Software .....	D-1



## List of figures

Number	Page
Figure 1.1. Space of all designs.....	9
Figure 2.1. Typical VLSI design flow in three domains (Y-chart representation). ....	20
Figure 2.2. A more simplified view of VLSI design flow.....	21
Figure 2.3. Level of design.....	23
Figure 2.4. Structure view of a finite state machine. ....	26
Figure 2.5. An example of a State Transition Graph (STG) .....	28
Figure 2.6. Traditional sequential synthesis trajectories. ....	31
Figure 3.1. Use of the evolutionary algorithms (EA) to create electronic circuits .....	43
Figure 3.2. General outlines of any evolutionary algorithms method. ....	43
Figure 3.3. Execution of the genetic algorithms.....	45
Figure 3.4. Coding of three parameters in a single string.....	47
Figure 3.5. A Random initial population with chromosome length of 12 bits.....	48
Figure 3.6. The fitness of each string has been computed. ....	49
Figure 3.7. The new population with the rank method. ....	50
Figure 3.8. The fitness of the strings is normalized. ....	51
Figure 3.9. The biased roulette wheel. ....	51
Figure 3.10. Single point crossover operator. ....	53
Figure 3.11. Two point crossover. ....	53
Figure 3.12. The mutation operator changed a random bit of the string.....	54
Figure 3.13. GP example. ....	56
Figure 3.14. Simplified GP crossover operators.....	57
Figure 3.15. Evolutionary programming algorithms.....	58
Figure 3.16. The evolution strategy algorithm. ....	59
Figure 3.17. Mapping between genotype space and phenotypic space. ....	60
Figure 3.18. The concept scheme of Evolvable hardware. ....	63
Figure 3.19. Circuit design problem in EHW. In EHW approach, the circuits in initial population are generated randomly and do not implement the desired logic function. Therefore, an evolutionary algorithm design a circuit that correctly implements given	

logic function and optimises fully functional circuit. In other words, evolutionary algorithm evolves a logic circuit [24].	64
Figure 3.20. Taxonomy of Evolvable Hardware	65
Figure 3.21. On-line EHW	68
Figure 3.22. Basic structure of a generic FPGA circuit	69
Figure 3.23. The primitive logic gates	71
Figure 4.1. Karnaugh maps for D flip-flops realization and output for assignment 1	83
Figure 4.2. Karnaugh maps for D flip-flops and output for assignment 2	84
Figure 4.3. Karnaugh maps for D flip-flops and output for assignment 3	85
Figure 4.5. Roulette wheel parent selection	93
Figure 4.7. Effect of Crossover and mutation on chromosome Fitness	95
Figure 4.8 Effect of P (Crossover) for example modulo12	102
Figure 4.9. Effect of P (Mutation) for example modulo12	102
Figure 4.10. Effect of P (Crossover) for example dk512	103
Figure 4.11. Effect of P (Mutation) for example dk512	103
Figure 4.12. Effect of P (Crossover) for examples modulo12, dk512 and dk16	104
Figure 4.13. Effect of P (Mutation) for examples modulo12, dk512 and dk16	104
Figure 4.14. Effect of Population Size	107
Figure 4.15. Effect of Population Size	107
Figure 4.16. Variation of No. Generations with No. Gates	108
Figure 4.17. Variation of No. Generations with No. literals	110
Figure 4.18. Variation of No. Gates with No. Generations	110
Figure 4.19. Variation of No. literals with No. Generations	111
Figure 4.20. Comparison of GAs with Nova based on number of gates	114
Figure 4.21. Comparison of GAs with Nova based on number of Literals	114
Figure 4.22. Comparison of GA with Mustang, Partitioning and codable column	119
Figure 4.23. Comparison of GA with Random and binary state assignments	120
Figure 5.1. The automatically circuit designed using an adaptive algorithm	124
Figure 5.2. How assemble-and-test reaches the unknown regions of the space of all representations	126
Figure 5.3. Comparison between two-level and multilevel structures	128
Figure 5.4. RM form is an exclusive-OR sum of products	129
Figure 5.5. Evolving an FPGA configuration using a simple genetic algorithm	132
Figure 5.6. Procedure of the proposed approach to design sequential logic circuit using genetic algorithm and extrinsic evolvable hardware	137
Figure 5.7. Description of the circuit's parts	138
Figure 5.8. The procedure of generating the circuits from the symbolic state transition table	139

Figure 5.9. The representations for a hypothetical inputs/outputs.....	140
Figure 5.10 Schematic of the chromosome structure used in EHW approach with layout 2x2.....	141
Figure 5.11. Three mutation operators used by the genetic algorithm.....	144
Figure 5.12. The geometry mutation process (3x3 circuit geometry) [24].....	145
Figure 5.13. Test vectors.....	150
Figure 5.14. Circuit structure is implemented according to state table given in Table 5. 3. ....	152
Figure 5.15. Serial adder (a) State transition graph, and (b) state transition table ; where. #I/p inputs, #Ps present state, #Ns Next state, #O/p Outputs. ....	155
Figure 5.16. Evolved Serial adder using (a) functional set (0-6, 15,16), (b) functional set (0-6, 10).....	156
Figure 5.17. Module-4 counter a)-state transition graph, b) State table and c) State assignment generated by GA.....	157
Figure 5.18. Evolved optimal circuit solution of the model-4 counter.....	158
Figure 5.19. A sequence detector described as a) state transition graph, b) state table, c) State assignment generated by GA. ....	159
Figure 5.20. Evolved optimal circuit solution of the sequence detector with circuit layout 4x5. ....	161
Figure 5.21. 1010 Detector a) state transition graph, b) state transition table, c) state assignment .....	161
Figure 5.22. Evolved optimal circuit solution for 1010 detector using 4x5-circuit layout. ....	162
Figure 5.23. Fitness distribution in 10 runs .....	165
Figure 5.24.The graphic illustration how the circuit layout and evolves functionality affect the solutions of sequence detector. ....	166
Figure 6.1. Procedure of the proposed approach to design FSM circuit using genetic algorithm and extrinsic evolvable hardware.....	170
Figure 6.2. Gate level representation of a sequential circuit.....	171
Figure 6.3. DK27 State Diagram and symbolic state transition table .....	175
Figure 6.4.The procedure of generation the *.pla file from the state transition table based on example of dk27 (Kiss2 benchmark). Step1 shows the initial symbolic State Transition Table, Step2 generates State Assignments using the genetic algorithm and Step 3 generates the PLA files (*.pla) based on the state assignments obtained. ....	176
Figure 6.5. An example of the phenotype and corresponding genotype of a chromosome with 3x4 circuit layout. Functional set (0-15). ....	178
Figure 6.6. An example of the phenotype and corresponding genotype of a chromosome with 3x4 circuit layout. Functional set (0-15) .....	179
Figure 6.7. Evolved dk27 design using (a) functional set (0-10) (b) functional set (0-15).....	180

## LIST OF TABLES

<i>Number</i>	<i>Page</i>
Table 1.1 Some key publication in evolutionary electronics.....	6
Table 2.1. An example of a State Transition Table (STT).....	28
Table 2.2. Some Commercial design Synthesis tools. ....	39
Table 3.1. Paradigms in Evolutionary Algorithms.....	44
Table 3.2. The different between design process and Evolution process. ....	73
Table 4.1. Number of Different state assignments. ....	80
Table 4.2. State Table with three State Assignments.....	82
Table 4.3. State transition t able for assignment 1.....	82
Table 4.4. State transition table for assignment 2. ....	83
Table 4.5. State transition table for assignment 3. ....	84
Table 4.6. Gate comparison for three state assignments. ....	85
Table 4.7. Summary of available approaches to find optimal state assignment.....	88
Table 4.8. Example state mutation .....	95
Table 4.9. Statistics of benchmark examples.....	99
Table 4.10. Compression crossover and mutation. ....	101
Table 4.11. Population size numbers of gates and number of literal.....	106
Table 4.12. Result number of generation.....	109
Table 4.13. Comparison of GAs with Nova based on the number of gates. ....	113
Table 4.14. Comparison of GA with Nova based on the number of literal.....	115
Table 4.15. Results for some Benchmark circuits.....	117
Table 4.16. State assignments for each algorithm.....	117
Table 4.17. Comparison of GA state assignment with optimal algorithms. ....	118
Table 4.18. Comparison of GAs state assignment with mechanical algorithms. ....	118
Table 5.1. Recent EHW approach used to evolve sequential logic circuits .....	135
Table 5.2. The full set of components, gates 0 to 15 and their logic functions.....	142
Table 5.3. The transformation process of STT in to PLA file format. Where <i>i</i> inputs = input + present state bits, <i>.o</i> designed the number of outputs calculated, outputs =next state +output bits, <i>.p</i> is the number of product terms, <i>.e</i> is end of file. ....	151
Table 5.4. Initial parameters used to evolve a circuit. ....	152
Table 5.5. Solution obtained for serial adder using EHW approach and manual method [1].....	155
Table 5.6. Solution obtained for model-4 counter using EHW approach and manual method.....	158

Table 5.7. Solutions obtained for sequential detector produced by proposed approach and manual method.....	160
Table 5.8. Solution obtained for 1010 detector produced using proposed method and manual design. ....	162
Table 5.9. Initial parameters used to evolve the circuits and their results.....	164
Table 5.10. Sequence detector experimental results the algorithm performance during the circuit layout and functionality distributions with variable circuit structure and Function set: 2,6,7,8.....	165
Table 5.11. Module-4 counters with fixed circuit geometry and Function set: 2-8,12.....	167
Table 6.1. Initial parameter used to evolve sequential logic circuit (Dk27.kiss2) .....	174
Table 6.2. A typical netlist chromosome for 100% functional of Dk27A next state logic.....	179
Table 6.3. State assignments generated by GA.....	181
Table 6.4. Experimental results of extrinsic EHW approach.....	183
Table 6.5. Experimental results of extrinsic EHW approach. #in, #out and #stat are the number of inputs, outputs and states respectively. #100 cases is the number of fully functional solutions obtained after 100 runs of GA. The evolved circuits which are more optimal in comparison with SIS [73] are shown in bold.....	185

## GLOSSARY

### List of Abbreviations

AI	Artificial Intelligent
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
ASM	Algorithm State Machine
BDD	Binary Decision Diagram
BDT	Binary Decision Tree
BE	Boolean Expression
BED	Binary Expression Diagram
CAD	Computer Aided Design
CMOS	Complementary Metal-Oxide-Semiconductor
CNF	Conjunctive Normal Form
CPU	Central Processing Unit
CSP	Constraint Satisfaction Problems
DNA	Deoxyribo Nucleic Acid
DNF	Disjunctive Normal Form
DSP	Digital Signal Processing
EA	Evolutionary Algorithms
EECD	Evolutionary Electronic Circuit Design
EHW	Evolvable Hardware
EP	Evolutionary Programming

ES	Evolutionary Strategy
ECAD	Electronic Computer Aided Design
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FBDD	Functional Binary Decision Diagram
GAL	Generic Array Logic
GAs	Genetic Algorithms
GP	Genetic Programming
HDL	Hardware Description Language
ICs	Integrated Circuits
ITE	If_Then_Else normal form
KISS	Keep Internal State Simple
LSI	Large Scale Integration
LUT	Look-Up Table
MOS	Metal oxide semiconductor
MCNC	Microelectronic Centre of North Carolina
NP	class of Non-Deterministic
OX	Output Circuit Crossover
OFDD	Ordered Functional Decision Diagram
PALs	Programmable Array Logic
PBIL	Population Based Incremental Learning
PLD	Progranatnable Logic Device
PLA	Programmble logic Array

POS	Product Of Sums normal form
R-M	Reed-Muller
ROM	Read Only Memory
ROBDD	Reduced Ordered Binary Decision Diagram
RDAG	Rooted and Directed Acyclic Graph
RTL	Register transfer level
SA	Simulated Annealing
SIS	Sequential Interactive System
SOC	System On Chip
SOP	Sum Of Products
SPICE	Simulation Program with Integrated Circuit Emphasis
STG	State Transition Graph
STT	State Transition Table
TT	Truth Table
TSP	Travelling Salesman Problem
ULM	Universal Logic Module
VGA	Variable Length Chromosome
VLSI	Very Large Scale Integration
XOR	Exclusive OR



## TERMINOLOGY USED IN THE THESIS

**Logic design:** A design as a network of logical components

**Logic synthesis:** A usual way of carrying out logic design in which gates are generation from an abstract behavioural/structure description.

**Netlist:** A complete structural description of a circuit, which is ready to implement.

**Combinational circuit:**

A circuit is combinational if it computes a function, which depended only on the inputs applied to the circuit; for every input value, there is a unique output value.

**Sequential circuit**

Sequential circuits are basically combinational circuits with additional storage elements and feedback.

*State minimization:* Combine equivalent states of a state machine to reduce the number of states. For most cases, minimizing the states results in smaller logic, although this is not always true.

*State assignment:* Assign a unique binary code to each state of a finite state machine. *State* a collection of state variable whose value at any one time contains all the information about the past value necessary to account for future behaviour.

*An evolved logic circuits:* A logic circuit is evolved mean that the evolutionary algorithm started with randomly generated non-functional circuit and through evolution, finds a fully functional circuit. The circuit is evolved based on the test-and-assemble method.

**Evolvable Hardware**

*A function set* of logic gates, is a set of logic gates from which the circuit can be assembled. For example the AND-OR, the function set of logic gates contains AND, OR and NOT logic gates. In evolvable hardware any set of logic gate can be chosen. Each logic gate in a functional set of logic gates is encoded with integer.

**Adaptation**

The process of generating a set of behaviours that more closely match or predict a specific environmental regime. An increased ecological-physiological efficiency of an individual relative to others in the population. The response of an organism to the present stimulus and its present state. It is the total sum of behaviours of an organism that define the fitness of the organism to its present environment; thus it is the operative function against which selection operates.

**allele**

One of a set of possible values for a gene. In a binary string genome, the alleles are 0 and 1.

**Chromosome**

Rod-shaped bodies in the nucleus of cells, most visible particularly during cell division, which contain the hereditary units or genes. A data structure (e.g. binary bit-string or an array of integers), which holds a "string" of task parameters, or genes. In evolutionary algorithms, chromosome is often used to refer to a genome.

**Crossover**

In genetic algorithms, a reproduction operator which forms a new chromosome by combining parts of each of two "parent" chromosomes.

**Ecosystem**

Biological community of interacting organisms and their physical environment.

**Evolution**

The process of change which is assured given a reproductive population in which there are (1) varieties of individuals, with some varieties being (2) heritable, of which some varieties (3) differ in fitness (reproductive success).

**Evolutionary algorithms**

Search and optimisation techniques based on the principles of natural evolution.

**Evolution strategy**

A type of evolutionary algorithm developed in the early 1960s in Germany. It employs real-coded parameters, and in its original form, it relied on mutation as the search operator and a population of size one. Since then it has evolved to share many features with genetic algorithms.

**Evolutionary computation**

Encompasses methods of simulating evolution on a computer.

**Extrinsic EHW** the evolution is simulated in software, and only the elite chromosome (i.e. the configuring bit string) gets written. Thus the circuit is configured only once.

**Exploration**

The process of visiting entirely new regions of search space, to see if anything promising may be found there.

**Exploitation:** Local search.

**Exploration:** Global search.

**Fitness**

A summation of the quality of environmental prediction by an organism throughout its range of regimes. The probability of or propensity for survival of an individual or population. A value assigned to an individual who reflects how well the individual solves the task at hand.

**Genetic programming**

Genetic algorithms applied to programs. Genetic programming is more expressive than fixed-length character string genetic algorithms, though genetic algorithms are likely to be more efficient for some classes of problems.

**Gene**

A unit of heredity located on a chromosome and composed of DNA. *Gene is* the smallest unit in a genome. In a binary string genome, the bits are genes.

In an array of characters, each character in the array is a gene.

**Genetic operator**

A search operator acting on a coding structure that is analogous to a genotype of an organism (e.g. a chromosome).

**Genotype**

The sum of inherited characters maintained within the entire reproducing population. Often also used to refer to the genetic constitution underlying a single trait or set of traits.

**Heredity**

The transmission of characteristics from parent to offspring through the gametes.

**Hyperspace**

A Cartesian co-ordinate space of high dimension, typically higher than three dimensions.

**Individual**

A single member of a population. In evolutionary computation, each individual represents a potential solution to the problem.

**Intrinsic EHW** the circuit gets configured for each chromosome for each generation (on-line).

**Mutation**

A reproduction operator, which forms a new individual by making alterations (usually small) to the parent.

**Natural selection**

The result of competitive exclusion as organisms fills the available finite resource space.

**Netlist**

For an electronic circuit is a data structure consisting of unordered list that define both topology and sizing of the circuits.

**Normal distribution**

A probability distribution that approximates a bell-shaped curve in shape.

**Offspring**

An individual generated by any process of reproduction.

**Phenotype**

The behavioural expression of the genotype in a specific environment. The realised expression of the genotype.

**Pleiotropy**

The capacity of a gene to affect a number of different phenotypic characteristics.

**Poisson distribution**

A discrete probability distribution commonly used to define arrivals in a queuing system.

**Polygeny**

The circumstance where a single phenotypic is affected by multiple genes.

**Population**

A group of individuals which may interact together, for example, by producing offspring.

**Probabilistic**

Models developed under conditions of uncertainty.

**Reproduction**

The creation of a new individual from two parents (sexual reproduction). Asexual reproduction is the creation of a new individual from a single parent.

**Reproduction operator**

A mechanism, which influences the way in which genetic information is passed on from parent(s) to offspring during reproduction. Reproduction operators fall into three broad categories: mutation, crossover and reordering operators.

**Search space**

Generally, if the solution to a problem can be represented using a representation scheme R, then the search space is the set of all possible configurations, which may be represented in R.

**Selection**

The process by which some individuals in a population are chosen for reproduction, typically on the basis of favouring individuals with higher fitness.

**Species**

A group of similarly constructed organisms that is capable of interbreeding and producing fertile offspring. A population whose members are able to interbreed freely under natural conditions.

**Stochastic**

A situation in which imprecise or random events affect values of variables, so that results can be given only in terms of probabilities.

## Chapter 1

### INTRODUCTION

*Chips that evolve and adapt to an environment*

*The processor inside a computer is designed to be a jack-of-all-trades and master of none, as people do many different things with their computers, from playing games to word processing. However, researchers are currently creating computer chips that can adapt themselves to particular software called Evolvable Hardware (EHW). Rather than being programmed, the chips learn as needed. The first likely use of the new chips will be in supercomputers or satellites, but eventually they will be used in everyday computers.*

*However, before this happens, the basics of designing chips using evolution need to be mastered.*

*Telegraph Connected, 8 April 2000*  
[www.telegraph.co.uk](http://www.telegraph.co.uk)

#### 1.1 Overview

This thesis addresses several problems in sequential circuit optimisation, that is, transformations on finite state machines which produce machines having equivalent behaviour, but whose implementations are either more economical, faster, or both. In particular, the focus of this thesis is a suite of algorithms and tools, which synthesize small, high performance realizations of finite state machines, given a suitable specification.

##### 1.1.1 Objectives

Biological organisms are among the most intricate structures known to man, exhibiting highly complex behavior through the massively parallel cooperation of huge numbers of relatively simple elements, the cells. As the development of computing systems approaches levels of complexity such that their synthesis begins to push the limits of human intelligence, more and more engineers are beginning to look at nature to find inspiration for the design. This thesis will present one such endeavor, notably an attempt to draw inspiration from biology in order to design novel digital circuits, endowed with a set of features motivated and guided by the behavior of biological systems: *self-replication* and *self-repair*.

The first phase of this research looks at the state assignment problem of automated synthesis of synchronized sequential circuits and highlights the importance of considering the genetic algorithm and its parameters and how it is implemented to solve the state assignment problem for logic synthesis and optimisation. The genetic algorithms (GA) approach to state assignment has been evaluated by comparing with both industry standard tools and other research published in this area [1].

The second goal of the research is to develop Evolvable Hardware (EHW) approach capable to design practical digital logic circuits.

In order to achieve this goal it was necessary to:

1. Investigate the evaluation process in the extrinsic gate-level evolvable hardware apparatus
2. Develop high-level self-adaptive EHW approach
3. Design an extrinsic EHW approach that is capable of evolving large benchmark circuits.

The proposed approach consists of four main stages. The first stage is concerned with the use of GA for the state assignment problem to compute optimal binary codes for each symbolic state to construct the state transition table of finite state machines (FSM). The second stage defines the subcircuits required to achieve the desired functionality. The third stage evaluates the subcircuits using extrinsic Evolvable Hardware. During the fourth stage, the final circuit is assembled. The obtained results compare favourably with those produced by manual methods and other methods based on heuristic techniques.

The focus of the research has mainly been directed towards traditional technology and applying evolutionary techniques to evolve new and hopefully

better circuits and thus prove the worth of EHW as a design technique. The EHW will then be compared with the traditional techniques.

The motivation behind the study of digital circuit evolution is to design circuits that are more efficient than the conventional designs, and then, to learn new principles of design.

Thus new methods and principles of evolving digital circuits are inferred.

### **1.1.2 Generalities**

Evolutionary Algorithms, as defined in [2] is a "collection of computational search, learning, optimization and modeling methods loosely inspired by biological evolution". Some of those methods are evolutionary programming (EP) [3], evolution strategies (ESs) [4] and genetic algorithms (GA) [5]. In the first part of this research genetic algorithms will be used for solving the state assignment problem of finite state machine. A more detailed description of the genetic algorithms and their use will be given in chapter 3. It is sufficient here to say that since their initial development in the 1970 genetic algorithms, as a tool of optimization, have had many successes in solving problems where the search space is huge. The fact that they evaluate in parallel many different solutions confers on them the ability to solve some optimization problems much faster than more conventional optimization techniques.

In the past, evolutionary algorithms methods have essentially been applied to software applications. It is only recently that those methods have been applied to the design of hardware circuits. Therefore in the second part of this research we use the evolutionary algorithms for the design of circuits. One of the main reasons for trying to design hardware circuits using alternative methods is to find some better circuits (for example smaller, faster or less power than those which could be designed using conventional techniques).

A circuit designer is limited by the set of mathematical models, the rules and techniques that he/she learned, whilst by freely exploring the space of all possible circuits new unconventional designs may be found. Of those



alternative methods, evolutionary computation, and more precisely genetic algorithms, come naturally to mind: they are very efficient methods for search in large spaces. The result of applying evolutionary algorithms to the design of hardware circuits is named Evolvable Hardware [1, 6]. There are presently two different trends in the field of EHW. On one side there are those who believe in a truly unconstrained evolution: the space of all possible circuits is freely explored. This approach can lead to spectacular results but also to failures. By tackling a too complex task the risks are high to the extent that no solution can be found [7]. On the other side there are those who perform hardware evolution within some constraints, generally by giving a structure to the circuits being evolved, thus reducing the size of the search space. The search space being smaller, the evolutionary process can explore it faster than with unconstrained evolution [1]. Some very interesting results have also been found, such as finding combinational logic circuits using fewer gates than what is obtained by using Karnaugh Maps and Boolean algebra or the Quine-McCluskey procedure [6].

Evolutionary techniques require evaluation and modification of large populations of individuals until a solution is found. For this reason, reprogrammable FPGAs (Field Programmable Gate Array) are usually the tool of choice, but not the only one, for EHW implementation [8].

### **1.1.3 Evolutionary Electronics**

Evolution of electronic circuits has been intensively investigated for the last decade. Evolutionary Electronics is a research area, which involved application of evolutionary algorithms in the domain of electronics. The main challenge of the area is the evolution of circuits for industrial applications and also to find methods to improve the performance of Evolutionary Algorithms in electronic circuit synthesis.

Evolutionary Electronics Circuit design applies the concepts of genetic evolution to the evolution of electronic circuits [9]. The main idea behind this research field is that each possible electronic circuit can be represented as an

individual or a chromosome of an evolutionary process, which performs standard genetic operations over the circuits. Due to the broad scope of the area, researchers have been focusing on different problems, such as optimization of combinational and sequential digital circuits [1, 10], synthesis of digital circuits [11], placement and routing [12], synthesis of passive and active analogue circuits [13], synthesis of operational amplifiers [14], and transistor size optimisation [15]. Table 1.1 summaries some relevant work on evolutionary Electronics. This is not a complete list of research work in the area, there are many other no less important work that have not been mentioned here. The study attends to identify the research application that started innovative trends in this area.

However, the evolutionary design of digital circuits is a process of evolving configurations of logic components for some prespecified computational program. Often the aim is for a highly efficient electronic circuit to emerge in a population of instances of the program. Digital electronic circuits have been evolved intrinsically [16] and extrinsically [17]. The former is associated with an evolutionary process in which each evolved electronic circuit is built and tested in hardware, while the latter refers to circuit evolution implemented entirely in software using computer simulations. The motivation behind the study of digital circuit evolution is to design electronic circuits that are more efficient than the conventional designs, and then, to learn new principles of design. It is well-accepted fact that to evolve circuits with increasing size is a difficult task even for evolution. However, the evolution of small circuits is feasible. The studies have revealed that often the evolved solutions are unusual in construction, and can be efficient in terms of number of gates used. Learning new principles of design is beneficial for the design of electronic circuits. Thus new methods and principles of evolving digital circuits are inferred.

Table 1.1 Some key publications in evolutionary electronics.

Date	Authors	Application
1991	Louis and Rawilis [18]	Evolution of basic digital function
1993	H. De Garis [19]	Introduction of the concept of Evolvable Hardware
1995	Thompson et al. [9]	Evolution of a hardware sensorimotor control structure
1995	Hemmi et al. [20]	used of hardware description language to evolve circuits
1996	Koza et al. [21]	Evolution of low-pass filter and bipolar transistor amplifies
1997	Miller et al. [6]	Evolution of novel arithmetic digital circuits
1998	Zebulum et al. [22]	Evolution of a digital circuit for CPU control
1999	Chongstitvatana et al. [23]	Learning finite state machine synthesis from partial input/output sequences.
2000	T. Kalganova [24]	EHW Design for combinational logic circuits.
2002	B. Ali et al. [1]	Design of sequential logic circuit

**ALL MISSING PAGES ARE BLANK**

**IN**

**ORIGINAL**



correct solution by concentrating on too small an area. That is, too much exploration means too much time, and too much exploitation means that the correct solution may not be found. Striking a good balance between exploration and exploitation is critical for a successful search algorithm.

Wasting time in unpromising areas of the search space may seem a small price to pay on the way to find the correct solution. Unfortunately, the size of the search space that arise in design domains is often so large that a human lifetime is short compared to the time needed by the fastest supercomputer to look at a significant fraction of the space. At the other extreme are algorithms that do no exploration and have knowledge available to solve the problem fast and directly. These algorithms make strong assumption about the search space and sufficient exploitable information is available to avoid searching. However, because these strong assumptions do not usually hold across-domain (search space) these algorithms work only on the particular space they were designed for and marginally on others.

In between these extremes of random and/or exhaustive search and no search, every other search algorithm makes assumptions of varying kinds about search space. These assumptions correspond to knowledge about the space and may be correct, incorrect, or misleading, implicit or explicit, and known or unknown, when used to search a particular space. This knowledge is exploited to guide exploration, speeding up search for generating a search is manifested in the set of generated solution.

Genetic algorithms belong to a class of algorithms, called blind search algorithms, which make the assumption that there is enough knowledge to compare two solutions and tell which is better [25]. There are different tasks when using EHW to evolve:

1. Because EHW are not subject to the limitations of human designers they can thus explore a much larger set of designs, maybe even the whole space of all designs (Figure 1.1).

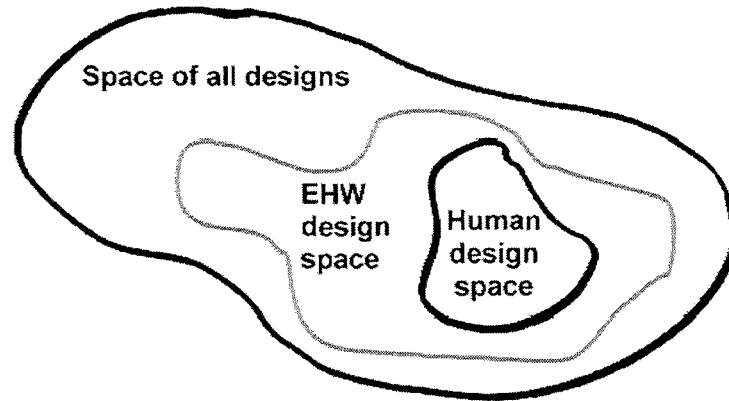


Figure 1.1. Space of all designs.

2. The EHW design space might grasp a larger part of the space of all designs than the human design space.
3. Evolutionary Electronics deals with a huge search space, requiring, therefore powerful search techniques to handle the task. Naturally, when the search space is very large, only random search has some chance to succeed. Hence, with this new approach to circuit designs a search technique procedures have to be followed:
  - The search space sampled by the algorithm must have its size limited.
  - It is usually necessary to adapt the search technique to the particularities of the design problem.

## 1.2 Scope of the Thesis

This section discusses the scope and limitations of the present body of work. This thesis focuses exclusively on sequential logic synthesis and outlines the use of the extrinsic EHW approach to evolve FSM. Both GA and EHW are combined together to produce optimal logic circuit. GA is used to optimise the state assignment problem. EHW is used to design the combinational parts of the desired circuit. The approach is tested on a number of FSMs. These circuits have been evolved using different functional sets of logic gates and

GA parameters. The sequential behaviour of FSMs is represented by a set of logic components.

We believe that an extension of each of the methods presented herein to extended sequential design is relatively straightforward. This is an area for future work. In addition, the algorithms in this thesis are all based on a single encoding model: the input-encoding model (described in detail in Chapter 4). This limits their effectiveness under certain cost metrics, because input encoding is only an approximation of optimal state encoding. As such, we expect that considerably better logic would result under these metrics from using a more precise model. In particular, a more potent model, namely output encoding could be substituted in the various steps, without disturbing the remainder of the synthesis path. As a result, input encoding is actually quite effective for this application.

Finally, algorithms presented here are computationally too expensive for the full range of sequential machines encountered in practice. Implicit methods, such as those introduced in this work, would extend its capacity enough to handle any practical design.

### **1.3 The position of this thesis within the field**

In preparing for this research program, it is intended to take a specific viewpoint: That small circuits evolved with as few constraints as possible may be of increased utility to engineering applications if they are properly understood. In addition to the factors governing a circuit's operation, understanding encompasses the influence an evolutionary design process has on its structure and properties, and whether altering the process for example by changing the types of basic element or evolutionary algorithm operators could improve the circuit in some way. It will be clear to the reader from the preceding pages that there are many ways of tackling this, and that some of the foundations have already been laid by the authors cited above. The first evolvable design suitable for general logic circuit, the first examples of extrinsic hardware evolution at the gate level, and the discovery and partial understanding of an entirely new form of evolved circuits. While the study



acknowledge the importance of large finite state machine benchmarks for the evolution of complex circuits, this work is intended to provide insights into the types of circuit at which evolution excels, and further foundations for the methodologies by which they can be produced. It is believed that the most appropriate starting point for achieving this is through the evolution and analysis of relatively small circuits. If computer time and memory limitations can be overcome, large design can be evolved.

#### **1.4 Outline of the thesis**

The thesis is about relationship between redesign structure and search. The relationship has been studied in the context of digital circuit evolution. In general the thesis can be divided into two parts. The first summarizes the theory of logic design processing and introduces an information analysis by which the structure of a circuit is characterized. The analysis is applied to a well-investigated model of design and thus the theoretical finding is supported empirically.

The use of the evolutionary algorithms to the problem of digital circuits design is discussed in the second part of the thesis.

Three goals are pursued in this part:

Firstly, demonstrate the advantages of the method.

Secondly, to show how the study of the structure of design that originates from a particular optimization problem could help to improve the evolutionary search. Furthermore, to identify principles that help to overcome the problem of the very fast growth in the number of gates used in the target circuit as the number of inputs to the evolved logic function is increased.

Thirdly, special attention is paid to the role in digital circuit evolution.

*Chapter 2* is a brief introduction to the problem of design of logic circuits. Further, it describes various forms of synchronous circuits, which are the focus of the bulk of the thesis. It begins with a discussion of conventional methods of digital design and closes with a description of logic synthesis tools needed for computer aided. It offers a general overview of sequential synthesis and traditional approaches. Provides a survey of digital circuit design and optimisation technique, which speed up the design cycle and enhance design quality. The chapter presents the basic definition necessary for understanding the work presented in this thesis. The chapter summarize the commercial tools available for design. Synthesis of finite state machines method for both two-level and multilevel logic implementation of the combination logic section of finite state machine is presented. Then, it presents one of the central themes of this thesis: the state assignment problem and previous approaches to this problem. The main contribution of the chapter is the algorithm for the state assignment problem that will be discussed further in chapter 4.

*Chapters 3* introduce natural selection, evolutionary algorithm theory and compares evolutionary algorithm and other search algorithms. This sets the stage for describing the innovative powers of a genetic algorithm and elaborating on encoding and evaluation of design for genetic algorithms. Further the chapter will discuss evolvable hardware. It will show how evolvable hardware has emerged from the combination of evolutionary algorithm and flexible electronic devices. It will also give an overview of the different subdivisions of EHW and their applications. After reading this chapter, the reader should have an understanding of the domain of evolvable hardware and the particular problems involved.

*Chapter 4* highlights the importance of considering the genetic algorithm and its parameters and how it is implemented to solve the state assignment problem for logic synthesis and optimization. The more general problem of logic circuit optimisation is explored. Synthesis based on a GA allows a

designer to minimise the actual area, power, or delay while performing state assignment. These GAs compensate for most of the unpredictability inherent in the logic reduction synthesis tools, providing a better measure of the circuit's characteristics. The genetic algorithm converges faster and finds better state assignment compared to previous reported methods. The genetic algorithm finds the best solutions for the encoding of all medium size MCNC benchmarks and tests result for these benchmarks are given. The synthesis based on the GA approach results in designs with the smallest area to date. Test results on standard benchmarks are given and compared with previously reported results.

*Chapter 5* Studies the evolutionary design of combinational and sequential logic circuits, particularly the sequential circuit. Thus digital circuits are evolved using evolutionary algorithms. The structure of the resulting circuits is investigated and it is shown that the principles of evolving digital circuits are valid. In this chapter an approach based on an evolutionary algorithm to design sequential logic circuits with minimum number of logic gates is proposed.

*Chapters 6* outlines the use of the extrinsic evolvable hardware approach to evolve finite state machines. Both the genetic algorithm and Evolvable Hardware are combined together to produce optimal logic circuits. A GA is used to optimise the state assignment problem. EHW is used to design the combinational parts of the desired circuit. The approach is tested on a number of finite state machines from MCNC benchmark sets. These circuits have been evolved using different functional sets of logic gates and GA parameters. The results show promise for the use of this approach as a design method for sequential logic circuits.

*Chapter 7* contains concluding remarks, which review the contribution of this work and include topics for further research.

### **1.5 Summary**

With the continuous increase in the complexity of electronic circuits, there is increased demand for effective methodologies for the design of such electronic circuits. These increases in complexity together with the increasing number of design objective (e.g. low power, high throughput, small area) provide the designer with a very hard task. For that reason there is a demand for effective ECAD tools, which perform some of the design tasks leaving the designer to concentrate on performance optimisation issues.

The complexity of the electronic design search space has encouraged using all computer-based techniques in the design procedure. One such technique, which is used in this thesis, is called Genetic Algorithms. These algorithms have shown a high degree of flexibility in dealing with problems with complex and computationally hard Problems, such as the electronic circuit design problem.

This thesis describes investigations carried out in order to develop a genetic based digital circuit design/synthesis ECAD tool. The investigations have led to the development of a novel custom genetic algorithm for the structural design of circuits. A number of circuit specific genetic operators have been produced and their use has been investigated together with conventional genetic operators.

## *Chapter 2*

### REVIEW OF THE LOGIC DESIGN PROCESS

#### **2.1 Introduction**

The impact of Very Large Scale Integration (VLSI) circuits in modern life can be seen in various electronic facilities. Over the years, growth in the complexity of VLSI design has enable designer to include well over a million transistors on each chip. Designers are faced with the daunting task of packing more functionality into a smaller area and creating a circuit that operates faster than the previous generation. Design Automation (DA) technique play an invaluable role in this complex process.

The first phase of this project looks at the state assignment problem of automated synthesis of synchronized sequential circuits. Automated synthesis for digital logic is rapidly increasing in importance in industrial design. This is due to the huge increase in possible circuit size over the past few years, making it extremely difficult to design efficiently by hand. Synthesis tools such as Synopsis free the designer to operate at a high level, performing the transition to lower level of design automatically, along with relevant optimisations. Optimisation of sequential circuits was first studied in detail in the late 1950s and 1960s [26], but has recently increased in importance.

This chapter starts by introducing the reader to the relevant subject area. The concept of logic synthesis can be defined as the process of transforming a verbal description of required functionality into a working circuit. Traditionally, it has been divided into combination and sequential synthesis. The research is focused on sequential synthesis and in particular the state assignment problem due to its impact on the complexity of the resulting circuit. The details of the algorithm implemented for the state assignment

problem using genetic algorithm are also given. This algorithm has been evaluated by comparing it with both standard industrial tools and other research published in this area.

## 2.2 Overview of Optimal Logic Synthesis

The benefits of automating the logic design process may be compromised if the result does not meet its area, speed, or power constraints, while optimising the design trade-offs. Effort is made to enable the design tools to make informed decisions as well as an expert designer could [27,28]. Therefore, a critical aspect of automatic logic synthesis is the optimization problem of deriving a high-quality design from the initial specification. The accepted optimization criteria for multi-level logic are to minimize some convex function of:

1. Area occupied by the logic gates and interconnects.
2. The Critical Path Delay of the longest path through the logic.
3. The Degree of Testability of the circuit, measured in terms of the percentage of faults covered by a specified set of test vectors, for an appropriate fault model (e.g. single stuck-at faults, multiple stuck-at faults, etc.).
4. Power consumed.

This minimization is to be performed while simultaneously satisfying upper or lower bound constraints placed on these physical quantities. While humans are superbly equipped to solve such problems once they are clearly formulated, the sheer mass of detail in VLSI designs makes the use of synthesis tools imperative.

Delay constrained area minimization has an immediate effect on practical microchip design. The design has to physically fit on the chip, which typically is of the order of 1cm square or less. We note also that area minimization has an economically important impact on yield, because net yield is known to decrease exponentially with the size of the chip. Alternatively, algorithms for area-constrained delay minimization are often used to maximize performance.

Another criterion, which is increasingly important especially for mobile technologies such as laptop, computers, and cellular phones, is to minimize the power consumption of the final circuit. The area, delay and power of a design before layout are estimated using models, which predict the effects of physical design, based on the cells and nets in the final design. Often power is minimized by just making the design slower, but transformations exist which reduce power while not increasing delay [132].

Another important part of integrated circuit design is the manufacturing test, which determines if a fabricated chip works as expected. A connection (wire) is untestable (or redundant) if replacing the connection with a constant value does not affect the functionality of the circuit.

If the connection is not testable, it is called redundant and when a redundant connection is removed, a smaller circuit is formed. There is the further problem that redundancies interfere with the production line testing of the integrated circuit. Therefore, another goal for logic synthesis is to produce designs with no redundancies.

Synthesis tools are also in an ideal position to tackle the testing problem. These tools create and alter designs, so that they can easily modify them for testability [28]. It makes sense for synthesis tools to insert the special structures needed for test, if any, and then optimise the whole design, including the test structures, for minimum speed and area penalty. Similarly, it is logical for synthesis tools to produce vectors that are inherently integrated with a design's test structures, and to minimize the vector set required to test the design. Thus the marriage of synthesis and test can achieve a fully automated test solution, and serves to move test forward in the design cycle.

The design of the optimal circuit that meets all of these constraints is a difficult problem due to the tremendous number of potential solutions for even a small set of logic equations. The size of VLSI circuits makes its logic synthesis a difficult optimization problem, which usually requires automated

tools for practical designs. The next sections outline the tools needed to design and efficiently utilize computer aided design tools for automatic logic synthesis.

### **2.3 Logic Synthesis with ECAD Tools**

Electronic Computer-Aided Design (ECAD) of microelectronic circuits is concerned with the development of computer programs for the automated design and manufacture of integrated electronic systems, with emphasis today on VLSI circuits [29]. Synthesis of VLSI circuits involves transformation of the specification of circuit behaviour into mask-level layout, which can be fabricated using VLSI manufacturing processes, usually a number of representations between abstract behaviour and mask-level layout. Optimisation strategies, both manual and automatic are vital in VLSI synthesis in order to meet the required specifications. However, the optimisation problems encountered in VLSI synthesis are typically NP hard. Therefore, solutions to the optimisation problem are incorporated in heuristic strategy, the development of which requires a thorough understanding of the problem at hand. Thus, automatic optimisation-based VLSI synthesis has evolved in to a rich and exciting area of research.

Direct application of synthesis in industry has been a significant drive force for research in ECAD. Simple marketing principles that other factors being equal, a product available sooner would capture a large share of market and would remain in use longer. The desire to reduce time to market has led to the initial investment of considerable money and effort into the development of ECAD tools capable of producing designs competitive with the best manual design. Today, constantly shrinking geometry and increasingly reliable manufacturing process have led to complex systems being implemented on a single chip making the use of ECAD tools commonplace and mandatory. In its turn, the rapid automation of the VLSI design phase has allowed companies to keep pace with advances in other areas of VLSI like computer architecture and manufacturing. As a consequence of this rapid development



in VLSI technology, it is currently possible to produce an ASIC, microprocessors and other type of circuits that contain millions of transistors. ASIC designer and recent advances in systems on chip (SOC) make the use of ECAD tools commonplace and mandatory.

## **2.4 VLSI Design Flow**

Design is the process of translating an idea into a product that can be manufactured. In the design of electronics this is simply a product formed from electronic equipments, software, and electromechinics. Effective design allows this translation to be done quickly cheaply and accuracy to produce a product that is commercially viable, competitive, fit for purposes, without problem and adaptable.

The design process, at various levels, is usually evolutionary in nature. It starts with a given set of requirements. Initial design is developed and tested against the requirements. When the requirements are not met, the design has to be improved. If such improvement is either not possible or too costly, then the revision of requirements and its impact analysis must be considered. The Y-chart shown in Figure 1.2 illustrates a design flow for most logic chips, using design activities on three different axes (domains), which resemble the letter Y [30]. The Y-chart consists of three major domains, namely:

- Behavioural domain,
- Structural domain,
- Geometrical layout domains.

The design flow starts from the algorithm that describes the behaviours of the target chip. The corresponding architecture of the processor is first defined. It is mapped onto the chip surface by floor planning. The next design evolution in the behavioural domain defines finite state machines, which are structurally implemented with functional modules such as registers and arithmetic logic units (ALUs).

These modules are then geometrically placed onto the chip surface using ECAD tools for automatic module placement followed by routing, with a goal of minimising the interconnection area and signal delays.

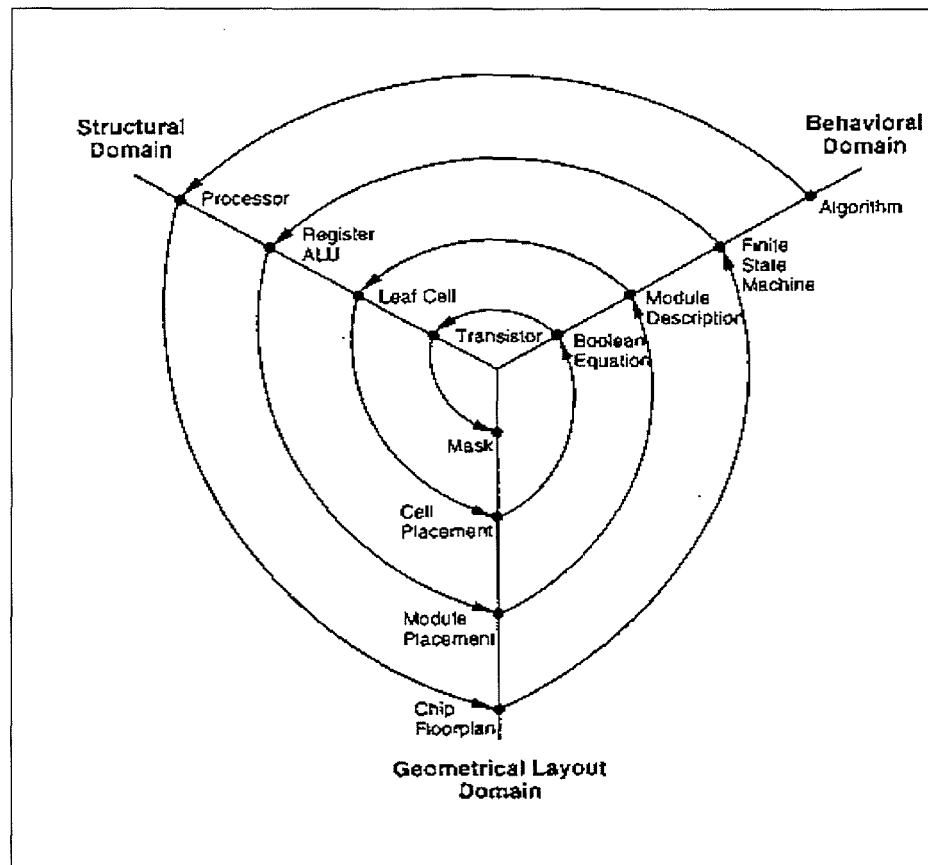


Figure 2.1. Typical VLSI design flow in three domains (Y-chart representation).

The third evolution starts with a behavioural module description. Individual modules are then implemented with leaf cells. At this stage the chip is described in terms of logic gates (leaf cells), which can be placed and interconnected by using a cell placement & routing program. The last evolution involves a detailed Boolean description of leaf cells followed by a transistor level implementation of leaf cells and mask generation. In standard-cell based design, leaf cells are already pre-designed and stored in a library for logic design use.

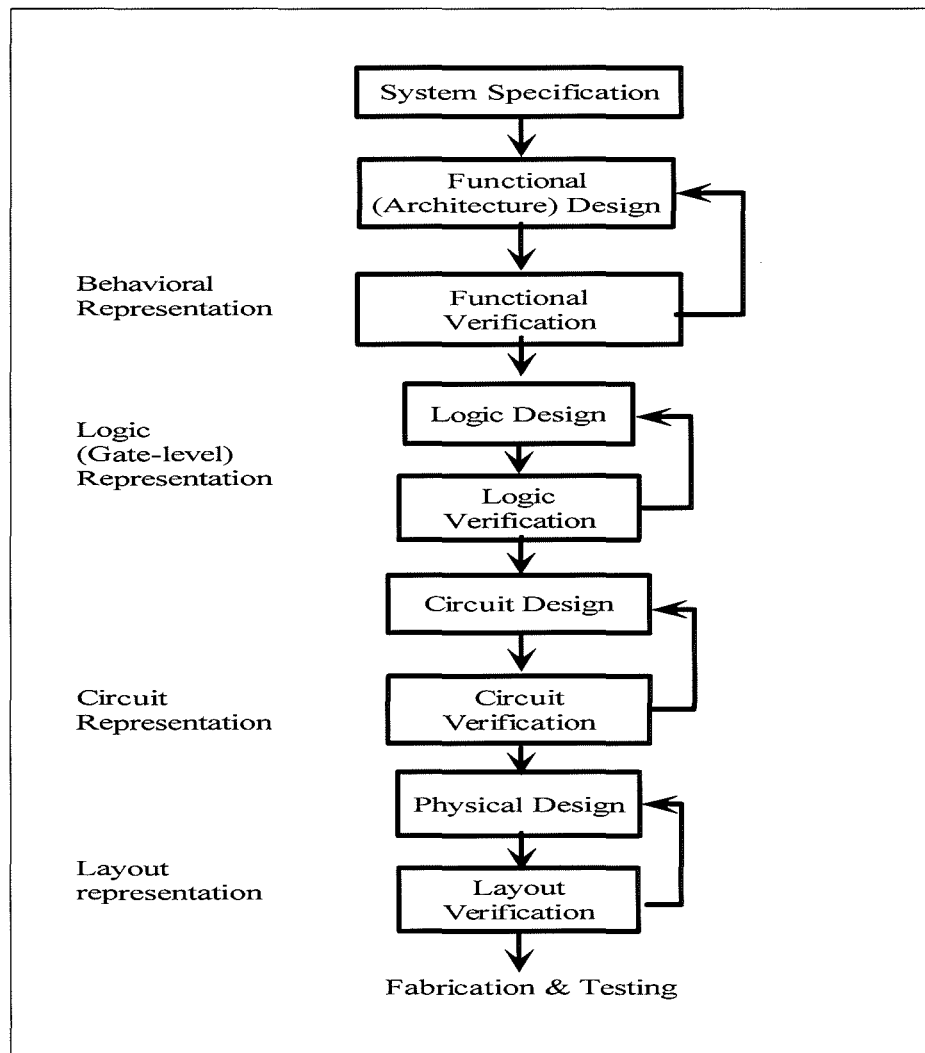


Figure 2.2. A more simplified view of VLSI design flow.

Figure 2.2 provides a more simplified view of the VLSI design flow; taking into accounts the various representations, or abstractions of design behavior, logic, circuit and mask layout. Note that the verification of design plays a very important role in every step during this process. The failure to properly verify a design in its early phases typically causes significant and expensive re-design at a later stage, which ultimately increases the time-to-market. Although the design process has been described in linear fashion for simplicity, in reality

there is much iteration back and forth, especially between any two neighbouring steps, and occasionally even remotely separated pairs. Although top-down design flow provides an excellent design process control, in reality, there is no truly unidirectional top-down design flow. Both top-down and bottom-up approaches have to be combined. For instance, if a chip designer defined architecture without close estimation of the corresponding chip area, then it is very likely that the resulting chip layout exceeds the area limit of the available technology. In such a case, in order to fit the architecture into the allowable chip area, some functions may have to be removed and the design process must be repeated. Such changes may require significant modification of the original requirements. Thus, it is very important to feed forward low-level information to higher levels (bottom up) as early as possible.

The most important message here is that the logic complexity per chip has been (and still is) increasing exponentially. The monolithic integration of a large number of functions on a single chip usually provides:

- Less area/volume and therefore, compactness;
- Less power consumption;
- Less testing requirements at system level;
- Higher reliability, mainly due to improved on-chip interconnects;
- Higher speed, due to significantly reduced interconnection length;
- Significant cost savings.

It is now possible to design complex digital circuits systematically on a computer, simulate the entire design down to component level and verify that everything works before even considering making any hardware. Design of a complex system can be approached at different levels. Figure 2.3 shows the division between them with some relevant explanations.

Design level	Design Description	Primitive Components	Theoretical Techniques
Algorithmic	Specification High-level Lang. Math. Equations	Functional Block Black boxes	Signal processing theory Control theory Sorting algorithm
Functional	VHDL, Verilog FSM language C/Pascal	Register Counter ALU	Automata theory Timing analysis
Logic	Boolean equations Truth Tables Timing diagrams	Logic gates Flop-flops	Boolean algebra K-map Boolean minimization
Circuit	Circuit equations Transistor netlist	Transistor Passive comp.	Linear/non-linear eq. Fourier analysis

Figure 2.3. Level of design

## 2.5 Probing the limits of Logic Synthesis

Logic synthesis has freed designers from the complexities of gate-level design by converting RTL descriptions to optimised gate-level logic [29]. But ASIC, FPGA, and CPLD designers are still constrained by a dependence on silicon. Designers will need to pay more-not less-attention to layout as silicon densities continue to increase.

A few years ago, logic synthesis seemed a first step into a world of higher-level design. Back then, designers imagined being able to move higher and higher up the synthesis chain, until they could specify a design behaviourally and just push a button-and the software would do the rest. Nice dream, but not a reality. Why? Because design, even with high-level HDLs and simulation, must eventually meet silicon “reality.” And that reality, especially at sub micron or deep-sub micron levels (below 0.5  $\mu\text{m}$  L-effective) is not a nice, well-behaved world. Design rules will migrate down to 0.18  $\mu\text{m}$ , with chip voltages moving down to less than 1V as well. But that’s not the only reason for a re-evaluation of logic synthesis. Silicon’s higher densities bring

new system-level problems. Larger ASICs can be likened to systems, and as in to systems, must be partitioned for design ease and clocking. And last, hardware design is still hardware design; writing code in VHDL or Verilog, even code that simulates well, does not guarantee working silicon.

Today's engineers use logic synthesis primarily for control logic, optimising and mapping combinatorial logic (equations) into a netlist-and ready for layout [31]. They also use synthesis to instantiate major RTL components such as registers. Some tools, such as Synopsys Design Compiler, Cadence's Synergy, Exemplar's Core, allow designers to use module generators and megacell/cell libraries to select the correct element. Megacells can be hefty, including  $\mu$ Ps, FPU's, ALUs, and DSPs. In effect, the synthesis tools provide a single interface to specify a design. Some synthesis tools, such as Synopsys' Design Compiler, Cadence's Synergy, and the forthcoming Viewlogic View-Synthesis (was SilcSyn) provide some higher-level synthesis capabilities. These capabilities include resource allocation and sharing for key RTL blocks, such as adders or registers.

Mainstream logic synthesis tools from Synopsys, Mentor Graphics, Exemplar, Cadence, and Viewlogic also provide state-machine generators and mappings to optimized state machines. Many engineers find these tools work for general state machines, but, typically, they turn to hand design for highly optimized state machines.

Industry consensus seems to say it's still a bit early for efficient state-machine synthesis. However, engineers can define complex controls by defining multilevel state machines (state machines within state machines, etc); these can be defined with current synthesis tools. Most synthesis users describe designs with a HDL, such as Verilog or VHDL. However, when using a HDL, it's easy to lose touch with the design; you can define major RTL blocks with simple statements. Thus, a few lines of code can trigger major effects on a design's timing or performance. Good logic designers like master programmers, have to keep foremost in their minds the major flows of their

designs, continually monitoring any changes that add, delete, or modify RTL blocks.

Finally, writing Verilog or VHDL code does not automatically stop you from violating propagation delays or logic constraints such as set up or hold. Moreover, many constraints are functions of the ASIC process (voltage and temperature) as well as of the signal characteristics (slow or fast edges). Consequently, you cannot realistically estimate these timing delays until floor planning or place and route. You'll have fewer problems downstream with synthesis if you keep these logic realities in mind when coding. Static timing analysers can catch timing errors, but it's far easier to design it right the first time.

## **2.6 Sequential logic synthesis**

### **2.6.1 Sequential circuits**

Almost all VLSI circuits are sequential circuits; i.e. they contain memory or storage element in the form of flip-flops or latches as well as combinational (or switching) circuitry. Sequential circuits can be either synchronous or asynchronous sequential logic. While asynchronous design has certain advantage, such as speed, synchronous design is more widely used and is the subject of this research. Considerable progress has been made in the understanding of combinational logic optimization in the recent past and consequently a large number of university and industrial ECAD programs are now available for the optimal synthesis of combinational circuits [30, 31]. These optimization programs produce results competitive with manually designed logic circuits.

The understanding of sequential circuit optimization, on the other hand, is considerably less mature. The presence of internal states adds considerably to the complexity of the optimization problem. While the primary inputs and outputs are typically binary, internal states are represented in symbolic form.

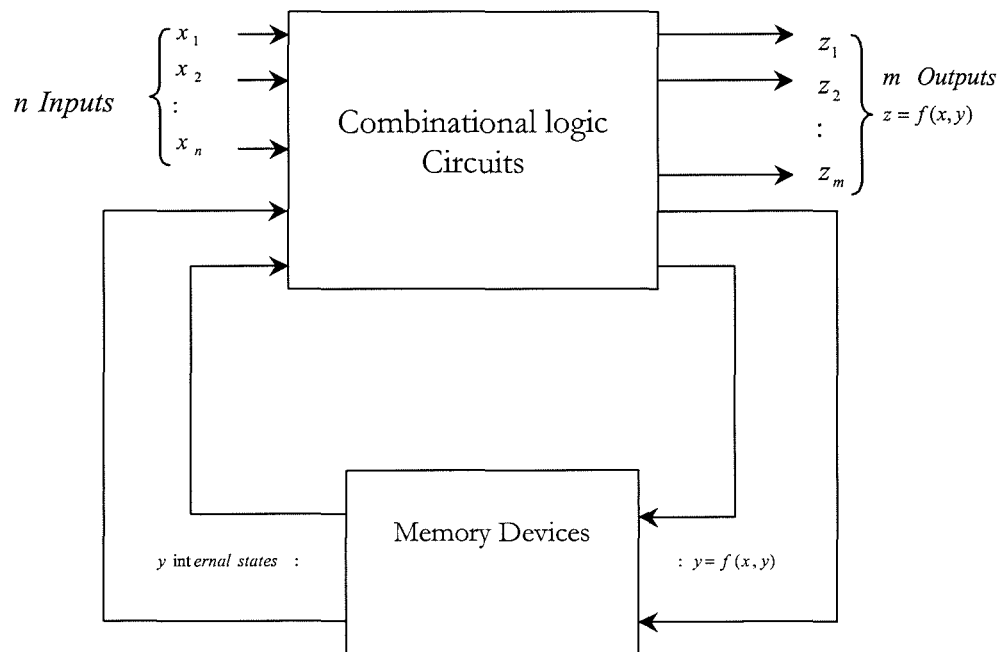


Figure 2.4. Structure view of a finite state machine (FSM).

Sequential, circuits are most often modelled using finite state machines. A FSM is a mathematical model of a system (in our case, a switching circuit). The states of a system completely summarize the information concerning the past input to the system that is needed to determine its behaviour on subsequent inputs.

More formally, a FSM,  $M$ , is a quintuple  $M = (Y, X, Z, \alpha, \beta)$

$Y$  is the set of all possible secondary states stored within the machine memory.

$X$  is the set of all possible primary inputs to the machine from external sources.

$Z$  is the set of all possible output states from the machine.



$\alpha$  is the next states Boolean function define as  $\alpha: (Y_t x X_t) \rightarrow Y_{t+1}$

$\beta$  is the output Boolean function, which determines the current state of the output Z. It has two possible forms:

In Moore Machine [34], the output logic is a function of the secondary state only:

$$\beta : (Y)_t \rightarrow Z_t, \text{ Secondary state change only at clock times.}$$

In a Mealy machine [35], the output logic is a function of machine total state defined as:

$$\beta \rightarrow (Y_t x X_t) \rightarrow Z_t, \text{ Since primary input may change at any time.}$$

It is convenient to visualize a FSM as a direct graph with nodes representing the states and the edge representing the transitions between states. Such a graph is known as a State Transition Graph (STG) Figure 2.5. An edge in the STG is labelled by the input causing the transition and the output asserted on the transition. In the State Transition Table (SST), Table 2.1, each row of the table corresponds to single edge in the STG. Conventionally, the left most column in the table corresponds to the primary inputs and the right most columns to the primary outputs. The column following the primary inputs is the present state column and the column following that is the next state column. A STT is a tubular representation of the FSM.

As well as state/output table, the FSM can be represented by Algorithmic State Machine (ASM) chart [32, 33]. The ASM chart, however, is not used in this thesis as it tends to hide the state assignment problem, which is under investigation.

To deal with complexity, VLSI circuits are invariably specified in a hierarchical fashion. Large sequential circuits are typical modelled by smaller, interacting FSMs.

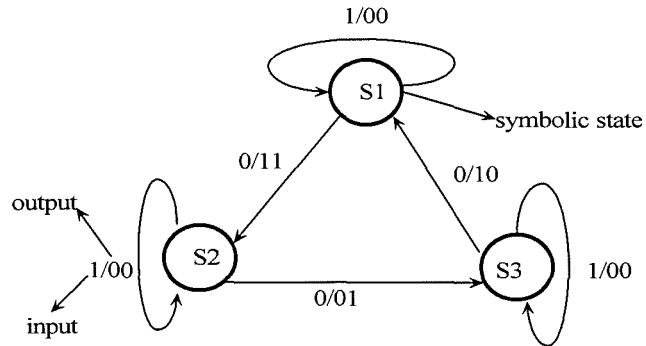


Figure 2.5. An example of a State Transition Graph (STG).

Table 2.1. An example of a State Transition Table (STT).

InputPr	esent States	Next States	Output
0	S1	S2	11
0	S2	S3	01
0	S3	S1	10
1	S1	S1	00
1	S2	S2	00
1	S3	S3	00

Synthesis tools are required to encode the internal state of FSMs as binary strings. This encoding determines the complexity and the structure of the sequential circuit, which realize the FSM, and therefore has profound effect on its area, testability and performance.

Stated differently, synthesis tools have the freedom of encoding states in such way that the design constraints are satisfied. The notation of structure is generally associated with the manner in which a machine can be realized from interconnected smaller component machines. It may be desired, for example to construct the circuit with the minimum amount of logic or to build it from an interconnection of smaller circuit to obtain superior performance.

A second degree of freedom available to sequential synthesis tools is based on the fact that the STG corresponding to a given functionality is not unique. Transformations to the STG like state splitting, state merging or STG partitioning, enable movement from one STG to another without changing the functionality. These transformations guide the encoding of states in a particular direction, in many cases into directions that would not have been taken otherwise. Such transformations are sometimes necessary to achieve the desired objectives.

### **2.6.2 Early work on sequential logic synthesis**

Work on sequential logic synthesis dates back to the late '40 discrete off the shelf components (relays and vacuum tubes) were used. In the 1960's, Small-Scale Integrated circuits (SSI) became popular and much of the work in that period was motivated by the need to reduce the number of latches in the circuit since that meant a reduction in the number of relatively expensive chips on the circuit board. Also, since combinational logic synthesis was still in its infancy, techniques for state encoding and FSM decomposition were unable to target the combinational logic complexity of the sequential circuit effectively [137].

The minimization of the number of states in completely specified FSMs was first investigated by Moore [34], Huffman [35] and Mealy [36]. Ginsburg [36] and Unger [37] extended this work late on to the reduction of states in incompletely specified machines. An interesting technique for deriving maximal capability in state reduction using Boolean algebra was reported in a short communication by Marcus [38].

The relationship between state encoding, and the structure of the resulting sequential circuit was first investigated in terms of the algebraic theory of partitions by Hartmanis [26] and later by Hartmanis and Stearns [39]. Karp [40], Kohavi [27], Krohn and Rhodes [41], Yoeli [42] and Zeiger [43] also made contributions to machine structure theory. Hartmanis and Stearns [39] developed the concept of state splitting to augment the possibilities of finding desirable decompositions and state assignments, among others, such as Zeiger

[43]. The book by Hennie [44] provides a bright and intuitive description of the above contributions.

State encoding was treated from a different point of view by Armstrong [45] and Dolton and McCluskey [46]. The procedure of Armstrong formulated the state-encoding problem as one of assigning codes so that pre-derived adjacency relationships between states are satisfied in the Boolean domain. To a certain extent, the procedure of [45] inspired some state assignment algorithms developed in recent work (e.g. Devadas et al [47] for targeting multilevel implementation. The discussion of more recent developments is given in chapter 4.

### 2.6.3 Classic Synthesis Trajectory

This section described the sequential synthesis problem, and the traditional approach to this problem as a sequence of decomposition steps. A brief recount of the history of sequential synthesis and the current state of the art for FSMs is also given.

Sequential synthesis is the process of creating a suitable implementation for a given FSM. It is an extremely difficult problem, for several reasons. First, the number of possible implementations for even a modest-sized specification is staggering. Second, although many tools approximate the cost criterion as a simple metric, in reality it is often a complex, multi-dimensional function. Moreover, trade-offs involving several characteristics of the solution (e.g. area, power) are typical. Furthermore, addressing even simplified cost metric often require algorithms of high computational complexity. In fact, many of the problems described below are NP-complete.

As a result, the synthesis problem is typically broken into a sequence of decoupled subproblems, as shown in Figure 2.6. We first sketch the overall steps, and then give some more detailed background on certain key steps.

The process starts with a specification of the FSMs behaviour in some form, such as a flow table. The machine's internal states are normally identified by symbols, so that the specification focuses as much as possible on the intended behaviour and not some particular implementation.

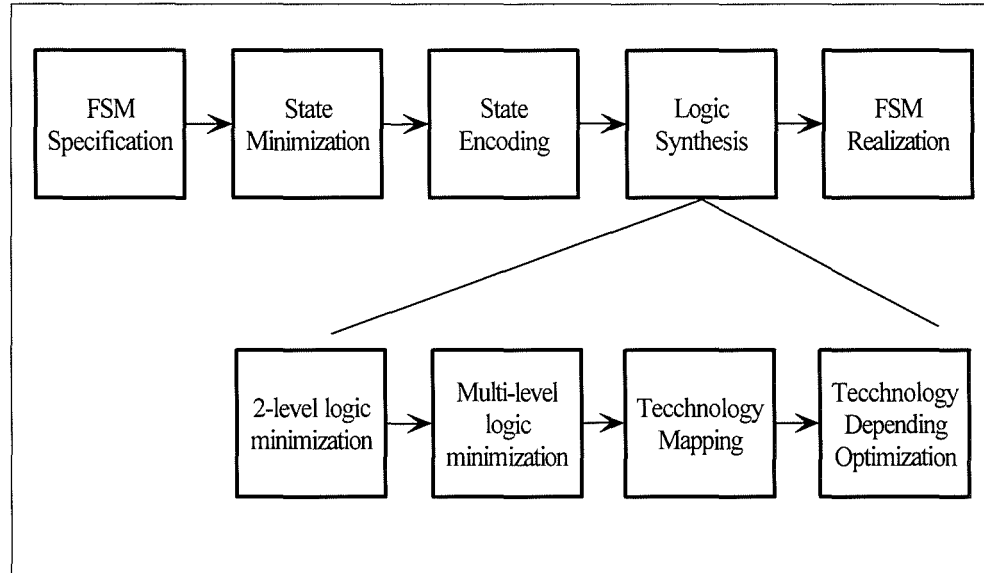


Figure 2.6. Traditional sequential synthesis trajectories.

#### 2.6.3.1 State Minimization

The first step in the flow, state minimization, is an optimization that makes use of a classic observation [39] that don't cares in an incompletely specified FSM specification often permit a realization with fewer states. In fact, even when the FSM is completely specified, states having indistinguishable behaviour can be merged, producing smaller equivalent machines. Often, the FSMs resulting from state minimization are fundamentally simpler than their unminimized counterparts, and can thus be implemented with better logic, regardless of the cost metric involved. Through state reduction is generally desirable, it was found that in some cases it can reduce the chance of decomposition based on closed partitions [39]. Many researchers have studied state minimization over several decades [48]. The bulk of this research has striven to find efficient algorithms for solving the classic problem as stated by Luccio and Grasselli [49], both exactly and heuristically. A small minority of recent work has attempted to solve the more difficult problem of optimal state minimization, which targets logic complexity [48], rather than the fewest number of states.

### *2.6.3.2 State assignment of finite state machines*

A State Assignment Problem is to assign codes to the internal states of a FSM in order to minimize some cost function related to the circuit's realization (like PLA area or number of gates). Most of the research papers discuss only the problem of assigning internal states, assuming that the user specifies codes for inputs and outputs. However, there are efforts based on very similar principles, which assign input states and outputs (specified initially by names) with binary codes. They have found, for instance, application in minimization of microprogrammed control units [50] or in designing control units with PLAs [51].

The problem is a classic in Switching Circuits Theory, but relatively few programs have been widely available until recent years. In the early sixties the basic main approaches to state assignment and structural theory of FSMs were formulated. Few of them were programmed and the programming results were rather unsatisfactory. Recently the state assignment problem is again gaining momentum because of widespread attempts to realize a logic level silicon compiler for VLSI [1,52,53,133].

#### **State assignment approaches**

Obviously, a realization in binary logic is required; hence, state encoding assign a suitable assignment of binary codes to the symbolic states. The encoding so derived effectively transforms the symbolic FSMs specification into a set of pure binary valued Boolean (combinational) functions.

Nominally, any encoding which maintains the distinction among the various states is valid, however certain codes permit more economical realizations than other. This realization spurred significant research into the problem of optimal state encoding. The resulting algorithms offer wide variety of heuristic and exact approaches. There are basically eight approaches to state assignment:

1. Partition theory of Hartmanis, Stearns, Kohavi and Almaini [26, 27, 54]. The theory is based on a concept of a partition of the state

machine into blocks. Partitions are found from state tables and used to find sets of partitions producing unique and optimal encoding. This theory is very elegant from mathematical point of view, gives an insight into the nature of structural properties of machines, and it permits the decomposition of FSMs. This approach is intractable for computer solutions of FSMs with more than 10 states, 10 inputs, and 10 outputs.

2. Column evaluation approach of Dolotta and McCluskey [46] extended by Curtis [55] Weiner [56] Vavilov [57] Torng [58]. The columns of the state table are scored with respect to many various criteria influencing quality of assignment. The scores are used to find good partitions and next the state assignment. This approach can produce very good realizations (also for various types of flip-flops); the results are even better than those from approach 1, but are also intractable for machines of more than 12 states.
3. Enumerative approach of Story [59]. All possible partitions are evaluated as candidates for assignment separately by calculating the complexities of realizations of the corresponding Boolean functions, and assuming that all other partitions have been optimally selected for them. The subset of partitions with the best scores, also mutually matching, is selected for assignment. This approach gives better results than the approach from two, but it is even slower.
4. Branch and bound approach of Perkowski, Lee and Zasowska [60, 61]. This approach has two variants. The first permits realization of machines with up to 8 inputs, 8 internal and 8 outputs states and gives optimum realizations for arbitrary technology and flip-flops. This is perhaps the program that generates most optimum solutions of all the published programs, but is very slow. The approximate method based on this method [62] does not evaluate all partitions, but only

heuristically selected best partitions. It permits for realization of machines with 12 states, but can be extended to 18-20 states. Both approaches permit for state assignment combined with state minimization.

5. Quadratic assignment approaches of Armstrong, DeMicheli and Perkowski [45,50,62]. All these approaches are based on embedding some graphs created from FSM table to hypercube graphs. This approach permits realization of machines to 100 states but according to evaluations from [62] it gave solutions too far from optimum. Some theoretical improvements were made in [62] but not programmed. DeMicheli implemented two algorithms for state assignment at UC Berkeley. The first of them was based on the quadratic assignment method in which state assignment is reduced to the graph-embedding problem. The second algorithm was based on other principles since he was not satisfied with the quality of results. The results from [53], where both creation of the graph for embedding and the embedding algorithm were done on new principles, seem to be very satisfactory (machines with 136 states have been assigned), but no detailed comparison with other approaches is yet available.
6. Approach of Moroz [64]. This is a very fast constructive algorithm that is widely used in design automation systems in the Soviet Union. The author has implemented this algorithm and observed that it can find assignments for machines with more than 100 states.
7. Approach of DeMicheli, Brayton and Sangiovanni-Vincentelli [65] (KISS program). This approach is based on minimization of multiple-valued Boolean functions to find state groups and then constructive assignment by embedding of these groups to the faces of a hypercube. Its strategy is very innovative and the computer results are



very good. It is perhaps the best product currently available from the point of quality/time ratio. It was applied to machines with up to 100 states, but the largest published result is for 27 states. The groups of states are found with the use of multi-valued Boolean minimization program Espresso-MV [66] applied to FSM before state assignment. This method of finding groups of states (blocks) combined with fast Boolean minimization was a source of programs success. The Kiss program is now widely available in universities and is used by many companies to build commercial software. A program extending this approach was built in Intel [67].

8. In recent year, considerable research has been on the application of genetic algorithm to the state assignment problem, particularly Almaini [1,68] and Amaral et al [69]. This approach is based on GA to find optimal state assignment for sequential machine. In this research, the GA approach proposed along with a set of operators needed for its implementation, as will discussed in more detail in chapter 4.

#### **2.6.3.3 A Problem with existing state assignment approaches**

We would like to have a program as fast as that of approach 6, as good as in 4 and as useful in VLSI design as approach 7. However, this is impossible. The user then has to implement a method selected among the above for his class of machines and system's speed and performance requirements, or implement many algorithms and select among them interactively for any particular problem [70]. In this section a summary of the advantages and disadvantages of state assignment approach given in the previous section are considers.

The method of DeMicheli fails when there are few or no groups of internal states those translate to the same state under some input state. Unfortunately, such types of machines often occur in industrial applications [70]. The new program developed by DeMicheli at IBM gives results of about 20-constraint assignment problem [53].

The idea of applying Boolean minimization to the non-assigned machine is not new. It was used in the systems of Story, Harrison and Reinhard [59] and of Perkowski and Zasowska [61], but because of the lack of fast Boolean minimizers, like Espresso-MV, this approach was applicable only to machines smaller than 9 internal states. However, the method also a secondary assignment minimizes the numbers of gates and can be used for multi-level logic realization. The enumerative approach of the method allows finding an absolutely minimum solution to the problem for various technologies, since the cost function for realization is user defined. Approximate variant for larger machines was also created. Availability of fast minimizers permits the improvement of method [71].

The approach 8 is a new algorithmic method developed as part of this research project and will be presented in chapter 4.

The disadvantages of the original approach of DeMicheli et al [65] are the following:

They do not take into account the outputs, various structures and modern methods of solving the quadratic assignment problem and also do not minimize secondary assignment. Attempts to improve Kiss were successful [70]. However, the program still cannot be applied to machines with more than 100-state.

The disadvantages of the approach of Armstrong are the following:

The method of reduction to quadratic assignment was doubtful. Armstrong didn't take into account the possibilities of fast logic minimizers (they didn't exist at that time), modern approximate approaches to quadratic assignment (they didn't exist either), and the assignment of the outputs.

The disadvantages of Moroz approach are the following:

The method of creating the assignment graph can be essentially improved. The method of solving the quadratic assignment problem can be used instead of his embedding, which should produce results of better quality, and he didn't take into account assignment of outputs.

Different papers applying the embedding methods use different approaches to create the assignment graph. Saucier [72] creates a nonoriented weighted graph, whose edges correspond to transitions between states of FSM.

Moroz [64] creates an oriented graph, whose edges correspond to oriented transitions between states. He writes about embedding, but his work can be treated as an approximate solution to quadratic assignment of a particular type, where the graph is oriented, costs of the edges are equal, and the cost function is defined as in the quadratic assignment.

Armstrong [45] formulates a nonoriented graph, whose edges are created according to several principles of adjacency.

The above authors use different constructive algorithms for embedding those graphs on hypercube. They do not give any, other than heuristic, explanations of adequacy of the proposed assignment (embedding) techniques. Also, the program of Saucier is designed for asynchronous machines.

Perkowski uses a combination of factors that take into account adjacency of states, inputs, and outputs, both from the view of primary adjacency (like De Micheli), and also secondary adjacency with use of methods similar to partition theory and tabular methods.

Although many interesting approaches to state assignment have been recently proposed [1], careful comparison of algorithms is necessary on large benchmarks of industrial machines. Many new interesting algorithmic concepts arise from the papers and evaluation results recently available. Perhaps in future systems, various algorithms will be used for small (less than 8 states), medium (9-40), and large (40-120) and very large (more than 120) machines to obtain good speed/performance ratios. Detailed analysis of reduction methods is necessary, as well as evaluation of the complexity of optimal and approximate algorithms for related mathematical problems, like hypercube embedding, embedding to faces and quadratic assignment.

#### **2.6.4 Concurrent state minimization and state assignment to improve area.**

One of the methods to improve a chips area is discussed in this research. Current approaches to structural synthesis of finite state machines are non-minimal. As described above, the currently used design approach is first to minimize the number of machine's internal states and follow it with the states assignment. The aim of these methods is generally to decrease the semiconductor area of the realization for some selected implementing technology. However, the methods apply very abstract cost approximations to achieve this. It would be more desirable to minimize some generalized technology dependent cost functions.

Minimization of the number of internal states results from the adopted assumption: "the more internal states, the more complicated is the realization, hence more memory elements are needed, there are more excitation functions, and therefore their realization is more complicated". Practical examples bear evidence that the flow table with the minimum number of internal states is not necessarily the appropriate starting point to achieve the circuit realization of the minimum total complexity (computed for instance as the weighted sum of the number of flip-flops and the number of combinational gates).

Practical examples show evidence that we should not seek a FSMs with the minimum number of internal states, or one with the excitation functions depending on the minimum number of variables. Examples of FSMs can be easily found that have minimum realizations with the greater than minimum number of flip-flops (because of much simpler excitation functions). Also, the attempt to find a realization of the set of excitation functions with the minimum number of argument variables is often useless, because such realizations can have more gates or connections than other realizations of these functions. Moreover, these assumptions do not take into account the realization of excitation functions for flip-flops other than D flip-flops. One of the possible approaches is to replace two stages.

Minimization of the number of the machines internal states and state assignment of the machines internal states are joined. In this joint process, the cost function, related to the realization of the excitation functions in a selected technology, is optimized. The optimum and approximate methods of this type are presented in [62].

## 2.7 CAD Algorithms and tools

Some of the commercially available synthesis systems are described in Table 2.2. In order to experiment with synthesis it is necessary to work with a system in which source code is readily available. Thankfully, due to the work of many people, the SIS synthesis system is available from the University of California Berkeley [73]. SIS is built around the earlier work of Rudell [66] the development of the MIS system for combinatorial logic synthesis [74]. SIS incorporates many different algorithms that have been developed for sequential logic synthesis, such as state assignment, state minimization, retiming and technology mapping.

Table 2.2. Some Commercial design Synthesis tools.

Company	Product name	Synthesis level	Chip type
Cadence Design Systems	Synergy (VHDL, Verilog)	RTL, test state machine	ASIC, PLD
Mentor Graphics	AutoLogic (VHDL)	Datapath, RTL test, state-machine	ASIC, FPGA
Synopsys	Design Compiler (VHDL, Verlog)	RTL, test state-machine	ASIC, FPGA
Altera (AHDL, VHDL)	Max-Plus-4	RTL, state-machine	ASIC, FPGA

## 2.8 Other approaches to FSM design and optimization

Because the traditional approach to FSM design either involves solving at least two difficult combinational problems: state minimization and state assignment, or produces possibly poor solutions, many attempts to implement other FSM design methodologies have been undertaken. They attempt to minimize total area/chip count and include:

- design of general-purpose FSMs based on shift-registers or other elementary machines instead of flip-flops [75],
- design with special flip-flops with multiplexed inputs,
- decomposition of FSMs into structures of FSMs [76,77],
- converting the form of machine (Mealy to Moore and Moore to Mealy) to select one of better performance,
- concurrent state minimization and state assignment [78, 79],
- special structures of FSMs, like different type of micro-programmed control units with many ROMs and multiplexers, or partitioned realizations of logic [80, 81].
- direct conversion of high level description like parallel FSMs to symbolic or geometric layout [82].

The above approaches are used selectively for various categories of machines: some of them can be used for all machines, some of them are reasonable for large machines only, some other can be applied only to small machines.

## 2.9 Summary

The chapter presents a review of logic synthesis and design process and includes the necessary basic definition for an understanding of the work presented in this dissertation. After much research into the above synthesis problems, it has become apparent that the linear flow of isolated steps, while simpler to engineer, produces globally suboptimal solutions, even when optimum solutions are found at each step. Optimal state encoding represents a partial (and early) attempt to remedy this situation.

A critical insight by De Micheli [50] states that optimal state encoding requires symbolic logic minimization. In other words, a better encoding method results when encoding and logic minimization steps are combined. The single most important development in recent times has been the availability of high quality tools and methods to support top-down design.

## EVOLUTIONARY ALGORITHMS: A TOOL FOR DIGITAL DESIGN

### **3.1 Introduction**

This chapter introduces evolutionary algorithm theory and show the difference between evolutionary algorithm and the classic design model. Evolutionary Algorithms have appeared as a general concept for developing new computational models for optimisation and design. The principles of evolution in nature have been modelled in a variety of different ways and thus a number of computational models have been developed. These are referred to as Evolutionary Algorithms. Evolutionary algorithms have a common conceptual basis associated with the simulation of two fundamental processes. These are the processes of random variation and selection within a population, described by Charles Darwin (1859) as the principles of evolution. The interplay of variation and selection gradually pulls the population to a target that in evolutionary computation is merely the solution of a computational problem. Thus evolutionary search is employed to solve difficult problems in optimization and design. In many problems, evolutionary algorithms have been found to produce solutions that are better than those produced by the traditional design and other search techniques [1,6]. Solutions obtained by evolution are often unusual in construction, since they are generated in a completely different manner from the conventional methods for optimization and design [13]. This concept of evolutionary design of efficient and novel solutions has also been adopted in electronic circuit design (Figure 3.1). In this case a process of natural selection evolves the design. The design starts as a set of instruction encoded in the DNA whose encoding region are first transcribed into RNA in the cell nucleus and then later translated into proteins in the cell cytoplasm [83]. The DNA carries



the instruction for building molecules using sequence of amino acids. Eventually after a number of extraordinarily complex and subtle biochemical reactions an entire living organism is created. The survivability of the organism can be seen as a process of assembling a larger system from a number of component parts and then testing the organism in the environment in which it finds it self.

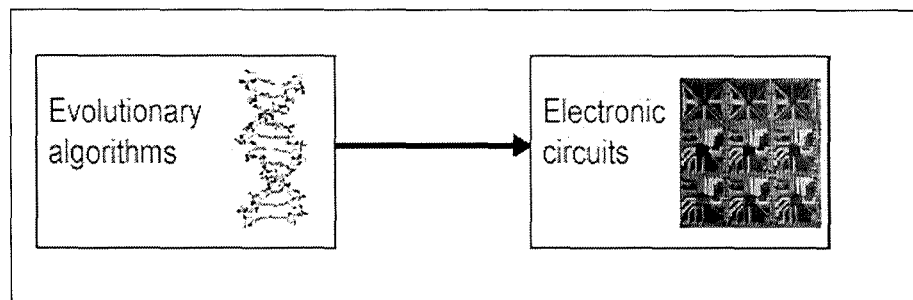


Figure 3.1. Use of the evolutionary algorithms (EA) to create electronic circuits.

### 3.2 Outline of Evolutionary Algorithms

Generally, all evolutionary algorithm methods follow the same procedures as shown in Figure 3.2.

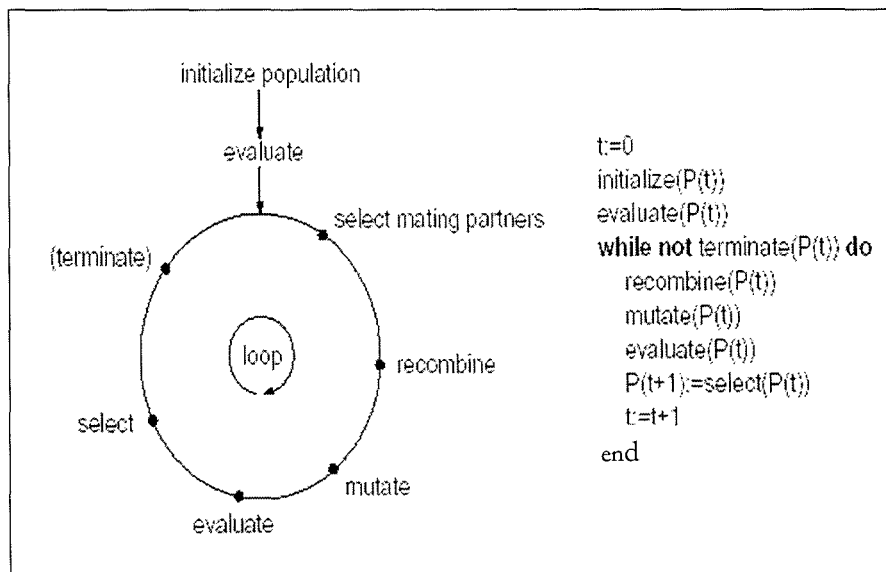


Figure 3.2. General outlines of any evolutionary algorithms method.

Seeding the population with random values normally carries out initialisation of the population. The fitness of each individual in the population is then evaluated. This fitness value is proportional to the value of the function being optimised. After that, selection is carried out to form a new population. Individuals in the population are normally selected for reproduction based on their fitness values. Those with higher fitness values are more likely to be selected for the new generation. Evolutionary operations (such as mutation and crossover) will then be applied on the population to randomly vary the individuals. This is carried out repeatedly as shown in the figure 3.2, until some terminating criterion is met. The paradigms of evolutionary Algorithms include genetic algorithm, genetic programming, and evolution programming and evolution strategies. Genetic programming applies the GA concept to the generation of computer programs. Evolution programming uses mutations to evolve populations. Evolution strategies incorporate many features of the GA but use real-valued parameters in place of binary-valued parameters. Their main different come from the operators they use and in general way they implement the three mentioned stages: selection, reproduction and representation. A population of individual structures is initialized and then evolved from generation to generation by repeated applications of evaluation, selection, recombination, and mutation. The population size  $N$  is generally constant in an evolutionary algorithm, although there is no a priori reason (other than convenience) to make this assumption. Table 3.1 outlines a typical paradigm of the evolutionary algorithms.

Table 3.1. Paradigms in Evolutionary Algorithms

Paradigms	Proposed by
Genetic Algorithms	J. H. Holland [5]
Genetic Programming	J. Koza [84]
Evolution Strategies	I. Rechenberg [4]
Evolutionary Programming	L.J. Fogel, A.J.Owens, M.J.Walsh [3]

### 3.2.1 Genetic Algorithms

Genetic algorithms are search algorithms that are based on the principles of genetics and biological evolution. John Holland first proposed it as an efficient search mechanism in artificially adaptive systems [5]. Recently, genetic algorithms have received considerable attention regarding their potential as an optimisation technique for complex problems and have been applied to many real world problems such as scheduling and sequencing, reliability design, vehicle routing and scheduling, group technology, facility layout and transportation [12].

A genetic algorithm is executed as shown in the Figure 3.3, which is adapted from Goldberg [25], Davis [85], for more details on how genetic algorithms are used in various applications [86].

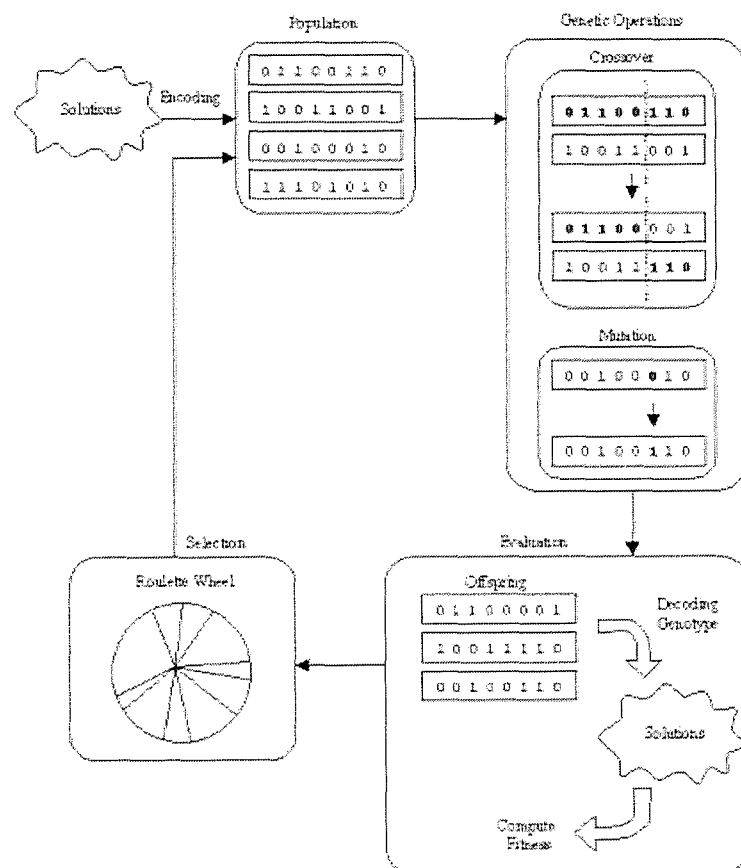


Figure 3.3. Execution of the genetic algorithms.

First, some terminologies used in genetic algorithms need to be explained. Genetic algorithms start with an initial set of randomly generated potential solutions called population. Each potential solution in the population is called a chromosome, and is normally represented as a string of symbols (binary bit string in conventional GA). The population of chromosomes evolves over successive iterations called generations. In each generation, chromosomes are evaluated using a fitness function. To form the next generation, applying the crossover and mutation genetic operators to the current population forms new chromosomes, called offspring. The entire population (parent and offspring) then undergoes selection based on their fitness value. Crossover, mutation and selection are discussed in more detail in following sections.

#### ***3.2.1.1 Usage Requirements of Genetic Algorithms***

In order to apply genetic algorithms to solve a problem, there are a few requirements, which needs to be satisfied. These requirements are discussed below.

##### ***Representation Scheme***

A representation scheme is needed for the chromosome. The length of the chromosome string, the alphabet size of the genes and the mapping that expresses each possible point in the search space of the problem as a fixed-length character string needs to be determined. Any chosen representation scheme must satisfy the sufficiency requirement (i.e. must be able to express a solution to the problem) in order for the problem to be solved using genetic algorithms. A decoding function is also required to decode the chromosome into a phenotypic expression that can be evaluated by the fitness function.

Genetic algorithm works with a coding of the parameters (the string or chromosome) and not with the parameters directly. Often, with integer parameters, plain binary coding is used [134]. If there is more than one parameter to be optimized then they are concatenated to give one string. Figure 3.4 illustrates this:

The problem has three parameters coded with 10, 4 and 6 bits respectively. The resulting string is obtained by concatenating those three parameters, resulting in a 20-bit length string.

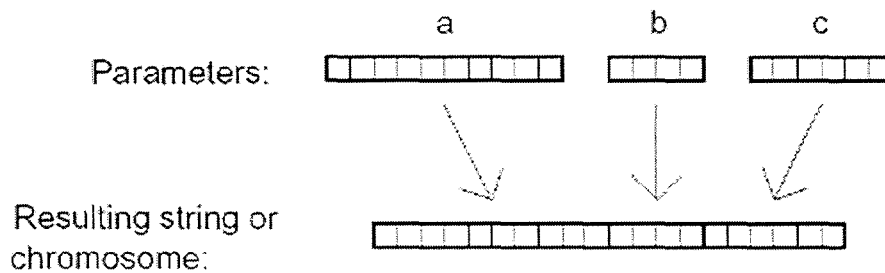


Figure 3.4. Coding of three parameters in a single string.

### ***Initial population***

The initial population is composed of random strings. It should sample sufficiently well the search space so that, ideally, no area of it possibly containing a solution is left behind. This is of course dependent of the size of the search space and/or the complexity of the problem. It is common for GA to have a population of 50 or more strings. Figure 3.5 shows an example of a random initial population of 6 chromosomes with a length of 12 bits.

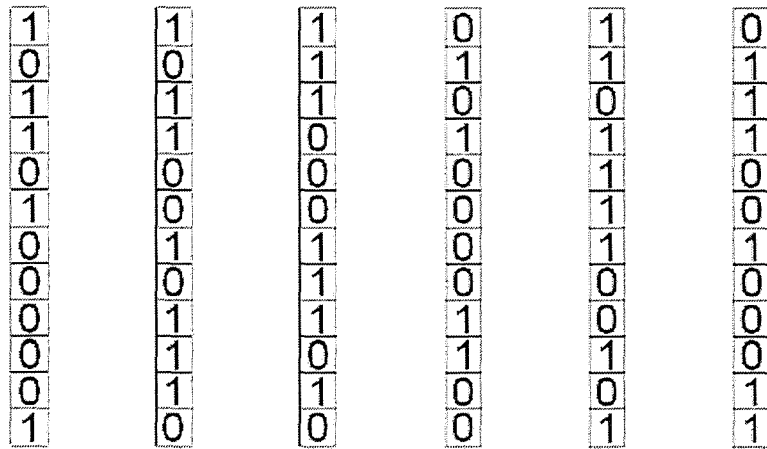


Figure 3.5. A Random initial population with chromosome length of 12 bits.

### Fitness Measure

A fitness measure that is able to assign a fitness value to each fixed-length character string it encounters in the population is required. This fitness measure (fitness function) must be fully defined (i.e. capable of evaluating any chromosome that can exist in the search space of the problem). The genetic algorithm needs to know the fitness of each string. To compute the fitness, the string has to be decoded to retrieve the parameters it contains, and then the problem can be evaluated with those parameters, resulting in the fitness value [134]. Keeping the same strings as in the example above, we could get the fitness shown in Figure 3.6.

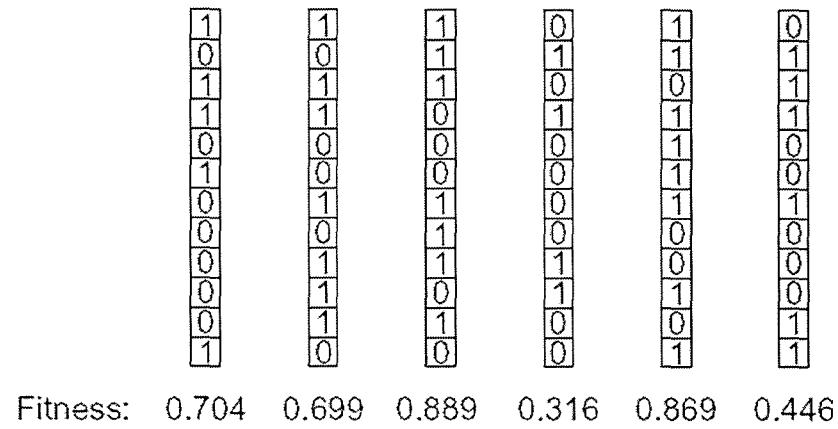


Figure 3.6. The fitness of each string has been computed.

### ***Selection***

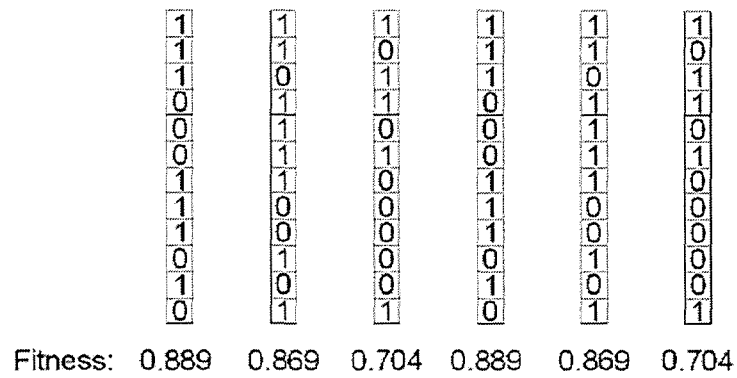
In this step we want to select the good strings for the next generation and discard the bad ones. Different methods can be used to perform this selection; two of those are the roulette wheel and the rank method.

#### ***A. Rank method***

In this method the strings are first sorted according to their fitness. Then, the first few good strings are kept and reproduced. The number of good strings to keep and reproduce is a parameter of the genetic algorithm. For example in a population of 100 strings the 25 best strings could be reproduced 4 times each to give the new population. In Figure 3.7(a) the first three strings are selected and are reproduced two times each. The resulting population is represented in Figure 3.7(b).



(a)



(b)

Figure 3.7. The new population with the rank method: (a) population in rank order, (b) the new population.

### B. Roulette wheel

The fitness value are first of all normalized such as their sum equals to one. Figure 3.8 illustrates this. The reproduction process can then be viewed as selecting the strings with a probability equal to their normalized fitness.



	<div>1</div> <div>0</div> <div>1</div> <div>1</div> <div>0</div> <div>1</div> <div>0</div> <div>0</div> <div>0</div> <div>0</div> <div>0</div> <div>1</div>	<div>1</div> <div>0</div> <div>1</div> <div>1</div> <div>0</div> <div>0</div> <div>1</div> <div>0</div> <div>1</div> <div>1</div> <div>1</div> <div>0</div>	<div>1</div> <div>1</div> <div>1</div> <div>0</div> <div>0</div> <div>0</div> <div>1</div> <div>1</div> <div>1</div> <div>0</div> <div>1</div> <div>0</div>	<div>0</div> <div>1</div> <div>0</div> <div>1</div> <div>0</div> <div>0</div> <div>0</div> <div>1</div> <div>1</div> <div>1</div> <div>0</div> <div>0</div>	<div>1</div> <div>1</div> <div>0</div> <div>1</div> <div>1</div> <div>1</div> <div>0</div> <div>0</div> <div>1</div> <div>0</div> <div>1</div> <div>1</div>	<div>0</div> <div>1</div> <div>1</div> <div>1</div> <div>0</div> <div>0</div> <div>1</div> <div>0</div> <div>0</div> <div>0</div> <div>1</div> <div>1</div>		
Fitness:	0.704	0.699	0.889	0.318	0.869	0.446	3.922	
Normalized fitness:	0.179	0.178	0.227	0.08	0.221	0.114	1.000	
								Sum

Figure 3.8. The fitness of the strings is normalized.

A simple implementation would be to use the roulette wheel (Figure 3.9). Roulette wheel spins a number of times equal to the size of the desired population to get the selected strings. In this example the wheel should be spun six times.

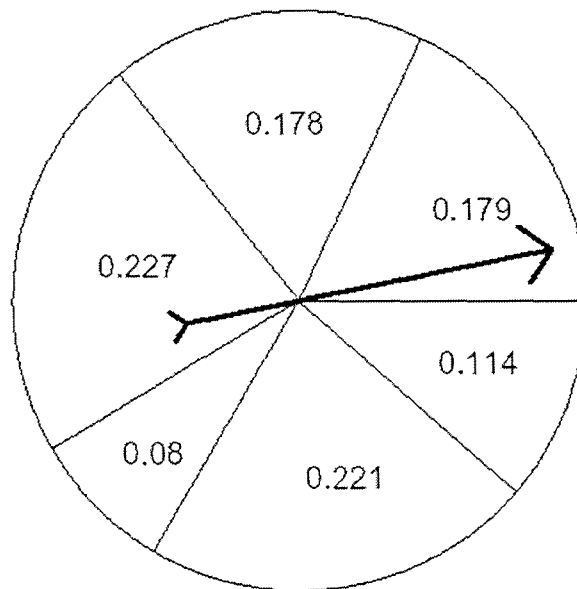


Figure 3. 9. The roulette wheel.

Whether to use the roulette wheel or the rank method generally depends on the problem. With the rank method the good strings are always reproduced, whereas with the roulette wheel the good strings only have a higher probability of being reproduced. Thus, with the roulette wheel, good strings might be discarded, resulting in a larger number of iteration for the genetic algorithm to find a solution. The advantage of roulette wheel would be to preserve a bit more diversity in the strings, by allowing, although with a small probability, some bad strings to survive. Whether this diversity is needed or not depends on the nature of the problem. If the initial population samples sufficiently well the search space this diversity may not be needed. On the contrary, if the initial population doesn't sample the search space well enough this added diversity could allow exploration at locations farther away from current good strings, possibly leading to the discovery of other points with a good fitness. There are a few parameters such as the population size, maximum number of generations and the terminating criteria that are required in order to execute a genetic algorithm. The terminating criterion is usually either that the maximum number of generations has been reached, or a solution that satisfies a problem-specific success predicate has been found.

#### **3.2.1.2. Genetic Operators**

There are two GA operators described as follows:

- **Crossover**

The crossover operator allows new individuals to be created by recombining portions of chromosomes from the parents to form the offspring. The purpose of this operator is to allow for new points in the search space to be tested. Crossover is carried out by first probabilistically selecting two parents from the population based on their fitness values. A crossover point is then randomly chosen (using uniform probability distribution), and the remainder portions of both chromosomes are swapped. This produces two new offspring with genetic material from both parents. Figure 3.10 shows more clearly how the crossover operation is performed.

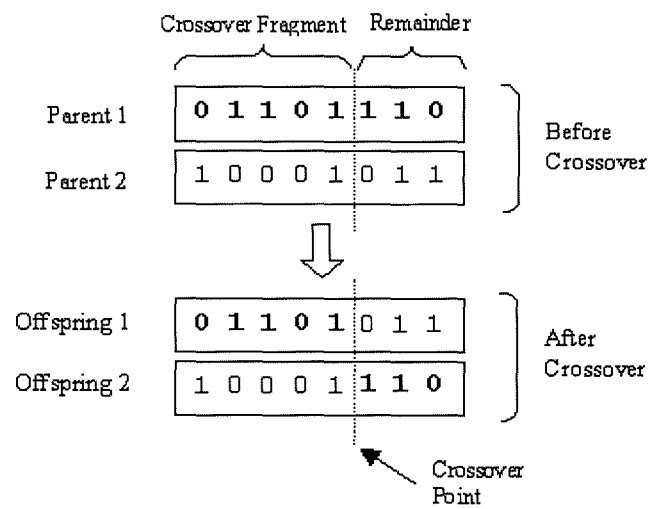


Figure 3.10. Single point crossover operator.

- **Two point crossover**

The process is essentially the same as with single point crossover, with the exception that two random locations are chosen. Figure 3.11 shows the two-point crossover where the bits between those two locations are swapped between the two strings.

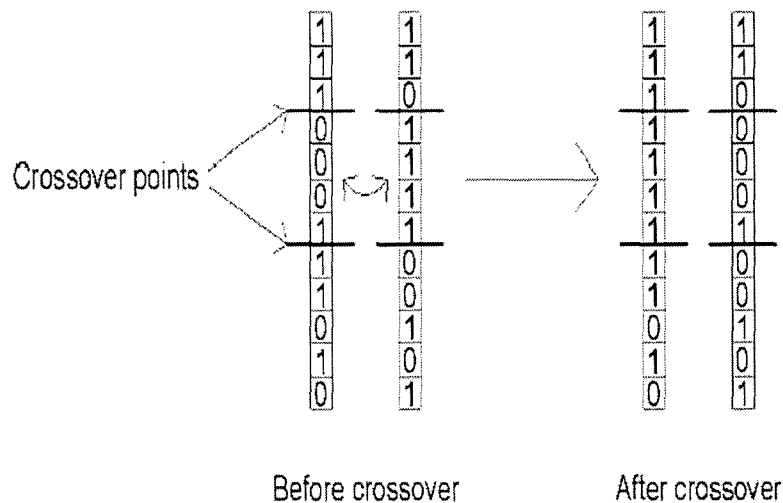


Figure 3.11. Two point crossover.

Multiple point crossover has been considered by De Jong [87] and his conclusions were that it decreased the performance increasingly with increased number of crossover points. An explanation is given in [25, p.119]: when the number of crossover points increase there are less structure that can be preserved and the crossover operator becomes more like a random shuffle.

- **Mutation**

This is carried out by probabilistically selecting an individual from the population, and randomly changing the single character (gene) at a randomly chosen mutation point (if the chromosome is a binary bit string, the bit at the mutation point is flipped). The probability of mutation is normally set a priori, and is usually very low. The purpose of mutation is to replace the genes lost from the population during the selection process so that they can be tested in a new context, or to provide new genes that were not present in the initial population [25]. This has the effect of increasing the search space and the chance of finding the global optimum solution rather than a local optimum.

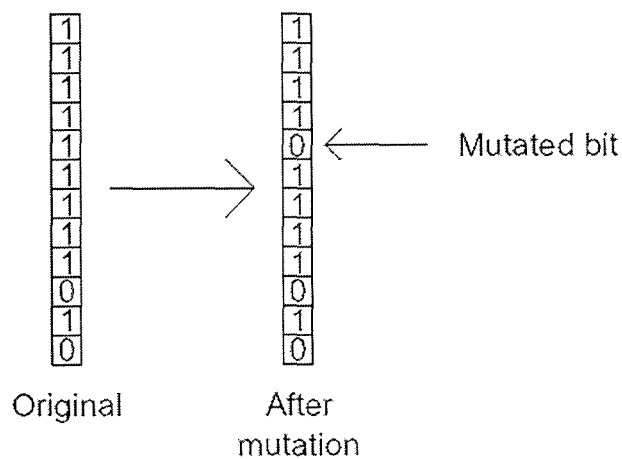


Figure 3.12. The mutation operator changed a random bit of the string.

Figure 3.12 show the mutation is the occasional random alteration of a bit in the string: changing a 1 in a 0 or vice versa.

### 3.2.1.3 Elitist Selection and Fitness Scaling:

In roulette-wheel selection, individuals have a probability of being selected that is proportional to its fitness value. This has some undesirable properties, one of which is the possibility that the best individual in a population may not be selected for reproduction. Elitist selection guarantees that the best individual in the population always survives by copying it directly into the next generation. This way, when a good solution is found, it will never be lost.

Also, as a population converges, the difference between fitness values of the individuals in the population decreases, resulting in a roulette-wheel with nearly equal sized slots. This means the chances of the best individuals being selected is about the same as the rest of the population, leading to a decrease in the rate of convergence. Fitness scaling provides a solution to this problem by scaling the fitness values to give fitter individuals an advantage. There are several methods of fitness scaling; of which "windowing" is probably the simplest. "Windowing" performs scaling by subtracting the lowest fitness value in the population from the fitness values of all individuals in the population.

### 3.2.2 Genetic Programming

John Koza [13] of Stanford University first introduced Genetic Programming (GP) (also called Evolutionary Programming, EP). GPs extend the GA approach to the space of programs in that the genotypes are effectively computer programs instead of fixed length strings. In principle, any language can be used but unfortunately, GP mutation and crossover would cause a large percentage of syntactic errors. Koza thus proposed a new language with syntax in prefix form (similar in structure to LISP).

For genetic programming, one must first define a set of functions  $F$  and a set of terminals  $T$  (constants and variables) assumed a priori to be useful for the problem at hand. The sets of  $F$  and  $T$  must exhibit syntactic closure. Therefore, each  $F$  in the set must accept as arguments any other  $F$  return value or any terminal in the set of  $T$ . Therefore, the search space is defined as

being the set of all possible composition of  $F$  that can be recursively formed from elements of  $F$  and  $T$ .

In order to illustrate this, it is best viewed as an example. Suppose the set of functions were defined as being  $\{+, -, *, /\}$  and  $T$  as  $\{A, B, C\}$ .

Thus, two example programs could be defined as being  $(+(A, B, *(C, B, B)))$  or  $-(A, /(b, C))$ . Both these programs are illustrated in Figure 3.13.

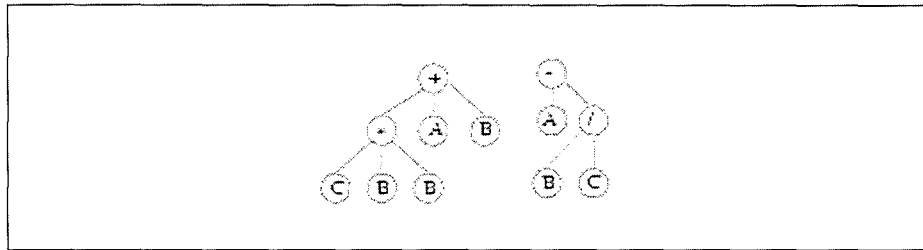


Figure 3. 13. GP example.

The advantage of using such a postfix form for program representation is that it easily lends itself to be structured in a tree-like fashion with  $F$  as the nodes and  $T$  as the leaves. As will be seen, this makes it easy to define the genetic operators.

#### 2.2.2.1 GP Crossover Operator

As illustrated in figure 3.14, crossover can be implemented in a straightforward fashion using this representation. A random sub tree is chosen in each genotype and then they are crossed creating two new programs.

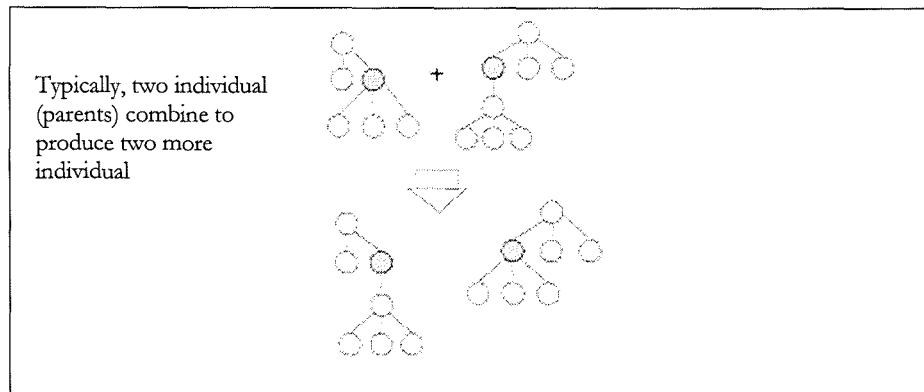


Figure 3.14. Simplified GP crossover operators.

#### 2.2.2.2 GP Mutation

Mutation is actually seldom used in GP but can be defined in several ways:

Sub tree destructive: Remove a sub tree and replace with a randomly generated one (or none at all). Node/sub tree swap: Randomly swap nodes or sub trees. An improvement in Genetic Programming that is often implemented is the generation of automatically defined functions. What essentially takes place is that during the course of a run, if a useful function is generated and identified then this function is packaged up and treated as one function in subsequent iterations.

### 3.2.3 Evolutionary Programming

Evolutionary programming (EP), developed by Fogel et al. [3] traditionally has used representations that are tailored to the problem domain. For example, in real valued optimization problems, the individuals within the population are real-valued vectors. Similarly, ordered lists are used for traveling salesman problems (TSP), and graphs for applications with finite state machines. EP is often used as an optimizer, although it arose from the desire to generate machine intelligence. The outline of the evolutionary programming algorithm is shown in Figure 3.15. After initialization, all  $N$  individuals are selected to be parents, and then are mutated, producing  $N$  children. These children are evaluated and  $N$  survivors are chosen from the  $2N$  individuals, using a probabilistic function based on fitness. In other

words, individuals with a greater fitness have a higher chance of survival. The form of mutation is based on the representation used, and is often adaptive. For example, when using a real-valued vector, each variable within an individual may have an adaptive mutation rate that is normally distributed with a zero expectation. Recombination is not generally performed since the forms of mutation used are quite flexible and can produce perturbations similar to recombination, if desired. As discussed in a later in section 4.8, one of the interesting and open issues is the extent to which an EA is affected by its choice of the operators used to produce variability and novelty in evolving populations.

```

Procedure EP; {
    t = 0;
    initialize population P(t);
    evaluate P (t);
    until (done) {
        t = t + 1;
        parent_selection P(t);
        mutate P(t);
        evaluate P(t);
        survive P(t);
    }}

```

Figure 3.15. Evolutionary programming algorithms.

### 3.2.4 Evolution Strategies

Evolution strategies (ESs) were independently developed by Rechenberg [4], with selection, mutation, and a population of size one. Schwefel [82] introduced recombination and populations with more than one individual, and provided a nice comparison of ESs with more traditional optimization techniques. Due to initial interest in hydrodynamic optimization problems, evolution strategies typically use real-valued vector representations. Figure 3.16 outlines a typical evolution strategy. After initialization and evaluation, individuals are selected uniformly randomly to be parents. In the standard recombinative ES, a pair of parents produces children via recombination, which are further perturbed via mutation. The number of children created is greater than  $N$ . Survival is deterministic and is implemented in one of two



ways. The first allows the  $N$  best children to survive, and replaces the parents with these children. The second allows the  $N$  best children and parents to survive. Like EP, considerable effort has focused on adapting mutation as the algorithm runs by allowing each variable within an individual to have an adaptive mutation rate that is normally distributed with a zero expectation. Unlike EP, however, recombination does play an important role in evolution strategies, especially in adapting mutation.

```

Procedure ES; {
    t = 0;
    initialize population P(t);
    evaluate P(t);
    until (done) {
        t = t + 1;
        parent_selection P(t);
        recombine P(t);
        mutate P(t);
        evaluate P(t);
        survive P (t);
    }
}

```

Figure 3.16. The evolution strategy algorithm.

### 3.2.5 Phenotypic versus Genotype Evolution

Evolvability strongly depends on a process of translation of genotypes to phenotypes. According to Fogel, "Living organisms can be viewed as a duality of their genotype and their phenotype"[88]. The genotype is the genetic constitution underlying a single trait or set of traits (the genetic coding), and the phenotype is the functional expression of a trait (behaviour, physiology and morphology).

In evolutionary computation, the parameters that are subject to optimisation constitute the phenotypic (behavioural) space, while genetic operators such as mutation and crossover in genetic algorithms work on abstract mathematical objects like binary strings that constitute the genotype (informational) space. A mapping (decoding function) between the phenotypic space and the genotype space is therefore required in genetic algorithms (Figure 3.17).

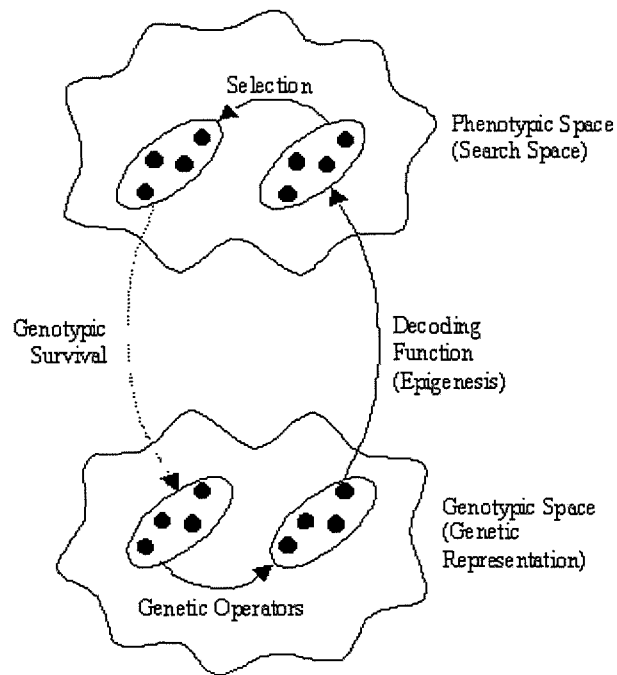


Figure 3.17. Mapping between genotype space and phenotypic space.

In addition, solutions in genetic algorithms are typically encoded in chromosomes with the assumption of a "one-gene/one-trait" model of the relationship between the genotype and the phenotype. This assumption is not however, entirely valid given the pleiotrophic and polygenic nature of the genotype [25] Pleiotrophy is the effect that a single gene may simultaneously affect several phenotypic traits and polygeny is the effect that a single phenotypic characteristic of an individual may be determined by the simultaneous interaction of many genes. Natural evolution, unlike genetic algorithms, evolves optimal solutions that are comprised of components that strongly interact seamlessly in some purposeful manner, without separate optimisation of the components. Also, because selection acts from the top-down, survival of an individual depends on the suitability of its collective behaviour, rather than the superiority of some individual behaviour. David Fogel [2] also argues similarly, that this "one-gene/one-trait" model of evolutionary genetics is an oversimplification given that naturally evolved

systems are both "extensively pleiotrophic and highly polygenic", as well as the fact that selection acts on the collection of phenotypic traits of an individual, not on singular traits in isolation (i.e. top-down) [89].

Biologist, Ernst Mayr has stated, "The genes are not the units of evolution nor are they as such the targets of natural selection" [88]. This means, only the behaviour of an individual in response to its environment has selective value, regardless of how this behaviour is generated. Evolutionary programming takes a top-down approach towards evolving solutions for solving problems, instead of the bottom-up approach taken by genetic algorithms. In evolutionary programming, the representations of the solutions are abstracted as vectors of behavioural traits, which reside in the phenotypic space, thereby avoiding the simplifying assumptions in genetic algorithms. This also means that there is no need for a decoding function. The significance of this is that a decoding function may introduce additional non-linearity and other mathematical difficulties, which can hinder the search process substantially. The mutation operator in evolutionary programming also operates at the phenotypic level, as opposed to the genetic operators in genetic algorithms.

Studies and research have shown that evolutionary programming and evolution strategies have repeatedly outperformed genetic algorithms in various optimisation problems. Lawrence Fogel gave a list of examples for this, which includes the travelling salesman problem, automatic control, pattern recognition, functions with multiple local optima and even the test suite of functions that have been used by the genetic algorithm community for tuning their procedures [88].

### **3. 3 Evolutionary Designs of Digital Circuits**

Digital design as opposed to analogue design does not use full properties of underlying technology. It works at an abstract level where signal value above or below given thresholds are categorised as logic values. This mean that circuit design is easier to achieve since slight variations in the low-level signal value do not affect the logic value.

In recent years with greater demand for more efficient methods for electronic circuits design a novel method has been introduced to design digital circuit called evolutionary hardware.

The evolutionary design of digital circuits is a process of evolving configurations of logic gates for some prespecified computational program. Often the aim is for a highly efficient electronic circuit to emerge in a population of instances of the program. The motivation behind the study of digital circuit evolution is to design electronic circuits that are more efficient than the conventional designs, and then, to learn new principles of design. Learning new principles of design is beneficial for the design of electronic circuits [1,11]. Thus new methods and principles of evolving digital circuits are inferred.

### **3.4 Evolvable Hardware**

#### **3.4.1 Introduction**

Evolvable Hardware (EHW) refers to hardware that can change its architecture, and thus its behavior, dynamically and autonomously by interacting with its environment [100]. This domain emerged from the combination of evolutionary Learning and the rapid progress of electronic hardware during the recent years. While this was impossible some years back, Reconfigurable Hardware like FPGA can be reprogrammed both very quickly and while in operation. If we combine this with techniques like genetic algorithms that can solve difficult learning and searching tasks, we come to the idea of hardware that can learn from expertise and be reprogrammed to perform better next time. However, few industrial applications or commercial products of evolutionary algorithms have been reported so far, because of their computational costs. EHW is a promising approach to overcome such problem, and makes use of the adaptive capability of GA by reducing its computational costs. EHW is a hardware device, which is built on a software-reconfigurable device.

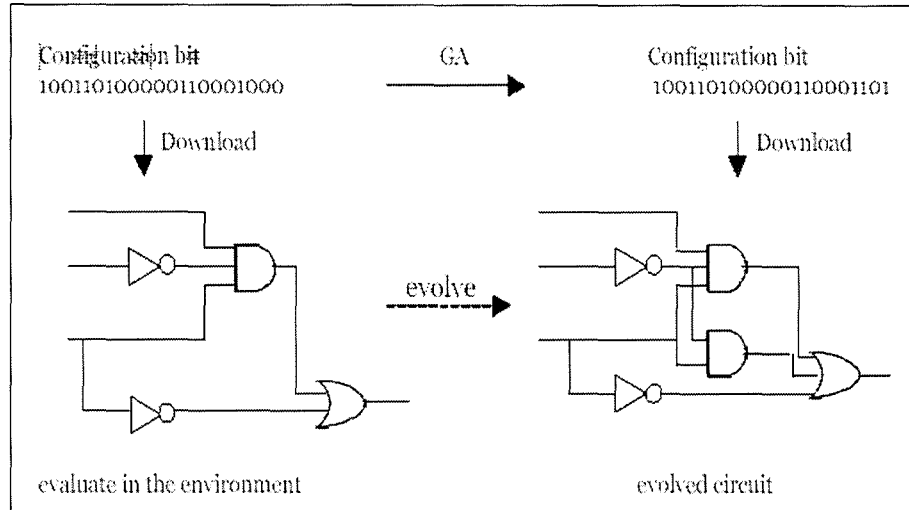


Figure 3.18. The concept scheme of Evolvable hardware.

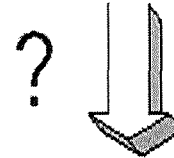
Its structure can be determined by downloading binary bit strings called the architecture bits. The architecture bits are treated as chromosomes in the population by the GA, and can be downloaded to the reconfigurable device resulting in changes to the hardware structure. The changed functionality of the device can then be evaluated and the fitness of the chromosome is calculated. The performance of the device is improved as GA evolves the population according to fitness [9,11].

The chromosome of EHW specifies two things. One is the function type of the evolution unit. In figure 3.18 the evolution units correspond to gates like AND/OR gates. The other is the interconnection among the evolution units. EHW can be classified into two classes according to the grain-size of an evolution unit, gate-level & function-level. Figure 3.19 shows the process of extrinsic EHW. The figure is an example of gate-level evolution and function-level evolution, where each evolution unit is higher hardware function than gate-level evolution

## EHW process:

1. **Initialisation** : randomly generated initial population
2. **Evaluation** : chromosomes in initial population
3. **Evolution** : change the genotype of chromosomes
4. **Evaluation** : chromosomes in current population
5. **If** terminate = "no", then GO to Step 3, **Else** GO to Step 6
6. **Evaluation** : chromosomes in final population

Desired logic function												
$X_1, X_2, \dots, X_M$						$Y_1, Y_2, \dots, Y_M$						
0	0	...	1			1	0	...	0			
0	1	...	1			0	1	...	0			
.	.					.	.					
.	.					.	.					
1	1	...	1			1	1	...	0			



### Evolved logic circuit

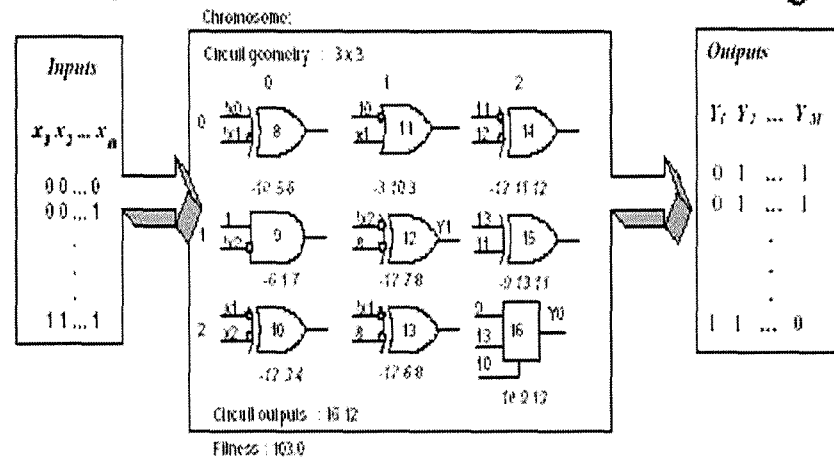


Figure 3.19. Circuit design problem in EHW. In EHW approach, the circuits in initial population are generated randomly and do not implement the desired logic function. Therefore, an evolutionary algorithm design a circuit that correctly implements given logic function and optimises fully functional circuit. In other words, evolutionary algorithm evolves a logic circuit [24].

### 3.4.2 Taxonomy of EHW

Each evolutionary electronic system can generally be characterised by five main key properties [100] (see Figure 3.20).

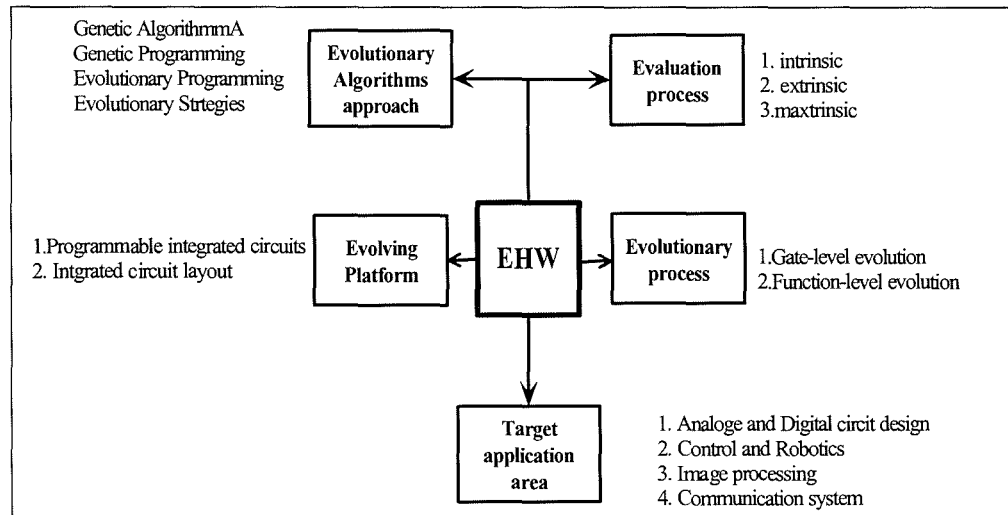


Figure 3.20. Taxonomy of Evolvable Hardware.

#### 3.4.2.1 Evaluation process

In EHW, evaluations of chromosomes can be performed on software or hardware. Depending on how the evolution process is implemented, two of evolutionary process can be defined:

**Intrinsic EHW**, the circuit gets configured for each chromosome for each generation [16].

**Extrinsic EHW** the evolution is simulated in software, and only the elite chromosome (i.e. the configuring bit string) gets written. Thus the circuit is configured only once [17].

##### 3.4.2.1.1 Extrinsic EHW

Extrinsic design marked the beginning of the field of evolutionary electronics. It is in essence the application of evolutionary methodologies to hardware design in simulation. The approach thus consists of using software simulation to evaluate evolving circuits. At the end of the evolutionary process the best

individual is downloaded into the reconfigurable device. The extrinsic approach is simpler than the intrinsic approach and helps to draw theoretical results into the EHW area. Extrinsic design as mentioned before should actually be referred to as Evolutionary Circuit Design. This is because extrinsic design is in fact just that the use of evolutionary techniques as an aid in designing circuits.

The main aim is the creation of an evolution based automation tool to design electronic circuits without the need of any kind of human intervention such that the evolutionary technique can replace the circuit designer. Using this approach, the designer must only worry about the specification of the system with the evolutionary process worrying about the rest [24].

Though a lot of the research in the area is moving towards intrinsic design, extrinsic hardware evolution does fulfill a niche in hardware design. Most importantly, as will be seen in the next section, intrinsic designs are not universal. This is due to the fact that analogue transients affect the evolutionary process implicitly. Thus, the same configuration downloaded onto a different programmable device, results in degradation in performance. As extrinsic design is done in simulation however, it is subject to the same abstractions placed on human designers. Thus, the solutions produced are as universal as conventional human designs are. A further important aspect of extrinsic evolution is that it is not limited to programmable devices. Thus, it can have a much wider scope for its applications. An evolved program can easily be designed to evolve towards a solution with particular characteristic advantages and thus several solutions for the same problem can be realized, each one optimized for a different aspect [13]

In this way, the evolutionary algorithm can find novel approaches to problems that may not have been seen before as well as giving some very useful insights into possible solutions. The extrinsic EHW approach has been studied more than the intrinsic approach [84]. In Koza et al. [101] analogue



electronic circuits are successfully evolved using the SPICE simulation program to perform for standard mathematical operation: cube root, square root, cube and square. The extrinsic approach is simpler than the intrinsic approach and helps to draw theoretical results into the EHW area [11].

#### *3.4.2.1.2 Intrinsic EHW (on line adaptation)*

EHW is only really interesting if it can be used to make systems adapt intrinsically, i.e. while in operation (see Figure 3.21). In this way, systems could cope with a changing environment or possible errors [100, 102, 103]. Unfortunately, such on-line adaptation is very difficult to achieve. Not only because of the learning process involved, but primarily because of the on-line requirement. The combination of fitness evaluation of a possible solution during the operation of the circuit is what causes part of the problem. Imagine for example the on-line adaptation of a robot controller circuit. Fitness evaluation would then load the possible solution into the controller and have it perform a test run to see how well the individual performs in the real environment. But because of the nature of crossover and mutation operators, very poor individuals can be created. Evaluating such a poor product of evolution in the real environment could cause the robot for example to hit obstacles, maybe even destroying things on its path. This is not the good approach to on-line evaluation. Although part of it is being solved by modern FPGAs, part of the problem still is the swapping of the program of a PLD during operation. In some way, the operation of the circuit has to be interrupted to allow reloading of data. This can only be done if the circuit is turned off from the outside, or it is made sure in some way that the circuit is temporarily not needed by the system it is in. Modern FPGAs, such as the Xilinx chip XC6216 model offer some interesting properties to improve the reloading of programs during operation, such as high reconfiguration speeds of reloading parts of the circuit, which can be used to improve reprogrammability. Another way of solving the reloading problem is having more than one circuit standing by, and using only the best one. The other circuit can then

learn from the environment, be reloaded, and take over from the first circuit, once its fitness improves.

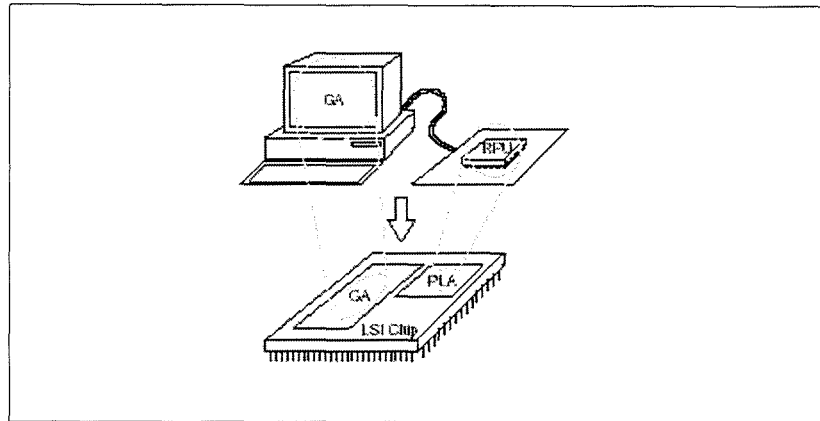


Figure 3.21. On-line EHW.

#### 3.4.2.2 *Evolving platform*

The field of EHW is relatively young but there have been many changes in the technology platform used. Some researchers have approached this problem by building their own platforms [8]. Evolving a VLSI design is either on a function level or a direct layout of oxides, metal and silicon. The function level can be more or less high level. Constructing circuits from a high level library of macro-cells or only by use of transistors (also a macro) are examples of functional EHW. The functional approaches do not handle the routing and the floor-planning problem of the chip design. These problems can be viewed as hard combinational problem that can be solved traditionally or by mean of evolutionary Algorithms [12]. The evolving platform is the physical media that the EHW system is intended to evolve circuit specification for. In the following different physical media for EHW are described.

#### **Field-Programmable Gate Arrays**

An FPGA (Field-Programmable Gate Array) is basically a chip that can be configured (i.e., programmed via software) to realize any given function (that is, to implement any digital logic circuits). They are two-dimensional arrays of

logic elements and, while the exact structure of an FPGA element can vary considerably from one manufacturer to the next, certain essential features are constant Figure 3.22; each element can implement a programmable function, usually consisting of combinational logic plus one or more memory elements (flip-flops) for sequential behavior. The complexity and the structure of the programmable function can vary considerably from one type of FPGA to the next (for example, the combinational part can be implemented using a lookup table, as in the Xilinx XC4000 and XC5200 families, or be hard-wired, as in the Xilinx XC6200 [104] family. Communication between the elements is handled through programmable connections, again of varying complexity depending on the type of FPGA. Experience has shown that connections (rather than functionality) are the main bottleneck for the layout of FPGA circuits, an observation that has led most designers to add long-distance connections distributed homogenously throughout the array.

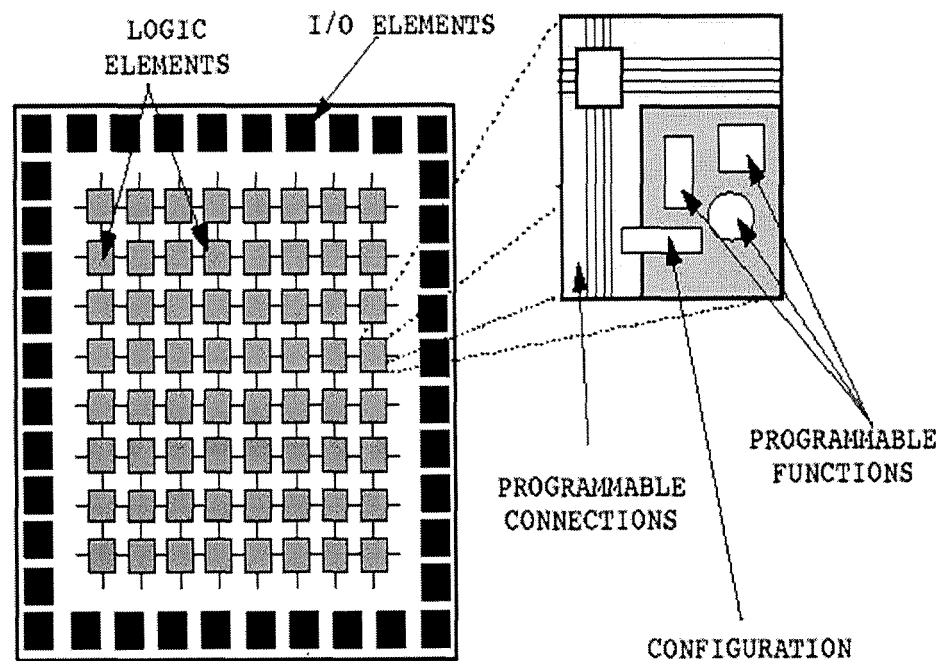


Figure 3.22. Basic structure of a generic FPGA circuit.

The functionality and the connections of an element are controlled by its configuration. The configuration bit stream (that is, the sum of all the configurations of the FPGA's elements determines the global behavior of the chip. An FPGA can be configured to implement any digital logic circuit (provided enough elements are available or the circuit can be subdivided among different chips) and in most cases is reprogrammable (that is, its configuration can be erased and replaced by a new one, implementing a different circuit).

The advent of FPGAs and, more recently, of open-architecture FPGAs, has given a considerable impetus to evolvable hardware research [91]. With FPGAs such as the Xilinx 6216, hardware architecture becomes as malleable as software. No longer is there any need to use a computer simulation to evolve electronic circuits, downloading only the final elite chromosome to a programmable chip. FPGAs allow online hardware evolution where all chromosomes can be downloaded to the FPGA and evaluated as configurations of real hardware [92,93,94].

Researchers configured FPGA circuits so that each cell within the system behaved in a random manner, then evolved cellular automata rules so that all cells oscillated between all 0s and all 1s in unison. The important point about this implementation is that the evolutionary algorithm operates solely within the FPGA circuits, with no reference at all to an external computer.

However, the most astonishing example of hardware evolution has to be the work of Thompson [102], who used genetic algorithms to coax a Xilinx XC6216 chip into distinguishing between the frequencies of two square wave inputs - without the aid of a clock input. Thompson believes that the structural and behavioural constraints inherited from conventional design methodologies can unconsciously become prejudices about what sorts of circuits are possible. As a result he has pioneered what he calls 'an unconstrained' approach to intrinsic hardware evolution.

### 3.4.2.3 Evolution process

Evolution at EHW can be performed in different level: -

#### Gate-level evolutions

The hardware evolution employing the primitive gate such as AND, OR and NOT gates in Figure 3.23. The works in [10,11, 18, 24] describe hardware evolution at gate level and demonstrate a combination of genetic learning for EHW which changes its own hardware structure in order to adapt to the environment in which it is embedded.

Logic synthesis tools use a library of gates to create circuits at the logic level. In [1] the benchmark circuits were synthesized using EHW from state tables into gate-level networks. More details are given in chapter 5.

Gate level evolved hardware is used in a number of real applications such as a Robot navigation system [9] and a pattern recognition system [105]. The main reason for using such hardware is to increase the speed of operation. Its generalizing ability has a considerable advantage due to the small number of observable data obtained by the robot interacting with its environment. However, the size of a circuit available to gate-level evolution is not so large because of the limitation of GA execution time. Thus the gate-level EHW can't be used for practical application. In order to solve this problem the function level evolution is proposed.

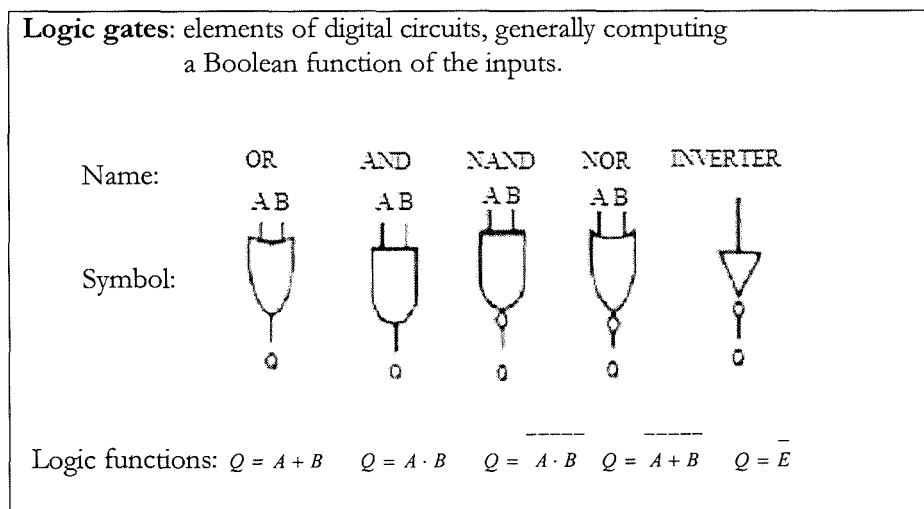


Figure 3.23. The primitive logic gates.

### **Function-level evolutions**

Function level evolution [24,13,63] is a natural extension of gate-level evolution. In the function level evolution, high level hardware functions, such as addition, subtraction, sine, etc., rather than simple logic function are used as primitive building blocks in the evolution. The function level EHW combines the advantage of both hardware and software, which make it suitable for applications that must be processed at very high speed. In [85] an evolutionary algorithm is applied to solve the system level synthesis (a mapping from a behavioural description) problem.

### **3.5 Differences from Traditional Methods**

Evolutionary Algorithms methods generally differ from traditional search and optimisation techniques in three main areas [13, 18] in that evolutionary algorithms method:

1. Utilise a population of potential solutions in their search.
2. Use direct "fitness" information instead of function derivatives or other related knowledge.
3. Use probabilistic, rather than deterministic transition rules.

Most traditional optimisation methods move from one point in the decision hyperspace to another using some deterministic rule. The problem with this is that it is likely to get stuck at local maximal. Evolutionary Algorithms methods start with a diverse set (population) of potential solutions (hyperspace vectors), and typically, a new population of the same size is generated in each generation after. This allows for exploration of many maxims in parallel, lowering the probability of getting stuck.

Even though evolutionary algorithm methods are probabilistic, they are not strictly random search. The stochastic operators used in the operations on the population direct the search towards regions of the hyperspace that are likely

to have higher fitness values. Table 3.2 described some differences between design process and Evolution process

Table 3.2. The different between design process and Evolution process.

Design	Evolution
<p>The design process is a top down approach</p> <p><b>Key characteristics:</b></p> <ul style="list-style-type: none"> <li>• The complete specification of the input-output relationships must be known (e.g. timing diagrams, truth tables)</li> <li>• Hierarchical decomposition in simpler blocks</li> <li>• Circuits designed to tolerate manufacturing process variations</li> <li>• Use of mathematical models and design methodologies</li> <li>• Inner working is chosen by the designer</li> </ul>	<p>The evolutionary process is a bottom up approach</p> <p><b>Key characteristics:</b></p> <ul style="list-style-type: none"> <li>• No need for a detailed specification: the global quality (fitness) of the circuit is used</li> <li>• Flattened approach (no redundancy between building blocks)</li> <li>• Circuits are often tuned to the hardware they were evolved on</li> <li>• Evolution does not use mathematical models or design methodologies</li> <li>• Inner working is chosen by the evolutionary process</li> </ul>

### 3.6 Evolvable hardware implementation

In recent year the application of EHW grew in different direction as mentioned in Figure 3.20. All the known ways to design circuits can potentially be subject to evolution. For example, digital circuits where more complex digital circuits are generally done at a more high level using some

description language. The basic components (inverters, gates, multiplexers, etc.) are used to design digital circuits.

For analogue circuits there is little to no choice other than to design directly at this level each transistor size can possibly be different; complex design rules are necessary to realize circuits that work despite the fluctuations in the manufacturing process; layouts have to be carefully thought so that the temperature gradient inside the chip do not alter the functioning of the circuit. Evolutionary techniques could be used to evolve the layout of the masks directly. To our knowledge, this has never been done for a number of reasons: the search space is really huge; to get the fitness of the circuit a simulation has to be done, but even if the circuit is working in simulation there is no certitude that it will work in silicon manufacturing a chip costs too much and takes way too long, ultimately, there are still many difficulties with approaches at a higher level, and working at a low level would not help, but only add problems. Even if this approach seems to be out of question today, this is where EHW might lead us in a few years.

Other implementations of EHW have also been explored such as, for example, evolving resistors and capacitors values to perform analog filtering. Antenna design is also a vast field of application of evolutionary techniques. They are now industrially used to evolve application specific antennae. See for example [106] where genetic programming is used to design antennae, or [107] which reviews antennae that have been evolved for unconventional applications. See [108] for a broader set of examples of evolutionary techniques put to use in industrial applications.

EHW already incorporates a large set of application. Special attention will be paid in this thesis to the progress made in the area of extrinsic EHW applied to digital circuit design. A number of works has been done in the area of evolving combinational circuits [24]. In this dissertation, an extrinsic EHW applied to combinational logic circuit will be extended to design sequential



circuits [1]. The needs of the EHW realm are discussed and the research area of this work drawn.

### **3.6.1 Advantages**

Designing a circuit using evolutionary techniques has some advantages over classical circuit design.

- Evolutionary design can explore a much wider range of solutions than human designers.
- People who have only basic knowledge of the domain in which the design is wanted can use evolutionary design.
- Evolutionary design can cope with varying degrees of constraints and special requirements. An extra constraint just needs to be incorporated in the chromosomes and fitness function to expand the requirements [109].

Circuits can be created even if the I/O relationships are not precisely known. This is especially useful when the problem is difficult to characterize (e.g. robot controller). A circuit can be evolved as long as the global performance of the circuit can be measured.

Traditional models and design methodologies do not limit evolution. More performant circuits that are tuned to the hardware can thus be evolved (e.g. smaller or faster circuits). This is especially true for what is called unconstrained evolution.

Environments where the logical behaviour of the hardware is not known in advance. In such situation, the traditional method of hardware design is ineffective. Furthermore, EHW will be very effective in situation where errors can occur, or where input data from the environment is unavailable due to accidents. Tradition hardware design overcame such problem with fault-tolerant techniques, usually by using redundant circuits. An exciting aspect of hardware evolution is that very high-speed tasks can be performed, for instance, pattern recognition or control [105]. The function level EHW

combines the advantage of both hardware and software, which make it suitable for applications that must be processed at very high speed.

### **3.6.2 Disadvantage of evolutionary Design**

The computational time complexity related to the length of chromosome has been mentioned in all EA used for problem solving [13,25].

Sometimes a fitness function is very difficult to find without paying a heavy computational cost in EAs. For example in digital circuit design, a circuit with a number of external inputs will require  $2^n$  (where  $n$  is the external input) possible input combinations when calculating functionality. It is thus difficult to obtain results for circuits with a large number of inputs.

In the same study on EHW experiments [12], the correctness of evolved circuit was terminated. Where in the experimental results, not every single run can generate a correct circuit.

Evaluating an EHW in a real physical environment can cause damage to the EHW or the physical environment.

Most researches on EHW, so far, have a common problem in that the evolved circuit size is small [21]. Further the hardware evolution is based on primitive gates such as the AND/OR gates. This gate level evolution is not powerful enough for industrial applications.

### 3.7 Summary

The area of Evolutionary Algorithms is a very rich area and has grown tremendously in the last decade. This chapter in no way attempts to cover the field in depth but rather to give an overview of the field and in particular, the areas of the field of particular importance for evolutionary electronics. As we shall see throughout this dissertation, the two most important issues when considering the application of GAs or ES involve the definition of a genotype phenotype mapping as well as the definition of a fitness criterion to guide the evolutionary process. Once these can be made, GA can be applied in a very straightforward manner to a large number of areas.

The chapter has also given a brief account of evolutionary algorithm used for circuit design aspects. A set of major algorithms exists and the major difference between them can be summarized as follows:

- GA: selection, crossover, mutation, and binary representation.
- EP: selection, crossover, mutation integer and binary representation.
- GP: selection, crossover, and mutation program representation.
- ES: mutation elitism, binary, integer and real representation.

The chapter discussed the main concepts of EHW and their application according to five important properties summarised in EHW taxonomy properties classification. And finally summary of the advantages and disadvantages of using EHW in real world application has been given.

## *Chapter 4*

### OPTIMAL STATE ASSIGNMENT OF FINITE STATE MACHINE

#### **4.1 Introduction**

Synthesis of VLSI circuits involves transforming a specification of circuit behaviour into a mask-level layout that can be fabricated using VLSI manufacturing processes. Optimization strategies are necessary in VLSI synthesis in order to meet the desired specification. The first order optimization criteria in this process are typically all a desired subset of area optimality, speed, power dissipation and testability. Considerable progress has been made in understanding combinational logic optimization in the recent past and large number of universities and industrial CAD programs are now available for optimal logic synthesis of combinational circuits. Optimization of sequential circuits is considerably less mature [28,29,52,103].

As explained in chapter 2, state assignment is an important step in the synthesis of a sequential circuit. This encoding determines the complexity and the structure of the sequential circuit realizing the state machine and therefore has a profound effect on its area. There is no exact algorithmic approach for finding the optimum state assignment for the internal states of a sequential circuit. Various heuristic methods have been proposed for obtaining reasonable assignments. State assignment algorithms assign binary codes to each symbolic state. These codes are used to create a logic-level implementation using a register per bit of binary code, so the encoding has a direct effect on the quality of the implementation. The state assignment problem has been studied periodically since the early 1960s [45,46], and is one of the classical unsolved problems in logic synthesis. Recently developed state assignment algorithms attempt to predict the effect of encoding on the size of resulting two level and multilevel implementations. Techniques for the

optimal state assignment of sequential circuits synthesis will be presented in this chapter. A more detailed discussion of these techniques and of genetic algorithms for state assignment will be discussed here.

#### 4.2 Complexity of the state assignment problem

In the design of switching systems or circuits, which are to handle digital information, a problem arise as to how that information is to be coded. In general the states of the FSM keep information of previous inputs. In an implementation of the FSM as a logic circuit, the present state of the FSM has to be stored in memory. Figure 2.4 in chapter 2 shows a generic block diagram of a FSM. In order to obtain this logic circuit, a binary code has to be assigned to each state. The only restriction for a valid state encoding is that each state has to be assigned a unique binary value.

If the numbers of states in the FSM is  $n$  then the number of states variables  $b$  is the smallest integer that is equal to or grater than  $\lceil \log_2 n \rceil$ . The assignment process is that of deciding which of the  $2^b$  codes provided by  $b$  state variables must be assigned to any particular state in the state machine. The first state of FSM can be allocated any one of the  $2^b$  combinations; the second state can be allocated any one of remaining  $2^b - 1$  combinations, etc.; hence the  $n^{\text{th}}$  state of FSM can be assigned any one of the  $2^b - n + 1$  combinations of state variables.

Then the total number of different possible encoding [110] is given by

$$T(n, b) = \frac{2^b!}{(2^b - n)!}$$

possibility of assigning  $2^b$  combination of state variable to the  $n$  states. The state variable can be permuted  $b!$  ways. In addition, each state variable can be complemented, so the set of state variables  $b$  can be complemented in  $2^b$

ways. Many of these possible encoding yields the same logic circuit. This is the case when the encoding differs only by having one column complimented ( $2^b$  cases) and when the encoding differs only by having two columns permuted ( $b!$  cases). Then, the total number of distinct encoding that yield different logic circuit is

$$A(n,b)=T(n,b)/(2^b \cdot b!)=\frac{(2^b-1)!}{b! (2^b-n)!}$$

Table 4.1 shows the value of  $T(n,b)$  and  $A(n,b)$  for different  $n$  and  $b$ . As can be seen; the number of combination is very large. Thus even for FSM with six states the number of possible state assignment to be considered is 420, and it rapidly rise to more than ten million for a circuit with nine states. Since the number of possible state assignments grows profusely with the number of internal states, it is almost impossible to try all assignment in order to select the one leads to the simplest logic circuit design.

To try a new state assignment every 100  $\mu$ s, it would take 66 years to try all possible assignments for a small FSM with 16 states [45].

Table 4. 1. Number of Different state assignments.

$n$	$b$	$T(n, b)$	$A(n, b)$
2	1	2	1
3	2	24	3
4	2	24	3
5	3	6720	140
6	3	20160	420
7	3	40320	840
8	3	40320	840
9	4	$\approx 4 \cdot 10^{19}$	$10^7$
10	4	$\approx 2 \cdot 10^{13}$	$\approx 5 \cdot 10^{10}$
11	4	-	-
12	4	-	-
13	4	-	-
14	4	-	-
15	4	-	-
16	4	-	-
17	5	$\approx 2 \cdot 10^{23}$	$\approx 10^{20}$

Any valid state encoding will lead to a correct implementation of a FSM, though the complexity of the associated logic circuit may depend heavily on the state assignments.

#### 4.2.1 Problem Formulation

The goal of optimum state assignment is to transform sequential circuits behaviours into functional circuits, which are less expansive according to some cost function. The cost function typically incorporates (area, speed, power consumption, testability) as the main objective of the optimization procedure. This chapter present concept and algorithm, which are mostly independent of specific cost function and are basic ingredients for a logic synthesis tool. In this work we will assume that minimum area is the primary objective of the optimization procedure. There are many possibilities for estimating the area requirement of physical circuit implementations if the circuit is described at gate-level. The gates, literals and connection count for next state and outputs equations are the optimization goals in this work.

A literal is defined as a Boolean variable or its complemented (e.g.  $A$  or  $\bar{A}$ ).

The numbers of literals needed to describe each function at individual nodes summed over all nodes of the Boolean network represent the circuit. A connection is defined to be a distinct input of a gate having at least two inputs, i.e. input to inverters are not counted. Obviously, the literal count depended on how the circuit description is and how the function at each node is represented. For example, if the function at a node is represented in a factored form the literal count will usually be smaller than if the node function is represented in the SOP form. Therefore, when evaluating optimization results with respect to literal counts, great care has to be taken in order to ensure a fair comparison.

#### 4.2.2 Motivation example

Consider an example, Table 4.2, that illustrates how the state assignment can influence the cost of synchronized FSM with four states (S0, S1, S2, S3) and one input  $x$ . The table shows three valid encoding for the FSM (Assign 1,2,3).

Table 4.2. State Table with three State Assignments.

Present state	Next state		Output		Assign #1	Assign #2	Assign #3
	$x=0$	$x=1$	$x=0$	$x=1$			
S0	S0	S1	0	0	00	00	00
S1	S0	S2	0	0	01	10	11
S2	S3	S2	0	0	10	11	01
S3	S0	S1	0	1	11	01	10

- **Assign 1:**

The transition table corresponding to the state assignment 1 is derived in Table 4.3. D flop-flops are used to implement the memory portion. The entries in the transition table represent the next state of D flip-flops for each combination of present state and input value. Karnaugh maps are for each of the flip-flop excitation input and output as shown in Figure 4.1.

Table 4.3. State transition table for assignment 1.

Input $x$	Present State $y_1 \ y_2$	Next State $y_1 \ y_2$	Output $z$
0	0 0	0 0	0
1	0 0	0 1	0
0	0 1	0 0	0
1	0 1	1 0	0
0	1 0	1 1	0
1	1 0	1 0	0
0	1 1	0 0	0
1	1 1	0 1	1



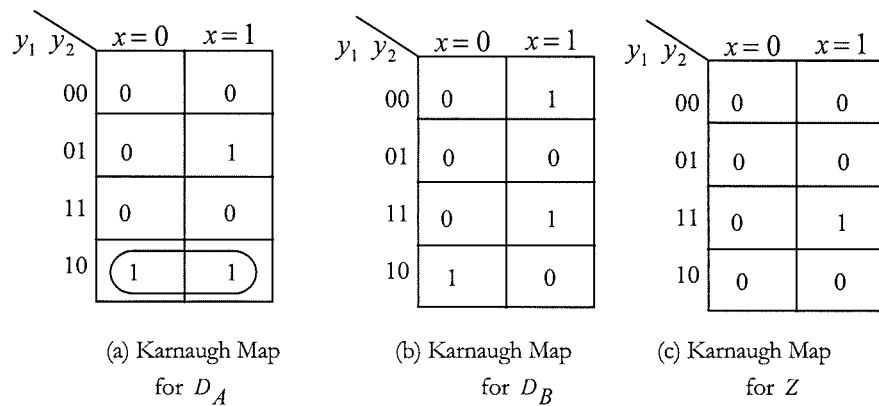


Figure 4.1. Karnaugh maps for D flip-flops realization and output for assignment 1.

Note that for each flip-flops, the D input is made equal to the required next state.

The assignment 1 yields the following equation:

$$D_A = y_1 \bar{y}_2 + x y_1 \bar{y}_2$$

$$D_B = \bar{x} y_1 \bar{y}_2 + x y_1 y_2 + x \bar{y}_1 \bar{y}_2$$

$$Z = x y_1 y_2$$

- **Assign 2:**

The state transition table corresponding to the state assignment 2 derived in Table 4.4.

Table 4.4. State transition table for assignment 2.

Input $x$	Present State $y_1 \ y_2$	Next State $y_1 \ y_2$	Output $z$
0	0 0	0 0	0
1	0 0	1 0	0
0	1 0	0 0	0
1	1 0	1 0	1
0	1 1	0 1	0
1	1 1	1 1	0
0	1 0	0 0	0
1	1 0	1 1	0

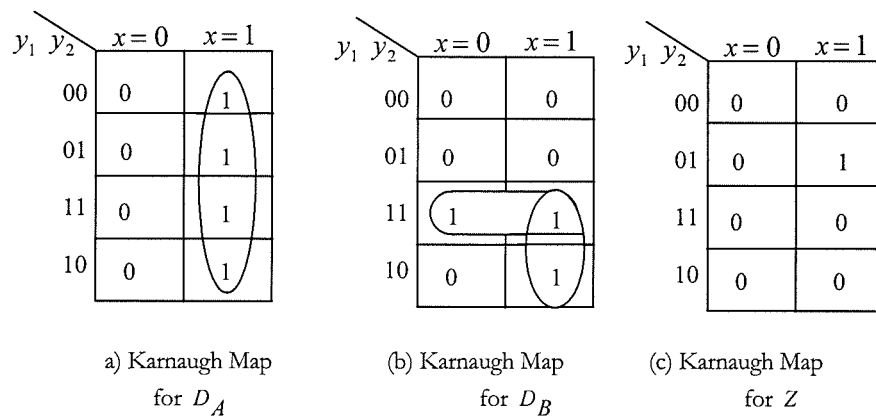


Figure 4.2. Karnaugh maps for D flip-flops realization and output for assignment 2.

The assignment 2 yields the following equation:

$$D_A = x$$

$$D_B = y_1 y_2 + x y_1$$

$$Z = x \bar{y}_1 y_2$$

- **Assign 3:**

The state transition table corresponding to the state assignment 3 derived in Table 4.5.

Table 4.5. State transition table for assignment 3.

Input $x$	Present State $y_1 y_2$	Next State $y_1 y_2$	Output $z$
0	0 0	0 0	0
1	0 0	1 1	0
0	1 1	0 0	0
1	1 1	0 1	0
0	0 1	1 0	0
1	0 1	0 1	0
0	1 0	0 0	0
1	1 0	1 1	1

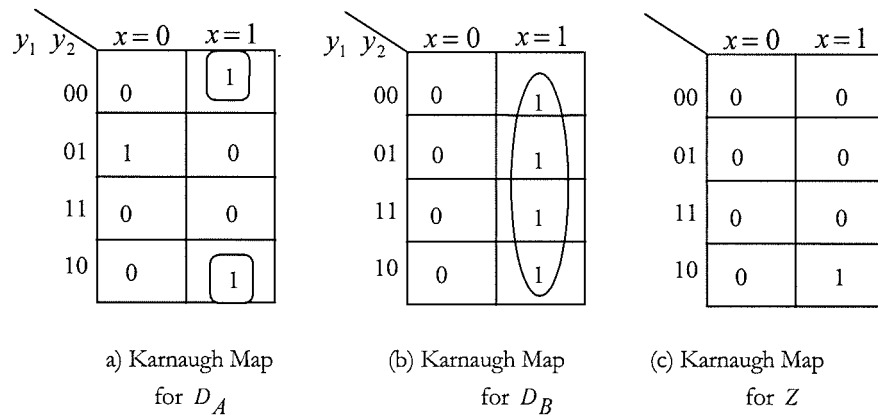


Figure 4.3. Karnaugh maps for D flip-flops realization and output for assignment 3.

The assignment 3 yields the following equation:

$$D_A = x \bar{y}_2 + \bar{x} \bar{y}_1 \bar{y}_2$$

$$D_B = x$$

$$Z = x \bar{y}_1 \bar{y}_2$$

This example illustrates that choosing an appropriate state assignment greatly reduce the cost of implementation. The cost of the circuit is defined here as the number of 2 input AND/OR/NOT gates needed to realize the circuit. However, state encoding that leads to simpler equations and therefore smaller area designs can be selected. This can be seen from Table 4.6, which shows a comparison of the number of gates required to implement the circuit for each of the three assignments.

Table 4.6. Gate comparison for three state assignments.

State assignment	# 2-input AND	# 2-input OR	#NOT	#Total gates
Assign 1	9	3	3	15
Assign 2	4	1	1	6
Assign 3	6	1	3	10

### 4.3 Automated state assignment

In this thesis optimal state assignment algorithm creates assignments that use the minimum number of logic components. A number of algorithms to find good state assignments have been discussed in detail in chapter 2.

The widely used algorithms in this area by researchers so far are:

- In MUSTING [47], JEDI [111] and MUSE [112] a graphic is built, with a vertex corresponding to each state in a FSM and edges connecting every pair of states. Each edge has a weight that measure how desirable it is to assign close codes to this pair of states. The problem called graphic embedding, then reduces to minimization of the weight sum of the states in the graph.
- In Nova [113] and KISS [114] symbolic minimization is used to obtain a set of constraints that the encoding must satisfy. Each uses a different approach for determining the state bit encoding. It is difficult to characterize which is best for a given FSM. A good approach is to try all of them, this possibly because execution time is very small.

Nova, proposed by Villa et al. [113], builds upon the work done on the KISS algorithm. Minimisation is achieved by combining states, which share the same next state and outputs. A number of the approaches depending on certain heuristics, a combination of the algorithms are executed to produce logic minimisation.

### 4.4 Mechanical state assignment

State assignment algorithms that don't attempt an optimal state assignment are usually grouped under the category of mechanical state assignment algorithms. These algorithms are referred to as mechanical because they assign state values based on some type of rule. There are four mechanical state assignment algorithms:

1. One-hot code
2. Random code

3. Gray code
4. Standard Binary sequence

#### **4.5 Previous Application of Genetic Algorithm to State Assignment**

In recent years, considerable research has been undertaken on an application of genetic algorithms to state assignment, particularly Almaini [68] and Amaral et al [69].

In [68] the Algorithm result was compared with NOVA, MUSTANG and SPECTRAL, with the assignment based on closed partitions. This work demonstrated that genetic algorithms are a valid method of finding good state assignments.

In [69] the fitness function used the Hamming distance for evolution. So the problem is reduced to minimizing the weighted sum of the distance of the states in the graph, a classic problem called graph.

There are some similarities between the state assignment and travelling Salesman Problem (TSP)[115]. Just as the goal of the TSP is to find a path that minimizes the travelling distance; the goal of the state assignment is to find an implementation that minimizes the implementation cost. The equivalence of the distance between two cities in the TSP is the connection value of desired adjacent graph arcs in the state assignment. However in the TSP only the distance between adjacent cities in a tour is computed to form the fitness of a given path. In the state assignment problem the connection of each state with all other states must be weighted by the corresponding distance between these states, and then summed to form the fitness. In other words, the TSP can be seen as a special case of the state assignment problem where the Hamming distance is reduced to a binary function whose value is one if the states are adjacent, and zero otherwise.

In the TSP, when the positions of two cities on a tour are swapped, only the connections with their neighbours change. Defining this property as locality,

we observe that the state assignment problem does not have locality: if two-state assignment are swapped, their connection with all other states is affected.

The fitness function used for an m-city TSP is the length of the tour. Due to the symmetry of this function, for each path there are at least  $2m-1$  other paths with same length. In this state assignment problem the symmetry of the fitness function result is  $2^k k!$  equivalent solutions, where K is the number of bits used in the synchronous sequential circuit. This large number of equivalent solution results in a search space with many local minima, Therefore, it's harder to find a good solution for the state assignment problem a for the TSP.

Table 4.7 presents a summary variety of available approaches for the state assignment problem. There is no completely satisfactory manual technique for finding the optimum state assignment. The problem of finding the optimum state assignment is NP-hard. The GA finds good optimal solutions short of complete enumeration and evaluation of all possible assignments [1].

Table 4.7. Summary of available approaches to find optimal state assignment

Approach	Theory	Merits	Comments
Partition Theory Hartmans and Stearns. 1967 [39].	Algebraic techniques are used to decompose the state machine.	The state assignment is based on closed partitions resulting in reduced dependency between the state variables.	Not all machines have close partitions.
Column evaluation approach. Dolota and McCuskey. 1964 [46].	The column of state table are scored with respect to various criteria to influence quality of assignment	This approach gives very good realisation, the result even better than approach in [26].	It is intractable for machine of more than 12 states.

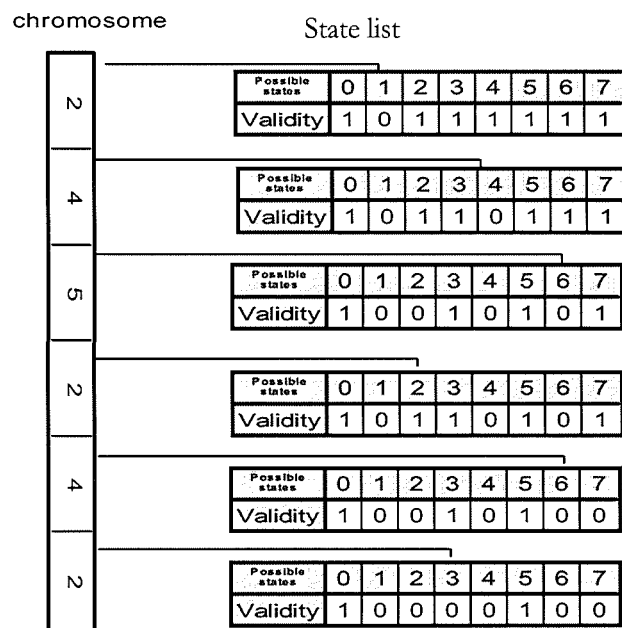
Approach	Theory	Merits	Comments
Enumerative approach of Story 1972 [59].	All possible partitions are evaluated as candidates for assignment separately by calculating the complexities of the corresponding Boolean function.	All partitions are optimally selected and the subset of best partition equally matching is selected for assignment.	The implementation of is slow.
Approach of Moroz 1970 [64].	The graph embedding created directly from the state graph whose edges corresponding to oriented transition between states.	The approach can be treated as approximate solution to quadratic assignment.	It doesn't solve the quadratic assignment but simply edge embedding problem.
Quadratic assignment approach. DeMicheli 1984 [65].	This approach is based on embedding some graphs created from FSM table to hypercube graphs.	This approach permits for realisation of machine up to 100 states.	The approach still can't be applied for machine with more than 100 states, and it doesn't take in to count output states for assignment.
Devadas "Mustang 1988" [47].	Commercial ECAD tools look for optimal state assignment.	Available	For multilevel implementation .
Approach of Villa "NOVA 1990"[113].	Commercial ECAD tools look for optimal state assignment.	Available	For two level implementation only.
"Genetic algorithm approach "Almaini , Amaral, 1995 [68,69] and Chalmeris, 2000 [133].	Using GA to generate optimal state assignment.	All possible assignment for a given problem maybe considered as search space of potential solution.	GA approach requires a set of operators for its implementation . The selection of these operators is crucial for the efficiency of the algorithm.
Proposed approach 2002 [1]	Using evolutionary algorithm to optimise and design sequential circuits.	Combination of both GA and EHW for sequential logic circuit.	

#### 4.6 Genetic Algorithms for state assignment problem.

This section describes in details how GA techniques are used to find efficient state assignment capable of minimizing the component count for a state machine.

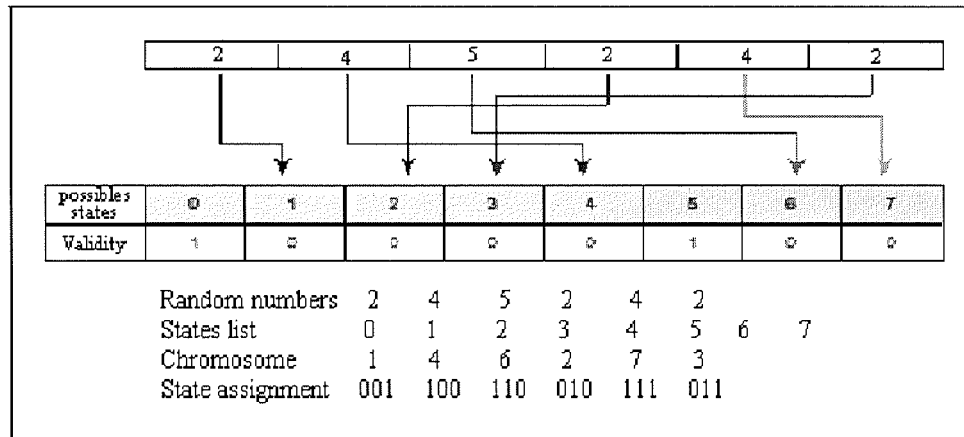
##### 4.6.1 Chromosome representation for GA State assignment problem

A finite state machine can be described by STT or state transition graph (STG). When implementing finite state machines, they are commonly represented as two-dimensional tables with a number of rows equal to the number of states and a number of columns equal to the size of the input alphabet. The intersection between a state and an input alphabet contains the next-state transition and the output symbol, if any.



(a) Chromosome representation





(b) An example coded assignment

Figure 4.4. (a, b) Chromosome representation.

The chromosome contains the encoded assignment for the internal state of the sequential machine. The length of the chromosome is equal to the number of the states used for the sequential machine. The initial population is randomly generated. The duplicate chromosomes are discarded. In order to encode the actual information, the FSM is represented as a list of  $n$  states; the  $i$ th element of the list is a number in the range from 1 to  $(2^b - i + 1)$ . Consider the example in Figure 4.4(a) where the genotype of the chromosome has been generated randomly. The genotype of a problem is represented by array of integers. The figure 4.4(b) shows how a six state sequential machine is encoded.

According to the Figure, the method can be summarised as follows:

- Random function is used to generate six integers (2,4,5,2,4,2).
- The states represented as list (0,1,2,3,4,5,6,7); the list starts with zero and contains all possible assignments for states of an FSM using minimum length code, i.e.  $b = \lceil \log_2 n \rceil$  bits to encode the set of  $n$  states; otherwise  $b$  is the integer that is greater than or equal to  $\log_2 n$  (or nearest higher integer). The content of the state list represents state assignments. The algorithm works through the status

validity table initially set to one. The numbers are counted from left to right.

The procedure is interpreted as follows: (a) The first random number is 2, take the second number from the possible state list 1 as first code for the initial state and set validity (0), so it will not be used or counted for future selections; (b) The next random number is 4, take the fourth number from the possible state list (4), and remove it from the list by setting validity (0); (c) The next random number is 5, take the fifth number from the possible state list (6) and set validity (0).

The procedure continues in the same way for the remaining numbers in the list. It can be seen from the figure that the random number 2,4,5,2,4,2 would map the states 0,1,2,3,4,5,6,7 to the assignment 1, 4, 6, 2, 7, 3 respectively to assign a unique code to each state. This method is applied to generate randomly the initial generation.

#### 4.6.2 Fitness function

The cost function typically incorporates (area, speed, power, testability) as the main objectives optimization procedure of logic synthesis tools. We will assume that the minimum area is the primary objective of the optimization procedure.

The number of 2-input AND/OR and NOT gates defines the fitness function, AND/OR logic gates that are used in the logic equations after being minimised using ESPRESSO logic minimisation [66]. The extent of fitness to the environment is evaluated by the following function:  $w_n G_n + w_a G_a + w_o G_o$ , where functions  $G_n$ ,  $G_a$  and  $G_o$  are the numbers of NOT, AND and OR gates, and  $w_n$ ,  $w_a$  and  $w_o$  are weight where the chromosome is realised as a circuit with NOT, AND/OR gates.

A selection mechanism is necessary to select the individual that will generate offspring, and also to select the individual that will survive the next generation. In this task the roulette wheel method was chosen. In the method presented in [25], the probability of selecting a given individual is given by its

fitness divided by the length of the roulette wheel, where the length of the wheel is the sum of the fitness of all individual.

For example the total fitness equal 100. The percentage of population total fitness is also shown in Figure 4.5. For this example the chromosome 1 had a fitness value of 30, which represented 0.3 of the total fitness. As a result, chromosome 1 is given 0.3 of the based roulette wheel, and each spin turned up chromosome 1 with probability 0.3. Each time we required another offspring, a simple spin of the weight roulette wheel the reproduction candidate. In this way more highly fit chromosome have higher number of offspring in the succeeding generation. Once the chromosome has been selected for reproduction, an exact replica of the chromosome is made.

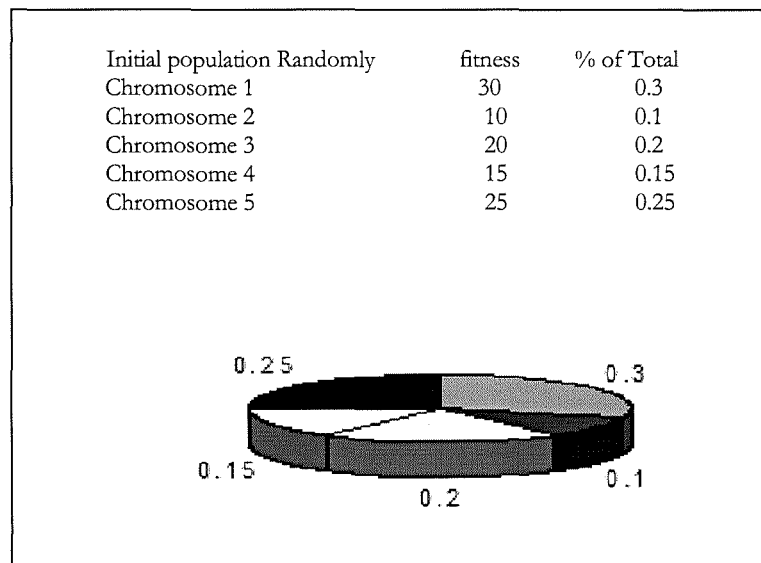


Figure 4.5. Roulette wheel parent selection.

#### 4.7 Genetic operators

Once gate count has been carried out for every chromosome, fitness values are assigned to each individual in the population. Roulette wheel selection is used to select the chromosomes from the previous population. Once the new generation is created, the recombination operations are applied. In this case, the two-point crossover operation is applied. A crossover operator that randomly selects two crossover points within a chromosome then interchanges the two chromosome genes between these points to produce two new offspring. The “|” symbol indicate the randomly chosen crossover point. This is illustrated in Figure 4.6:

Before Crossover	After Crossover
Chromosome 1= 2 1   3 5   6	Offspring1= 2 1   7 3   6
Chromosome 2= 4 2   7 3   1	Offspring2= 4 2   3 5   1

Figure 4.6. The crossover operation. Two offspring are produced from a pair of parents.

The mutation operation chosen was based on the interchange of two genes (states) in each chromosome.

The mutation rate controls the number of operations to be applied and in this way controls the amount of random information introduced into an individual. For example, Table 4.8 shows two swapping operations are performed during mutation. The first one swap the state with assignments 6 and 2, changing assignment of state S1 and S2, the second swapping is between the pattern 3 and 7, change assignment of state S3 and S5.

All solutions are reachable by this operation because given an assignment a finite number of single swapping operations can transform it to any other assignment in the solution space. The mutation rate was variable and increased with each generation if there had been no improvement in the gate count of the best chromosome.

Table 4.8. Example state mutation

State	Before mutation		After mutation	
S0	1	001	1	001
S1	<b>6</b>	<b>110</b>	2	<b>010</b>
S2	2	<b>010</b>	<b>6</b>	<b>110</b>
S3	3	011	7	111
S4	0	000	0	000
S5	7	111	3	011

Figure 4.7 shows the fitness values for different assignments. For example if the best chromosome corresponding to the assignment A. then the genetic algorithm will now search the local area for better solution by applying crossover. The best solution is stored in the first chromosome, so that moving towards other areas loses nothing.

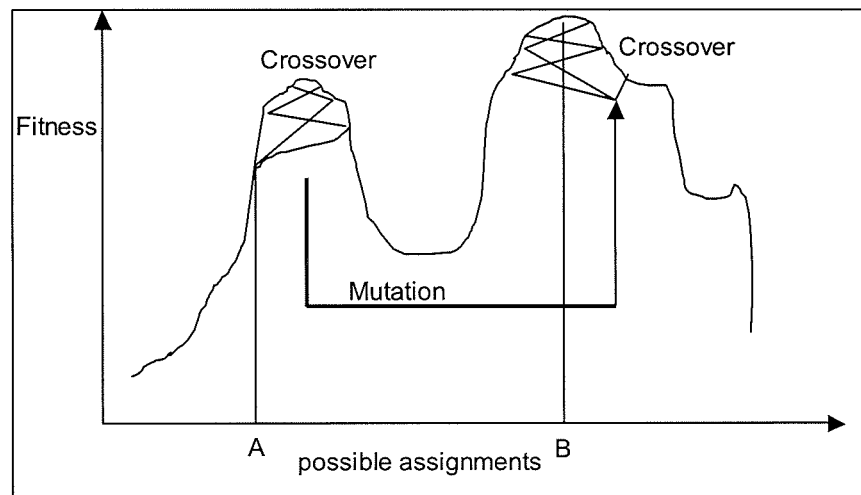


Figure 4.7. Effect of Crossover and mutation on chromosome Fitness.

Once the local maximum is found the population will converge. Convergence of the population causes an increase in mutation rate and the population may find away out of the local maxima. After finding a better solution then the previous local maximum representing the genetic algorithm will a gain search that area and may end up with assignment B. During the search of the local area the mutation rate is dropped to its lowest possible value.

Finally, the result of the application of this operator is always a valid assignment. Selecting the GA parameter values for optimal state assignments will be discussed in section 4.8.

The algorithm used is described as:

*Step 1.* Create *num\_chromosomes* randomly generated chromosomes  $c_i$ ;

*Step 2.* Evaluate the fitness of each chromosome  $f(c_i)$ ;

*Step 3.* *best\_chromosome* =  $c_i$  such that  $f(c_i) = \max \{f(c_j)\}$ ;

*Step 4* *No\_generations* = 1;

*Step 5* *num\_children* (*percent breeding* \* *num\_chromosomes*);

*Step 6* While (*max\_no\_generations* not exceeded) do  
begin  
select *num\_children* parent chromosomes by windowed roulette wheel; apply two-point crossover to create *num\_children* new chromosomes;  
replace *num\_children* parents with new chromosomes (children);  
mutate *percent\_mutation* genes of total genes;  
replace randomly selected chromosome with *best\_chromosome* (elitism); evaluate the fitness of new chromosomes  $f(c_j)$ ;  
if ( $f(c_j) > \max \{f(c_i)\}$ )  
*best\_chromosome* =  $c_j$ ;  
*no\_generations* = *no\_generations* + 1;  
end;

However, when creating a new population by crossover & mutation, the best chromosome might be lost. Hence, Elitism is utilised to copy the best chromosomes to the new population. Elitism rapidly improves the performance of the GA, by preventing the loss of the best-found solutions.

Several parameters control the way the GA optimises the state assignment of the FSM, allowing the users to vary their value. The parameters are:

- The population size of the genetic algorithm;
- The number of generations of the GA around the main loop;
- The initial number of runs of the GA to perform the optimisation;
- The probabilities of crossover rate ( $P_c$ ) and mutation rate ( $P_m$ ).

#### **4.8 Experimental results**

This chapter discusses several major points. It shows how GA parameters affect the results of the state assignment. The results obtained on the MCNC benchmark set of FSMs [116] are presented for different number of generation, population size, probability of crossover and probability of mutation. In the following subsections more results obtained by the genetic algorithm are compared with the results obtained by NOVA algorithm and other heuristic technique.

All the results given in this chapter are implemented in C on PC 450MHZ Pentium-III with 128Mb of RAM. The program seeks good assignment for the state variables. The users are required to input several parameters for the GA before running the program.

For a given state table, which is to be realized as a clocked sequential circuit, attempt is made to minimize the number of AND/OR gates necessary in the combinational part of the circuit under the following assumptions:

- No internal state merging will be attempted.
- The cost of the circuit is defined as the number of AND/OR gates needed to realize the circuit.
- The program attempts to find good assignment for internal state variable of a sequential machine within a reasonable time but is not guaranteed to find the best solution.

The following results show the performance, in terms of the number of two input (AND/OR) gates required, for the various state assignment algorithms considered.

#### *4.8.1 Parameter Effects*

This section covers two issues. Firstly, experiments were conducted to understand the effect of GA parameters before a suitable set is adopted.

Secondly, for the set of parameters found, several benchmark circuits are synthesised for different areas: number of AND/OR /NOT gates and number of literals.

#### **Benchmark Set**

A subset of the MCMN benchmark set given in Table 4.9 is used in this evaluation. The table shows the benchmarks used, together with their number of input/ output/states.



Table 4.9. Statistics of benchmark examples

Benchmark	# Input	# Output	# States
Bbara	4	2	10
Bbtas	2	2	6
dk15	3	5	4
dk16	2	3	27
dk27	1	2	7
dk512	1	3	14
donefile	2	1	24
Lion9	2	1	9
modulo12	1	1	12
S1	8	6	20
shiftreg	1	1	8
train11	2	2	11
tav	4	4	4

#### 4.8.2 Parameter space

Several parameters control the way GA operates a state assignment of the FSM, allowing the user to vary their value. The parameters that must be chosen are: the probabilities of crossover rate ( $P_c$ ) and mutation rate ( $P_m$ ), the population size, the number of generations of the GA around the main loop and the initial numbers of runs of the GA to perform the optimisation.

We cannot choose the control parameters until we consider the interaction between the genetic operators. Because they cannot be determine independently.

Therefore in choose these parameters, it was decided to search the space parameter by parameter except for crossover and mutation probability, which

are considered together. For each parameter chosen, the parameter is fixed, and the process continues for the next parameter.

#### *4.8.3 Crossover and mutation*

Crossover and mutation probabilities are dealt with as a single unit, as they are similar and each has a significant effect on a good choice of the other.

A too-small probability of crossover could be partially compensated for by too large probability of mutation. A too large probability of mutation is likely to mask the effect of crossover. Again, there is computation time problem with searching a large space for good parameters.

Firstly, the probable range for values were decided to be  $0.1 \leq P_c$ , as below 0.1 there is little chance of any change from generation to generation. Also  $0.001 \leq P_m$ , as below there is a little chance of having any effect.

Three benchmarks were used, modulo12, dk512 and dk16 from the MCNC benchmark set. It was important to choose three significantly different benchmarks. “Module12” number of state =12, while “dk512” number of state=14 and “dk16” number of state=27. As expected with any optimization method based on genetic algorithm, it is possible to choose  $P_c$  or  $P_m$  either too high or too low. If  $P_c$  or  $P_m$  are too high, not enough good chromosomes are retained from generation to generation.

Table 4.10. Compression crossover and mutation.

Benchmark Examples	$P_c$	$P_m$	No. Gates	No. literals
modulo12	0.10	0.08	27	31
	0.20	0.01	21	29
	0.40	0.01	22	25
	0.60	0.025	25	24
	0.80	0.05	24	31
dk512	0.10	0.08	48	52
	0.20	0.01	43	47
	0.40	0.01	50	54
	0.60	0.025	44	48
	0.80	0.05	60	47
dk16	0.10	0.08	745	622
	0.20	0.01	479	501
	0.40	0.01	538	522
	0.60	0.025	549	601
	0.80	0.05	773	650

Table 4.10 shows the effect of  $P_c$  and  $P_m$  on the number of gates. Figure 4.5 shows the plot of different probability of crossover  $P_c$  with the number of gates. The plot of probability of mutation  $P_m$  and the number of gates is shown in figure 4.8. However if  $P_c$  or  $P_m$  are too low, few changes are made from generation to generation, leading to static population. As can be seen in Figures 4.8 and 4.9, it was decided to choose  $P_c = 0.25$  and  $P_m = 0.01$ .

The results plotted in Figure 4.12 and 4.13 for modulo12, dk512 and dk16 show different probability of crossover  $P_c$  and mutation  $P_m$  with the number of literals.

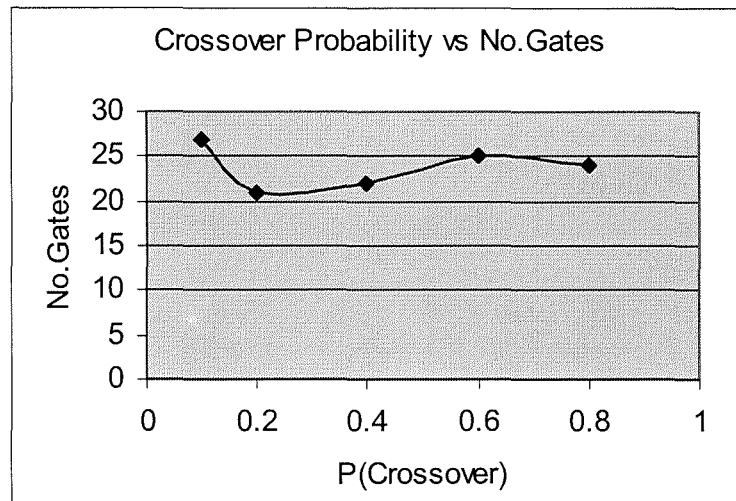


Figure 4.8 Effect of P (Crossover) for example modulo12.

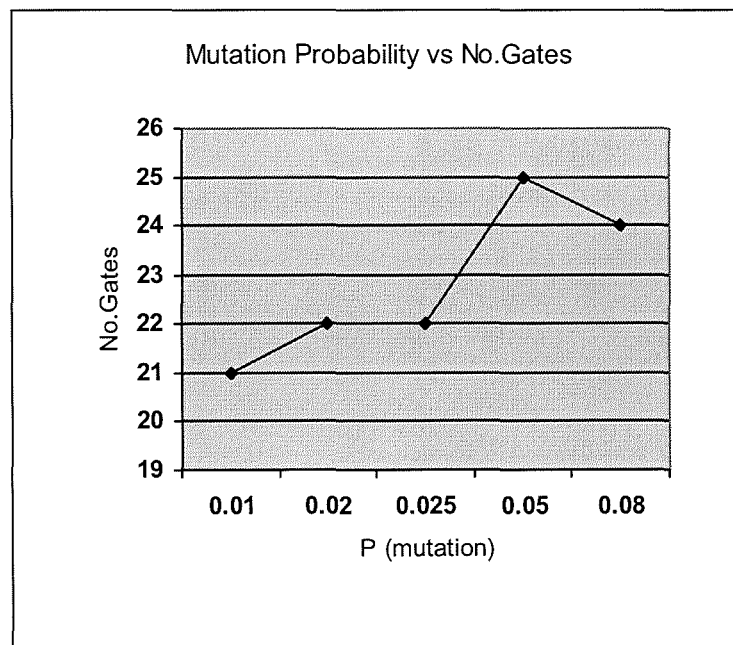


Figure 4.9. Effect of P (Mutation) for example modulo12.

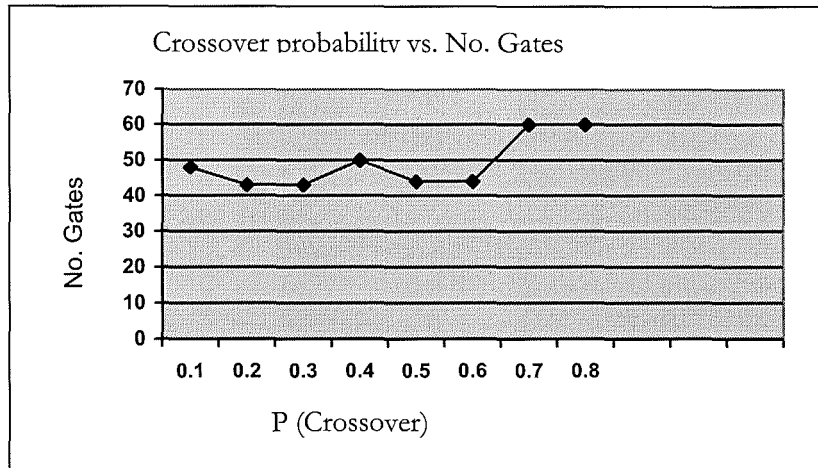


Figure 4.10. Effect of P (Crossover) for example dk512.

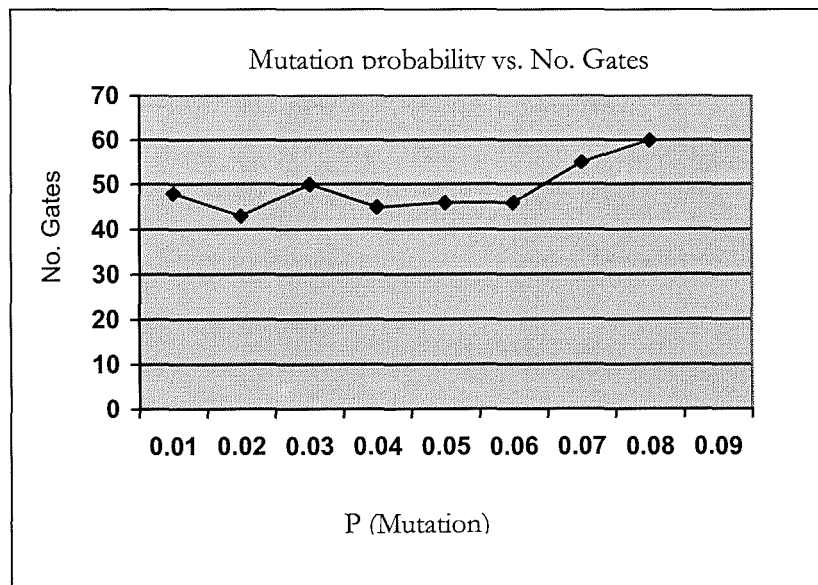


Figure 4.11. Effect of P (Mutation) for example dk512.

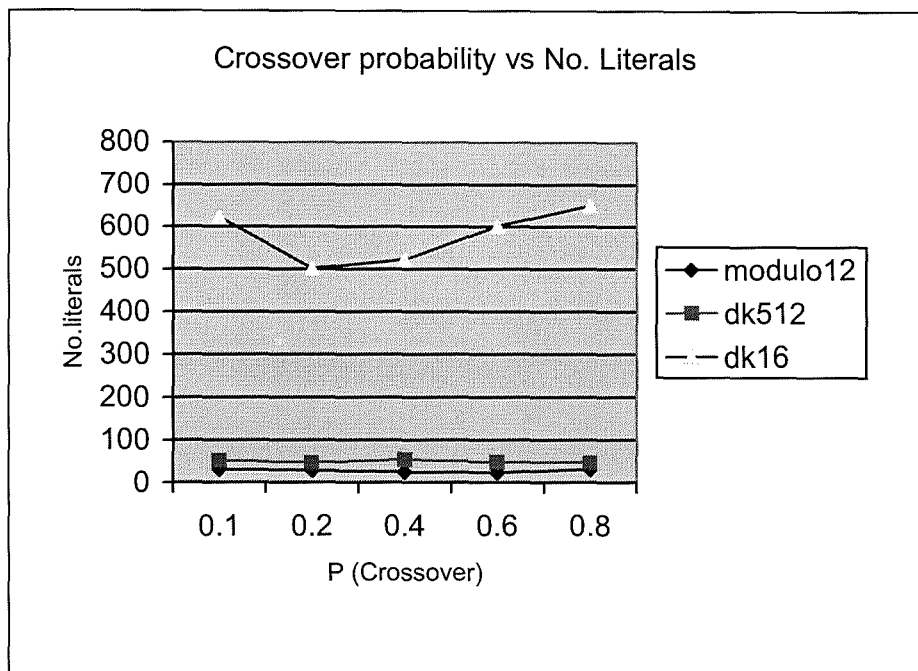


Figure 4.12. Effect of P (Crossover) for examples modulo12, dk512 and dk16.

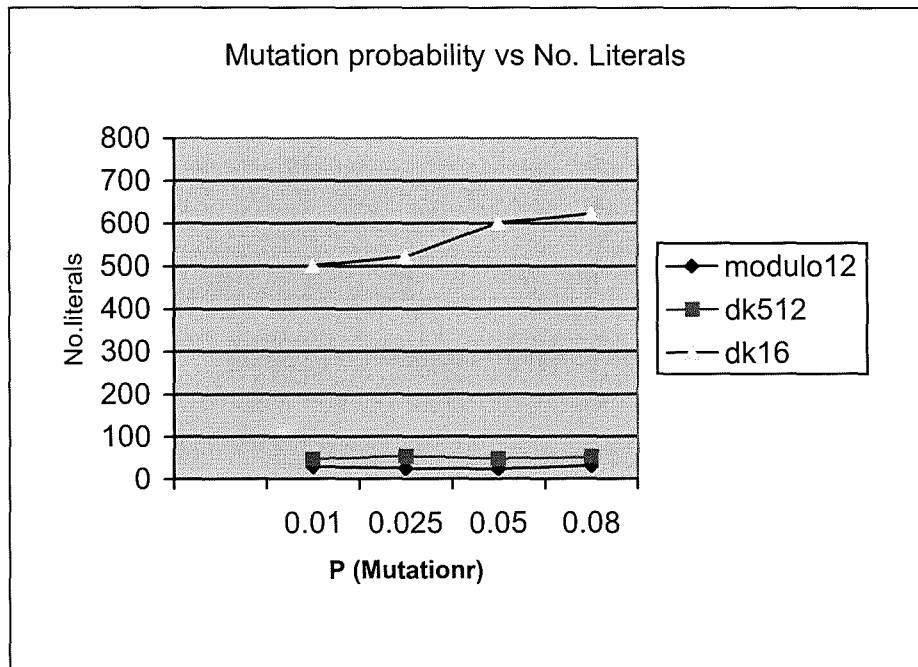


Figure 4.13. Effect of P (Mutation) for examples modulo12, dk512 and dk16.

#### 4.8.4 Population Size

The probability of crossover and mutation were then fixed at the values given above, and attention was turned to discovering a good value for population size. Increasing in the population size increase its diversity and reduces the probability that the GA will prematurely converge to a local optimum, but it also increases the time required for the population to converge to the optimal regions in the search space. This is a one-dimensional problem, requiring fewer execution runs, and therefore it was possible to use benchmarks which proved a good cross-section of the benchmarks, as seen in the benchmark table. The benchmark circuits modulo12, dk512 and dk16 were used and run with different population sizes. The population size was varied from an initial setting (15) by doubling and halving to explore the value around 15. Once results were obtain for 15 and 30, it was clear that this range was nearly large enough, and so the point 5,10 and 50 were added as in Table 4.11.

The aggregation of results has been treated in the same way as in the previous section. Also plotted are the numbers of gates (No. Gates) Vs Population sizes in figure 4.14 and the numbers of literals (No. Literals) Vs Population plotted in figure 4.15. The plots shows that increasing the population size will, on average, lead to decreased number of gates.

However an increase in the population size will increase the CPU time of the program. So, we decide to fix the defaults Population size =20.

Table 4.11. Population size numbers of gates and number of literal

Benchmark Example	Population_ size	No. Gates	No. Literals
modulo12	5	30	34
	15	23	25
	20	24	28
	30	23	27
	50	24	28
dk512	5	43	55
	15	39	47
	20	32	43
	30	41	36
	50	35	45
dk16	5	549	601
	15	538	522
	20	479	501
	30	477	495
	50	475	504



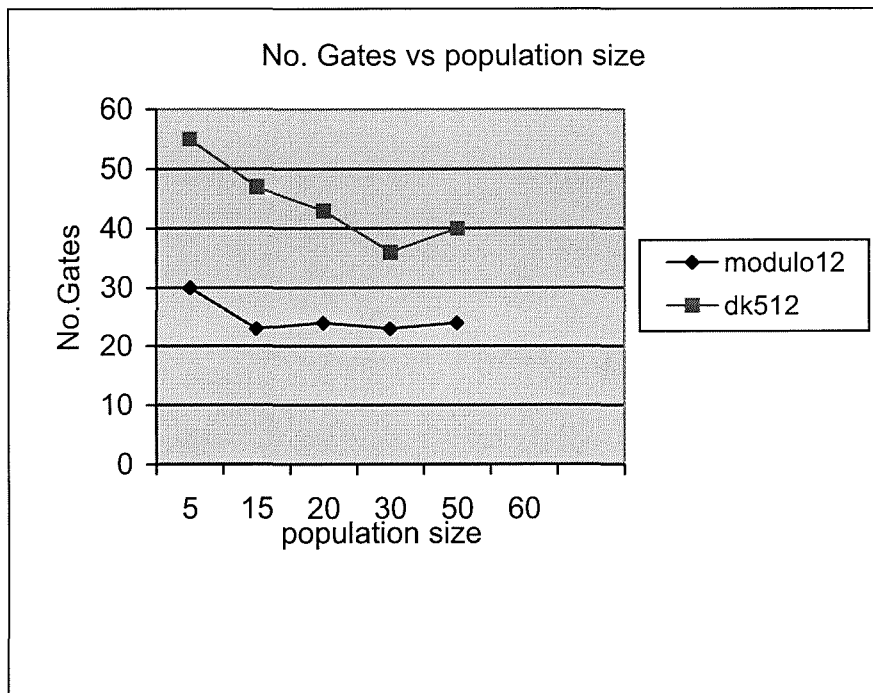


Figure 4.14. Effect of Population Size.

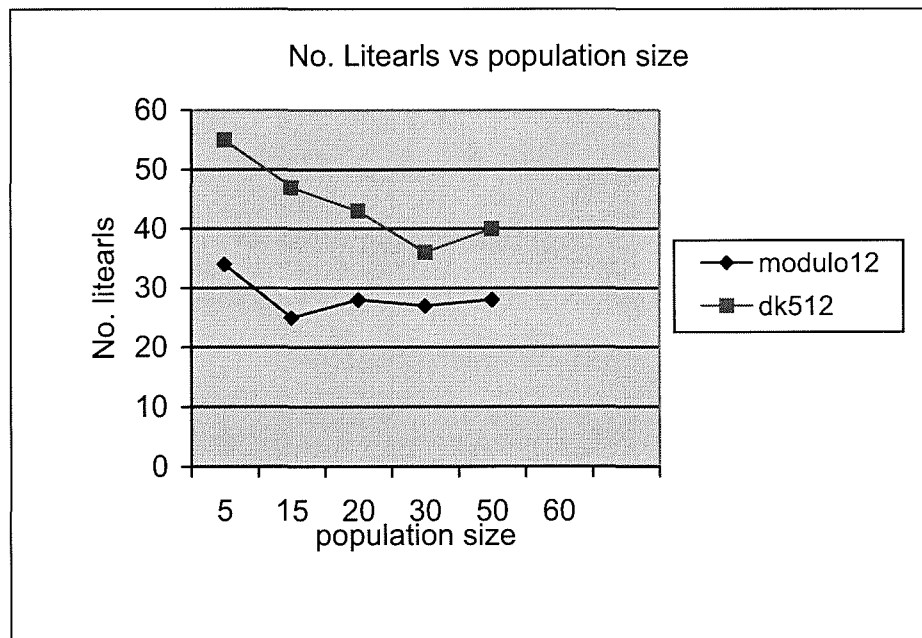


Figure 4.15. Effect of Population Size

#### 4.8.5 GAs Generation

In this section, we compare the effect of the number of generations on three benchmarks with the crossover and mutation rate fixed. The results obtained are tabulated and plotted as shown in figure 4.16. In the table 4.12, number of generations was varied from 10 to 500. It was clear this range is large enough to see the effect of this parameter on the GA.

We can see from the plot Figures 4.16 and 4.17, that the best results occur when number of generation  $>100$  for modulo12 and dk15. After this range, there is no significant effect when the number of generation increases. Figures 4.18 and 4.19 shows the affect of increasing the number of generations on the number of gates and literals. However when the number of generation is small this allows the algorithm to be run within a reasonable period of time.

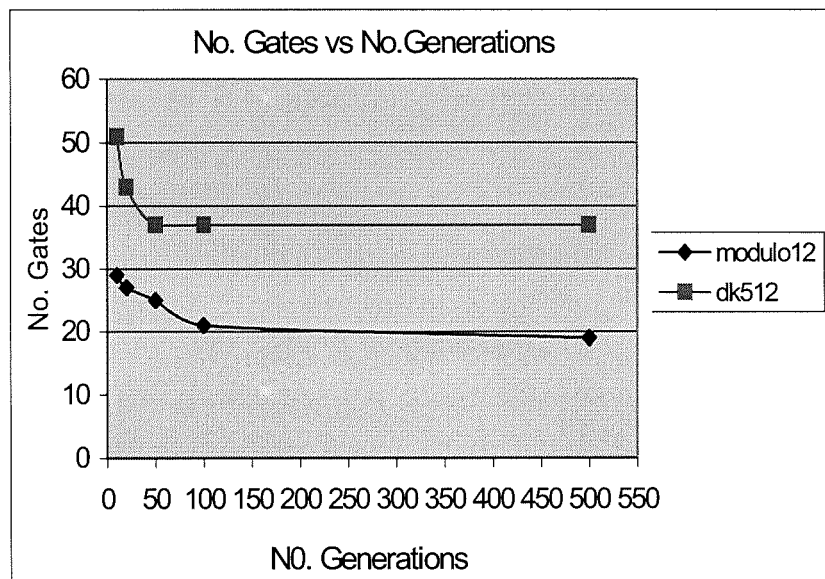


Figure 4.16. Variation of No. Generations with No. Gates.

Table 4.12. Result number of generation.

Benchmark examples	N0.Generation	No. Gates	No. Literals
modulo12	10	29	33
	20	27	31
	50	25	27
	100	21	25
	500	19	23
dk512	10	51	57
	20	43	49
	50	37	40
	100	37	51
	500	37	41
donfile	10	425	429
	20	435	430
	50	416	421
	100	340	435
	500	316	421
dk16	10	549	601
	20	538	522
	50	479	501
	100	455	393
	500	397	365

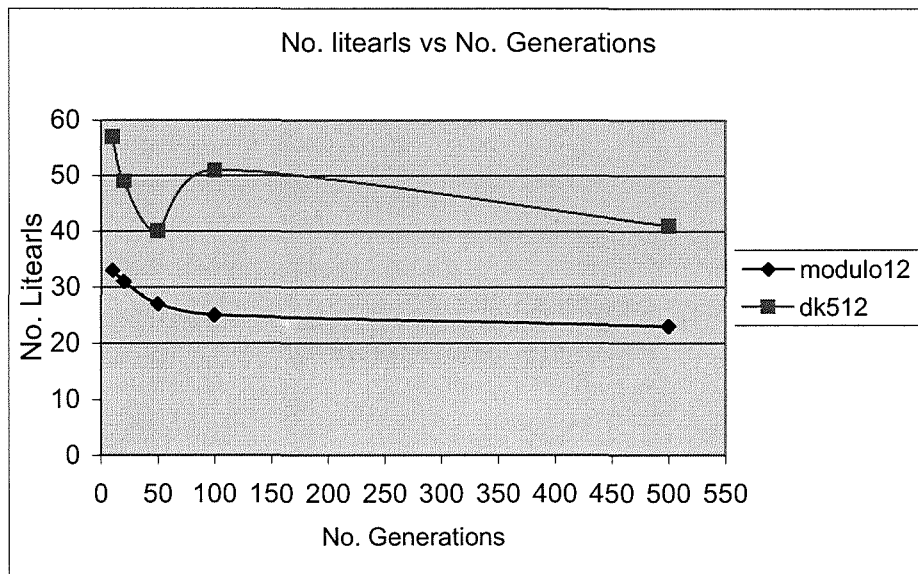


Figure 4. 17. Variation of No. Generations with No. liearls.

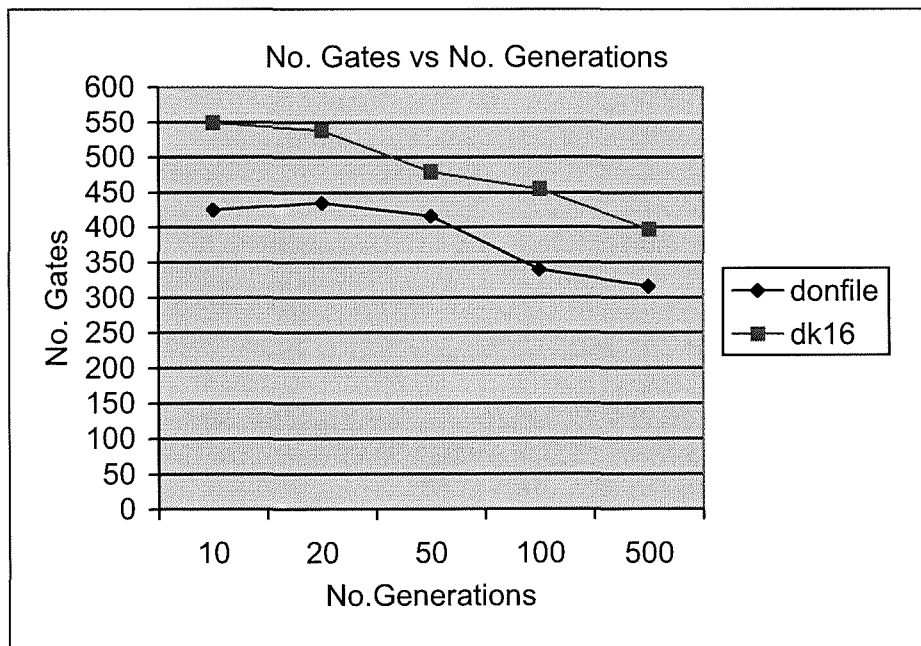


Figure 4.18. Variation of No. Gates with No. Generations.

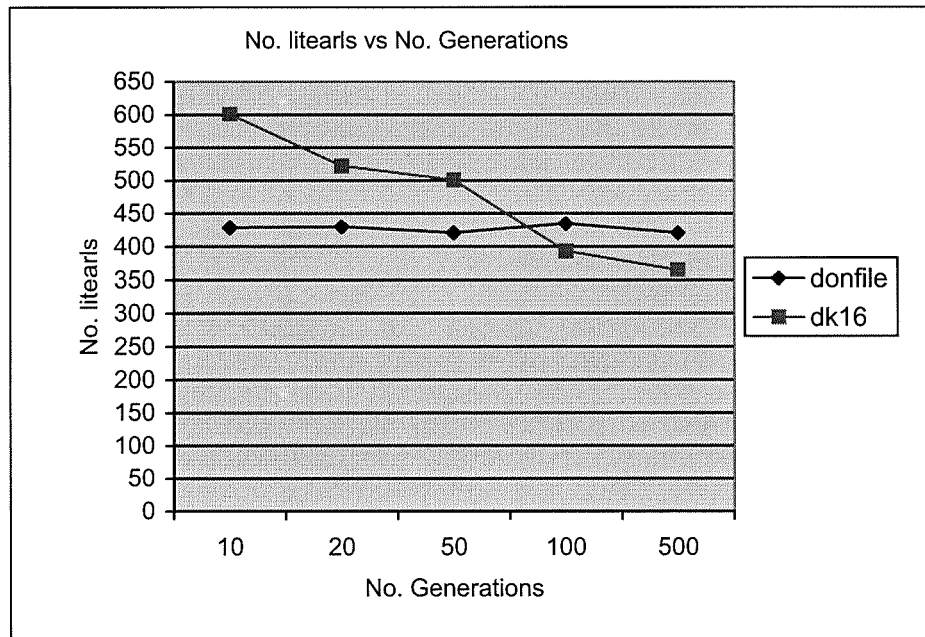


Figure 4.19. Variation of No. literals with No. Generations

From tabulated and graphical results it was decided to choose the number of generation=100. This gave us better results in less time.

We can see the function of GAs as a balanced combination of exploration of new regions in the search space and exploitation of already sampled regions. This balance critically controls the performance of GA. However, the performance of the GA is determined by the choice of control parameters as was discussed in the previous result.

Here the trade-offs that arise are pointed out:

- Increasing the crossover probability increases recombination of building blocks, but it also increases the disruption of good strings.
- Increasing the mutation probability tends to transform the genetic search into a random search, but it also helps reintroduce lost genetic material.

- Increasing the population size increases its diversity and reduces the probability that the GA will prematurely converge to a local optimum, but also increase the time required for the population to converge to the optimal region in the search space. So we can't choose control parameter until we consider the interaction between the genetic operators. The choice of control parameter itself can be a complex non-linear optimization problem and the choice of optimal parameter largely remains an open issue.

#### 4.9 Comparison of results

The second set of experiments aims to compare the GA results with NOVA using the system “SIS” disturbed by the department of EECS, UC Berkeley [73]. These comparisons were based on the number of literals for sum of products form and number of the gates.

The genetic algorithm with population size of 20 was demonstrated for the probability of crossover is 0.25 and the probability of mutation is 0.01. The number of generations over which GA was run was 100. However, it was observed that, for FSMs with less than 15 states, the number of generations required for GA to operate effectively was lower somewhere between 50 and 100. For the MCNC FSMs benchmark with number of states more than 15, the number of generations was over 100 and it took less than 30 minutes of CPU time to run. The results obtained from the GAs are compared with Nova in terms of two inputs gates required in Table 4.13 and with number of literals in Table 4.14. These results are plotted in Figures 4.20 and 4.21.

Table 4.13. Comparison of GAs with Nova based on the number of gates.

Benchmarks Examples	Number of states	Genetic algorithm	Nova [113]
Bbara	10	60	79
Bbtas	6	15	31
Dk15	4	20	21
Dk16	27	351	379
Dk27	7	10	19
Dk512	14	39	53
Donefile	24	186	207
Lion9	9	22	38
modulo12	12	27	31
Swma1	8	6	17
Swma2	8	4	12
Swma3	16	94	129
Table1	16	42	84
Table2	6	21	37
Table3	6	26	36
train11	11	38	45
Tav	4	20	25
<b>Total Number of gates</b>		<b>981</b>	<b>1243</b>

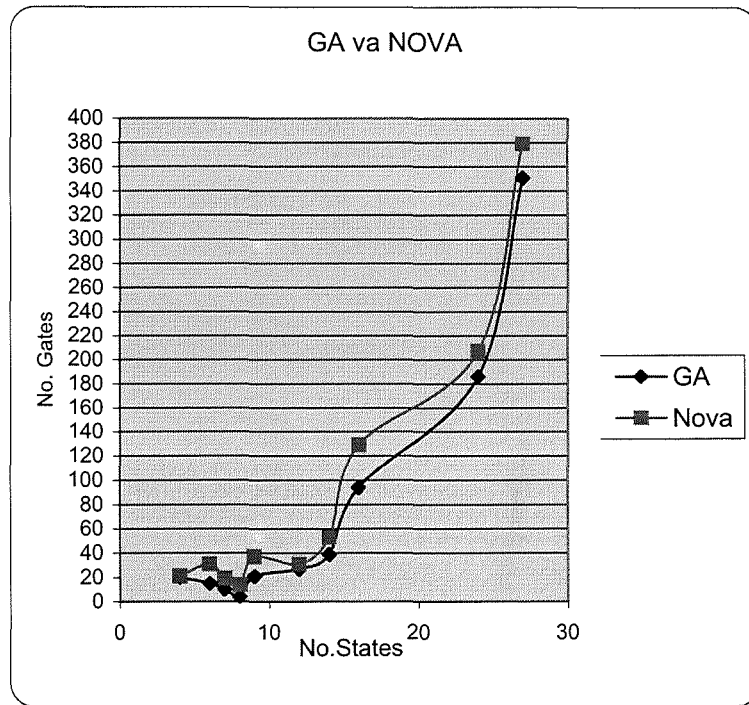


Figure 4. 20. Comparison of GAs with Nova based on number of gates.

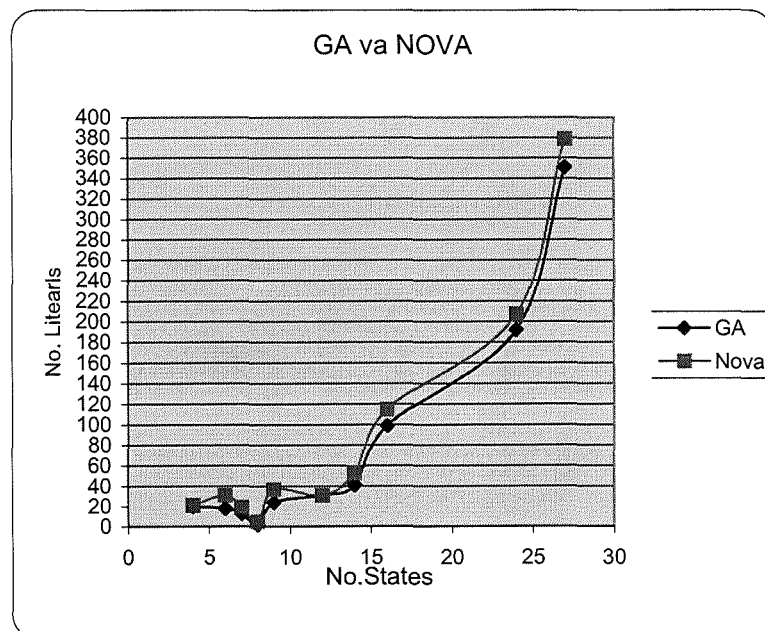


Figure 4. 21. Comparison of GAs with Nova based on number of Literals.



Table 4.14. Comparison of GA with Nova based on the number of literal

Benchmarks Examples	# States	Genetic algorithm	Nova [113]
Bbara	10	60	79
Bbtas	6	19	31
dk15	4	22	21
dk16	27	351	397
dk27	7	13	19
dk512	14	41	53
Donefile	24	192	207
Lion9	4	14	17
modulo12	12	31	31
Shiftreg	8	3	18
Swma1	8	11	17
Swma2	8	7	14
Swma3	16	99	115
Table1	16	46	75
Table2	6	24	37
Table3	6	24	36
train11	11	54	67
Tav	4	32	38
<b>Total Number of Literals</b>		<b>1043</b>	<b>1272</b>

The main observation to make from these tables is that, the genetic algorithm produces superior solutions to the state assignment problem compared with NOVA. The GA approach and NOVA results are presented for many benchmarks. In all benchmark tested in Table 4.13 the GA assignment produce better results compared to NOVA. Experimental results For the 17 benchmark tested showed that the GA could generate state assignments,

which required on average 15.44% fewer gates and 13.47% fewer literals compared with NOVA. After state assignment generated, area should then be the main concern. The network was mapped in to a library with NAND, NOR and NOT gates using SIS [73]. The area is estimated as the sum of gates area.

The following SIS script are used for area testing:

```
read_kiss benchmark.kiss2           #read FSM description

state_assign PROGRAM <OPTIONS>      #state assignment

extract_seq_dc                      #constrict logic network

source script.rugged;               #area minimizing logic reduction

source script.map_area_lib2         #area minimizing map to lib2

print_map_stats                     #print mapped area
```

This script performs the test state assignment on the benchmark and then lets SIS perform the logic minimization. A logic network is constricted from the state assignment and then minimized by running the script Rugged to produce the smallest possible circuit. The logic network is mapped to a gate representation, and then characterised about design area calculated and printed. Comparative results are presented in Table 4.15. The actual assignments are presented in Table 4.16. Each assignment is presented as tables of decimal numbers that represent the binary code of the states of the benchmark.

The tables show that the area-based state assignment running the algorithm of NOVA and the GA state assignment

Table 4.15. Results for some Benchmark circuits

Benchmark Examples	Area (GA)	Area (Nova)
Bbara	340	528
Bbtas	117	120
dk14	437	500
dk15	327	289
dk16	1336	1118
Donfile	858	984
modle12	156	184
Shifreg	101	132

Table 4. 16. State assignments for each algorithm.

<i>Benchmark</i>	<i>GA</i>	<i>NOVA</i>
<i>Bbara</i>	0-6-2-14-4-5-13-7-3-1	9-0-2-13-3-8-15-5-4-1
<i>Bbtas</i>	0-4-10-5-12-13-11-14-15-8-9-2-6- 7-3-1	2-3-6-15-1-13-7-8-12-4-9-0-5- 10-11-14
<i>dk14</i>	5-7-1-3-6-0-4	1-4-0-2-7-5-3
<i>dk15</i>	0-3-5-4	0-2-4-1
<i>dk16</i>	12-8-1-2-7-1-3-2-8-1-4-29-0-16- 26-9-2-4-3-10-11-17-24-5-18-7- 21-25-6-20-19	12-7-1-3-4-10-23-24-5-27-15- 16-11-6-0-20-31-2-13-25-21- 14-18-19-30-17-22
<i>donfile</i>	0-12-9-1-6-7-2-14-11-17-20-23-8- 15-10-16-21-19-4-5-22-18-13-3	12-14-13-5-23-7-15-31-10-8- 29-25-28-6-3-2-4-0-30-21-9- 17-12-1
<i>modulo12</i>	0-8-1-2-3-9-10-4-11-12-5-6	0-15-1-14-2-13-3-12-4-11-5- 10
<i>Shitreg</i>	0-2-5-7-4-6-1-3	0-4-2-6-3-7-1-5

#### 4.10 Comparison of GA algorithm with different state assignment approach

In order to verify the behaviour of the genetic algorithm, it was applied to six examples from the MCNC benchmark set. The results from these machines were compared to different methods, applied to state assignment problem, closed partition method Almaini [68], codable column method [32] and MUSTANG [47] in Table 4.17. In addition the results from genetic algorithm are tabulated against mechanical state assignment in Table 4.18. The table shows that the number of two input AND/OR gates using fixed GAs parameter value as indicated in section 4.8, produced minimal logic implementation than those produced by conventional methods. The comparisons results presented are plotted in Figure 4.22 and 4.23. From the figures, it can be seen that GA produce results comparable to those obtained by the heuristic algorithms.

Table 4.17. Comparison of GA state assignment with optimal algorithms.

Kiss2 file	#States	Genetic algorithm	Codable-Column	Partitioning	Mustang
Bbtas	6	15	35	--	25
dk16	27	351	--	--	455
dk27	7	10	30	16	19
dk512	15	39	82	72	86
Donfile	24	186	--	--	293
Modulo12	12	27	28	16	37

Table 4.18. Comparison of GAs state assignment with mechanical algorithms.

Kiss2 file	#States	Genetic algorithm	Random search	Standard Binary
Bbtas	6	15	21	39
dk16	27	351	812	743
dk27	7	10	12	13
dk512	15	39	57	94
Donfile	24	186	596	293
Modulo12	12	27	21	28

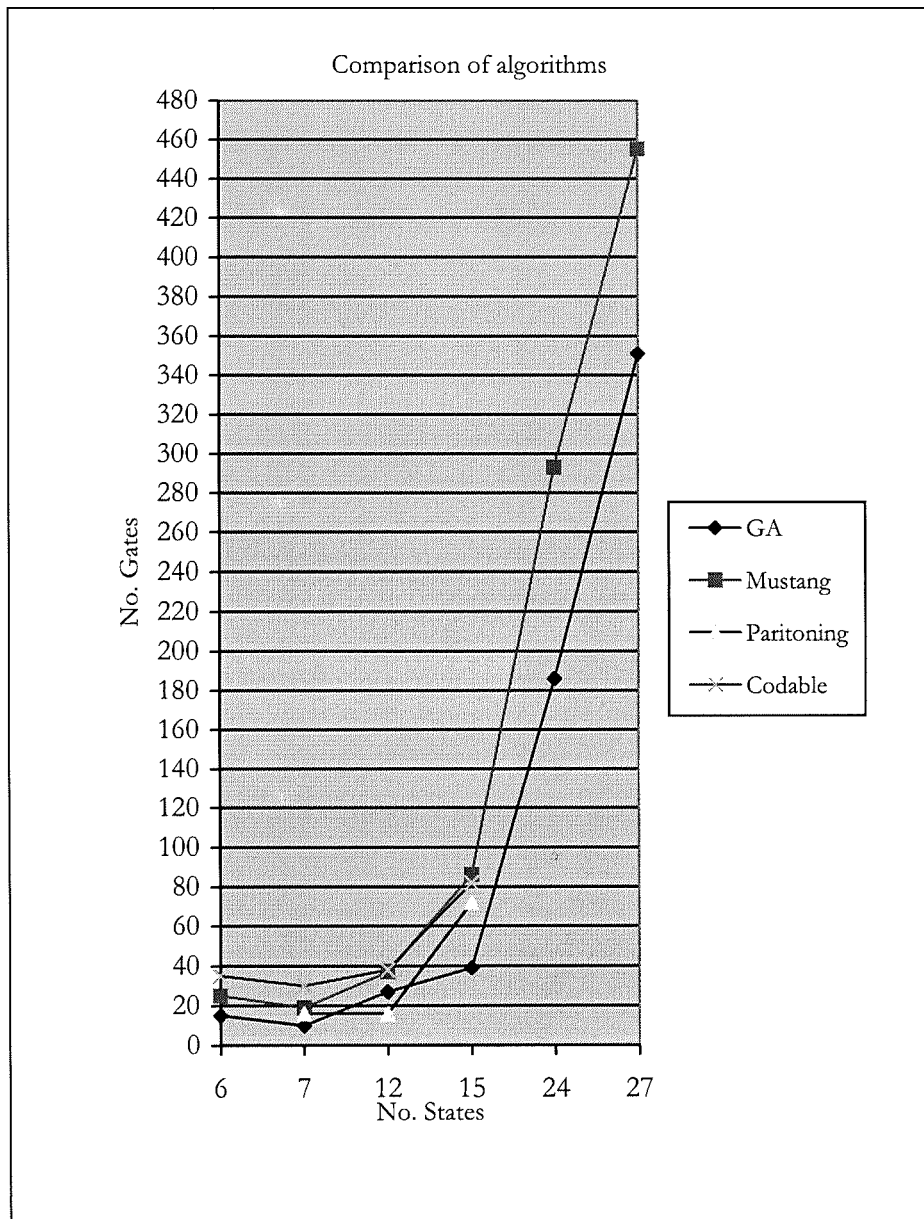


Figure 4.22. Comparison of GA with Mustang, Partitioning and codable column.

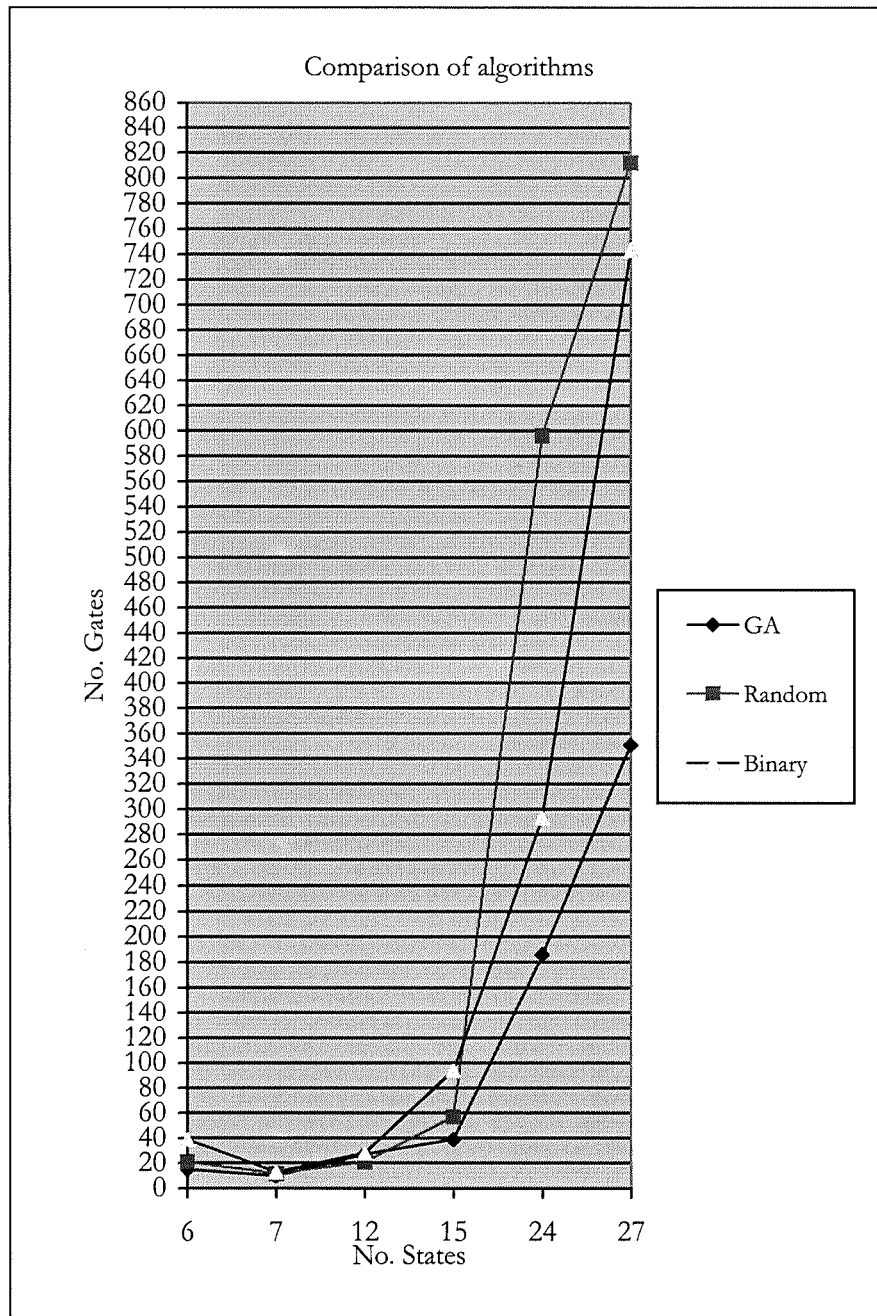


Figure 4.23. Comparison of GA with Random and binary state assignments.

#### 4.11 Discussion of Results

The results given in the previous section show that in most examples a better solution can be found for the state assignment by using GAs rather than a commercial package like NOVA. After a large number of experiments it was found that the GAs consistently finds solutions that are minimal. From the previous experiment the following GA parameters were chosen: population size = 20, breeding rate = 0.25, minimum mutation rate = 0.015. The mutation rate was variable and increased with each generation if there was no improvement in the literal count of the best chromosome. The number of generations over which the GA was run was a maximum of 500. However it was observed that for machines with less than 9 states, the number of generations required for the GA to operate effectively was lower at somewhere between 50 and 100 and time less than 30 sec on average. For the medium size benchmark of MCNC the number of generation was between 100 to 1000 and the time less than 30 min.

Specifically, the following three questions were answered.

1. Are the optimal values of the GAs parameter truly different for different circuits, or are the variations merely a function of the stochastic nature of the algorithm?
2. Are the optimal values of the GAs parameters truly different for different costs function for the same circuit?
3. If the parameters are different for different circuits or within a single circuit, is there any way of estimating what parameters should perform the FSMs and cost function specifications?

The logic equation produced by the fittest state assignments found was converted into equivalent 2-input AND/OR gates to allow comparison with other methods.

Comparison of results between genetic algorithms and different state assignments approaches has been discussed for the average area needed.

From experiments the average area produced by GA is 83% of the result produced by NOVA. The CPU time of GAs varied from 30s for small size benchmarks to 30.0m for large benchmarks depending on the GA parameters. Future, it is believed that better chromosome encoding and more effective parent selection may enhance the effectiveness of the genetic algorithm itself, possibly, and that as a consequence improved results may be obtained.

#### 4.12 Summary

In this chapter the use of genetic algorithms is proposed to solve the state assignment problem. Given the huge search space and the existence of many local minima, this problem is well suited for genetic algorithms. The results are presented to show that this approach can achieve solutions superior to previous methods. The observations show that the choice of the GA parameters is a very important issue.

A program using genetic algorithms to find the optimum state assignment for finite state machines has been written and it can produce a fairly good assignment. Through evolutionary operations of recombination, mutation, and selection new generations of search points are found that show a higher average fitness than their ancestors do. Evaluation of the performance of the search technique can be very compact depending on:

1. How quickly the search finds a solution.
2. How good the solution is.

So, there is a difference between finding a good solution and an optimised solution. The results show that the state assignments as found by the genetic algorithm are at least as good, but in most cases better, than those derived by NOVA and similar techniques. It has been demonstrated that the GA is a valid method of finding a good state assignment; it is competitive with commercial software, and is not dependent on any particular feature of the sequential machine.



## EVOLUTIONARY DESIGN FOR LOGIC CIRCUITS

### 5.1 Introduction

Design is the process of translating an idea into a product that can be manufactured. In the design of electronic circuits this implies a product formed from electronic components, software and electromechanics. Effective design allows this translation to be done quickly, cheaply and accurately to produce a product that is fit for the purpose.

The top-down automatic design method of electronic circuits is generally a complex task requiring knowledge of a large collection of domain specific rules [33]. It should be emphasised that during all these stages great care has to be taken to maintain the logical functionality of the original circuit specification. Hence, there is a demand for effective ECAD tools that perform some of the design tasks leaving the designer to concentrate on issues of performance optimisation. The complexity of the electronic design search space has encouraged the use of evolutionary computation in Evolutionary Electronic Design (EED) procedures.

Moore's law seems to be valid for the development of new computer hardware [90]. This implies that a larger number of transistors implementing digital logic gates are becoming available for designers. Earlier we have seen a limit in the size of hardware devices. However, we may very well soon see limits in design ability. That is, designers are not able to use all the transistors in the largest integrated circuits becoming available. To overcome this problem, new and more automatic design schemes would have to be invented. One such method is evolvable hardware.

Evolvable hardware was introduced recently [97,100] as a new way of designing electronic circuits. Instead of manually designing a circuit, only input/output-relations are specified. The circuit is automatically designed using an adaptive algorithm, which is illustrated in Figure.5.1

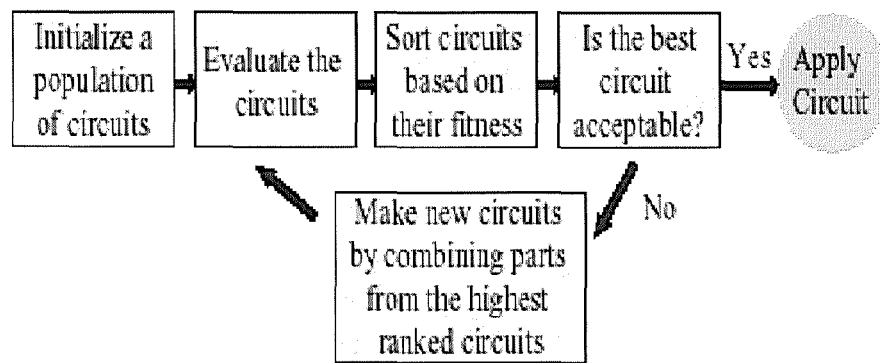


Figure 5. 1. The automatically circuit designed using an adaptive algorithm.

These techniques have shown a high degree of flexibility in dealing with complex and computationally hard problems [9]. The automated synthesis of digital logic to satisfy the function specification is a well-researched area [117]. A circuit synthesised using a functional specification is a relatively straightforward process. However, optimising either the size or the performance of such circuits is a considerably more difficult problem. There is very little research, which actually evolves the functionality of sequential logic circuits [9,118]. EHW approach has begun to show that it is possible to evolve such circuits in a radically different way [1].

The sequential circuits are divided into purely combinational blocks and registers. Large sequential circuits are typically modelled by smaller interacting finite state machines. The aim of this work is to look at the problem of automated synthesis of synchronous sequential circuits using a new approach

based on GA. EHW extends the concept of GA to evolution of electronic circuits [119,120].

This new approach can be expressed as a black box of the problem. From this point of view, one regards the problem of implementing the circuits as being equivalent to designing a black box with inputs and outputs. This black box should be such that on presentation of the original input signals the desired outputs are delivered. The essential new feature of this technique is that the details inside the box are encoded into chromosomes. The chromosome representing a circuit is subject to the usual processes of evolutionary algorithms. The advantage of EHW over the traditional circuit design approach is its capacity for dynamic and autonomous adaptation. EHW can reconfigure its structure dynamically (on-line) and autonomously, according to changes in the task requirement or the environment in which the EHW is embedded.

This chapter will consider the circuit design technique that is based on evolutionary algorithm. The extrinsic EHW approach to evolve sequential logic circuits is proposed. In the proposed approach both GA for state assignment and EHW are combined together, to produce optimal logic circuits. GA is used to optimise the state assignment problem. EHW is used to design the desired circuit. The aim of this approach is to generate a gate level netlist for the target circuit.

## **5.2 The space of all representations**

Every binary function is specified by a truth table. The truth table specifies what values the outputs of a function are for all values taken by the function inputs. There are certain special collections of operators that act on a binary function that have the property that any function can be represented by expressions involving these operators and the input variables. The collection of these operators and the sets they operate on is often referred to as algebra. In the case of binary functions there are two well-known algebras: standard

Boolean which uses AND, OR, and NOT, and Reed-Muller which uses AND, EX-OR and NOT. When these algebras are used, a particular class of expressions can only represent a given function. The basic concept is shown in Figure 5.2.

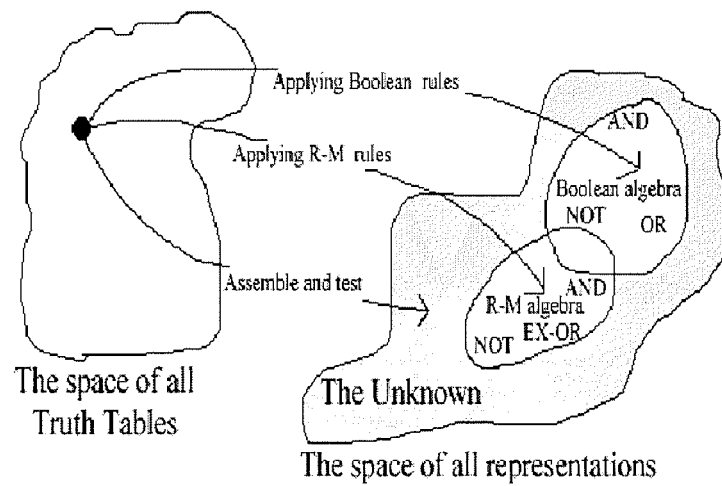


Figure 5.2. How assemble-and-test reaches the unknown regions of the space of all representations [24].

The unknown region in Figure 5.2 depicts all the representations of logic functions which are written as an expression which does not use operations taken from the set {NOT, AND, OR, EX-OR}. Any expression in this region once known could be manipulated to become either an expression in the R-M or Boolean regions [24].

### 5.2.1 Representation of Boolean function

There are different ways of representation Boolean function, which can be classified into tabular forms, logic expressions and decision diagrams. In the logic level of design synthesis, the two-level minimization is a mature and very popular approach. Boolean expression can be simplified with use of Karnaugh maps [135]. The Karnaugh map is a way of representing a Boolean function so that logically adjacent terms are physically adjacent. The Karnaugh

map approach is not suitable for minimising Boolean functions which a higher number of input variables (more than six variable). For such functions, Quine laid down the basic theory that was adapted later by McCluskey, known as Quine-McCluskey procedure [136]. The Quine-McCluskey algorithm involved the systematic and exhaustive reduction of Boolean expressions. The method is applied repeatedly using a formal tabular procedure, which is suitable for minimisation of a Boolean expression containing a large number of variables. This procedure can easily be programmed into a computer. The difficulty is that computation time and memory space are of exponential order, with respect to the number of inputs to above 16 variables.

Two-level and its Programmable logic Array (PLA) implementation shown figure 5.3 (a) provide good solution to a wide class of problem in logic design. However, there are situations, especially for large multiple output circuit, where multilevel design is desirable and more effective. It facilitates sharing and simplifies testing.

For example, simple logic circuit,

$$f = (x_0 \bar{x}_3 + \bar{x}_2)(\bar{x}_0 + \bar{x}_1)$$

Which can be shown in Figure 5.3 (b), has three levels. In the first level, the product term  $x_0 \bar{x}_3$  is generated, which has an AND level. In the second level, both  $(x_0 \bar{x}_3 + \bar{x}_2)$  and  $(\bar{x}_0 + \bar{x}_1)$  are generated with OR level.

Finally, they are combined by an AND gate to produce the output for the function. However, multilevel logic circuits are much more difficult to synthesise than two-level circuits. In 1964, Lawler proposed an approach for exact multilevel logic minimization [121]. All the multilevel prime implicants are first generated, then a minimum subset is found by solving a covering problem using any method for two-level minimization. As an exact optimization method, it is only suitable for small Boolean function on

account of high complexity. In the last two-decades, many heuristic techniques have been developed. An example of a heuristic technique for logic minimization is ESPRESSO [66]. ESPRESSO is a technique that, when applied to an arbitrary sum of product will produce an equivalent sum of products expression and, when applied repetitively, a succession of expressions, each with fewer products than its predecessor.

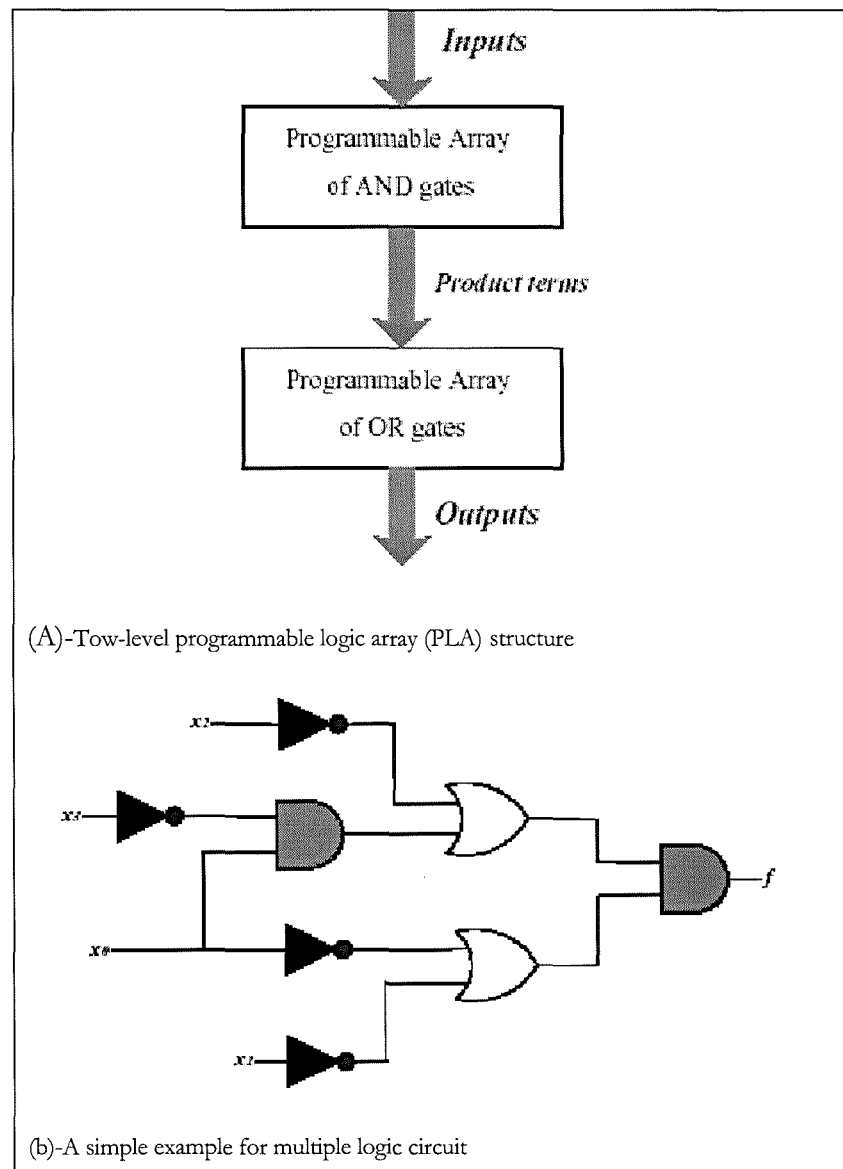


Figure 5.3. Comparison between two-level and multilevel structures.

### 5.2.2 Reed-Muller functions

It is well known that many Boolean functions, which can be easily implemented using exclusive-OR gates, are very inefficiently represented in canonical Boolean logic. When a Boolean logic function is expressed using XOR gates and uncomplemented variables it is called a Reed-Muller (RM) canonical form [122] (see figure 5.4). There are also generalized Reed-expansions where the variable can be fixed or mixed polarizes [33]. Traditionally functions are represented in logic synthesis system using the SOP representation as inclusive OR sum of AND product terms. However, there has been interest recently in using the Reed-Muller representation, which is an Exclusive-OR sum of product terms.

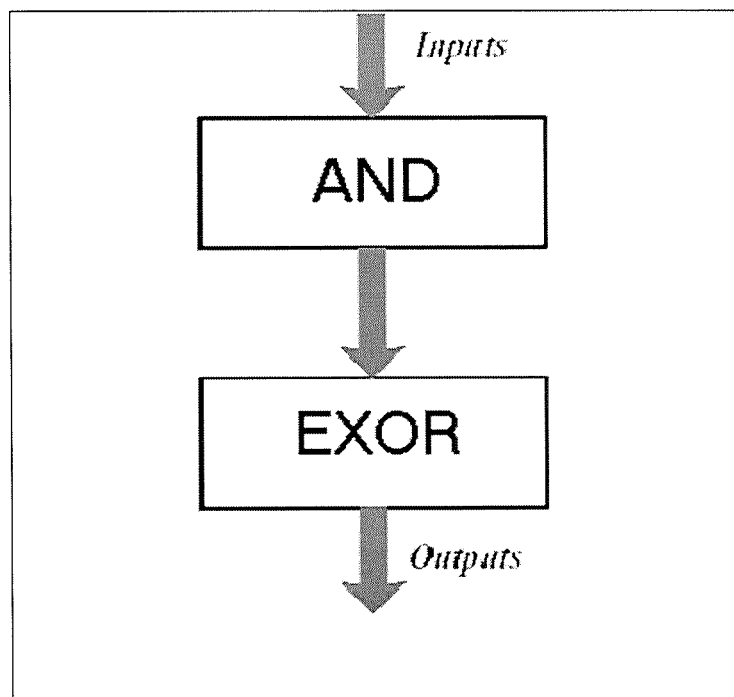


Figure 5.4. R-M form is an exclusive-OR sum of products.

There are two reasons for this interest: firstly in using it as a means of design classes of circuits based on exclusive OR gates, and secondly as an efficient way of representing and manipulating functions. Implementations based on Exclusive-OR gates can be more economical in some application, such as

arithmetic circuits, coding schemes for error control [123]. Further the RM approaches result in circuits that are generally easier to test [33].

### 5.2.3 Mixed functions

The Boolean function represented by (AND, OR and NOT) and Reed-Muller function (AND, EX-OR and NOT) gates. In mixed functions, both type of representations are used to represent the logic function. The new approaches of EHW introduce new methods to design circuits using mixed functions by creating a library of components for the EHW to select from [24].

## 5.3 Backgrounds and Motivation

EHW has only recently been applied to the synthesis of combinational and sequential logic circuits [1,18,84]. Though a number of authors made useful contributions; the subject area is still in the early stage of development. At the earlier stage of EHW investigation, the main purpose of EHW was to evolve fully functional circuits starting with randomly generated populations. The requirements to EHW change with time and current research intend not only evolve fully functional solution but also optimise the obtained solution by a number of optimisation criteria. In respect of these requirement a dynamic fitness function has been proposed by Kalganova [124], where the quality of evolved circuits is estimated in terms of the number of logic gates actually used in the circuits. Thesis by Kalganova [24] investigates a number of issues in extrinsic EHW. They include the circuit layout evolution, function level evolution and increment evolution of combinational logic circuits.

In this section, an overview of work done in this area is outlined.

**Louis [117]** is one of earliest sources that reports the use of GA to design combinational logic circuits. He has made use of genetic algorithms on design structures to attempt to solve the combinational logic circuit design problem.

**Coello et al. [125]** extended the research done by Louis, and used a GA to automate the design of combinational logic circuits. They modeled logic circuits with matrices, with each element representing a gate and the inputs



from previous elements. The gate was selected from a list of five fundamental gates: AND, NOT, OR, XOR and WIRE, where the first four are self-explanatory, and the last representing a physical wire (and thus, the absence of a gate). Logic minimization was thus solved by a constraint on the matrix elements: maximizing the wire element, and hence minimizing the total number of gates necessary for the implementation of the circuit.

**Koza [13]**, on the other hand, used genetic programming to assist in the design of combinational circuits. The main goal of Koza research was the successful generation of the circuits, and not their minimization, thus does not completely apply to the task at hand.

**In Higuchi [126]** work's, the input/output sequence is generated by feeding random input sequence to a given state diagram and gets corresponding output sequence. The existing optimal solution can be reached in reasonable time, but the disadvantage is that correct input/output sequences are not equivalent to given state diagram.

**Fogel [127]** evolved the state machine that can predict the next output symbols from one given input output symbol sequence using Evolutionary Programming (EP).

**Thompson et al. [9, 102]**, another interesting question is how the ideas of artificial evolution can be used to evolve hardware. Currently there are technological limitations mainly regarding time and costs, which prevent the application of such ideas in many areas. Nevertheless there are interesting developments in this field such as the configuration of Field Programmable Gate Arrays (FPGA) using Evolutionary Algorithms. An FPGA is VLSI Silicon Chip containing a large array of components and wires. Switches distributed throughout the chip determine how each component behaves and how the components are connected to the wires. Configuring these switches determines the behavior of an FPGA. The specific arrangement is stored in its configuration memory. Since the FPGA can be interfaced to a host

computer an Evolutionary Algorithm running on the host computer can write to its configuration memory. By setting the switches the EA creates a physically real electronic circuit, which can be tested and evaluated according to its real-world performance.

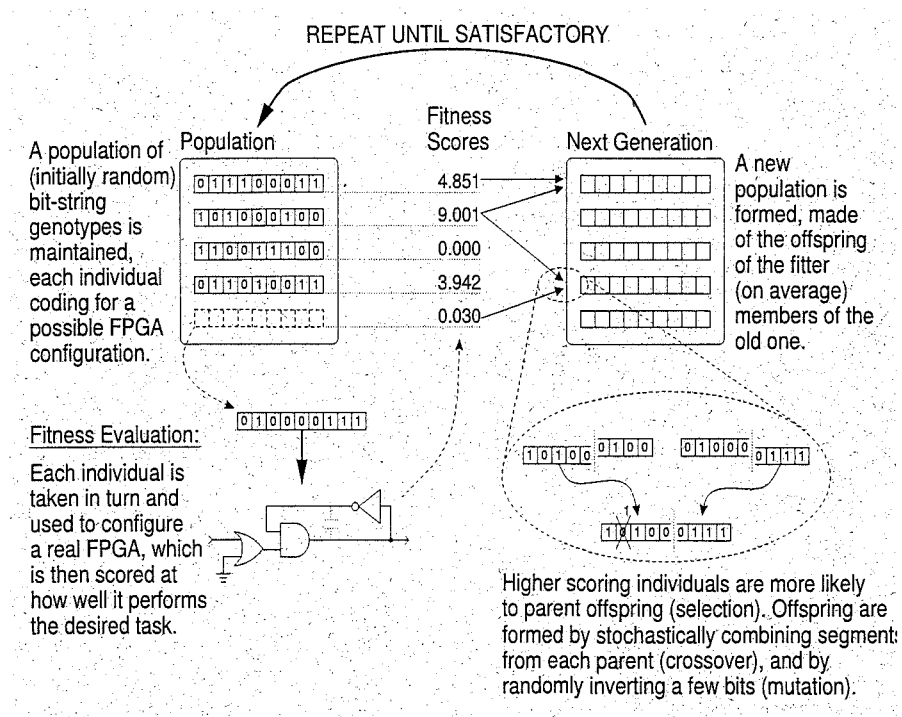


Figure 5.5. Evolving an FPGA configuration using a simple genetic algorithm.

By automatically designing electronic circuits using an evolutionary algorithm major differences between the circuits designed according to conventional goal-oriented methods and those configured using evolutionary methods could be discovered. Perfect behaviour was achieved using only a 10x10 array out of the whole 64x64 array to implement an electronic circuit (see Figure 5.5) whereas conventional design would need a much larger area to achieve the same performance. Within the 10x10 array not all cells contributed to the circuits behaviour and among them are cells which are not even connected to the main part of the circuit but do still influence the behaviour of the system. The detailed physical properties of the silicon the spatial relationships of the

components and their interactions have been exploited by evolution in a fascinating way. A circuit evolved with an EA can use the electronic resources more effectively than an equivalent circuit designed by conventional methods. Thompson [128] has attained a control system in which the original finite state machine used to control the robot movement, was substituted by a Dynamic State Machine, in which synchronous and asynchronous variable are mixed. Thompson is concentrating on the evolution of robust circuits. Robustness is especially important for the use of so evolved circuits in real-world applications. Robustness means in this context that the circuit is able to operate in a satisfactory way even when variations in its environment or implementation occur.

**Hemmi et.al [20]** described hardware with software and came to be able to generate hardware and are proposing hardware evolution system AdAM (Adaptive Architecture Methodology) based on hardware description language SFL of CAD system PARTHENON (Parallel Architecture Refiner Theorized by NTT Original Concept) that NTT developed. The FPGA is used in the field where prototype design and a small amount of other electronic circuits should be made.

**P. Chongstitvatana et.al [23, 118, 129]** show that the evolutionary process has been used to synthesise synchronous sequential circuits that perform according to observed partial input/output. GA was applied to synthesis circuits that are based on Moore and Mealy's model on PLDs and FPGAs (programming logic device and field programming gate arrays).

The authors indicated that the sequential logic circuit modeled in FSM constructed as lookup table (LUT) representing state transition and output function.

Since the FSM can be directly translated into RAM, it can be seen that the RAM content is evolved. With experiment they show that the results can be classified into two categories, complete solution and incomplete solution

depended on the correctness of partial input/output. Without the behaviour description of the target circuit, the correctness of resulting circuit cannot be verified. The correctness percentage is defined as the number of runs that yield a complete solution divided by the total number of runs.

The correctness varied with the length of input/output sequence and it is increased with the number of input/output sequences. Their experiment was conducted to synthesise a number of simple FSMs such as serial adder, counter and sequence detector.

The disadvantage of this approach is that a GA requires a large number of generations to synthesize a large circuit. Due to the cost of the selection method, which required sorting and a large number of generations, the GA took several hours, sometime days to find the target FSM.

Table 5.1 summarises the recent work concerning the evolution of sequential logic circuits. Analysing the table it can be noticed that EHW has been used mainly to synthesise relatively small sequential logic circuits. The hardware evolution is based on primitive logic gates, which is not powerful enough for industrial applications. The development of large circuits is still an important challenge to EHW researchers. The table shows that there is no developed EHW method, which can be applied to evolve real-world sequential circuits.

Table 5.1. Recent EHW approach used to evolve sequential logic circuits

Author	Year	Type of sequential circuit	Evolving platform	Target Application	TYPE OF EHW
T Higuchi [126]	1993	State transition graph		Digital logic circuit	Extrinsic
H Hemmi [20]	1994	Digital Sequential adder	AdAM system	Serial Adder	Extrinsic
A Thompson [128]	1995	Dynamic state Machine (DSM)	Simulator"Mr. Chip robot"	Robotics	Intrinsic
C. Manovit [118] And P. Chongstitvatana [23]	1998 1999	Synchronous sequential logic circuits partial input/output sequence [118] and Finite-state machine synthesis from multiple partial input/output sequences [23].	PLD, GAL	Frequency detector, Odd Parity Detector, Module-5 counter, Serial Adder.	Intrinsic
C. Apornthewan et al. [129]	2000	Learning finite state machine synthesis from partial input/output sequences.	PLD, FPGA	Serial Adder, 0101 Detector, Module-4 counter, Reversible 8-counter.	Intrinsic
B. Ali [1]	2002	Combine both state assignment and EHW to design sequential logic	Logic gates	Serial adder, module-8 counter, Detector and MCNC benchmark [116]	Extrinsic

#### **5.4 Basic idea of the proposed approach**

In this work, a new approach is introduced to evolve sequential logic circuits. This approach treats the synthesis of entire sequential circuits, breaking a circuit up to sub-blocks of combination logic and flip-flops [1].

The proposed approach is based on evolving the functionality and connectivity of a rectangular array of logic gates. The complexity of a logic circuit is a function of the number of gates in the circuit. The central idea of this approach is to represent the circuits in such a way that the genetic operations can be carried out. The proposed approach consists of 4 main stages as follows:

1. Optimisation of the state assignment problem using GAs,
2. Formulation of the task for the combinational logic circuit,
3. Design of the combinational logic circuits using an extrinsic EHW approach.
4. Assembly of the sequential logic circuit.

From the previous chapters in this thesis, it can be seen that the design of sequential circuits involves optimisation of the state assignments, generation of the circuit structure and synthesis of the combinational logic in the next-state and output functions of the sequential logic circuit using extrinsic EHW. In other words, the idea is to identify the sequential logic circuit structure and generate the combinational parts of the circuit using the evolutionary algorithm approach. Hence, both the GA for state assignment problem (SAP) and extrinsic EHW approach are combined to create powerful tools to design sequential logic circuits.

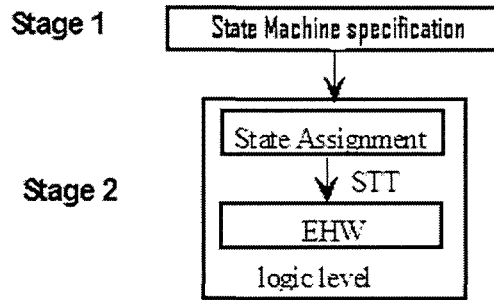


Figure 5.6. Procedure of the proposed approach to design sequential logic circuit using genetic algorithm and extrinsic evolvable hardware

In Figure 5.6, the architecture of the genetic synthesis of sequential logic circuits is described. This architecture contains several stages. The first stage represents the specification of the target sequential circuit behaviour using symbolic state table. The state minimisation, if required, can be done using existing tools [73]. In the next stage, the genetic algorithm uses this symbolic state table to generate optimal state assignments to assign a binary code for each state. Therefore, the state transition table (STT) of the sequential circuit is formatted as a two-level logic PLA file. The extrinsic EHW uses software models to evaluate the fitness function of the resulting circuit. The genetic synthesis creates circuits at the gates-level by using function sets of logic device such as AND, OR, NOT and D flip-flops. It is up to the evolutionary algorithm to choose among these components to create the best possible desired circuits.

### 5.5 Problem Modelling

In this work, the state transition table has been chosen to describe the behaviour of the synchronous sequential logic circuit. The structure of sequential logic circuits shown in Figure 5.7 comprises a set of two sections of combinational logic and D flip-flops (DFFs). The state transition table gives the desired function of the circuit. The circuit is generated using a given set of available logic gates. The combinational parts of sequential logic circuits are

generated using EHW approach, where the state machine truth tables for the combinational parts are defined using GA. Note that in the proposed approach the synthesis procedure of sequential logic circuits is similar to the synthesis of the combinational logic circuit. The desired functionality for the combinational parts of the logic circuit is evaluated using STT. The search space is defined by a number of different factors: (1) building blocks presented to the framework; (2) the number of logic elements used to generate the circuit; (3) the application for which the circuit is being evolved.

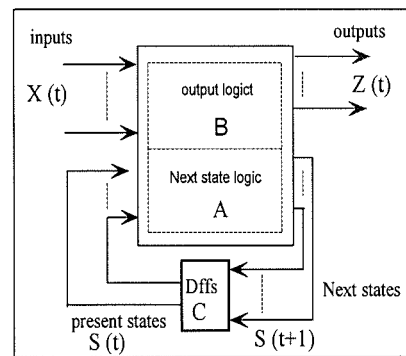


Figure 5.7. Description of the circuit's parts

In the design process, it has been long accepted that the best way to solve a problem is to decompose the problem into several simpler sub-problems. The structure of a sequential circuit in the proposed approach contains 3 subcircuits as shown in Figure 5.7. Each subcircuit is evolved separately. For instance the chromosome representing circuits C is evolved by state table T, T can be decomposed in to smaller state table  $T1$  for subcircuit A and  $T2$  state table for subcircuit B. Once the subcircuits have been evolved, the sequential circuit is assembled. In this case, the 2-combinational logic circuits A and B have to be evolved. Each combinational circuit is represented as a rectangular array of logic gates. Each logic gate in this array is uncommitted and can be removed from the network if it is proved redundant.



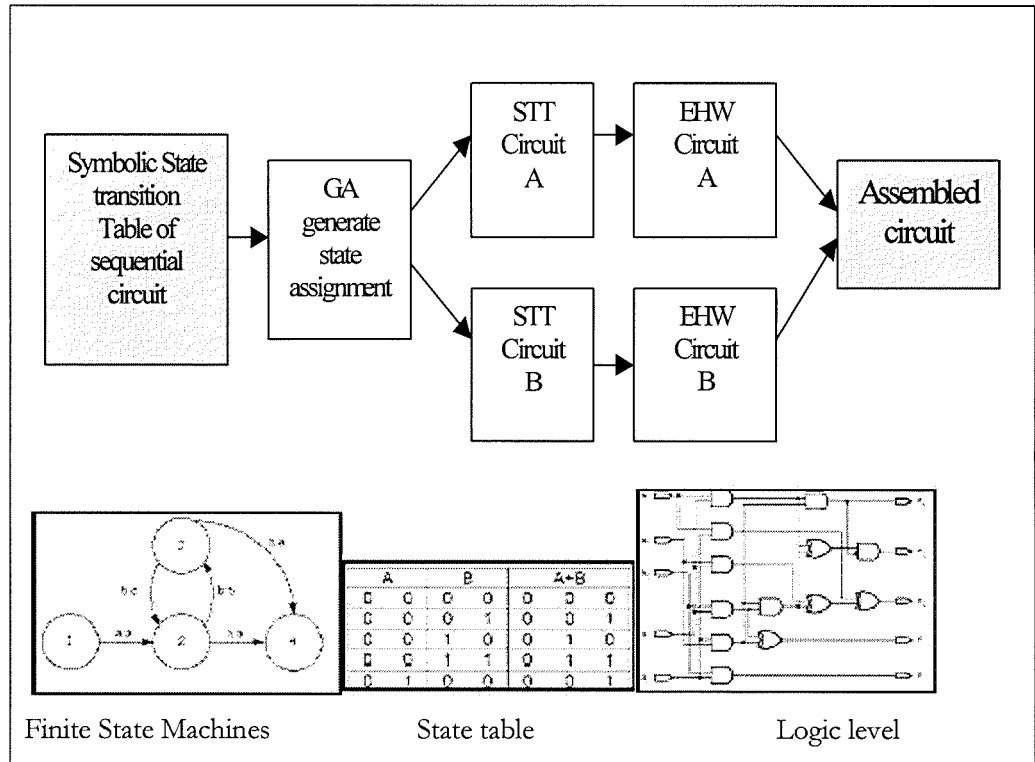


Figure 5.8. The procedure of generating the circuits from the symbolic state transition table.

Figure 5.8 described the procedure of generating the circuit from the symbolic state transition table.

## 5.6 Encoding of evolution

This section describes both the representation and evaluation used within our evolutionary approach. The gate level representation [24] has been used to encode each circuit into an integer string. Figure 5.9 illustrates an example of this kind of representation for a hypothetical output. The circuit (phenotype) is constituted by a combinational part (arrangement of Boolean gates) and a sequential part (D flip-flop). The latter provides the means whereby a delayed version of the output signals can be used as feedback for the same or other circuits.

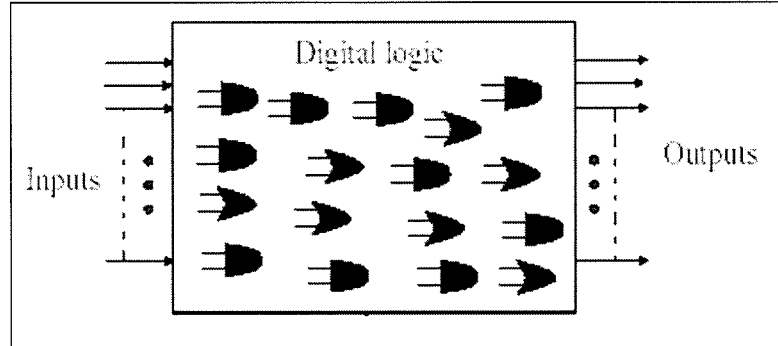


Figure 5.9. the representations for a hypothetical inputs/outputs.

### 5.6.1 Chromosome representation and connectivity

The chromosome defines the connection in the network between the primary inputs and primary outputs. Figure 5.10 shows the representation of the chromosome connection between the 4-primary inputs and 2-primary output. The network is designed using logic gates. The chromosome layout described by  $2 \times 2$  ( $n_{\text{columns}} \times m_{\text{rows}}$ ) geometry of uncommitted logic cells and netlist numbering. Each logic cell is represented by a triple of integers  $\langle c^1 c^2 c^3 \rangle$ , where  $c^1$  defines functional gene and  $c^2, c^3$  define the gate inputs. The genotype is made up of blocks of integer numbers or genes that encode the type of each particular logic gate. The genes associated with the gates of the first layer will encode its nature and also the source of the input signals.

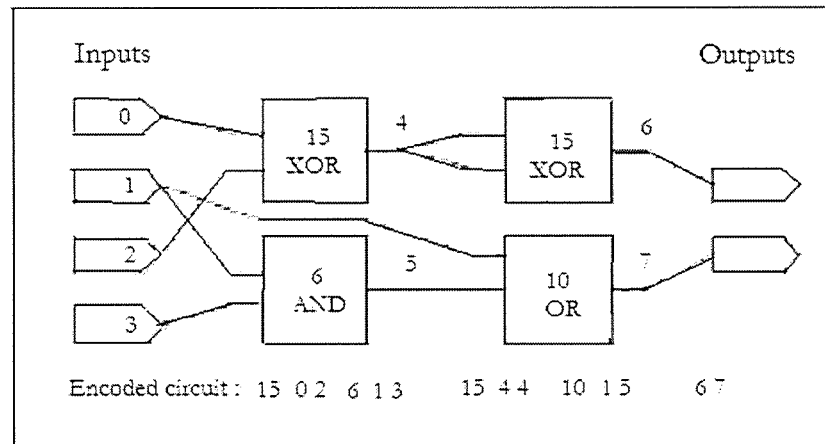


Figure 5.10 Schematic of the chromosome structure used in EHW approach with circuit layout equalled to 2x2.

The circuit structure is encoded as connectivity list of rectangular array of components. The genotype is characterised by three parameters: the number of columns, the number of rows and the level back parameter [1]. The first two parameters are merely the dimension of the rectangular array and the last one is a parameter, which controls the internal connectivity. The level back parameter determines how many columns of cells to the left of a particular cell may have their outputs connected to the inputs of that cell. The maximum cell connectivity can be achieved if the number of rows is one and the level back parameter is equal to the number of columns. At the same time, if the number of rows is one and the level back parameter is one then each cell must be connected to its immediate neighbour to the left. Further, cells within any particular column cannot be connected to each other and each logic gate has two-inputs and one output.

To encode a digital electronic circuit into a genotype, the array is defined as a composition of cells each of which can be any allowed two-input logic gate. The allowed cell functions are listed in Table 5.2. The sixteen letters in the list represent all Boolean functions of two arguments.

Table 5.2. The full set of components, gates 0 to 15 and their logic functions.

Gene	Function	Gene	Function
0	0	8	$a \cdot \bar{b}$
1	1	9	$\bar{a} \cdot \bar{b}$
2	$a$ "wire"	10	$a + b$
3	$b$ "wire"	11	$\bar{a} + b$
4	$\bar{a}$	12	$a + \bar{b}$
5	$\bar{b}$	13	$\bar{a} + \bar{b}$
6	$a \cdot b$	14	$a \oplus b$
7	$\bar{a} \cdot b$	15	$\bar{a} \oplus \bar{b}$

### 5.6.2 Genetic operators

Crossover and mutation are the operators through which new circuit solutions are generated. Due to the complex interaction of logic elements, which comprise a circuit, the effects of both crossover and mutation can be highly disruptive to the search.

Recombination is implemented with uniform crossover. The whole logic cell is considered as a gene. Two cells are chosen from two chromosomes and swapped. The number of chromosomes selected for breeding is defined by the crossover rate, which is carried out on a cellular level. In order to preserve the interconnection conditions, the repair algorithm checks the parameters of logic gates for correctness. When two chromosomes with different geometries undergo crossover it is very likely that merely swapping genes to produce the offspring. These would have to be repaired (randomly initialised), and this would introduce a considerable amount of randomness into the recombination process. Thus, the selection of correct crossover rate and its type is very important.

There maybe result of broken connection between logic elements caused by recombination. So as to minimize the negative effects of crossover, chromosome repair is used to reconnect any element connection broken during the operation.

Mutation is used to maintain diversity within the population. It operates directly after crossover and is analogous to a copying infidelity as material is transferred from parent to offspring.

**PARAMETER CIRCUIT MUTATION:** The parameter circuit mutation allows us to change the following three parameters of the circuit: (1) Cell input, (2) Cell type, and (3) Circuit output. Each of these parameters is considered as an elementary unit of the genotype. The parameter circuit mutation rate defines how many genes in the population are involved in mutation. The chromosome contains 3 different types of genes. The three mutation operators applied are highlighted in Figure 5.11.

**GEOMETRY MUTATION:** Geometry mutation allows us to change the number of rows and columns in chromosome. Geometry mutation can be applied to each chromosome with the geometry mutation probability. The number of rows and columns are treated as an elementary unit of the genotype. Each such unit can be changed with a probability 0.5. The geometry mutation consists of two main steps: (1) Gene mutation, (2) Repair algorithm. On the first step the new number of columns or rows of the chromosome is randomly defined. This number cannot exceed the maximum number of columns or rows. On the second step, the repair algorithm is applied to ensure that a chromosome with new geometry represents a valid genotype.

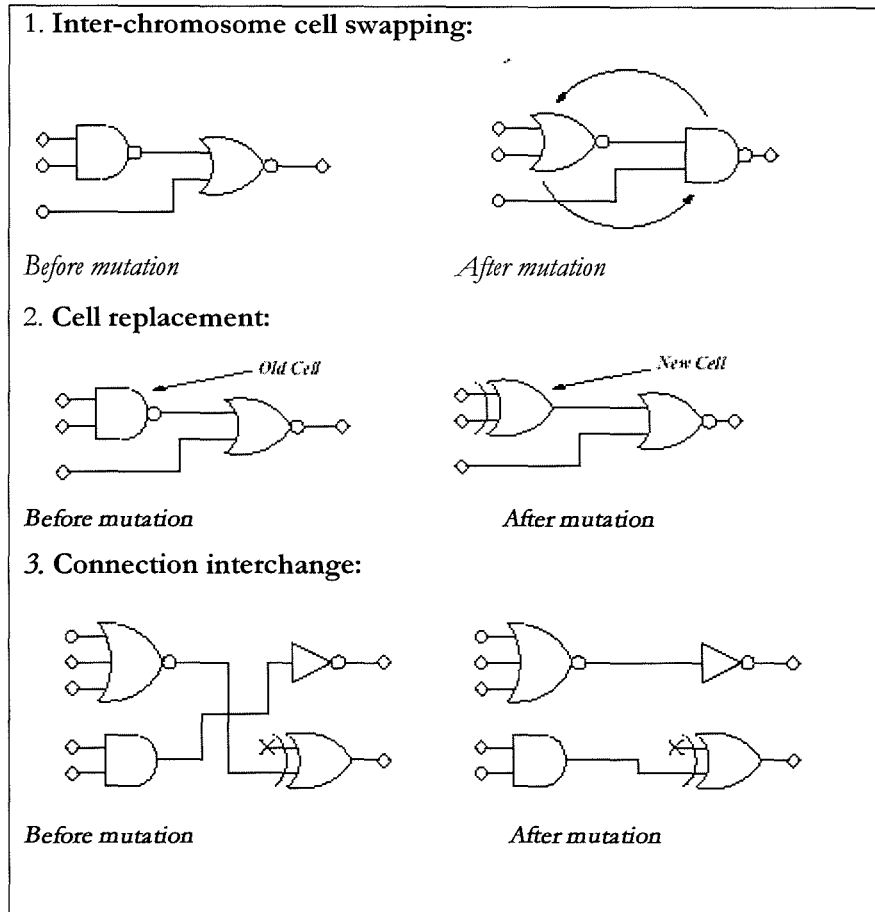


Figure 5. 11. Three mutation operators used by the genetic algorithm.

Let us consider geometry mutation process for chromosomes with 3x3 circuit geometry. Let  $N_{columns}$  and  $N_{rows}$  be the number of columns and rows of chromosome assigned to be mutated and *new\_gene* is the new value of mutated genes, which is synthesised randomly. The gene mutation procedure is the following:

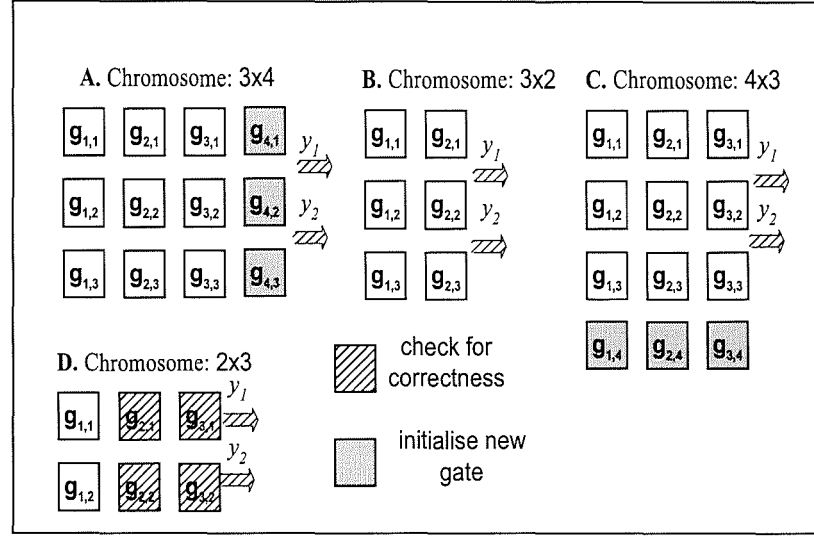


Figure 5.12. The geometry mutation process (3x3 circuit geometry) [24].

Define the circuit mutation rate  $P_{mg}$ .

Generate random number for each chromosome,  $rand1 \in \{0, 1\}$ .

If ( $rand1 < P_{mg}$ ) the geometry mutation is applied to the current chromosome

Generate random number  $rand2 \in \{0, 1\}$ .

If ( $rand2 < 0.5$ ) the number of columns in chromosome is chosen to be mutated and the new number of columns ( $new\_gene$ ) is generated from the range  $[1, N_{columns}^{max}]$ . Else the number of rows is considered as mutated gene and the new number of rows ( $new\_gene$ ) is generated from the range  $[1, N_{rows}^{max}]$ .

When  $new\_gene$  is defined, the geometry mutation is performed in the following manner. Let us consider the case when the mutated gene is the number of columns. In this case the new circuit structures, shown in Figure 5.12 (A and B), can be synthesised. If ( $new\_gene > N_{columns}$ ) we have to add new columns in the chromosome representation (Figure 5.12 (A)). The gates in new columns are initialised using initialisation procedure. It is possible, however, that the circuit output disobeys the levels-back constraint. Thus, the

chromosome may need to be repaired. The repair algorithm checks whether the circuit outputs obey the levels-back constraint, and whether all the cell inputs are valid. If the circuit output does not satisfy this condition a new circuit output is initialised. If ( $new\_gene < N_{columns}$ ) we have to remove some columns in the circuit structure (Figure 5.12 (B)). After the new structure is obtained, a repair algorithm is applied to the circuit output, because the circuit output can refer to a gate which no longer exists in the circuit. In the case when the mutated gene is the number of rows, the structures C and D given in Figure 5.12 can be synthesised. If ( $new\_gene > N_{rows}$ ) the new rows of gates are added to the circuit structure (Figure 5.12 (C)). Again, these gates are initialised. There is no need to apply repair algorithm to the circuit outputs in this case because connections are not changed and the circuit outputs will still refer to the correct logic cells in the circuit structure. If ( $new\_gene < N_{rows}$ ) some rows are removed from the circuit structure (Figure 5.12 (D)), then the inputs of the remaining gates as well as circuit outputs can refer to gates, which are no longer present. The repair algorithm has to be applied to each genotype of the gate and to the circuit outputs.

### 5.6.3 Connection repair algorithm

The result of GA operation (mutation and crossover) correctness of genotype can be broken if the functionality of any logic gate has been changed. Therefore, once the number of outputs in the logic cell has been changed, the gate input connections and circuit output connections are checked for correctness.

The repair algorithm needed for:

- Circuit output connections;
- Gate input connections;
- Gate output connections.



The repair algorithm is activated once the number of output logic cells has been changed. If the repair algorithm is activated the circuit output connections and the gate input connection are checked for correctness.

$N_{connect}$  is defined as the connectivity parameter representing the number of columns on the left to which a cell of a particular column or an output may be connected.

#### 5.6.4 Criteria used in an extrinsic EHW

The main purpose of the extrinsic EHW approaches is to evolve the fully functional circuit with optimal parameters. Any type of circuit parameter can be chosen to define the quality of evolved circuit. In this work the number of primitive logic gate is used as optimization criteria.

1. Let the percentage of correct output bits be  $F_1$ ;
2. Let the number of active primitive logic gates in the circuit be  $F_2$ ;

One objective of circuit design is to construct a fully functional circuit optimised by given criteria. In EHW approaches the search for the desired circuit begins with the randomly generated circuits. This means that initially the chosen circuits are not fully functional. Hence, there are two main objectives in EHW:

- to evolve a fully functional circuit;
- to optimise the evolved circuit by given criteria;

The first objective of EHW can be achieved by checking the actual circuit for correctness. This can be provided by criteria  $F_1$ . These criteria show how close the actual circuit functionality is to the requested one.

#### 5.6.4.1 Fitness function

An evaluation function, called fitness function needs to be defined for a problem to be solved in order to evaluate chromosomes. A chromosome with a high fitness value is likely to be a good solution to the problem. Therefore, the goal of EHW can be defined as generating fully functional circuits in which the whole manufacturing cost is minimum. At the beginning of the search, only compliance with the state table is taken into account. Once the first fully functional solution appears, the evaluation process switches to a new fitness function in which fully functional circuits that have less manufacturing cost are rewarded. Each of these two fitness functions can optimise the circuit by a number of criteria. For example, Dynamic fitness function variables ( $F_1 + F_2$ ) are used to evaluate the circuit [1, 24].  $F_1$  uses Hamming distance to measure the 100% functionality of the circuit between a given set of inputs and outputs.  $F_2$  defines the number of primitive logic cells that are used in the circuit.  $F_2$  is activated when  $F_1$  reaches 100% functionality. The following steps are required in order to evaluate the chromosome.

-Set the initial inputs to circuit.

-The output of the circuit is mapped with the corresponding next state columns in the STT.

The first fitness function compares the corresponding outputs of subcircuit A and subcircuit B with given next state column of STT. The percentage of correct next state bits corresponding to the  $j$ -th output,  $Fy_j$  is calculated as follows:

$$Fy_j = \left( \frac{\sum_{i=1}^p |y_j - d_j|}{p} \right) * 100 ;$$

where  $p$  is the number of input combinations in the given logic function and  $|y_j - d_j|$  is the absolute difference between the actual next state output and

the desired output  $d_j$ ,  $y_j$  is the vector of the j-th circuit outputs. The circuit completely implements the output  $y_j$  if  $F_1$  reaches 100% functionality. Once the solution has been evolved the circuit optimisation criteria is activated. The second fitness function  $F_2$  minimises the number of logic gates by rewarding those circuits with the least number of active logic gates. The procedure described above applies for both subcircuits A and B.

#### 5.6.4.2 Test vectors

The experiments with test vectors aim to evolve the cell such as for a given input it gives the correct output. Figure 5.13 illustrates the procedure that is used to compute the fitness based on the test vectors. For each line of the truth table the following is done: the entry in the input column is sent to the cell as its input; the output of the cells is then compared with the entry in the output column; and, finally, this comparison gives the fitness for the current ( $F_1$ ). The process is repeated for each line and the whole fitness can be taken as the sum of the fitness obtained for each line. For the experiments with the test vectors, the fitness is taken as the number of output bits of the cell that match the truth table. The three input circuit, for example, would be evaluated over  $2^3 = 8$  possible input combinations.

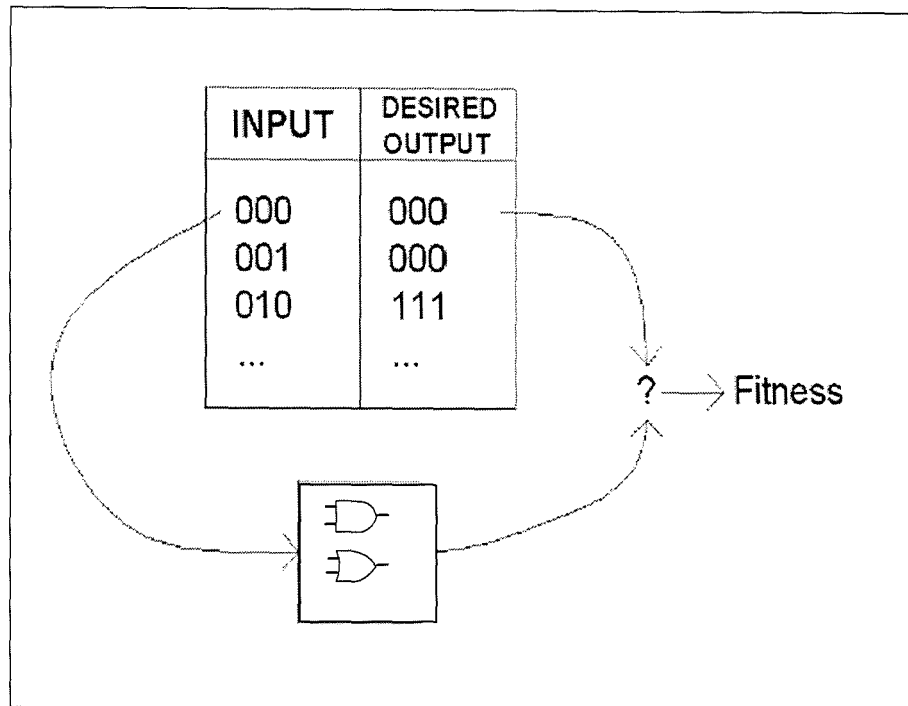


Figure 5.13. Test vectors.

### 5.6.5 Definition of subcircuits for combinational logic circuits

In this section, the components in Table 5.2 with the D flip-flops are used to implement the example given in Table 5.3. In order to evolve the whole FSM, the genetic algorithms have to be executed, one for each circuit output adopting the following strategy:

1. Run the GAs for each input, assuming a delay flip-flop in each circuit output.
2. Find the outputs which were hardest to evolve, and store the delayed output used as inputs to these cells.
3. Re-run the GA for the other inputs, keeping available only the delayed outputs used by the circuits representing the signals mentioned in the second item,

The aim of this strategy is to minimize the amount of time, by placing a delay flip-flop only in those inputs to the cells, which have been more difficult to evolve.

For that reason, the initial state is read and subcircuits A and B are constructed using the primitive logic gates to implement the next state function. The initialisation procedure contains several steps: (1) Define circuit geometry of chromosomes in the population; (2) Initialise the genotype of cells; (3) Generate the circuit outputs for each of the chromosomes.

The first step is constrained to observe the maximum number of rows and columns in the chromosome. During the second and third steps, the initialisation of cell inputs and circuit outputs is performed according to the levels-back constraint and to the type of variables.

Table 5.3. The transformation process of STT in to PLA file format. Where *i* inputs = input + present state bits, *.o* designed the number of outputs calculated, outputs =next state +output bits, *.p* is the number of product terms, *.e* is end of file.

				STT of the circuit	STT of subcircuit A	STT of subcircuit B
i/p	Ps	Ns	o/p	.i 4	.i 4	.i 4
0	S0	S1	0	.o 4	.o 3	.o 1
0	S1	S4	1	.p 10	.p 10	.p 10
0	S2	S4	1			
0	S3	S4	0			
0	S4	S0	0	0000 0010	0000 001	0000 0
1	S0	S2	0	0001 0101	0001 010	0001 1
1	S1	S3	1	0101 0101	0101 010	0101 1
1	S2	S3	0	0110 0100	0110 010	0110 0
1	S3	S4	1	0010 0000	0010 000	0010 0
1	S4	S0	0	1000 1010	1000 101	1000 0
				1001 1101	1001 110	1001 1
				1101 1100	1101 110	1101 0
				1110 0101	1110 010	1110 1
				1010 0000	1010 000	1010 0
i/p inputs				.e	.e	.e
o/p outputs						
Ps Present state						
Ns Next state						
Step 1				Step 2	Step 3	

Table 5.4. Initial parameters used to evolve the circuit.

Parameters	GA	EHW
Population size	20	10
Number of generations	100	5000
Number of GA runs	10	100
Crossover rate	0.25	0.6
Crossover type	Two-point	Uniform
Mutation rate	0.015	0.05

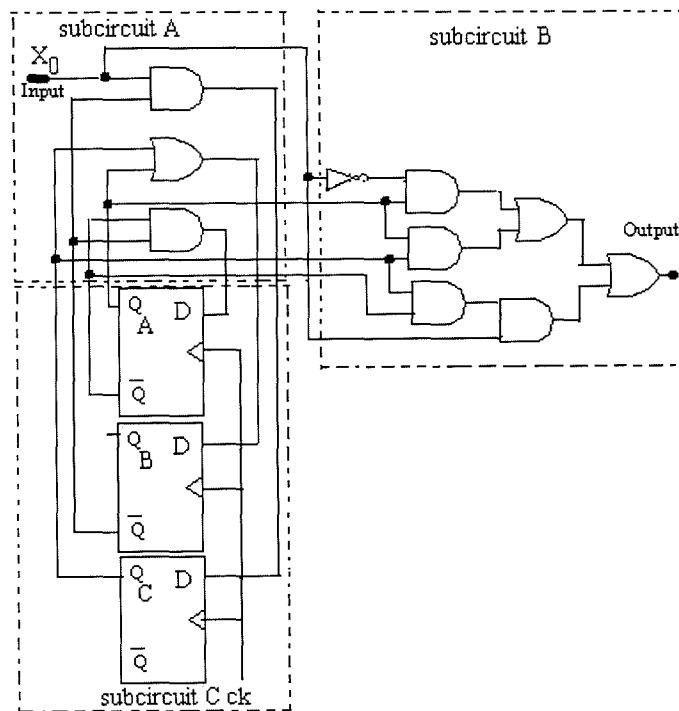


Figure 5.14. Circuit structure is implemented according to state table given in Table 5. 3.

In Table 5.3, step 1 shows the symbolic state table of FSM and the state assignment generated by GA. In step 2 the encoding is used to obtain the standard two-level PLA format. In step 3, the STT of the circuit is divided into input combinational logic subcircuit (A) and the output combinational logic subcircuit (B). Once the decomposition is completed, the fully functional circuits can be generated using EHW. The initial data of GA parameters is given in Table 5.4. The graphical representations of evolved subcircuit A and B are shown in Figure 5.14. The total number of logic gates in the assembled circuit is 10 (6 AND, 3 OR, 1 NOT). The most efficient evolved subcircuit consists of 3 logic gates in subcircuit A, 7 gates in subcircuit B and 3 D flip-flops. The functional set of logic gates contains all gates encoded from 0 to 15 (see Table 5.2).

### 5.7 Experimental results

In this section, the circuit structure synthesised using the proposed approach is considered and compared to manual designs. We present the problem of designing a digital circuit in terms of a set of examples. Where in this experimental result, the GA parameters in Table 5.4 are used for the state assignment problem. A number of experiments have been carried out in order to determine suitable values for the EHW parameters.

This section will describe the actual experiments conducted. For every experiment we will describe the following:

- General description
- Implementation aspects
- Experiment settings
- Number of runs
- Number of generations
- Population size

- Creational scheme
- Selection techniques
- Evolutionary operators
- Fitness function
- Success and termination predicate
- Fitness distribution
- Structural variety
- Fitness evolution

#### **5.7.1 Example 1: Serial Adder**

In this synchronous sequential circuit there is a combinational logic comprised of 1-bit full adder, with latch storing the value of the adders carry output. By definition, a full adder adds two logic variable A and B and carry input  $C_i$ . Many different approaches have been developed to evolve full adder using primitive logic gates [11]. The serial adder accepts as input two-serial strings of digits of arbitrary length, starting with low order bits, and produces the sum of the one-bit stream as its output. So this device can be easily described as state machine has 2 inputs (AB), one output (S) and number of internal state is 2 ( $S_0, S_1$ ) where generated carry ( $C_0$ ) is feed back. However, in this case we don't have to implement GA for state assignment because the circuit has only two states (see figure 5.15).



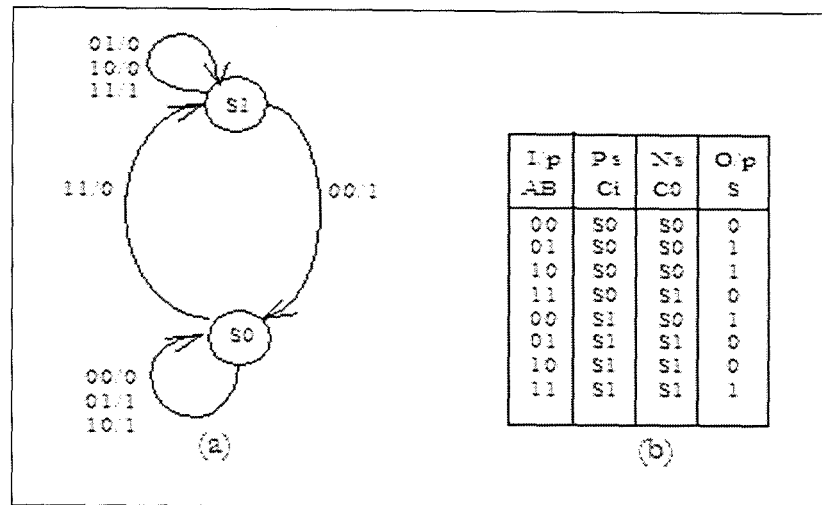


Figure 5.15. Serial adder (a) State transition graph, and (b) state transition table ; where #I/p inputs, #Ps present state, #Ns Next state, #O/p Outputs.

Table 5.5 shows the comparison of functional circuit of serial adder generated by human designer [33] using Karnaugh maps and the one evolved using 2-input AND/ OR, NOT gates.

Table 5.5. Solution obtained for serial adder using EHW approach and manual method [1].

EHW	Human design
$C_o = (A + B) C_i + A B$	$C_o = A B + C_i A + C_i B$
$S = \bar{A} \bar{B} C_i + A \bar{B} \bar{C}_i + \bar{A} B \bar{C}_i + A B C_i$	$S = \bar{A} \bar{B} C_i + A \bar{B} \bar{C}_i + \bar{A} B \bar{C}_i + A B C_i$
Number of 2-input AND&OR	Number of 2-input AND&OR
Gates =16	Gates =16

In this case, the circuits are evolved for one output sum (S) and two inputs (AB) with next state carry Co. Miller [11] shows it was possible to evolve the one-bit adder by evolving the functionality and connectivity of interconnected AND, OR and EX-OR gates. The gate-level representation of this circuit required 5 logic gates, where the sum output implemented with EX-OR gates (see Figure 5.16a). Further, in our case we concentrate on using two-level gate PLA format representation to evolve the serial adder, which required more

gates than the previous work done using multilevel function representation of full adder. For simplicity, we used full adder structure, which connected to D flip-flops and the specification of desired circuit is in the form of STT in PLA format. However, the gate-level encoding would require geometry 3x10 circuit layout. The most efficient logic gates representation is shown in figure 5.16. In Figure 5.16(b) a gate array representation of an evolved serial adder is given. The inputs A, B, and  $C_i$  are binary inputs. The allowed cell functions can be chosen to be any subset of those shown in Table 5.5.

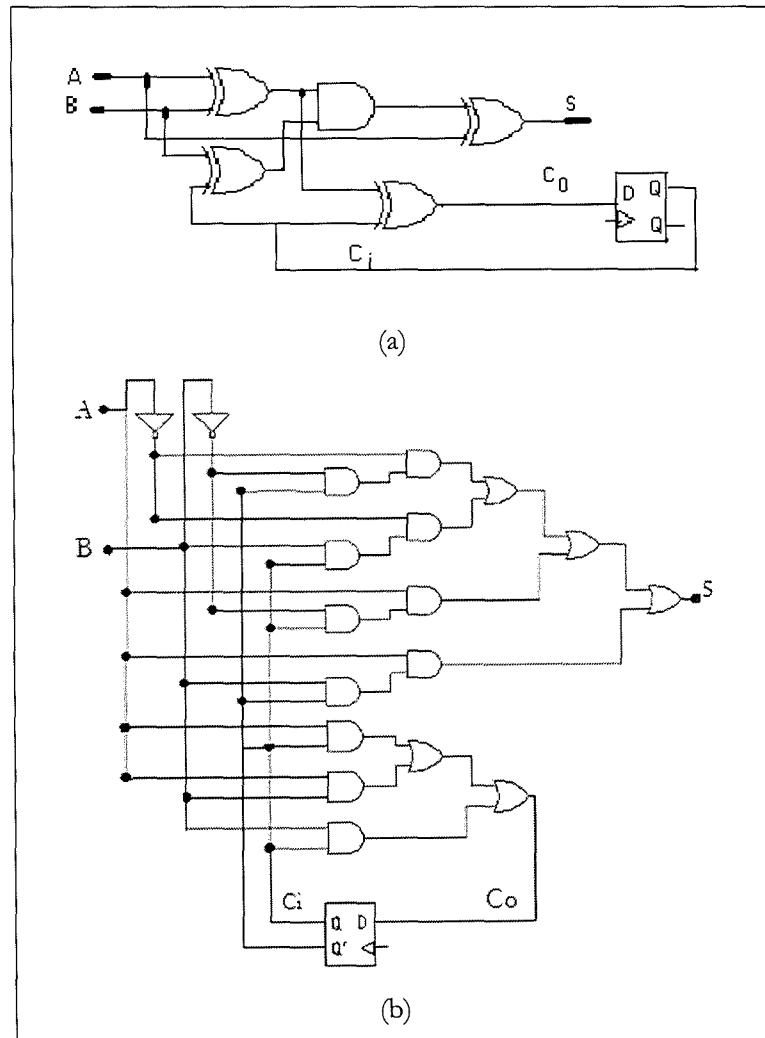


Figure 5. 16. Evolved Serial adder design using (a) functional set (0-6, 15,16), (b) functional set (0-6, 10).

It should be noted that the circuit was actually evolved by separating the carry function from the sum and evolving them separately. The genetic algorithm had the following parameters: population size =10, maximum number of generations=5000, crossover rate=0.6, mutation rate=0.05, elitism, levels-back= 10. In the 10 runs, three solutions were obtained which were 100% functional.

### 5.7.2 Example 2: Module-4 counter

The simplest synchronous digital system is the binary counter. Module-4 counter has four internal states as shown in Figure 5.17. The optimal state assignment has been identified using GA as in Figure 5.17(c). The evolutionary algorithm parameters to design the combinational parts of the GA are: the population size is 5, the maximum number of generation is 5000, the total of runs is 100, the crossover rate is 0.6, the mutation rate is 0.05, layout described by 1x10 in proposed approach (a) and by 4x4 in proposed approach (b) and the resulting equations are given in Table 6.

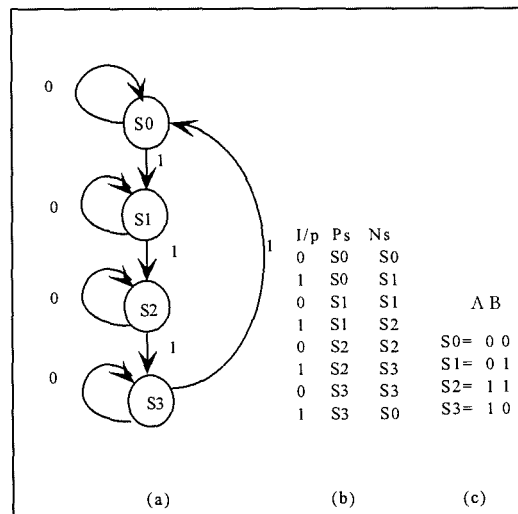
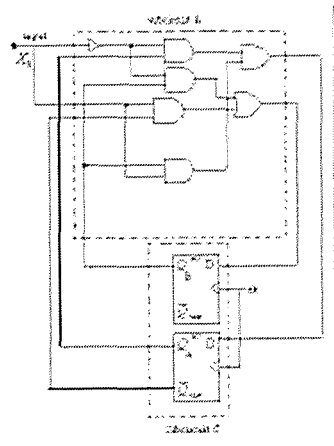


Figure 5.17. Module-4 counter a)-state transition graph, b) State table and c) State assignment generated by GA.

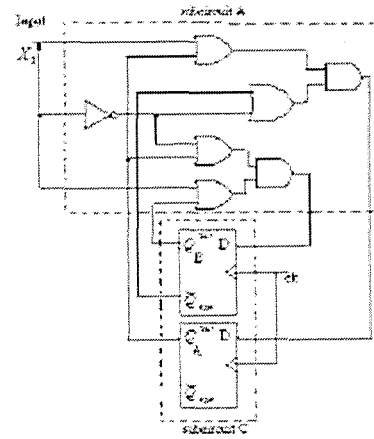
Table 5.6. Solution obtained for model-4 counter using EHW approach and manual method.

Proposed approach (a)	Proposed approach (b)	Manual method
$D_A = \bar{X}_0 A + X_0 B$	$D_A = \bar{X}_0 A + X_0 \bar{B}$	$D_A = \bar{X}_0 B + X_0 \bar{B}$
$D_B = X_0 \bar{A} + \bar{X}_0 B$	$D_B = X_0 A + \bar{X}_0 B$	$D_B = X_0 \bar{A} B + \bar{X}_0 A + A \bar{B}$
Subcircuit A=5 gates	Subcircuit A =5 gates	Subcircuit A=9 gates
Subcircuit C= 2-D flip-flops	Subcircuit C=2 D flip-flops.	Subcircuit C=2 D flip-flops

In this example, small size of population and large numbers of generations are used. The circuit shown in Figure 5.18 has also been tested using two layout sets with the same state assignment. Choosing too small circuit layout run the risk that no 100% functional solution could be found because it is physically impossible to build the circuit of required functionality with few logic gates. Choosing too large a circuit layout gives the evolutionary algorithm too many possibilities to work with. This is has been proved for combinational logic circuit in [24].



(a) Layout 1x10



(b) Layout 4x4

Figure 5.18. Evolved optimal circuit solution of the model-4 counter.

Figures 5.18a and 5.18b show how the numbers of the columns and rows influence the evolution of circuits. The search space of evolutionary algorithm

increases with increase of the number of rows and columns. It can be seen that in Figure 5.18b the evolved optimal circuit solution with circuit layout 4x4 required 2 AND, 4 OR and 1 NOT logic gates. Note that there is no fixed solution of evolved circuit structure with the same state assignment. Three evolved circuits of the problem are given with different circuit structures. The module-4 counter has been evolved and the experiment results show that it is possible to improve the GA performance considerably by careful choice of the number of columns of the cell and level-back parameter.

### 5.7.3 Example 3: Sequence detector

In this example, we synthesise a sequence detector. This circuit has one-input and one-output. The behaviour of circuit can be described as shown in Figure 5.19. When the input sequence 011 occurs, the outputs become 1 and remains 1 until the sequence 011 occur again. In this case, the output returns to 0. The output then remains 0 until the sequence 011 occurs a third time, etc. The comparison of result produced by the proposed approach and manual design based on a random assignment are shown in Table 5.7.

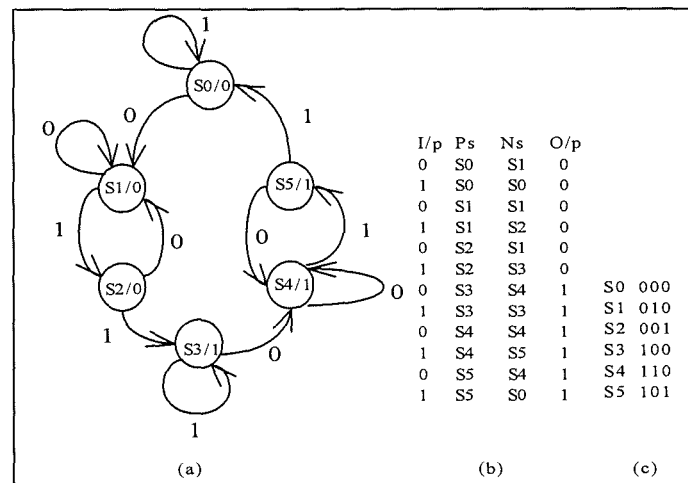


Figure 5.19. A sequence detector described as a) state transition graph, b) state table, c) State assignment generated by GA.

Table 5.7. Solutions obtained for sequential detector produced by proposed approach and manual method.

Proposed Approach	Manual method
$D_A = XB$	$D_A = A\bar{C} + A\bar{X} + BC\bar{X}$
$D_B = \bar{X}$	$D_B = BX + \bar{A}CX$
$D_C = XA\bar{C} + \bar{X}C + \bar{A}C$	$D_C = BX + \bar{A}\bar{C}\bar{X} + \bar{A}B\bar{X} + A\bar{C}X$
$Z = C$	$Z = A + BC$
Subcircuit A =8,	Subcircuit A =17,
Subcircuit C=3 D flip-flops	Subcircuit B =2
	Subcircuit C=3 D flip-flops

Once the EHW generate the same output as the target machine, the sequential network of subcircuit A can be structured using 5 AND, 2 OR, 1 NOT and subcircuit C implemented by 3 D flip-flops as shown in Figure 5.20. The manual design uses a random state assignment and Karnaugh maps to minimize the equations. The parameters of the evolutionary algorithm for this example are as follows: the cell mutation rate is 0.05, the population size is 20, and the maximum numbers of generations is 5000. A circuit layout structure of 4 rows and 5 columns is used with maximum level back parameter equal to 5. The parameter help to obtain a reasonable probability of achieving 100% correct solution.

Analysing the evolved circuit structure in Figure 5.20 it can be seen that the resulting circuit has a more efficient structure than the circuit obtained by manual method.

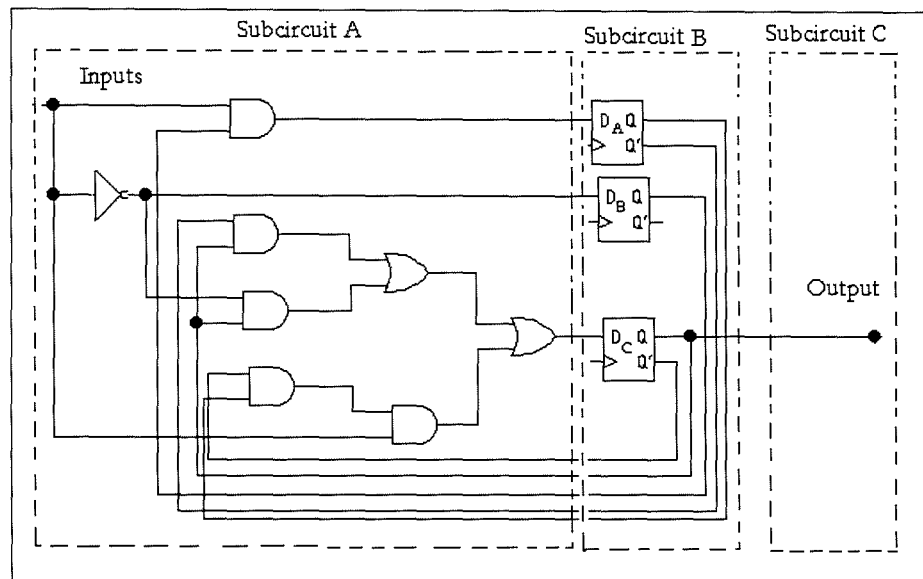


Figure 5.20. Evolved optimal circuit solution of the sequence detector with circuit layout 4x5.

#### 5.7.4 Example 4: 1010 detector

The example demonstrates the synthesis of 1010 detector. This circuit has one input, one output and 4 internal states, as shown in Figure 5.21. The results produced by the proposed method are compared with results produced by a random assignment [33] as shown in Table 5.8.

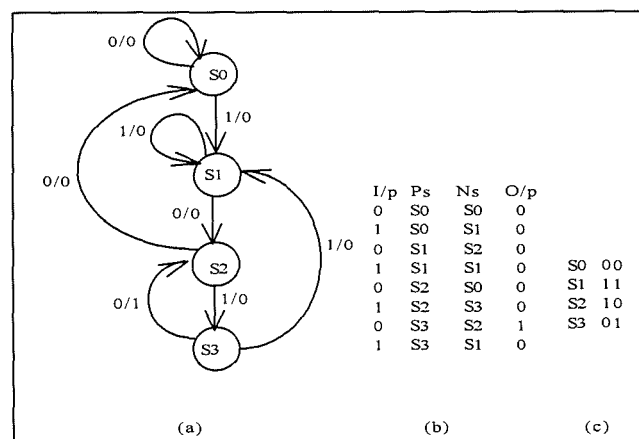


Figure 5.21. 1010 Detector a) state transition graph, b) state transition table, c) state assignment.

Table 5.8. Solution obtained for 1010 detector produced using proposed method and manual design.

Proposal approach 2001	Almaini, 1994 [33]
$D_A = X \bar{B} + A$	$D_A = \bar{X} \bar{A} B + \bar{X} A \bar{B} + XAB$
$D_B = X$	$D_B = \bar{A} B + A \bar{B} + X \bar{B}$
$Z = \bar{X} \bar{A} B$	$Z = \bar{X} A \bar{B}$
Subcircuit A =2	Subcircuit A =12
Subcircuit B =3	Subcircuit B =2
Subcircuit C= 2 D flip-flops	Subcircuit C= 2 D flip-flops

The solution reported in [33] uses almost 3 times more gates than the circuit produced by the proposed approach. It is interesting to note that the outputs are implemented in the same way for both circuits. The evolved circuit requires 5 logic gates in comparison with 14 logic gates for manual design based on random assignment (00,01,10,11). Note that the circuit was actually evolved by separating each subcircuit and evolving each subcircuit separately to obtain the target circuit. The probability of the cell mutation rate is 0.05, the population size is 20 and the number of generations is 5000 with 1 x 10 circuit layout.

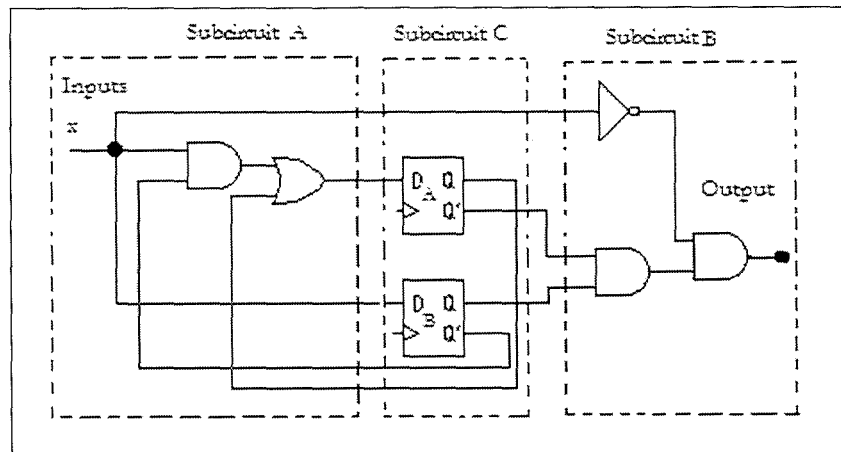


Figure 5.22. Evolved optimal circuit solution for 1010 detector using 4x5-circuit layout.



## 5.8 Analysis of the results

In this chapter we have put forward the view that evolutionary algorithms together with the assemble-and-test methodology can be regarded as a discovery engine or creative machine for new designs. We studied this idea in the context of different examples of sequential logic circuits. From results it can be seen that unusual structure may be able to be discovered by examining a series of evolved designs. We examined the concept of the space of all circuit representations but feel that similar ideas may well carry over to the general field of design. The human designed algebras, forms a subsets of the space of all representations employing evolutionary techniques.

The new proposed approach is based on using GA to generate optimised state assignment and EHW to design the logic circuit. All results have been discussed in the pervious examples used these methods. The results given compared with manual designed method. The comparison results are based on the number of logic gates at gate level. In term of Boolean gates, the evolved circuits use number of gates given in Table 5.2. Nevertheless, the experiments shows that the amount of hardware (gates) can be reduced compared with manual design method.

The results of some of the experiments showed that it is possible to improve the evolutionary algorithm performance considerably by choosing carefully the number of columns of cells and the number of rows as discussed in example 2.

In the experimental results the different population size is used and consider the sensitivity of EHW approach to this parameter is discussed. The number of generations also grows with the size of the circuit. The most significant results have been obtained by analysing the circuit structure. We showed that combinations of different logic gates allowed us to evolve more optimal circuit structures. Moreover because our algorithm is based on the state table analysis the GA finds the substitutions of logic operators, which is impossible

to define using well-known algebra logic rules. We contend that evolving the functionality and connectivity of logic gates can only discover such strange non-algebraic circuits.

Table 5.9. Initial parameters used to evolve the circuits and their results.

Circuit	Example1	Example 2	Example 3	Example 4
Circuit layout	3x10	1x10	4x5	1x10
#Population size	5	10	10	20
#Generation	5000	5000	5000	5000
Crossover	0.5	0.5	0.5	0.5
Mutation	0.06	0.06	0.06	0.06
#Run	10	10	10	10
100%functionality	3	2	2	3
#Active gate	15	5	8	5

### **Evolving logic circuit using criteria F1 and F2**

In this section we will consider some experimental results obtained for the sequence detector. The main idea of this experiment is to define whether using different optimization criteria during the second stage of dynamic fitness function affect the evolved circuit structure and algorithm performance or not. The initial data of evolutionary algorithm for this experiment are given in Table 5.9. Only primitive logic gates of one-and two inputs are allowed to participate in evolution. Each gate is considered as a separate chromosome element. The obtained experimental results are summarised in Table 5.10. As mentioned in the previous section that the number of primitive logic gates in the circuit defines the quality of evolved circuits.

Table 5.10. Sequence detector experimental results the algorithm performance during the circuit layout and functionality distributions with variable circuit structure and Function set: 2,6,7,8.

#Run	1	2	3	4	5	6	7	8	9	10
F1	50	62.5	75	62.5	65	75	100	75	75	100
F2	3	9	3	5	7	6	8	4	6	9
Circuit Geometry	4x1	10x1	4x1	8x1	9x1	6x1	4x5	7x1	6x1	4x5

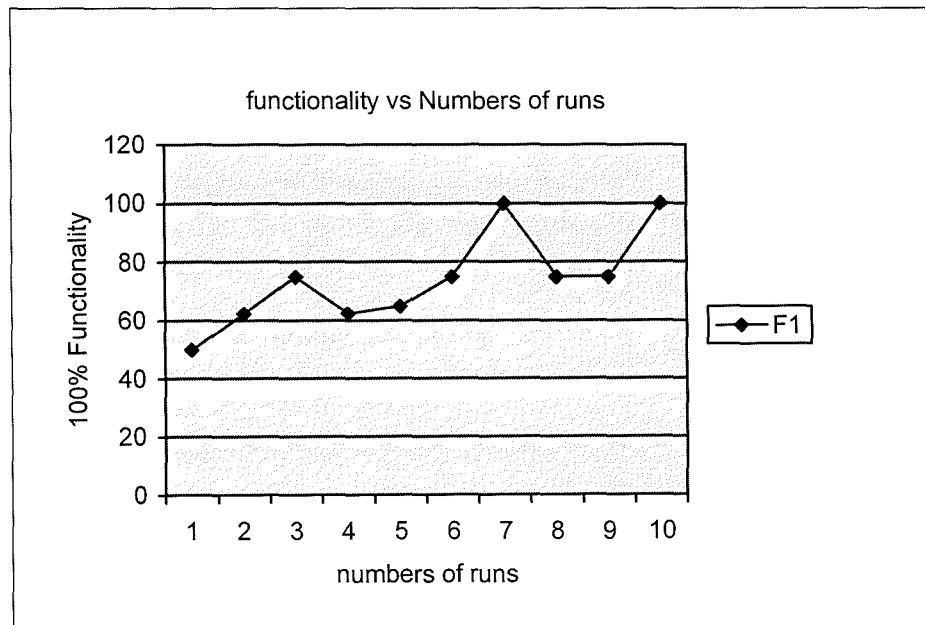


Figure 5.23. Fitness distribution in 10 runs

Justification of evolved circuit layout together with circuit functionality can be derived empirically from the results reported in the Table 5.10 and plotted in Figure 5.23. It can be seen from the result how the connectivity parameters and circuit layout affect the algorithm performance. The layout is characterized by the following parameters:

1. Connectivity parameters;
2. The numbers of rows and columns in rectangular array.

Each results has different circuit layout parameters. The experiments have yielded some very interesting results that performance may depend on the number of columns and connectivity parameters rather than evolutionary algorithm only. In this case the algorithm performance is defined by number of fully function circuit evolved during 10 runs and the quality of evolved circuits. Moreover the experiment investigates how the numbers of columns in rectangular array influence the algorithm performance.

It can be seen that the number of logic gates involved in the circuit increases as the number of columns in the rectangular array is increased (see Figure 5.24). Therefore, in order to evolve logic circuits with minimum number of logic gates, a suitable number of columns must be chosen.

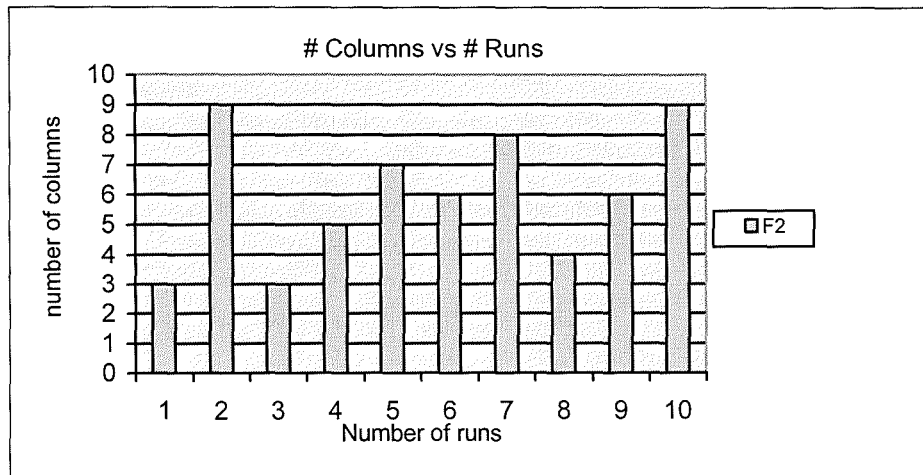


Figure 5.24. The graphic illustration how the circuit layout and evolved functionality affect the solutions of sequence detector.

The experiments shows how the fitness function with different optimisation affect the algorithm performance and the quality of evolved circuits. The percentage of correct bits F1 and the number of active primitive logic gates in circuits, F2 have been chosen as the criteria to compare the algorithm performance. It is clear that F2 criteria have to be minimized during the evolutionary process when F1 criteria is maximised.

Table 5.11. Module-4 counters with fixed circuit geometry and Function set: 2-8,12.

#Run	0	1	2	3	4	5	6	7	8	9
F1 %	100	100	100	100	100	100	100	100	100	100
F2	7	8	6	7	6	5	6	5	4	4
Circuit Geometry	1x10	1x10	1x10	1x10	1x10	1x10	1x10	1x10	1x10	1x10
Time (sec)	23	41	39	39	39	38	38	38	39	39

Table 5.11 shows a summary of the experimental results received for algorithms with the dynamic fitness function for Module 4-counter. The numbers of columns and rows in rectangular array have been chosen to be constant. The quality of circuit evolved with fixed circuit geometry comparing with 100% function circuit evolved (F1). It can be noticed that the number of active logic gates is not fixed. This proves that there is no fixed circuit when we use evolution algorithm. The experimental results show that in terms of the 100% cases evolved the chromosome representation with flexible and permanent circuit geometry give similar results. Analysing the structure of circuits evolved allows us to make the conclusion that proportion of the number of available gates in the circuit to the number of actual gates becomes fixed with increasing numbers of available gates in circuit

The CPU times of model 4-counter required for ten runs are presented in the Table 5.11. In general, the time required for each run tends to increase as we solve more complex circuit.

In this experiment some issues concerning analysis and verification of evolved circuits at gate-level EHW have been considered. Firstly, some function representation and their suitability for extrinsic EHW have been examined. Secondary, the dynamic fitness function strategy has been introduced to improve the quality of evolved circuit. Two criteria to define the functionality of the logic function are introduced. These are:

- The type of primitive logic function describing the behaviour of building block (AND, OR, NOT);
- The type of input used in the building block (primary or inverted).

The experimental results show that these two functionality types have different influence the algorithm performance.

### 5.9 Summary

The chapter presented a new synthesis method which provide for both GA to define optimal state assignment and EHW to design the combinational part of sequential logic circuits. The EHW approach described is based on the extrinsic evolution at gate-level, in the sense that each gene of a chromosome corresponds to a primitive logic gate. The problem of how to evolve a sequential logic circuit that performs a desired function (specified by state table) in a given set of available logic gates has been discussed. Most of the previous work is summarised in Table 5.1. The extrinsic evolution EHW is used to evolve sequential logic circuit and to our knowledge no similar work has been published in this area to compare with. The experimental results obtained are compared with results produced by the manual design methods. The minimum numbers of AND, OR, NOT logic gates in the combinational block of the circuit is the criteria set by the user to choose the optimal solution. The state assignment of state machine is often critical and a small change in the codes assigned to the state can lead to very wide difference in the number of logic gates and in the topological structure of that logic. The work shows how an evolutionary algorithm could be used to produce a novel and efficient design for digital logic circuit some time difficult for human design to expect because of its unusual structure. As it has been verified through the presented EHW approach implementations, the proposed approach can be successfully applied to the design of the sequential logic circuits. Further we believe that the GA-based approach has a great potential to provide a practical tool for assisting designers of logic circuits.

## EXTRINSIC EVOLUTION OF FINITE STATE MACHINES

### **6.1 Introduction**

The chapter outlines the use of the extrinsic evolvable hardware approach to evolve MCNC finite state machine benchmarks. The approach introduced in chapter 5 is tested on a number of finite state machines from MCNC benchmark set. These circuits have been evolved using different functional sets of logic gates and GA parameters. The results show promise for the use of this approach as a design method for sequential logic circuits. The approach is divided into two stages using GA as encoding scheme, which described in chapter 4 and EHW to design general FSMs from MCNC benchmark set [1]. The developed approach is the first attempt to evolve sequential logic circuits from the standard benchmark set.

### **6.2 Evolution of MCNC FSMs**

We are ready to complete the FSM design processing introduced in Chapter 5. In order to evolve the MCNC FSMs benchmarks, two stages are combined. The first stage depends on the state table to represent the chromosome for state assignment. The second stage depends on the layout structure of the circuit at the functional level. The architecture of the genetic synthesis of FSM benchmark set of sequential logic circuits is similar to one described in chapter 5 (see figure 5.7). In chapter 5, small size sequential circuits were evolved and compared with manual design methods. Here the main design processing is the same but we try to evolve more general circuits, large size of MCNC benchmark sets. Therefore a third stage is added to Figure 6.1 labelled as stage 2. In this stage the symbolic state minimization is used to produce the smallest possible circuit. This involves examination of the symbolic state

transition table to see if there are any equivalent states. Equivalent states are those, which have the same output and next state transition, and these can be merged to minimise the numbers of states.

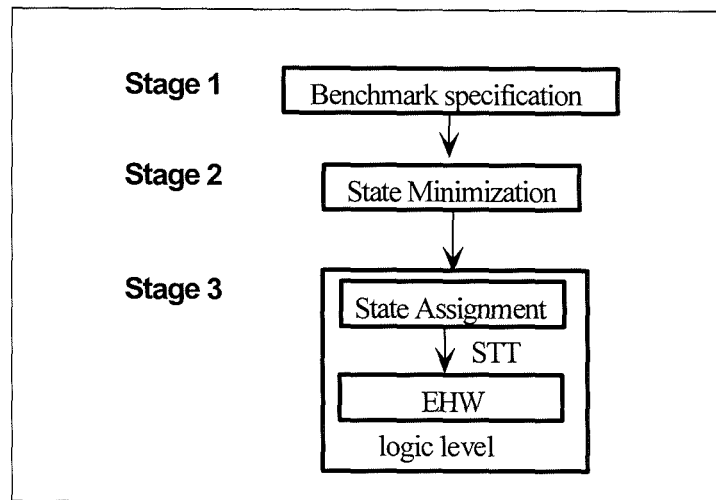


Figure 6.1. Procedure of the proposed approach to design FSM circuit using genetic algorithm and extrinsic evolvable hardware

The evolutionary algorithm representation had been illustrated in chapter 5, where we saw that an important task in any evolutionary experiment is to be able to define a genotype-phenotype mapping. In this case, the phenotypes were made up of one or more of the cells shown in Figure 6.2.

As can be seen from Figure 6.2, an integer representation is being employed. In this particular example, a predefined structure composed of layers must be evolved. In addition, only 2-input gates are being used. The total number of cells is a representation parameter. In the above figure, there are eight inputs, from  $i_0$  to  $i_7$ . These inputs may come from two different sources: external inputs applied to the circuit, feedback coming from the D flip-flops. The gate level representation has been used to encode each circuit into an integer string as described in section 5.6.1. The circuit (phenotype) is constituted by a combinational part (arrangement of logic gates) and a sequential part (D flip-flop).



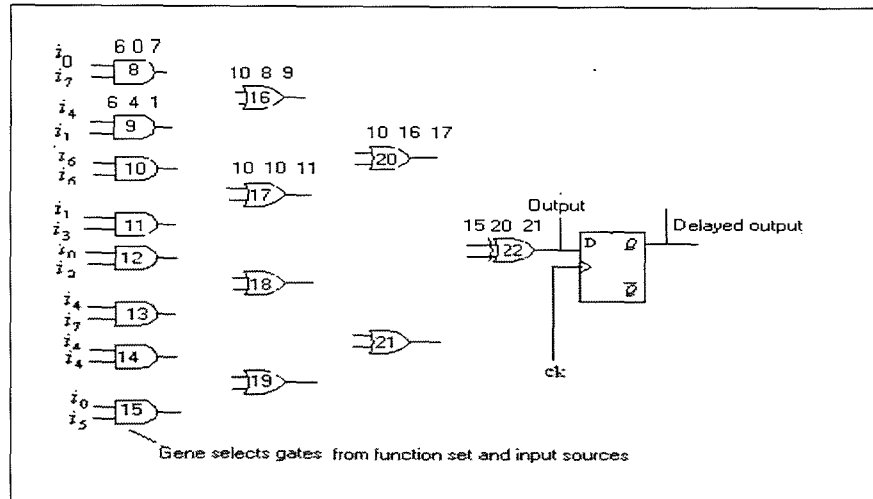


Figure 6.2. Gate level representation of a sequential circuit.

This representation provides the following interesting feature:

- Both combinational and sequential circuits can be encoded. In the case of pure combinational circuits, the D flip-flop is not considered.
- There is no feedback path in the combinational parts of the cell,
- It is a very flexible representation, allowing the implementation of a wide variety of combinational and sequential functions. The sum of products representation is used. Allows this scheme to also include the possibility of using XOR logic gates.
- The user may change the depth (number of columns and rows) of the circuit. In Figure 6.2 there are an 8x4 array of logic cells between eight required inputs and one output. Depending on the particular application, it may be desirable to use only two or three level, due to speed requirements.
- The total number of cells that makeup the circuits can be changed by circuit layout structure.

### 6.3 EHW to design the combinational part of the circuit

For efficiency, a simple tabular representation for the FSMs is chosen. Rows in a table correspond to states, and columns correspond to inputs. This circuit layout of FSMs is represented as a rectangular array of components. These components are uncommitted and can be removed from the actual circuit design if they prove to be redundant.

#### Genetic operators

The circuit evolution is performed using a rudimentary  $(\lambda + 1)$  evolutionary strategy with uniform mutation [24]. The population consists of  $(\lambda + 1)$  genotype. Initially the elements of population are chosen at random. Once the fitness values of the genotype are evaluated a mechanism of population update is applied. The algorithm is as follows:

**Step 1** Randomly initialise a population of genotype

**Step 2** Evaluate fitness of genotype,

**Step 3** Copy fittest genotype into new population;

**Step 4** Fill remaining places in the population by mutated version of fittest genotype,

**Step 5** Replace old population by new and return to step 2 unless stopping criterion is reached.

Once this genotype phenotype mapping has been identified, the next step is to be able to identify a suitable fitness function. In this case, the functionality of the FSM is very well defined [1]. Thus the fitness function can be very precise, essentially testing to see whether or not the desired outputs are generated and if not, how many correct desired outputs are generated and awarding a score. We use dynamic fitness strategy disused in chapter 5, first we are trying to find the circuit with 100% functionality ( $F_1$ ) and second we

are trying to minimise the number of active gates in functionally completed circuit ( $F_2$ ).  $F_2$  Adds reward for the 100% functionally correct circuits that have minimum number of logic gates.

The parameter circuit mutation is used to change the type of genes in chromosome excluding the number of columns and rows. The mutation rate defines how many genes in the population are involved in mutation.

#### 6.4 Motivating Example

The proposed approach described in Figure 6.2 is tested against dk27 benchmark with seven states (S0, S1, S2, S3, S4, S5, S6), one input and two outputs. The experimental plan is outlined and results are given. A number of experiments have been carried out in order to investigate the specific features of the proposed method.

The initial data for the experiment are given in Table 6.1 for both GA state assignment and extrinsic EHW. The benchmark is given to the system in a file containing the objective state table in the form of a programmable logic array file (PLA). The structure of dk27 circuit in the proposed approach contains 3 subcircuits. The combinational part of the benchmark circuit is decomposed into subcircuits A for next state combinational logic and B for outputs combinational logic. Subcircuit C represents the D flip-flops. Each subcircuit has been evolved separately using the extrinsic EHW approach.

Table 6.1. Initial parameter used to evolve sequential logic circuit (dk27.kiss2)

Problem	State assignment (GA)	Combinational logic design (EHW)
Population size	20	15
The number of generations	100	5000
The number of GA runs	10	100
Type of crossover	Two-point	-
Crossover rate	0.25	-
Mutation rate	0.015	0.05
The number of rows	-	4, 8
The number of columns	-	3, 8
Target function	dk27.kiss2	dk27.kiss2

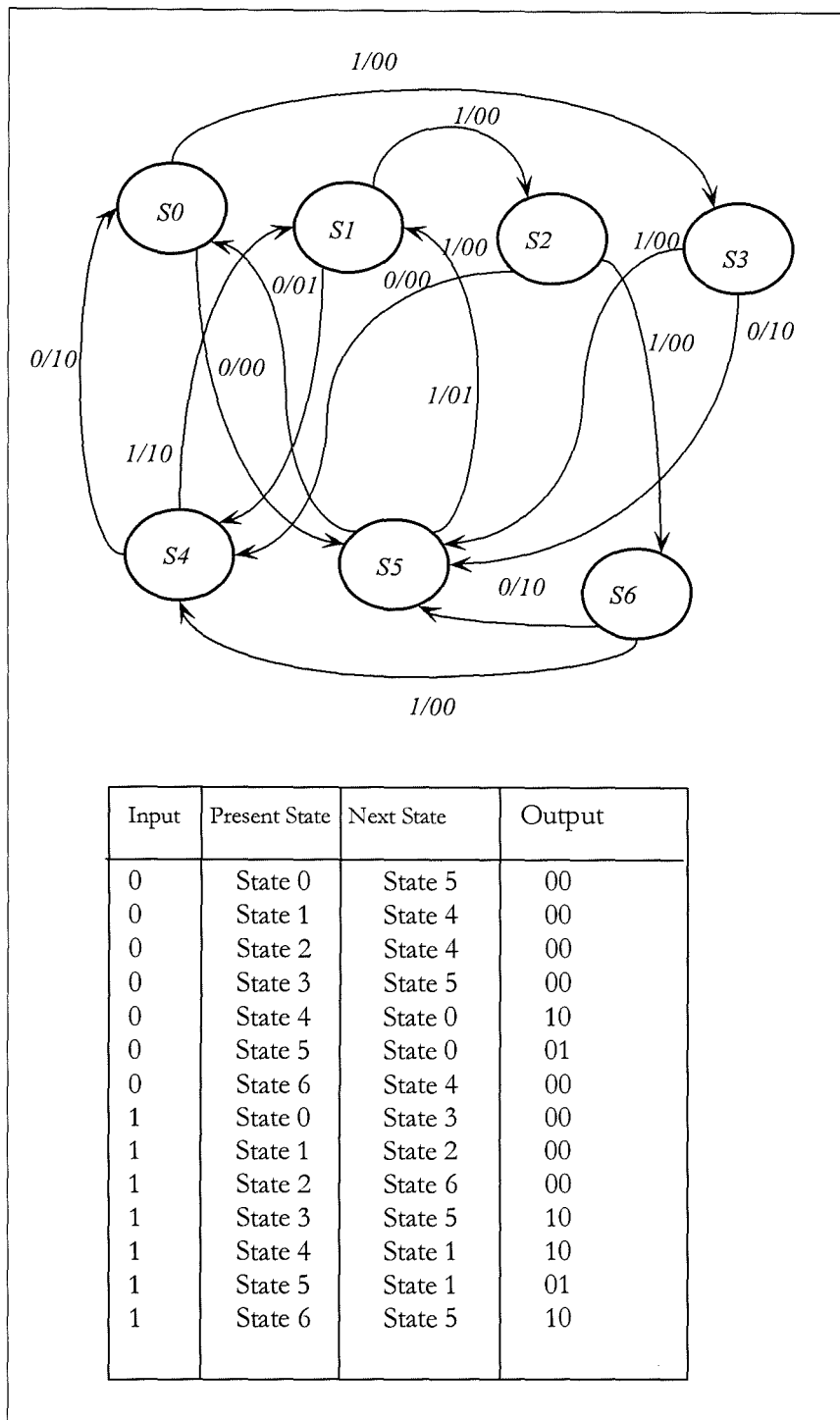


Figure 6.3. DK27 State Diagram and symbolic state transition table

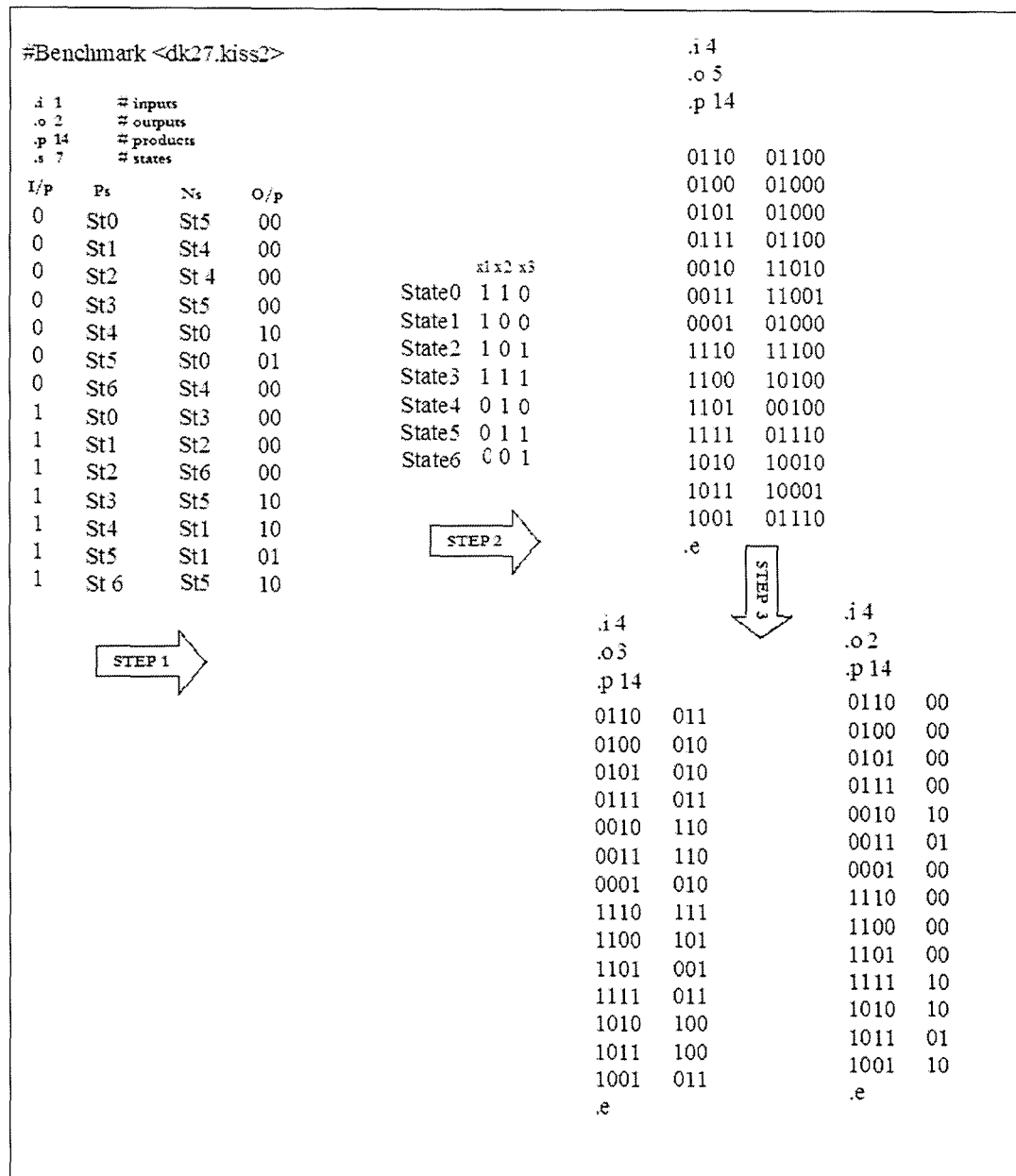


Figure 6.4. The procedure of generation the \*.pla file from the state transition table based on example of dk27 (Kiss2 benchmark). Step1 shows the initial symbolic State Transition Table, Step2 generates State Assignments using the genetic algorithm and Step 3 generates the PLA files (\*.pla) based on the state assignments obtained.

In Figure 6.4, step1 shows the valid encoding for the benchmark by simply replacing the symbols of the states in the SST by the respective state binary code generated by GA. Step 2 generates the state assignment. Step 3 partitions the STT of the benchmark circuit into next state combinational logic circuit A and output combinational logic circuit B. Once the EHW decomposition is completed, the fully functional circuit can be generated.

An example of chromosome representation with actual circuit structure is given in Figure 6.5 and Figure 6.6. These circuits represent a dk27A next state combinational logic and dk27B outputs combinational logic evolved using functional set in Table 5.2. The function circuit of dk27A has 4 inputs, 3 outputs and dk27B has 4 inputs, 2 outputs both implemented here on a combinational network with 3x4 circuit layout ( $N_{col} \times N_{row}$ ). The labels of dk27A circuit inputs 0,1,2,3,4 corresponding to the inputs variables  $x_0, x_1, x_2$  and  $x_3$  respectively. Each gates is assigned an individual address. Thus the logic gates located in 0<sup>th</sup> column and 0<sup>th</sup> row is labelled as 4 in Figure 6.5. The logic gate located in 3<sup>th</sup> column 1<sup>st</sup> row is labelled as 14. Each output of a logic gate is labelled with a real number. The integer part of this number defines the code of logic gate and the functional part determines the position of output in logic gate. The number of outputs in the logic function implemented defines the number of circuit outputs. The outputs of circuit are connected to the outputs of logic gates 10, 14 and 15 for dk27A. These integer values, whilst denoting the physical location of each input, gate or output within the structure, now also represent connections or routes between the various points. The chromosomes is merely a netlist of these integer value, where the position on the list tells us the gate or output which is being referred to, while the value tell us the connection to which that point is connected, and the gate functionality. Table 6.2 shows four chromosomes with 100% evolved circuit functionality and the numbers of gates required are listed.

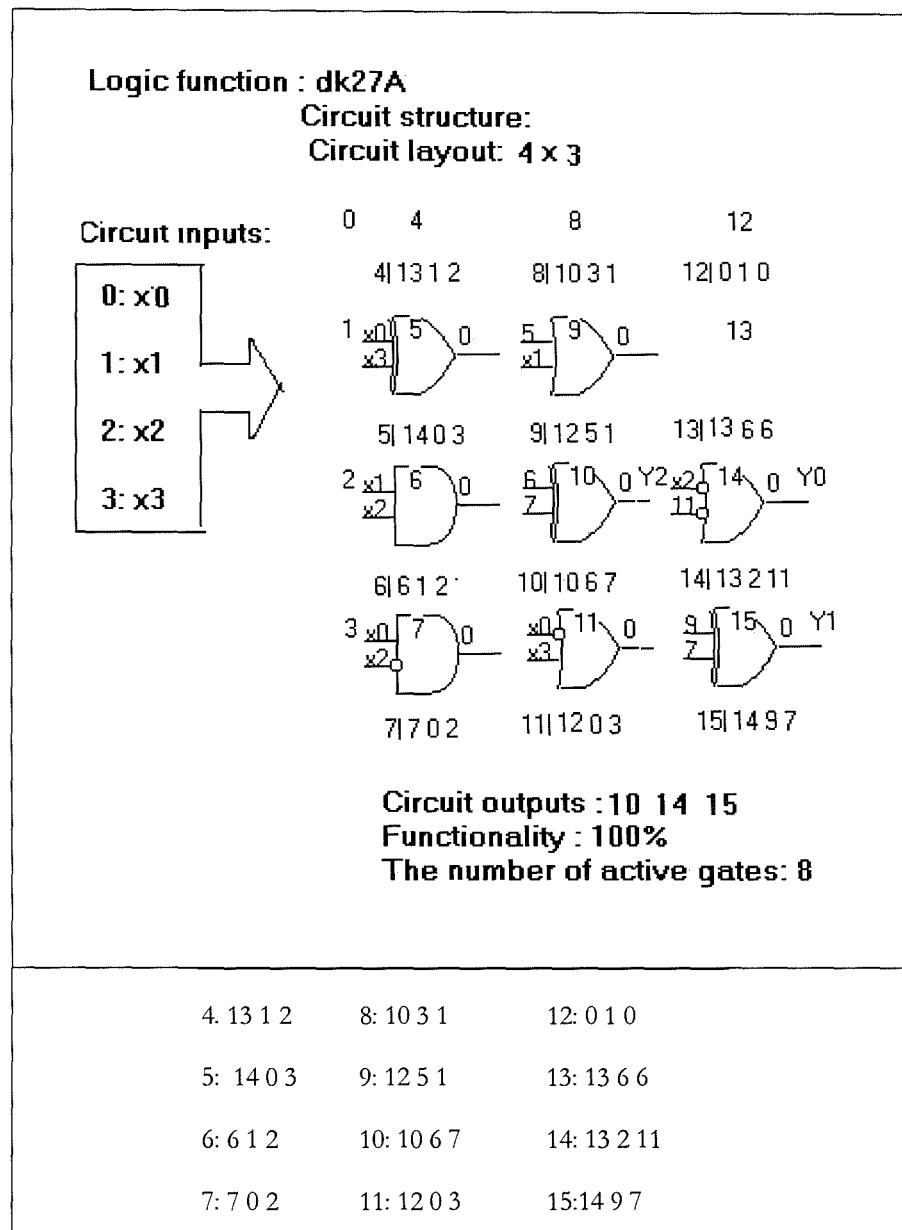


Figure 6.5. An example of the gate array representation and corresponding genotype of a chromosome with 4x3 circuit layout. Functional set (0-15).



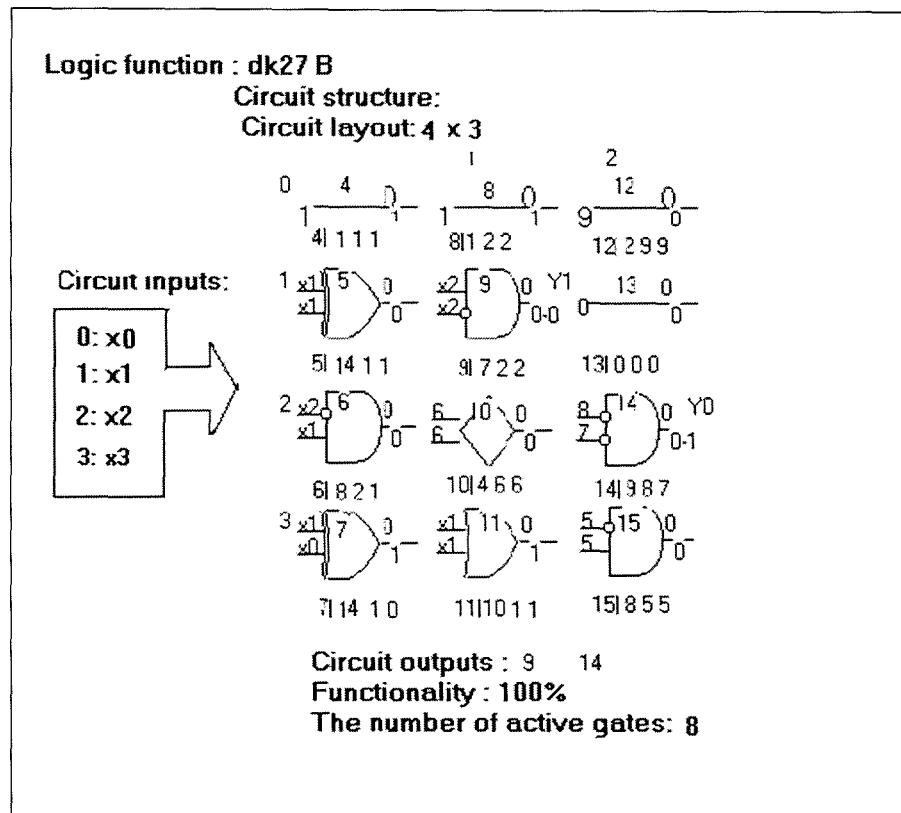


Figure 6.6. An example of the phenotype and corresponding genotype of a chromosome with 4x3 circuit layout. Functional set (0-15).

Table 6.2. A typical netlist chromosome for 100% functional of dk27A next state logic.

#Generation	# Gates	Chromosome
70	12	1 1 1 2 10 1 1 1 8 2 1 2 10 1 0 11 2 2 7 2 2 14 6 6 12 1 1 7 9 5 0 0 0 0 9 8 7 8 5 4
90	10	13 1 6 3 0 0 2 0 3 3 3 1 3 3 12 15 0 2 12 3 2 2 2 1 12 3 2 1 12 2 3 0 1 2
500	13	13 1 2 10 0 3 6 1 2 7 0 2 10 3 1 12 5 1 10 6 7 14 0 3 4 1 1 10 7 9 0 1 0 13 6 6 12 3 0
5000	9	13 1 2 10 3 1 6 1 2 7 0 2 12 5 1 10 6 7 14 9 7 13 2 1 12 0 3

## Evolving circuits using different function set

To see how choosing the functional set of logic gates affects the evolved circuit structures the obtained experimental results are shown in Figure 6.7 (a) and Figure 6.7(b). In Figure 6.7(a), the circuit has been evolved using functional set (0-10) and the circuit consists of 11 gates in sub-circuit A, 10 gates in sub-circuit B and 3 D flip-flops in sub-circuit C. The total number of logic gates in assembled circuit is 21 (12 AND, 7 OR, 2 NOT). Figure 6.7 (b) shows the circuit evolved using functional set (0-15). The most efficient evolved circuit consists of 10 logic gates in sub-circuit A, 5 gates in sub-circuit B and 3 D flip-flops. The total number of logic gates in the circuit is 15 (5 AND, 3 OR, 4 XOR, 3 NOT). The two circuits discussed above illustrate how choosing the functional set of logic gates affects the evolved circuit structures. The functional genes are encoded according to Table 5.2.

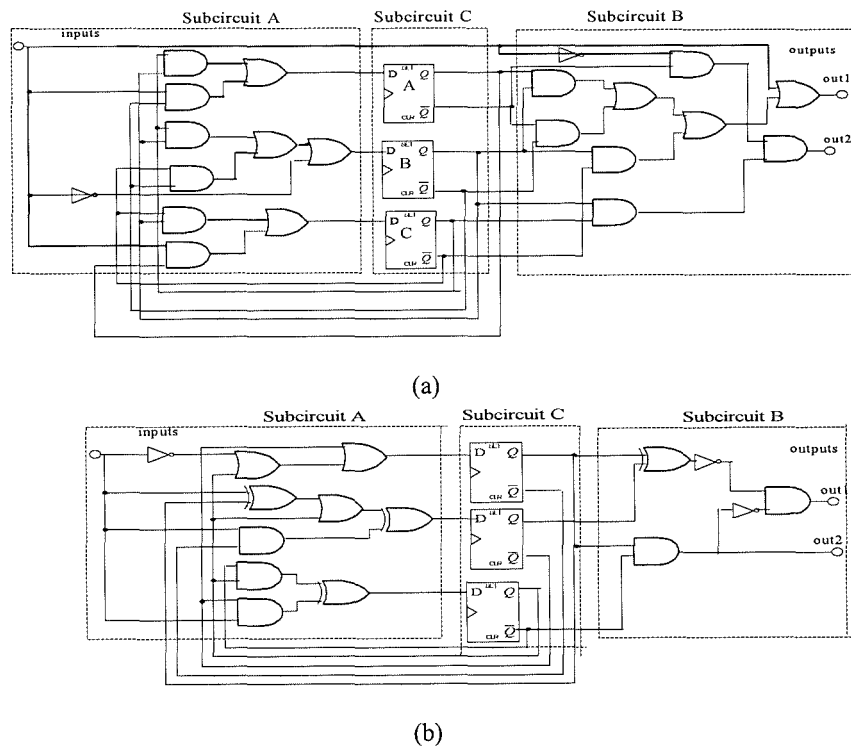


Figure 6.7. Evolved dk27 design using (a) functional set (0-10) (b) functional set (0-15)

It should be noted that after a chromosome has been obtained it is subject to analysis and any redundantly gate are removed. This may occur when gates are not involved in a circuit or connected to an output or when because of a certain input a gate is made redundant as in case of an input of 0 to an AND gate or an input of 1 to an OR gate.

## 6.5 Experimental results

In this section the experimental results are presented for a set of machine chosen from MCNC and literature benchmarks [116]. Table 6.3 shows the state assignment generated by GA and used to design the circuit.

Table 6.3. State assignments generated by GA.

FSM	#State	State assignment
bbara	10	2,3,5,4,7,8,9,0,1,11
beecount	7	7,6,1,3,4,0,2
dk15	4	0,2,1,3
dk16	27	12,8,1,27,13,28,14,29,0,16,26,9,2,4,3,10,11, 17,24,5,18,7,21,25,6,20,19
dk27	7	6,1,5,7,4,3,0
ex3	10	3,15,7,1,13,11,2,6,10,8
ex5	9	8,13,11,10,4,9,0,12,2
lion9	9	1,0,4,6,7,5,3,1,11
mark1	15	4,3,15,14,9,12,7,2,1,0,10,13,8,5,6
opus	12	11,14,4,3,10,12,7,2,8,15,6,0
Sse	16	6,15,14,10,8,0,2,4,12,7,9,11,3,1,5,13
Shiftreg	8	6,2,4,0,7,3,5,1
train11	11	8,13,11,10,4,9,0,12,2,1,3

The state assignment of a state machine is often critical and a small change in the codes assigned to the state can lead to very wide difference in the number of logic gates and in the topological structure of that circuit.

However EHW begins from randomly connected and randomly chosen logic gates and gradually evolves the target functionality. The evolutionary algorithm does not guarantee that 100% functionality circuit of the resulting connections will be achieved in all cases. So, the results reported here are the averages of the correct solutions from 100 runs. In this section, some experimental results obtained for the MCNC benchmark circuits are given.

In order to evolve better circuit in terms of the number of active logic gates, we use two stages. At First stage, the objective in digital evolution behaviour is to merely produce a 100% functionality correct circuit (F1). Second stage, continues evolving the circuit to optimise number of active logic gates in the circuit (F2). The approach performance can be estimated by:

1. The mean functionality fitness of best chromosome over 100 runs (Av.F1).
2. The mean numbers of active gates in fully function design evolve over 100 runs (Av.100F2), #100% cases is the number of fully functional circuits evolved.

The experimental results obtained are summarised in Tables 6.4. Each benchmark circuit has been performed 100 times. The function sets of logic gates are chosen according to encoding in Table 5.2. It can be seen from the table that large FSM benchmarks are difficult to evolve.

We have discussed the possible ways to improve the quality of evolved circuits. The choice of suitable circuit geometry is a very complicated task and is intimately linked with the complexity of the function implemented. So, in order to avoid this we have investigated the possibility of evolving the circuit

layout at the same time as trying to evolve the fully functional circuits. The circuit geometry defines the length of the chromosome, thus we worked with chromosomes of variable length. The main purpose of circuit layout evolution was to try to evolve the best circuit layout together with evolving circuit functionality.

Table 6.4. Experimental results of extrinsic EHW approach.

Benchmark circuit	Function set	Av.F1	Av.100F2	#100 cases	$t_{\text{EHW}}$ (S)
bbara	0,1, 2, 3, 4, 5, 6, 10,15	93.437	60	7	660
bbtas	2, 3, 4, 5, 6,7, 10,11,15,16	99.078	15	24	480
dk15	0,1, 6,7,10,11,13,14,15,16	90.530	53	11	300
dk16	0,1,2,3,4,5, 6,7,8,10,11,13,14,15,	89.00	265	1	3180
dk27	0,1, 2, 3, 4, 5, 6, 10,15	98.875	20	28	420
dk512	2, 3, 4, 5, 6, 9,10,11,13,14,16	93.421	25	31	484
donefile	2, 3, 4, 5, 6, 10,11,12,13,14,15	94.0625	105	2	410
lion9	0,1, 2, 3, 4, 5, 6, 10,15	94.7500	50	7	100
modulo12	0,1, 2, 3, 4, 5, 6,7,8,9, 10,11,16	98.678	15	36	360
shiftreg	0,1, 2, 3, 4, 5, 6, 10,11,12,13,15	97.531	18	21	660
tav	0,1, 2, 3, 4, 5, 6, 10,15,16	95.743	29	19	480

In the table we show the measure of computing CPU time  $t_{\text{EHW}}$ . The result shows that for the benchmark with small numbers of states the EHW required significantly less CPU time. The  $t_{\text{EHW}}$  run on a 450MHZ PC 128MBRAM for all the benchmark appears in the literature

The experimental results obtained are summarised in Table 6.5. The table shows the numbers of gates used to evolve each subcircuit after 100 runs. The particular set of logic gates used is fixed in advance, but whether or not any particular gate is used, or how many times a gate is used, is entirely free. The advantage of this approach is that it allows us to synthesise the benchmark circuits using any set of logic gates. Consequently, it permits the synthesis of compact and unusual circuit structures. The number of logic gates in the circuit defines the quality of evolved circuits.

It can be seen from Table 6.5 that large FSM benchmarks (dk16) are difficult to evolve with one valid solution after 100 runs. These benchmark sets results are compared against SIS [73] for sequential logic synthesis and optimisation. The inputs to SIS are given in state table format and the library is given in genlib format. The circuit was first optimized using script.rugged, which performs combinational optimization on the network. The optimized circuit was mapped with a library consisting of 2-input gates and inverters. The sequential redundancy removal algorithm was run on the mapped circuit. The output is a netlist of gates for the target technology. It can be seen that in some cases the evolved circuits are much better than the circuits generated by SIS.

EHW is no doubt a very promising technique for solving optimization problems with a tradeoff between speed and perfection. However, not all problems lend themselves very well to a solution with EHW. In case of problems involving design of benchmark consisting of more than 30 states, the barrier to applications of the EHW technique is that the possible combinations' are too vast to search using a conventional genetic algorithm.

Table 6.5. Experimental results of extrinsic EHW approach. #in, #out and #stat are the number of inputs, outputs and states respectively. #100 cases is the number of fully functional solutions obtained after 100 runs of GA. The evolved circuits which are more optimal in comparison with SIS [73] are shown in bold.

Benchmark. Kiss	Specification			Functional set	Estimation of the best				#100 cases	SIS [73]
	#in	#out	#stat		Sub-circuit A	Sub-circuit B	Sub-circuit C	Total		
bbara	4	2	10	0-5, 6, 10,15	32	28	3	<b>60</b>	7	79
bbtas	2	2	6	2-7, 10, 11, 15, 16	15	4	3	<b>19</b>	24	28
dk15	3	5	4	0,1, 6, 7, 10, 11-16	20	33	2	<b>53</b>	11	66
dk16	2	3	27	0-6, 8, 10, 11, 13, 14, 15	265	40	5	305	1	285
dk27	1	2	7	0- 6, 10, 15	11	5	3	<b>16</b>	28	20
dk512	1	3	14	2-6, 9-14, 16	25	22	4	<b>47</b>	31	58
Lion9	2	1	9	0-6, 10,15	29	21	4	50	7	35
shiftreg	1	1	8	0-13,15,13		5	3	<b>18</b>	21	19
tav	4	4	4	0-6, 10,15,16	3	23	2	<b>26</b>	19	29
Total number of gates					413	181	29	<b>594</b>		<b>619</b>

In the experiments reported here, the number of logic gates chosen is equal the number required to build conventional circuit over 100 runs. The evolved circuit use D flip-flops, which described as subcircuit C in the Table 6.5, meaning that evolutionary circuit minimize number of the D flip-flops in the

synthesis of state assignment. It may be concluded that the result found by this approach are at least as good as CAD tools, but in some case better than those derived by the available methods in small size benchmarks. The advantage of this approach is that it allows us to synthesise the benchmark circuits using any set of logic gates. Consequently, it permits the synthesis of compact and unusual circuit structures.

## **6.6 Summary**

This chapter proposes a new approach to evolve FSM benchmark circuits and consider circuit structures designed by applying different function set of logic gates. The basic idea of this approach is to use the strength of an evolutionary algorithm at both state assignment and circuit design stages. The standard genetic algorithm has been used in order to identify the optimal state assignment for the given problem. The extrinsic evolvable hardware with rudimentary evolutionary strategy has been applied to synthesise the combinational parts of FSMs. Former results are associated with an evolutionary process in which each evolved FSM benchmark circuit is built and tested in software using computer simulations. The implemented EHW is able to design logic circuits with size and complexity, which have not been demonstrated in published work so far on structural genetic and evolutionary algorithms. This automated approach has the added advantage of reduced dependency on the designer's knowledge and experience. Finally it can be concluded that because we produce valid logic circuits by merely connecting logic gates together and then testing to see if the resulting circuits is correct it becomes possible to design circuits which lie outside the space of circuits produced by ECAD tools. The method allows the synthesis of very novel circuit structures, which have never been seen before.



## CONCLUSIONS AND FURTHER WORK

### 7.1 Conclusions

The principle objectives of this thesis have been fulfilled. A novel technique, using Evolutionary Algorithm is implemented to design combinational and sequential digital circuits. The technique provides improvement over existing tools for circuit design. This automated approach has the added advantage of reduced dependency on the designer's knowledge and experience.

The sequential circuit design methodology has been divided into two main parts.

In the first part of our design process the GA has been used for optimised state assignment. The work has tended to look for assignment that can be automated and give optimum result. Using genetic algorithm, all assignments may be considered as a search space of potential solutions.

In this work a set of methods for assigning code to FSMs is presented. The main concern has been the encoding of internal states of sequential circuit with a view to minimizing circuit complexity.

Chapter 2 includes literature survey and summary of previous work done on sequential logic synthesis and the methods for encoding the internal states of FSM. The optimal state assignment problem has been the object of intense research for many years but no completely satisfactory solution has been found. This problem has been proven to be NP-complete. Thus, all methods proposed to solve it are heuristic approaches as discussed in chapter 2.

Synthesis based on a GA allows a designer to minimise the actual area while performing state assignment. These GAs compensate for most of the

unpredictability inherent in the logic reduction synthesis tools, providing a better measure of the circuit's characteristics.

The use of genetic algorithms is proposed to solve the state assignment problem in chapter 4. Given the huge search space and the existence of many local minima, this problem is well suited to genetic algorithms.

In addition results are presented to show that this approach can achieve solutions superior to previous methods. And the observations show that the choice of the GA parameters is a very important issue.

A program using the genetic algorithm to find the optimum state assignment for finite state machines has been written and it can produce a fairly good assignment within a fraction of the time required by other methods. Through evolutionary operations of recombination, mutation, and selection new generations of search points are found that show a higher average fitness than their ancestors do.

So, there is a difference between finding a good solution and an optimised solution. A FSMs state assignment can significantly affect the quality of the synthesized circuit. This observation is significant since the state assignment is relatively independent of other optimisations used during the synthesis process. Much of the state assignment research has been concentrated on reducing the circuit area. They select state encoding that leads to simpler equations and therefore smaller area designs.

The results reported in chapter 4 show that the state assignments as found by the genetic algorithm are at least as good, but in most case better, than derived by NOVA and similar techniques. In all benchmark tested in Tables 4.13 and 4.14 the GA assignment produce better results compared to NOVA. Experimental results For the 17 benchmarks tested showed that the GA could generate state assignments, which required on average 15.44% fewer gates and 13.47% fewer literals compared with NOVA. The experiment with

FSMs of various sizes show that the optimum parameters do not change significantly from one machine to the next. It has been demonstrated that the GA is a valid method of finding a good state assignment; it is competitive with commercial software, and is not dependent on any particular feature of the sequential machine.

In the second part of the thesis, extrinsic evolvable hardware has been proposed as a new method for designing circuits for complex real world applications. This work highlights some of the reasons that can explain why EHW has not yet been widely applied.

The idea of evolvable hardware is to use evolutionary techniques to develop new electronic circuits. There are at least three motivations for EHW:

When tackling a problem a human designer tries to split the initial complex problem in to many simpler elementary tasks to find basic building blocks that can be used in the design. The designer can then assemble those elements to build more and more complex designs. One of the limitations of this approach is that there might be a lot of redundancy between those basic blocks. Evolved hardware is not subject to this limitation and can explore a much larger space than what can be conceptualized by a designer, possibly leading to more efficient circuits (size, speed, etc).

When designing at a low level (at the gate level, or in analog circuits) the designer must rely on a mathematical model of the hardware.

That is the reason why there are rules of design to facilitate the design of working circuits in all (or most of) the cases. EHW has the advantage that it does not need a model: the circuit is tested and its fitness is the only requested information to evaluate its performance. Not being limited to those rules allows EHW to tune itself to the hardware it is running on, possibly getting more performance from the same hardware than a human-designed circuit. This tendency of EHW to tune itself to the hardware at hand can also be a

disadvantage: a circuit evolved on one chip might not work on another one because of slight changes in some characteristics (propagation time, parasitic capacitance, etc.).

Miller et al. suggested in [11], the above three points can be summarized by saying that the space of all the human-designed circuits is just a subset of the space of all possible designs. Because EHW are not subject to the limitations of human designers they can thus explore a much larger set of designs, maybe even the whole space of all designs.

The evaluation and evolutionary process are studied in detail in order to define the specific features of the extrinsic EHW approach and apply them to overcome a number of problems.

Extrinsic evolvable hardware still has two useful features not provided by intrinsic evolvable hardware. Extrinsic evolvable hardware can work with any design paradigm, and so any bias towards a particular kind of circuit behaviour is limited only by the researcher's imagination and the quality of the simulation used for evaluation.

Secondly, circuit designs evolved extrinsically can be simulated with only a net list description of the circuit. This avoids any potentially computationally expensive technology mapping stage needed to implement a circuit from a more general genotype representation. However, the time saved comes at the expense of the speed of solution evaluation

In chapter 5 the research has put forward the view that evolutionary algorithms together with the assemble-and-test methodology can be regarded as a discovery engine or creative machine for new designs. This idea is studied in the context of digital logic. The study suggested that new principles might be discovered by examining a series of evolved designs. During this phase we developed a gate library that contains the circuit components, as well as an easy parameterised evolutionary algorithm.

In chapter 6, the proposed method to synthesise the sequential logic circuits has been tested on the standard benchmarks circuits.

From the experimental results presented in this dissertation, we examined the concept of the space of all circuit representations but feel that similar ideas may well carry over to the general field of design. We also looked at the difficult problem of principle extraction from evolved data. However, a method from evolutionary algorithms as optimization procedures, or more generally as design methods have proven very useful. Using evolutionary algorithms solutions have been found that humans could not easily have derived.

Finally, we feel confident that the process of learning new principles from a blind evolutionary process is inevitable; it is just a matter of time and evolutionary algorithm is a rapidly growing branch of science that introduces new means and methods to find new solutions to existing problems.

The main contribution of the thesis can be summarised as follows:

- A review of the theoretical basis and generic techniques generally applied in the state assignment problem was carried out.
- A literature review of the techniques and developments used and the use of EHW design for sequential logic circuit was carried out and reported work critically evaluated.
- The approaches are used to evolve the sequential benchmark circuits and combined both the state assignment and EHW to design the circuit.
- The detailed results of the application of the approach to each benchmark machine are analyzed.
- The obtained results compare favourably against those produced by manual methods and other methods based on heuristic techniques.

- The approach is tested on a number of finite state machines from MCNC benchmark set. These circuits have been evolved using different functional sets of logic gates and GA parameters.
- The circuits are designed by using an evolutionary based scheme rather than by traditional manual design.
- Another aspect that has to be mentioned is that the computation time of the proposed method depends on evolutionary algorithms parameters.

## 7.2 Future work

There are several directions stemming from this research that could be followed for further exploration.

First, in chapter 4 a method for using genetic algorithm to solve the state assignment problem is developed. Therefore further improvement in genetic state assignment algorithm might be carried out to appropriate fitness function, crossover and mutation methods. Portable applications have made low power design an important issue. Besides, the power consumed may be the limiting factor in high performance systems, due to cooling costs and reliability concerns. The state assignment phase of digital system design affects the structure and complexity of the internal logic as well as the number of clocked elements. Thus, it affects both the power consumed and the area taken up by the synthesized logic. In this work, we deal with the problem of optimizing state assignment for area. Alternatively a state assignment, which explores power tradeoffs during state assignment, would be useful.

Second, The thesis proposes a new approach to evolve sequential logic circuits. Not enough work has been done in this direction and it is necessary to investigate the evolution of sequential logic circuits more closely. Further, several applications have been developed based on EHW. This is both in digital and analog target technology. There seem to be two major directions for the future:

**First**, evolution can be applied to tune the parameters of a circuit.

**Second**, the evolution can be applied to make online adaptable real-time systems. However, the evolutionary schemes would have to be improved to overcome the limitations described in this work. It seems like evolution will be introduced as a substitute to traditional analog design earlier than it is applied in traditional digital design.

It can be concluded that there are a number of areas of future study. These are briefly explained as follows:

- In the current work, the GA is used in the first stage to find state assignments with an optimised solution based on cost function. Then EHW is used to design the circuits as the second stage. Design in this way may introduce additional constraints. If the GA based state assignment is ignored, the search space will be enlarged for EHW consequently, EHW may find better results.
- Developing new library for model level components. One of the disadvantages of using gate design in GA is that the designed circuits are small. This problem can be avoided by using an advanced library, as shown in chapter 6.
- Much larger benchmark sets of sequential logic circuit can be evolved using decomposition technique
- The work can be extended to logic circuit synthesis on-line on programmable logic devices.
- Most of the work so far concentrated on using gate level EHW. Evolving circuits at transistor level would be useful. Further, the optimisation could be tailored to target area, power dissipation or both.

## PUBLICATIONS

1. Ali B, Benhamida S, Roli A, Saadah S, Tavares J. Solving CSP using Evolutionary Algorithm and PIBL approaches. EvoNet Summer School, Technological Educational Institution of Thessaloniki (TEI), Thessaloniki, Greece, August 2001.
2. Ali B, Kalganova T., Almaini A.E.A. Extrinsic Evolution of finite state machines. In I.C. Parmee, editor, proceedings of Adaptive Computing in Design and manufacture V, ACDM, pp. 157-167, UK, 2002.
3. Xia Y, Ali B, Almaini A.E.A. Area and power optimization of FPRM functions based circuits, IEEE International Symposium on Circuits and Systems, vol.329, pp.329-332, Bangkok, 25-28 may 2003.
4. Ali B, Kalganova T, Almaini A E A. Evolutionary algorithms and their use in the design of sequential logic circuits. (Accepted and to be published in international journal of Genetic Programming and Evolvable Machines 2003 (GENP)).



## REFERENCE

1. Ali B., Kalganvoa T. and Almaini A.E.A. Extrinsic evolution of finite state machine. In I.C. Parmee, editor, proceedings of Adaptive Computing in Design and manufacture V, ACDM, pages 157-167, UK, 2002.
2. Fogel D. B. An Introduction to Simulated Evolutionary Optimization. IEEE Trans. on neural networks, vol. 5, No. 1, pages 3-14, 1994.
3. Fogel L. J, Owens A. J. and Walsh M. J. Artificial Intelligence through Simulated Evolution. New York: John Wiley, 1966.
4. Rechenberg I. Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Stuttgart: Frommann-Holzboog, 1973.
5. Holland J. H. Adaptation in natural and artificial systems. Ann Arbor, MI: The University of Michigan Press, 1975.
6. Miller J. F., Thomson P., and Fogarty T. C. Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study, in Genetic Algorithms and Evolution Strategies in Engineering and Computer Science: D. Quagliarella, J. Periaux, C. Poloni and G. Winter (eds), Wiley, 1997.
7. Thompson A. n evolved circuit, intrinsic in silicon, entwined with physics. Proc. 1<sup>st</sup> Int. Conf. on Evolvable Systems (ICES96), T. Higuchi, M. Iwata, L. Weixin, eds., pp. 390-405, Springer, 1997
8. Layzell, P. The 'Evolvable Motherboard': A Test Platform for Research of Intrinsic Hardware Evolution. Csrp 479, School of Cognitive and Computing Sciences, University of Sussex, 1998.

9. Thompson A. Silicon evolution. In Koza, J. R., editor, Proc. of the Int. Conference on Genetic Programming, MIT Press, pages 444-452, 1996.
10. Miller J. F., Thomson P. Combinational and sequential logic optimization using genetic algorithms. In Proc. 1st IEE/IEEE Int. Conf. on Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA'95), pages 34-38. IEE Conf. Publication No. 414, 1995.
11. Miller. J, Kalganvoa T., Lipnitskaya N. and Job D. The Genetic Algorithm as a Discovery Engine: Strange Circuits and New Principles. (CES'99). Edinburgh, UK. Published by The Society for the Study of Artificial Intelligence and Simulation of Behaviour (AISB), pages 65-74, 1999.
12. Mazumder P., Rudnick E. M. Genetic Algorithms for VLSI Design, Layout & Test Automation. Prentice-Hall, 1999.
13. Koza J. R., Andre D, Bennett III F. H. and Keane, M. Genetic Programming III. Morgan Kaufmanns 1998.
14. Kruiskamp W., Leenaerts D. Darwin. CMOS opamp synthesis by means of a genetic algorithm. In 32nd Design Automation Conference, pages 433-438. Association for Computing Machinery, 1995.
15. Robert R., Hubert K., Tobias B. Stochastic Methods for Transistor Size Optimization of CMOS VLSI Circuits. PPSN, pages 849-858, 1996.
16. Kajitani .I .et al. An evolvable hardware chip for prosthetic hand controller. In Proc. of MicroNeuro'99, pages 179-186, 1999.
17. Higuchi T. et al. Evolvable hardware: A first step towards building a Darwin machine. In Proc. of the 2nd Int. Conf. on Simulated Behavior, pages 417-424. MIT Press, 1993.
18. Louis S., Rawlins J. G. Designer genetic algorithms: Genetic algorithms in structure design. In Proceedings of the Fourth

- International Conference on Genetic Algorithms, pages 53-60. Morgan Kaufman, San Mateo, CA, 1991.
19. DeGaris H. Evolvable hardware: Genetic Programming of a Darwin machine, in Artificial neural Nets and Genetic Algorithms, Albretch, R.F., Reeved, C. R, and Steele, N. C., Eds. Springer-Verlag, New York, 1993.
  20. Hemmi H., Mizoguchi, J., Shimonara K. Development and Evolution of Hardware Behaviors. Artificial Life IV, R.A. Brooks and P. Maes, ed., pages 371-376, MIT Press, 1994.
  21. Koza J. R., Andre D., Bennett III F. H., et al. Use of automatically defined functions and architecture-altering operations in automated circuit synthesis with genetic programming. In Genetic Programming 1996: Proc. 1st Annual Conf. (GP96), J. R. Koza et al., Eds., pages 132-140, Cambridge, MA: The MIT Press, 1996.
  22. Ricardo S. Zebulum, M.A.C. Pacheco, Marley M.B.R. Vellasco, Evolutionary Systems Applied to the Synthesis of a CPU Controller, The Second Asia-Pacific Conference on Simulated Evolution and Learning - Canberra, Australia, 24-27, November 1998.
  23. Chongstitvatana P., Apornthewan C. Improving Correctness of Finite-State Machine Synthesis from Multiple Partial Input/Output Sequences. Proc. of the First NASA/DoD Workshop on Evolvable Hardware, Pasadena, California, pages 262-266, 1999.
  24. Kalganova T. Evolvable Hardware Design for Combinational Logic Circuits. Ph.D. thesis, Napier University, Edinburgh, UK, August 2000.
  25. Goldberg D. E. Genetic Algorithms in Search Optimisation & Machine Learning. Addison-Wesley, 1989.

26. Hartmanis J., Stearns, R. E. Algebraic Structure Theory of sequential Machines. Prentice-Hill, Englewood Cliffs, New Jersey, 1966.
27. Kohavi, Z. Switching and finite state automata theory. McGraw-Hill, 1970.
28. Rudell R. L. Logic Synthesis for VLSI Design. UCB/ERL M89/49, April, 1994.
29. Hill J. F., Peterson G.R. Computer Aided Logical Design with Emphasis on VLSI. 4<sup>th</sup> Edition, John Wiley & Sons, inc, 1993.
30. Gajski D., Kuhn R. Guest Editor Introduction-New VLSI Tools. IEEE Computer, Vol.16, No.12, pages 11-14,1983.
31. Mazumder P., Rudnick E. Genetic Algorithms for VLSI Design, Layout and Test Automation. Prentice Hall, December 1998.
32. Clare C. R. Designing logic systems using state machines. McGraw-Hill, 1973.
33. Almaini A. E.A. Electronic logic systems. Prentice-Hall, 3<sup>rd</sup> Ed 1994.
34. Moore E. F. Gedanken-experiment on sequential machines. In Automata Studies, pages 129-153, Princeton, New Jersey, 1956. Princeton University Press.
35. Huffman D. A. The Synthesis of Sequential Switching Circuits. In J. Franklin institute, volume 257, No. 4, pages 275-303, 1954.
36. Ginsburg S. A synthesis Technique for Minimum State Sequential Machines. In IRE Transactions on Electronic Computers, volume EC-8, pages 13-24, March 1959.
37. Paull M. C, Unger S. H. Minimizing the number of state in Incompletely Specified Sequential Circuits. In IRE Transactions on Electronic Computers, volume EC-8, pages 356-357, September 1959.

38. Marcus M. P. Deriving Maximal Compatible using Boolean algebra. In IBM Journal of Research and Development, volume 8, pages 357-538, November 1964.
39. Hartmanis J., Stearns R. E. On the state assignment problem for sequential machines II. In IEEE Trans on Electronic Computer, Volume 16, pages 593-604, August 1967.
40. Karp R. Some techniques for state assignment for synchronous sequential machines. In IEEE Trans. Elect. Comput., vol. EC-13, pages 507-518, Oct. 1964.
41. Krohn K., Rhodes J. Algebraic Theory of Machines. In Proceeding Symposium on Mathematical Theory of Automata. Polytechnic Press, N.Y., 1962.
42. Yoeli M. Cascade Parallel Decomposition of Sequential Machines. In IRE Transaction on Electronic Computers, Volume EC-12, pages 587-592, April 1963.
43. Zeiger H.P. Loop-free Synthesis of Finite-state Machine. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, 1964.
44. Hennie F. C. Finite-state Models for logical Machines. Wiley, New York, 1968.
45. Armstrong D.B. A programmed algorithm for assigning internal codes to sequential machines. IRE Trans. Elect. Comput., vol. EC-11, pages 466-472, Aug. 1962.
46. Dolotta T. A, McClusky E. J. The coding of internal states of sequential machines. In IEEE Trans. EC-11, pages 549-562. 1964.
47. Devadas S., H.K. Ma, Newton A. R., and Sangiovanni-Vincentelli A. MUSTANG: State assignment of finite state machines targeting multi-level logic implementations. IEEE Transactions on CAD, CAD-7 (12), pages 1290-1300, December 1988.
48. Avedillo M. J, Quintana J.M. and Huertas J.L. Smas: program for the concurrent state reduction and state assignment of finite state machines. In ISCAS, pages 1781-1784, 1991.

49. Grasselli A., Luccio F. method of minimizing the number of internal state in incompletely specified sequential network. IEEE TEC, EC-14, pages 350-359, June 1965.
50. De Micheli G. Optimal Encoding of Control Logic. Int. Conf. on Circ. and Comp. Des. Rye NY, Sept. 1984.
51. Kalnberzin A.Ya and Chapenko V.P. Method of Input State Assignment for Digital Devices Implemented Using Programmable Logic Devices, *Avtomatika i Vychislitel'naya Tekhnika*, Vol. 17, No.1, pages 41-47, 1983.
52. Almaini A.E.A. Sequential machine implementations using universal logic modules. IEEE Trans Computers. Vol. C-27, No 10, pages 951-960, Oct 1978.
53. De Micheli G. Symbolic Minimization of Logic Functions, Proc. ICCAD 85, Santa Clara, California, pages 293-295, Nov.18-21, 1985.
54. Almaini A.E. A, Woodward M, E. Computer program for S.P Partition of sequential machine, *Electronic letter*. Vol. 10, No 21, pages 951-960, 17<sup>th</sup> Oct 1978.
55. Curtis H.A. Systematic Procedures for Realising Synchronous Sequential Machines Using flip-flop Memory: Part 1. IEEE Trans. on Comp. Vol. C-18, pages 1121-1127, December 1969.
56. Weiner P., Smith E. J. Optimisation of Reduced Dependencies for Synchronous Sequential Machines. IEEE Trans. on Electr. Comp. Vol. EC-16, pages 835-847, December 1967.
57. Vavilov E.N and Portnoy G. P. Synthesis of Circuits of Electronic Computers. Energia Publishers, Moscov, 1966.
58. Torng H. C. Introduction to the Logical Design of Switching Systems. Addison-Wesley, Reading, Massachusetts, 1964.
59. Story J. R., Harrison H.J and Reinhard E.A. Optimum State Assignment for Synchronous Sequential Circuits. IEEE Trans. on Comp, Vol. C-21, No.12, pages 1365-1373, December 1972.

60. Perkowski M. A System for Automatic Design of Digital Systems. Magyar Tudományos Akadémia. Számítástechnikai És Automatizálási Kutató Intézet. Budapest Tanulmányok 99/1979 pages 93-112. Hungary, 1979.
61. Zasowska A and Perkowski M. The Computer-Oriented Method for Joint Minimization and State-Assignment of Synchronous and Asynchronous Automata. Proc. of the Conf., Application of Computers in Engineering Design. Katowice, Poland, 1979.
62. Lee E.B and Perkowski M. Concurrent Minimization and State Assignment of Finite State Machines. Proceedings of the 1984 Intern. Conf. on Systems, Man, and Cybernetics, IEEE, Halifax, Nova Scotia, Canada, October 9-12, 1984.
63. Murakawa M., S. Yoshizawa I. Kajitani, et al. Hardware evolution at function level. In Proc. 4th Int. Conf. on Parallel Problem Solving from Nature (PPSN IV), W. Ebeling et al., Eds. vol. 1141 of LNCS, pages 62-71, Springer-Verlag. 1996.
64. Moroz D.Z. An Algorithm for Encoding the States of an Automaton. Avtomatika i Vychislitel'naya Tekhnika, Vol.4, No. 4, pages 21-24, 1970.
65. De Micheli G., Brayton R., Sangiovanni-Vincentelli A.L. KISS: A Program for Optimal State Assignment of Finite State Machines. Int. Conf On Comp. Aid. Design. Santa Clara, November 1984.
66. Rudell R.L and Sangiovanni-Vincentelli A.L. Espresso-MV: Algorithms for Multiple-Valued Logic Minimization. IEEE Custom Integrated Circuits Conference, pages 230-234, 1985.
67. Coppola A.J. An Implementation of a State Assignment Heuristic. Proc. of the 23-rd Design Automation Conference, pages 643-649, June 29 - July 2, 1986.
68. Almaini A. E. A, Miller J.F, Thomson P, Billina S. State assignment of state machine using genetic algorithm. IEE Proc

- Comp and Digital Techniques, Vol. 142, No 4, pages 279-286, July 1995.
69. Amaral J. N, Tumer K. And Ghosh J. Design genetic algorithm for the state assignment problem. IEEE Trans, SMC-25, (4), pages 689-694,1995.
  70. Perkasić M. A New Approach to the Structural Design of Finite State Machines. Department of EE, PSU, Report, 1986.
  71. Lee. E.B and Perkowski. M. A New Approach to Structural Synthesis of Automata. University of Minnesota, Department of Electrical Engineering, report, 1982.
  72. Saucier G. State Assignment of Asynchronous Sequential Machines Using Graph Techniques. IEEE Trans. on Comp. Vol. C-21, pages 282-288, March 1972.
  73. Sentovich E. M., et al. SIS: A System for Sequential Circuit Synthesis. Tech. Rep. UCB/ERL M92/41, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, May 1992.
  74. Brayton R., Sangiovanni-Vincetelli R., Wang, A. MIS: A Multiple-level Optimization System. In IEEE Transaction on Computer-Aided Design of International Conference on Computer Aided Design, pages 66-69, November 1987.
  75. Davis W.A. Single Shift-Register Realization for Sequential Machines. IEEE Trans. on Comp., Vol 1-17, No. 5, pages 421-431, May 1968.
  76. Krohn K., Rhodes J.L. Algebraic Structure Theory of Machines I: the Decomposition Results. Trans. American Math Soc., CXVI 1965.
  77. Hurson A.R. A VLSI Design of the Parallel Finite State Automaton and its Performance Evaluation on a Hardware Scanner. Intl. J. of Comp. and Info. Science, Vol. 13, No. 5, pages 481-505, 1984.



78. Lee E.B., Perkowski M. A New Approach to Structural Synthesis of Automata. University of Minnesota, Department of Electrical Engineering, report, 1982.
79. lee E.B., Perkowski M. Concurrent Minimization and State Assignment of Finite State Machines. Proceedings of the 1984 Intern. Conf. on Systems, Man, and Cybernetics, IEEE, Halifax, Nova Scotia, Canada, October 9 - 12, 1984.
80. Sklyarov V.A. Design of Automata Using Programmable Logic Arrays with Memory. Kibernetika, pages 840-848, Plenum Publishing Corp., 1985.
81. Zeiger H.P. Cascade Decomposition of Automata using Covers. In the Algebraic Theory of Machines, Languages, and Semigroups, by M.A. Arlib, Academic Press, Netherlands, 1968.
82. Schwefel H. P. Numerical Optimization of Computer Models. New York. John Wiley & Sons, 1981.
83. Zalzal A.M.S., Fleming P.J. Genetic algorithms in engineering system. IEE, Control Engineering Series, QA402.5.G3, 1997.
84. Koza J. R. Evolving a computer program to generate random numbers using the genetic programming paradigm. Proceedings of the Fourth International Conference on Genetic Algorithms, pages 37-44. La Jolla, CA: Morgan Kaufmann, 1991.
85. Davis L. Adapting operator probabilities in genetic algorithms. Proceedings of the Third International Conference on Genetic Algorithms, pages 60-69. La Jolla, CA: Morgan Kaufmann, 1989.
86. Gen M. and Cheng R. Genetic Algorithms and Engineering Design. John Wiley & Sons, 1997.
87. De Jong K.A. An analysis of the behavior of a class of genetic adaptative systems. Doctoral dissertation, University of Michigan, 1975.

88. Fogel D.B. Phenotypes, Genotypes, and Operators in Evolutionary Computation. In Proceedings of the 1995 IEEE Conference on Evolutionary Computation, Perth, Australia, IEEE Press, pages 193-198, 1995.
89. Fogel L.J. Evolutionary Programming in Perspective: The Top-Down View, in ZURADA, J. M., et al., eds. Computational Intelligence: Imitating Life, Piscataway, IEEE Press, pages 135-146, 1994.
90. Moore G.E., An Update on Moore's law, <http://developer.intel.com/>, Intel Developer Forum Keynote, San Francisco, 1997.
91. Haddow P. and G Tufte. An evolvable hardware FPGA for adaptive hardware. In congress on Evolutionary computation (CECOO), pages 553-560, 2000.
92. Lienig J. A parallel genetic algorithm for performance-driven VLSI routing. IEEE Transactions on Evolutionary Computation, vol. 1, no. 1, pages. 29-39, Apr. 1997.
93. Fourman M. P. Compaction of symbolic layout using genetic algorithms. In Proc. 1st Int. Conf. on Genetic Algorithms and their Applications, J. J. Grefenstette, Ed., pages 141-153, Lawrence Erlbaum Associates, 1985.
94. Kalganvoa T. Bi-directional Increment Evolution in Evolvable Hardware. Proc. Of The second NASA/DoD workshop on Evolvable hardware. Palo Alto, California, USA. Published by IEEE Computer Society, EH2000.
95. Coen E. The Art of Genes. How organisms make themselves. Oxford, UK, Oxford University Press, 1999.
96. Torresen J. A Divide-and-Conquer Approach to Evolvable Hardware. Second International Conference on Evolvable Hardware (ICES98), Lausanne, Switzerland, September 1998.

97. Torresen I. Evolvable Hardware: The Coming Hardware Design Method? In the book: "Neuro-fuzzy techniques for Intelligent Information Systems", page 435-449, published by N. Kasabov and R. Kozma (editors), Physica-Verlag (Springer-Verlag), 1999.
98. De Micheli G., R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment for finite state machines. IEEE Transactions on CAD, CAD-4 (3), pages 269-285, July 1985.
99. De Micheli G., Sangiovanni-Vincentelli A., Villa T. Computer-Aided Synthesis of PLA-Based Finite State Machines. Proc. IEEE 1983 Intern. Conf on Computer Aided Design, pages 154-156, September 1983.
100. Ricardo Z, Marco P, Marley V. Evolvable Systems in Hardware Design: Taxonomy, Survey and Applications, Proceedings of The First International Conference on Evolvable Systems: From Biology to Hardware (ICES'96), Lecture Notes in Computer Science 1259, pages 344-358, Tsukuba, Japão, October 1996.
101. Koza J. R. Future work and practical applications of genetic programming. In Handbook of Evolutionary Computation, page H1.1: 3. IOP Publishing Ltd and Oxford University Press, 1997.
102. Thompson A. An evolved circuit, intrinsic in silicon, entwined with physics. In Proc. of int. Conf. on Evolvable Systems (ICES'96), 1996.
103. Apornthewan C., Chongstitvatana P., An on-line evolvable hardware for learning finite-state machine. Proc. of Int. Conf. on Intelligent Technologies, Bangkok, December 13-15, pages 125-134, 2000.
104. XC6200 field programmable gate arrays, Data Sheet, Xilinx Inc., April 1997.
105. Higuch T, Iwata M., Kajitanai J., Iba N., Hirao J., Furuya T., and Manderick B. Evolvable hardware and its application to pattern recognition and fault tolerant systems. In Sanchez E. and

- Tomassini M., editors, Towards Evolvable Hardware. The evolutionary Engineering approaches, volume 1062 of lecture Notes and Computer Sciences, pages 118-135. Springer-Verlag, 1996.
106. Comisky W., Yu J. and Koza J. Automatic synthesis of a wire antenna using genetic programming. Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference, pages 179-186, Las Vegas, Nevada, 2000.
  107. Linden D. S. Evolving Wire Antennas Using Genetic Algorithms: A Review. Proc. of the First NASA/DoD Workshop on Evolvable Hardware, pages 225-232, Pasadena, Calif., 1999.
  108. Quagliarella D., Périaux J., Poloni C., Winter. G., eds. Genetic Algorithms and Evolution Strategies in Engineering. Winter (eds), Wiley, 1997.
  109. Davis L. Handbook of Genetic Algorithms. by Van Nostrand Reinhold. QA402.5 D3, 1991.
  110. Hartmanis J. On the state assignment problem for sequential machines I. In IRE Trans., EX-10, pages 157-165, 1961.
  111. Lin B., Newton A.R. Synthesis of multi level logic from Symbolic High-level Description Language. In Proc. Int. Conference on VLSI, August 1989.
  112. Rudell R., Sangiovanni-Vincentelli A. Multiple-valued optimization for PLA optimization. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 6(5), pages 727-750, September 1987.
  113. Villa T., Sangiovanni-vincentelli A. NOVA: state assignment of finite state machines for optimal two level logic implementation. IEEE Trans., C-9, pages 905-924, 1990.
  114. De Micheli G., Brayton R., Sangiovanni-Vincentelli A.L. KISS: A Program for Optimal State Assignment of Finite State Machines. Int. Conf On Comp. Aid. Design. Santa Clara, November 1984.

115. Lawler E. L, Lenstra J. K., Rinnooy Kan A. H. G. and Shimoys D.B. The Travelling-Salesman Problem. John Wily & Sons, Chichester, 1985.
116. Yang S. Logic synthesis and optimisation benchmark user guide version 3.0. MCNC, 1991.
117. Louis S. Genetic Algorithm as computational Tool for design. Ph.D. Dissertation, Department of computer Science, Indiana University, 1993.
118. Manovit C., Aporntewan C., and Chongstitvatana P. Synthesis of synchronous sequential logic circuits from partial input/output sequence. In Proceedings of International Conference on Evolvable Systems, pages 98-105, 1998.
119. Blickle T. Theory of Evolutionary Algorithms and Application to System Synthesis, TIK-Schriftenreihe Nr. 17, vdf, Hochsch-Verl. an der ETH, Swiss Federal Institute of Technology, Zurich, 1997.
120. Hill A. M., Kang S.M. Genetic algorithm based design optimization of CMOS VLSI circuits. In Proc. 3rd Conf. on Parallel Problem Solving from Nature (PPSN III), Y. Davidor, H.P. Schwefel, and R. Männer, Eds. vol. 866 of *LNC3*, pages 546-555, Springer-Verlag, 1994.
121. Lawler E. L. An Approach to Multilevel Boolean Minimization, Journal of the Association for computing Machinery, Vol.11, No.3, pages 283-295,1964.
122. Green D. Modern Logic Design, Electronic System Engineering Series. TK7868, L6.G7, 1994.
123. Wang L. Automated Synthesis and Optimisation of Multilevel Logic Circuit. Ph.D. thesis, School of Engineering, Napier University, Edinburgh, UK, 2001.
124. Kalganvoa T. and J. Miller. Evolving more efficient digital circuits by allowing circuit layout and multi-objective fitness. IEEE Computer society Press, In A. Stoica, D. Keymeulen, and J. Iohn,

- editors, proceeding of the first NASA/DoD Workshop on Evolvable Hardware. Pages 54-63, Los Alamitos, California, 1999.
125. Coello C., Hernández-Aguirre A., Buckles B. P. Evolutionary Multiobjective Design of Combinatorial Logic Circuits, Proc. of the Second NASA/DoD Workshop on Evolvable Hardware, pp. 161-170, Palo Alto, Calif., 2000.
  126. Higuchi T., Niwa T., Tanaka T., and Iba H. A Parallel Architecture for Genetic Based Evolvable Hardware Proc. of 2<sup>nd</sup> Workshop on Parallel Processing for Artificial Intelligence, PPAI-93 (IJCAI-93 Workshop), 1993.
  127. Fogel D. B. Evolutionary Computation. IEEE Press, pages 75-84, 1995.
  128. Thompson A. Evolving electronic robot controllers that exploit hardware resources. 3<sup>rd</sup> Eur. Conf. on Artificial Life, 1995.
  129. Apornthewan, C. and Chongstitvatana, P. An on-line evolvable hardware for learning finite state machine. Proc. of Int. Conf. on Intelligent Technologies, Bangkok, December 13-15, pages 125-134, 2000.
  130. Darringer J., Brand D., Gerbi W, Joyner J. and Trevillyan L. LSS: system for Production logic Synthesis. In IBM J. Res. Develop, volume 28, pages 537-545, September 1984.
  131. Zbigniew M. Genetic algorithms + data structures = evolutionary programs. Berlin, Springer-Verlag, 1992.
  132. Xia Y, Ali B, Almaini A.E.A. Area and power optimization of FPRM functions based circuits. IEEE International Symposium on Circuits and Systems, vol.329, pp.329-332, Bangkok, 25-28 May 2003.
  133. Chalmers S. Study of methodology for fault tolerant state machine controllers. Ph.D. thesis, Robert Gordon University, Aberdeen, UK, June 2000.

134. Roggen *D.* Evolvable electronics for vision-based robots. Diploma project. Institute de system Robotiques (ISR), Lausanne, Switzerland, 2001.
135. Karnaugh *M.* The Map Method for synthesis of Combinational Logic circuits. Transaction of the AIEE, Vol.72, Pt.1, pages 593-598, 1953.
136. McCluskey *E.* Minimization of Boolean Functions. Bell System Technical Journal, Vol.35, No. 5, pages 1417-1444, 1956.
137. Ashar *P.* Synthesis of Sequential Circuits for VLSI Design. PhD thesis, University of California at Berkeley, November 1991.

# APPENDIXES



*Appendix A*

This appendix contains the FSM benchmark transition tables in standard “KISS” format as discussed in chapters 4&6.

**bbara.kiss2**

```
.i 4
.o 2
.p 60
.s 10
--01    st0    st0    00
--10    st0    st0    00
--00    st0    st0    00
0011    st0    st0    00
-111    st0    st1    00
1011    st0    st4    00
--01    st1    st1    00
--10    st1    st1    00
--00    st1    st1    00
0011    st1    st0    00
-111    st1    st2    00
1011    st1    st4    00
--01    st2    st2    00
--10    st2    st2    00
--00    st2    st2    00
0011    st2    st1    00
-111    st2    st3    00
1011    st2    st4    00
--01    st3    st3    10
--10    st3    st3    10
--00    st3    st3    10
0011    st3    st7    00
-111    st3    st3    10
1011    st3    st4    00
--01    st4    st4    00
--10    st4    st4    00
--00    st4    st4    00
0011    st4    st0    00
-111    st4    st1    00
1011    st4    st5    00
--01    st5    st5    00
--10    st5    st5    00
--00    st5    st5    00
0011    st5    st4    00
-111    st5    st1    00
1011    st5    st6    00
--01    st6    st6    01
--10    st6    st6    01
```

```
--00  st6  st6  01
0011  st6  st7  00
-111  st6  st1  00
1011  st6  st6  01
--01  st7  st7  00
--10  st7  st7  00
--00  st7  st7  00
0011  st7  st8  00
-111  st7  st1  00
1011  st7  st4  00
--01  st8  st8  00
--10  st8  st8  00
--00  st8  st8  00
0011  st8  st9  00
-111  st8  st1  00
1011  st8  st4  00
--01  st9  st9  00
--10  st9  st9  00
--00  st9  st9  00
0011  st9  st0  00
-111  st9  st1  00
1011  st9  st4  00
```

.e

```
states:      state-assignment:
  0          0000
  1          0110
  2          0010
  3          1100
  4          0100
  5          0101
  6          1011
  7          0111
  8          0011
  9          0001
```

**Bbtas.kiss2**

```
.i 2
.o 2
.p 24
.s 6
00    st0    st0    00
01    st0    st1    00
10    st0    st1    00
11    st0    st1    00
00    st1    st0    00
01    st1    st2    00
10    st1    st2    00
11    st1    st2    00
00    st2    st1    00
01    st2    st3    00
10    st2    st3    00
11    st2    st3    00
00    st3    st4    00
01    st3    st3    01
10    st3    st3    10
11    st3    st3    11
00    st4    st5    00
01    st4    st4    00
10    st4    st4    00
11    st4    st4    00
00    st5    st0    00
01    st5    st5    00
10    st5    st5    00
11    st5    st5    00
.e
states:      state-assignment:
  0          000
  1          010
  2          100
  3          110
  4          001
  5          011
```

**dk15.kiss2**

```
i 3
.o 5
.p 32
.s 4
000 state1 state1 00101
000 state2 state2 10010
000 state3 state1 00101
000 state4 state2 10010
001 state1 state2 00010
001 state2 state2 10100
001 state3 state2 00010
001 state4 state2 10100
010 state1 state3 00010
010 state2 state3 10010
010 state3 state3 00010
010 state4 state3 10010
011 state3 state1 00100
011 state4 state1 00100
011 state1 state2 10001
011 state2 state2 10001
111 state3 state1 00100
111 state4 state1 00100
111 state1 state3 10101
111 state2 state3 10101
100 state1 state1 01001
100 state3 state1 10100
100 state4 state1 01001
100 state2 state3 01001
101 state1 state2 01010
101 state2 state2 01010
101 state3 state2 01000
101 state4 state2 01010
110 state1 state3 01010
110 state2 state3 01010
110 state4 state3 10000
110 state3 state4 01010
.e
states:      state-assignment:
  0          00
  1          10
  2          11
  3          01
```

**dk16.kiss2**

```

.i 2
.o 3
.p 108
.s 27
00 state1 state3      001
00 state2 state1      001
00 state3 state4      001
00 state4 state4      010
00 state5 state1      010
00 state6 state3      010
00 state7 state9      010
00 state8 state15     010
00 state9 state1      000
00 state10 state14    000
00 state11 state3     000
00 state12 state20    000
00 state13 state3     101
00 state14 state1     101
00 state15 state4     101
00 state16 state20    000
00 state17 state15    010
00 state18 state4     100
00 state19 state18    100
00 state20 state19    100
00 state21 state2     100
00 state22 state3     000
00 state23 state2     100
00 state24 state14    000
00 state25 state15    010
00 state26 state20    000
00 state27 state15    010
01 state1 state10     001
01 state2 state2      001
01 state3 state5      001
01 state4 state5      010
01 state5 state2      010
01 state6 state21     010
01 state7 state18     010
01 state8 state26     000
01 state9 state5      000
01 state10 state13    000
01 state11 state23    000
01 state12 state19    000
01 state13 state10    101
01 state14 state2     101
01 state15 state5     101
01 state16 state19    000
01 state17 state23    000
01 state18 state5     010

```

01 state19 state23	010
01 state20 state20	010
01 state21 state1	010
01 state22 state3	010
01 state23 state1	010
01 state24 state13	000
01 state25 state3	010
01 state26 state19	000
01 state27 state3	010
10 state1 state11	001
10 state2 state8	001
10 state3 state6	001
10 state4 state6	010
10 state5 state16	010
10 state6 state10	010
10 state7 state19	010
10 state8 state13	010
10 state9 state6	000
10 state10 state1	000
10 state11 state24	000
10 state12 state18	000
10 state13 state11	101
10 state14 state8	101
10 state15 state6	101
10 state16 state13	010
10 state17 state18	000
10 state18 state6	100
10 state19 state24	100
10 state20 state9	100
10 state21 state13	100
10 state22 state15	100
10 state23 state13	010
10 state24 state13	100
10 state25 state15	000
10 state26 state18	000
10 state27 state13	100
11 state1 state12	001
11 state2 state9	001
11 state3 state7	001
11 state4 state7	010
11 state5 state17	010
11 state6 state22	010
11 state7 state20	010
11 state8 state14	010
11 state9 state7	000
11 state10 state2	000
11 state11 state25	000
11 state12 state15	000
11 state13 state12	101
11 state14 state9	101
11 state15 state7	101

11	state16	state14	010
11	state17	state27	000
11	state18	state7	100
11	state19	state25	100
11	state20	state26	100
11	state21	state14	100
11	state22	state15	000
11	state23	state14	010
11	state24	state14	100
11	state25	state15	000
11	state26	state21	000
11	state27	state14	100

.e

states:           state-assignment

1	00000
2	00010
3	00011
4	00100
5	00101
6	00110
7	00111
8	01000
9	01001
10	01001
11	01011
12	01010
13	01101
14	01110
15	01111
16	10000
17	00001
18	10010
19	10011
20	10100
21	10101
22	11010
23	10111
24	11000
25	10101
26	11010
27	11011

**donfile.kiss2**

```

.i 2
.o 1
.p 96
.s 24
00      st0 st0      1
01      st0 st6      1
10      st0 st12     1
11      st0 st18     1
00      st1 st1      1
01      st1 st7      1
10      st1 st12     1
11      st1 st18     1
00      st2 st2      1
01      st2 st6      1
10      st2 st12     1
11      st2 st19     1
00      st3 st3      1
01      st3 st6      1
10      st3 st13     1
11      st3 st19     1
00      st4 st4      1
01      st4 st7      1
10      st4 st13     1
11      st4 st18     1
00      st5 st5      1
01      st5 st7      1
10      st5 st13     1
11      st5 st19     1
00      st6 st0      1
01      st6 st6      1
10      st6 st14     1
11      st6 st20     1
00      st7 st1      1
01      st7 st7      1
10      st7 st14     1
11      st7 st20     1
00      st8 st0      1
01      st8 st8      1
10      st8 st14     1
11      st8 st21     1
00      st9 st0      1
01      st9 st9      1
10      st9 st15     1
11      st9 st21     1
00      st10 st1     1
01      st10 st10    1
10      st10 st15    1
11      st10 st20    1
00      st11 st1     1
01      st11 st11    1

```



10	st11 st15	1
11	st11 st21	1
00	st12 st2	1
01	st12 st8	1
10	st12 st12	1
11	st12 st22	1
00	st13 st3	1
01	st13 st8	1
10	st13 st13	1
11	st13 st22	1
00	st14 st2	1
01	st14 st8	1
10	st14 st14	1
11	st14 st23	1
00	st15 st2	1
01	st15 st9	1
10	st15 st15	1
11	st15 st23	1
00	st16 st3	1
01	st16 st9	1
10	st16 st16	1
11	st16 st22	1
00	st17 st3	1
01	st17 st9	1
10	st17 st17	1
11	st17 st23	1
00	st18 st4	1
01	st18 st10	1
10	st18 st16	1
11	st18 st18	1
00	st19 st5	1
01	st19 st10	1
10	st19 st16	1
11	st19 st19	1
00	st20 st4	1
01	st20 st10	1
10	st20 st17	1
11	st20 st20	1
00	st21 st4	1
01	st21 st11	1
10	st21 st17	1
11	st21 st21	1
00	st22 st5	1
01	st22 st11	1
10	st22 st16	1
11	st22 st22	1
00	st23 st5	1
01	st23 st11	1
10	st23 st17	1
11	st23 st23	1

.e

states:	state-assignment
1	00000
2	01100
3	00101
4	00001
5	00110
8	00111
9	01001
10	00100
11	01011
12	01100
13	01101
14	01011
15	01111
16	10000
17	11001
18	10001
19	10011
20	10100
21	10101
22	10100
23	10111
24	11000

**lion9.kiss2**

```
.i 2
.o 1
.p 25
.s 9
10 st0 st1 0
00 st0 st0 0
00 st1 st0 0
10 st1 st1 0
11 st1 st2 0
10 st2 st1 0
11 st2 st2 0
01 st2 st3 0
11 st3 st2 1
01 st3 st3 1
00 st3 st4 1
01 st4 st3 1
00 st4 st4 1
10 st4 st5 1
00 st5 st4 1
10 st5 st5 1
11 st5 st6 1
10 st6 st5 1
11 st6 st6 1
01 st6 st7 1
11 st7 st6 1
01 st7 st7 1
00 st7 st8 1
01 st8 st7 1
00 st8 st8 1
.e
states:      state-assignment:
    0         0000
    1         0100
    2         1100
    3         1101
    4         1111
    5         0001
    6         0011
    7         0111
    8         0101
```

**module12.kiss2**

i 1

.o 1

.p 24

.s 12

0	st0	st0	0
1	st0	st1	0
0	st1	st1	0
1	st1	st2	0
0	st2	st2	0
1	st2	st3	0
0	st3	st3	0
1	st3	st4	0
0	st4	st4	0
1	st4	st5	0
0	st5	st5	0
1	st5	st6	0
0	st6	st6	0
1	st6	st7	0
0	st7	st7	0
1	st7	st8	0
0	st8	st8	0
1	st8	st9	0
0	st9	st9	0
1	st9	st10	0
0	st10	st10	0
1	st10	st11	0
0	st11	st11	0
1	st11	st0	0

**dk27.kiss2**

```

.i 1
.o 2
.p 14
.s 7
0      START      state6      00
0      state2      state5      00
0      state3      state5      00
0      state4      state6      00
0      state5      START      10
0      state6      START      01
0      state7      state5      00
1      state6      state2      01
1      state5      state2      10
1      state4      state6      10
1      state7      state6      10
1      START      state4      00
1      state2      state3      00
1      state3      state7      00
.e
states:      state-assignment:
0            110
1            100
2            101
3            111
4            010
5            011
6            001

```

**dk512.kiss2**

```
.i 3
.o 5
.p 32
.s 4
000 state1 state1 00101
000 state2 state2 10010
000 state3 state1 00101
000 state4 state2 10010
001 state1 state2 00010
001 state2 state2 10100
001 state3 state2 00010
001 state4 state2 10100
010 state1 state3 00010
010 state2 state3 10010
010 state3 state3 00010
010 state4 state3 10010
011 state3 state1 00100
011 state4 state1 00100
011 state1 state2 10001
011 state2 state2 10001
111 state3 state1 00100
111 state4 state1 00100
111 state1 state3 10101
111 state2 state3 10101
100 state1 state1 01001
100 state3 state1 10100
100 state4 state1 01001
100 state2 state3 01001
101 state1 state2 01010
101 state2 state2 01010
101 state3 state2 01000
101 state4 state2 01010
110 state1 state3 01010
110 state2 state3 01010
110 state4 state3 10000
110 state3 state4 01010
.e
states:      state-assignment:
0            01
1            10
2            00
3            11
```

**shifreg.kiss2****.i 1****.o 1****.p 16****.s 8****0 st0 st0 0****1 st0 st4 0****0 st1 st0 1****1 st1 st4 1****0 st2 st1 0****1 st2 st5 0****0 st3 st1 1****1 st3 st5 1****0 st4 st2 0****1 st4 st6 0****0 st5 st2 1****1 st5 st6 1****0 st6 st3 0****1 st6 st7 0****0 st7 st3 1****1 st7 st7 1****States****state-assignment:****0 110****1 010****2 100****3 000****4 111****5 011****6 101****7 001**

**tav.kiss2**

```
.i 4
.o 4
.p 49
.s 4
1000 st0 st1 1000
0100 st0 st1 0100
0010 st0 st1 0010
0001 st0 st1 0001
0000 st0 st1 0000
11-- st0 st1 0000
1-1- st0 st1 0000
1--1 st0 st1 0000
-11- st0 st1 0000
-1-1 st0 st1 0000
--11 st0 st1 0000
1000 st1 st2 1000
0100 st1 st2 0100
0010 st1 st2 0010
0001 st1 st2 0001
1100 st1 st2 1100
1010 st1 st2 1010
1001 st1 st2 1001
0110 st1 st2 0000
0000 st1 st2 0000
0011 st1 st2 0011
0101 st1 st2 0101
0111 st1 st2 0001
1011 st1 st2 1011
1101 st1 st2 1101
1110 st1 st2 1000
1111 st1 st2 1001
1000 st2 st3 1000
0100 st2 st3 0100
0010 st2 st3 0010
0001 st2 st3 0001
0000 st2 st3 0000
11-- st2 st3 0000
1-1- st2 st3 0000
1--1 st2 st3 0000
-11- st2 st3 0000
-1-1 st2 st3 0000
--11 st2 st3 0000
1000 st3 st0 1000
0100 st3 st0 0100
0010 st3 st0 0010
0001 st3 st0 0001
0000 st3 st0 0000
11-- st3 st0 0000
1-1- st3 st0 0000
1--1 st3 st0 0000
```



```
-11- st3 st0 0000  
-1-1 st3 st0 0000  
--11 st3 st0 0000  
.e
```

states:	state-assignment
0	00
1	10
2	11
3	01

## *Appendix B*

This appendix provides an over view of SIS [73]. It contains brief description of the sequential circuit model. SIS is an interactive tool for synthesis and optimization of sequential circuits. The research idea and code implementation contains there in the product of the efforts of many graduate students and professors at US Berkeley. Many of the research projects in the course involved exploration of new sequential optimization algorithm that consider logic and state together. This necessitated the development of a sequential circuit synthesis design program. Furthermore, such a program would be convenient for synthesis a sequential design from a symbolic state representation to a mapped implementation without user intervention in patching the input and output of various tools together. SIS produces an optimized netlist in a target technology given as input a state. A sequential circuit can be input to SIS in several ways (see Figure 1B), allowing SIS to be used at various stages of the design process. The two most common entry points are a netlist of gates and a finites machine in STT.

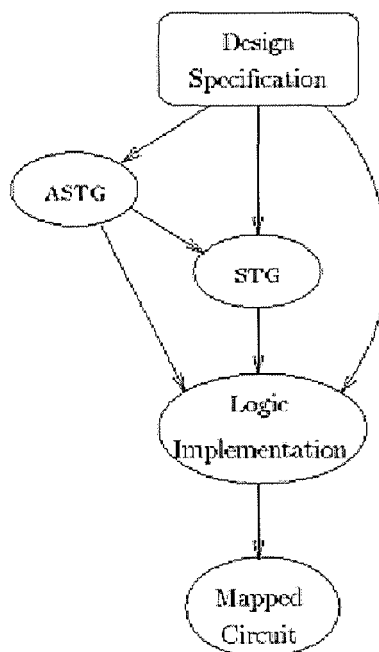


Figure 1B. A design is specified as an ASTG, STG, or logic implementation

## Finite State Machine Descriptions

A sequential circuit can be specified in BLIF (Berkeley logic interchange format) logic form, as a finite state machine, or both. An *fsm-description* is used to insert a finite state machine description of the current model. It is intended to represent the same sequential circuit as the current model (which contains logic), but in FSM form. The format of an *fsm-description* is:

```
.start_kiss
.i <num-inputs>
.o <num-outputs>
[p <num-terms>]
[s <num-states>]
[r <reset-state>]
<input> <current-state> <next-state> <output>
.
.
.
<input> <current-state> <next-state> <output>
.end_kiss
[latch_order <latch-order-list>]
[<code-mapping>]
```

*num-inputs* is the number of inputs to the FSM, which should agree with the number of inputs in the *.inputs* construct for the current model.

*num-outputs* is the number of outputs of the FSM, which should agree with the number of outputs in the *.outputs* construct for the current model.

*num-terms* is the number of ``<input> <current-state> <next-state> <output>'' 4-tuples that follow in the FSM description.

*num-states* is the number of distinct states that appear in ``<current-state>'' and ``<next-state>'' columns.

*reset-state* is the symbolic name for the reset state for the FSM; it should appear somewhere in the ``<current-state>'' column.

*input* is a sequence of *num-inputs* members of {0, 1, --}.

*output* is a sequence of *num-outputs* members of {0, 1, --}.

*current-state* and *next-state* are symbolic names for the current state and next state transitions of the FSM.

*latch-order-list* is a white-space-separated sequence of latch outputs.

*code-mapping* is newline separated sequence of:

.code <symbolic-name> <encoded-name>

*num-terms* and *num-states* do not have to be specified. If the *reset-state* is not given, it is assigned to be the first state encountered in the ``<current-state>" column.

The ordering of the bits in the *input* and *output* fields will be the same as the ordering of the variables in the *.inputs* and *.outputs* constructs if both an *fsm-description* and logic functions are given.

*latch-order-list* and *code-mapping* are meant to be used when both an *fsm-description* and a logical description of the model are given. The two constructs together provide a correspondence between the latches in the logical description and the state variables in the *fsm-description*. In a *code-mapping*, *symbolic-name* consists of a symbolic name from the ``<current-state>" or ``<next-state>" columns, and *encoded-name* is the pattern of bits ({0, 1}) that represent the state encoding for *symbolic-name*. The *code-mapping* should only be given if both an *fsm-description* and logic functions are given. *.latch-order* establishes a mapping between the bits of the *encoded-names* of the *code-mapping* construct and the latches of the network. The order of the bits in the encoded names will be the same as the order of the latch outputs in the *latch-order-list*. There should be the same number of bits in the *encoded-name* as there are latches if both an *fsm-description* and a logical description are specified.

If both *logic-gates* and an *fsm-description* of the model are given, the *logic-gate* description of the model should be consistent with the *fsm-description*, that is, they should describe the same circuit. If they are not consistent there will be no sensible way to interpret the model, which should then cause an error to be returned.

If only the *fsm-description* of the network is given, it may be run through a state assignment routine and given a logic implementation. A sole *fsm-description*, having no logic implementation, cannot be inserted into another model by a *model-reference*, the state assigned network, or a network containing both *logic-gates* and an *fsm-description* can.

**Example of an fsm-description:**

```
.model 101          # outputs 1 whenever last 3 inputs were 1, 0, 1
.start_kiss
.i 1
.o 1
0 st0 st0 0
1 st0 st1 0
0 st1 st2 0
1 st1 st1 0
0 st2 st0 0
1 st2 st3 1
0 st3 st2 0
1 st3 st1 0
.end_kiss
.end
```

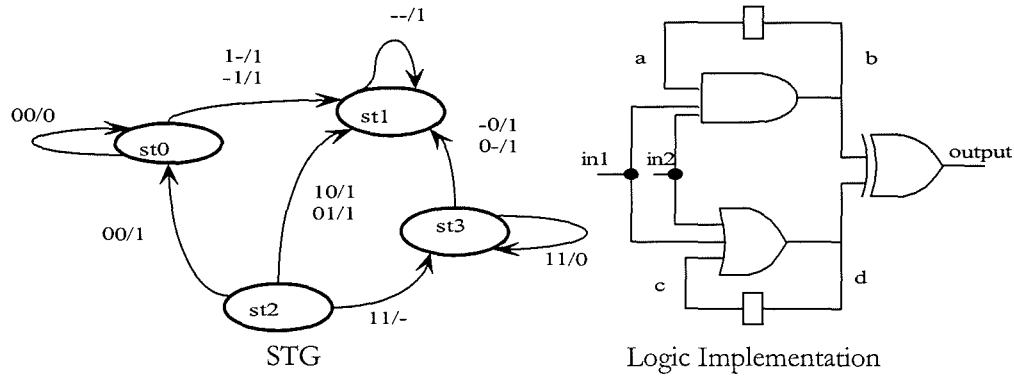
Above example with a consistent *fsm-description* and logical description:

```
.model
.inputs v0
.outputs v3.2
.latch [6] v1 0
.latch [7] v2 0
.start_kiss
.i 1
.o 1
.p 8
.s 4
.r st0
0 st0 st0 0
1 st0 st1 0
0 st1 st2 0
1 st1 st1 0
```

## Appendix B

## SIS for FSM

```
0 st2 st0 0
1 st2 st3 1
0 st3 st2 0
1 st3 st1 0
.end_kiss
.latch_order v1 v2
.code st0 00
.code st1 11
.code st2 01
.code st3 10
.names v0 [6]
1 1
.names v0 v1 v2 [7]
-1- 1
1-0 1
.names v0 v1 v2 v3.2
101 1
.end
```



Partial BLIF file format:

```
.model circuit
.inputs in1 in2
.outputs output
.start_kiss
.i 2
.o 1
00 st0 st0 0
-0 st0 st1 1
.....
.....
.....
.end_kiss

.latch b a
.latch d c

.name a in1 in2 b
111 1
.name c in1 in2 d
--1 1
-1- 1
1-- 1
.name b d output
01 1
10 1
.end
```

**Espresso for logic minimization**

```

.i 1      #Number of input
.o 2      #Number of output
.p 14     #Number of product terms
.s 7      #Number of states in machine

0 START   state6 00
0 state2   state5 00
0 state3   state5 00
0 state4   state6 00
0 state5   START  0
0 state6   START 01
0 state7   state5 00
1 state6   state2 01
1 state5   state2 10
1 state4   state6 10
1 state7   state6 10
1 START    state4 00
1 state2    state3 00
1 state3    state7 00

```

The result of a state assignment produced by SIS using NOVA program is:

```

$ NOVA state_assign -e h dk27.kiss2

.code START    100
.code state2    101
.code state3:   000
.code state4    111
.code state5    010
.code state6    110
.Code state7    011

```

The input file, which presented to ESPRESSO state minimization for the benchmark dk27.kiss sets up for a kiss-style minimization using state assignment, from NOVA is



```

.i 4                                #Number of input
.o 5                                #Number of output
.p 14                               #Number of product terms
0 100    110  00
0 101    010  00
0 000    010  00
0 111    110  00
0 010    100  10
0 110    100  01
0 011    100  10
1 100    111  00
1 101    000  00
1 000    011  00
1 111    110  10
1 010    101  10
1 110    101  01
1 011    110  10
.e

```

The result output file produced by ESPRESSO is shown as flows:

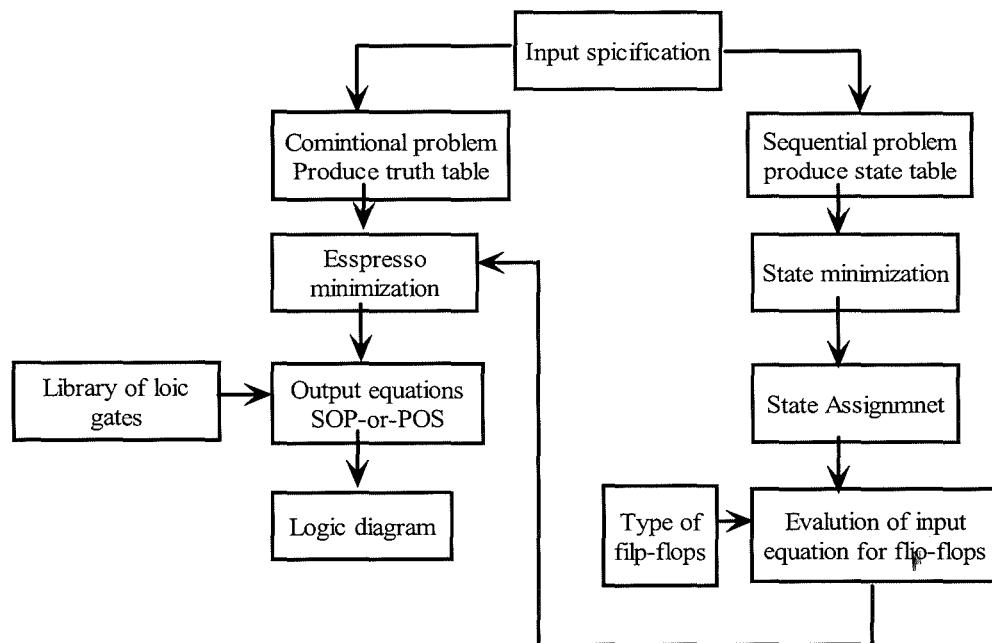
```

.i 4
.o 5
.p 9
0 1 0 - 01000
1 - 11  10010
-110    00001
-1- 0    10000
1- -0    00100
- 010    10010
- - 11    01000
- - 00    01000
-1 1-    10000
.e

```

## Design of Sequential Circuits

The design of a synchronous sequential circuit starts from a set of specifications and culminates in a logic diagram or a list of Boolean functions from which a logic diagram can be obtained. In contrast to a combinational logic, which is fully specified by a truth table, a sequential circuit requires a state table for its specification. The first step in the design of sequential circuits is to obtain a state table or an equivalence representation, such as a state diagram. The recommended steps for the design of sequential circuits are shown in the Figure below.

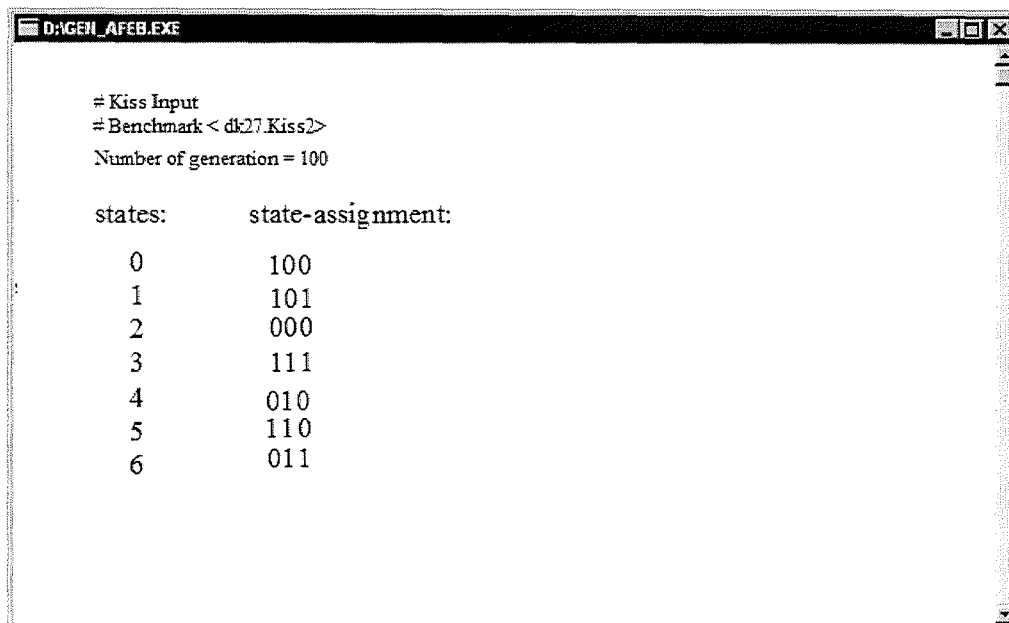


In this appendix we will consider a number of evolved dk27 benchmark designs. These circuits evolved using a gate-level EHW proposed in chapter 5 and it is extension to work done by Kalganova [24].

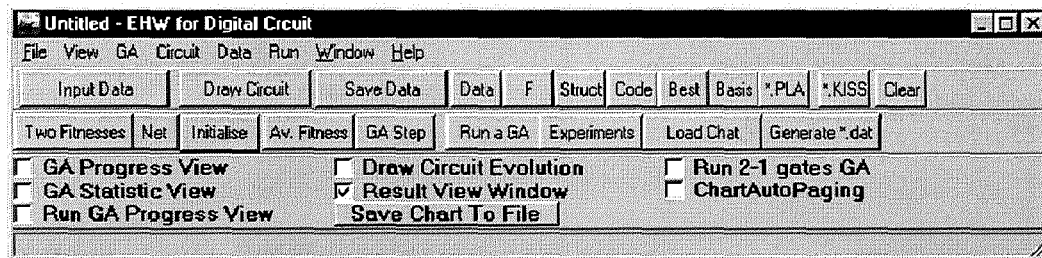
## 1. Generate state assignment for dk27.kiss2 benchmark

```
.i 1
.o 2
.p 14
.s 7
0 START state6 00
0 state2 state5 00
0 state3 state5 00
0 state4 state6 00
0 state5 START 10
0 state6 START 01
0 state7 state5 00
1 state6 state2 01
1 state5 state2 10
1 state4 state6 10
1 state7 state6 10
1 START state4 00
1 state2 state3 00
1 state3 state7 00
```

states:	state-assignment:
0	001
1	011
2	010
3	000
4	101
5	111
6	100



## 2.Exterinsic EHW to design circuits



### How to input data

Initial data can be defined using input data file with extension \*.dat or using input data forms.

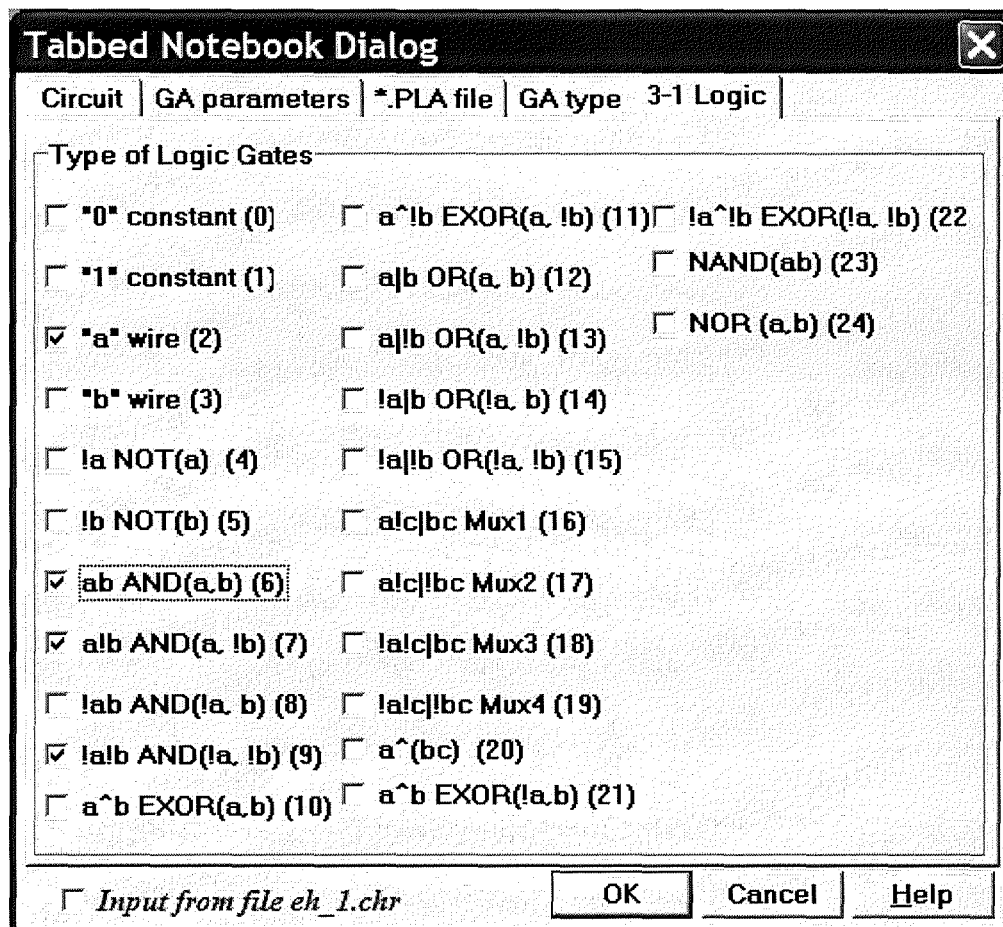
1. In order to import input data file (\*.dat), click "Input data", tick "Input from file eh1.chr", go to "\*.dat file" and then click "OK". The data from the file will be loaded to the memory. The PLA file processed has to be located in the folder "PLAfiles". In this case the initial population is generated randomly.

2. In order to import input data using forms, click "Input data", choose parameters that are required, click "OK". Input data file with name "File Name.dat" will be created in the directory specified in "\*.PLA file". In order to load data from the PLA file, choose required PLA file and click "Load PLA file". At the right, the initial data of PLA file will be appeared. In this case initial population is generated randomly.

In order to generate given initial population from the file, the data have to be written in the file "\*.chr". This file contains the initial parameters (the same as in \*.dat file) and the population of chromosomes that has to be loaded into memory. In order to download the chromosomes into memory click "Draw circuit", choose "Slide Best Circuits", click "Open File", choose the file with extension \*.chr and click "OK". The initial parameters and chromosome genotypes will be downloaded to the memory. The downloaded parameters remain the same if "Circuit Display Form" is closed.

Tabbed Notebook Dialog			
Circuit	GA parameters	*.PLA file	GA type
N-M Logic		3-1 Logic	
<b>TypeOfLogicUsed</b> <input type="radio"/> Combinational <input type="radio"/> Sequential		<b>Fitness Weight</b> Functionality <input type="text" value="0"/> # CorrectInOut <input type="text" value="0"/> # Active Gates <input type="text" value="0"/>	
<b>Circuit Options</b> # rows <input type="text" value="1"/> # columns <input type="text" value="10"/> levels back <input type="text" value="10"/>		<b>Dynamic Fitness</b> <input checked="" type="radio"/> # Gates <input type="radio"/> # VT (CMOS) <input type="radio"/> # VT (NMOS) <input type="radio"/> # VT (PMOS) <input type="radio"/> # VT (DMOS)	
<b>Cell</b> Max # ins <input type="text" value="2"/>		<b>Fitness Status</b> <input type="radio"/> Parallel Fitness F1+F2 <input checked="" type="radio"/> Parallel Fitness F1 <input type="radio"/> Consistent Fitness F1+F2 <input type="radio"/> Parallel And Consistent Fitness F1+F2 <input type="radio"/> Method of Objective Weighting <input type="radio"/> Method of Distance Functions (r=2) <input type="radio"/> Min-Max Formulation <input type="radio"/> Parallel Dynamic Fitness <input type="radio"/> Consistent Dynamic Fitness	
<b>Cell Representation</b> <input type="radio"/> Three Input One Output <input checked="" type="radio"/> Two Input One Output <input type="radio"/> MultiInput One Output <input type="radio"/> MultiInput MultiOutput		<b>File Name</b> <input type="text" value="dk27"/>	
<b>EHW Flexibility</b> <input type="checkbox"/> Circuit geometry <input type="checkbox"/> Levels-back parameter (n)		If the flexible circuit geometry is used the #rows and #cols define the MAXIMUM size of geometry used	
<input type="checkbox"/> Input from file eh 1.chr		OK	Cancel
		Help	

Input data from form



**Tabbed Notebook Dialog**

**GA parameters** | \*.PLA file | GA type | N-M Logic | \*.KISS file

**GA options**

Population size: 50

# generations: 5000

# runs total: 5

☐ History

☒ Elitism

Breeding rate: 0.6

Cell mutation rate: 0.05

Geometry mutation rate: 0.05

Selection pressure: 1

# 100% cases: 0

**Crossover**

☒ Gene Uniform

☐ Cell Uniform

☐ Geometry Uniform

**Mutation Level 1**

☒ Gene

☐ Geometry

☐ Gene And Geometry

**Mutation Low Level**

☐ [1, Nmax]

☐ [1, Ncur]

☐ Ncur+-1

**Selection**

☒ Tournament (2)

☐ Roulette Wheel

☐ Standard Proportions

☐ Input from file eh 1.chr

OK Cancel Help

\*.dat file contains the initial data that include the Evolvable Hardware and Evolutionary Algorithm parameters.

---

```

5 population_size
100 #_runs_total
5000 #_generations
0.05 ParameterMutationRate
0.05 GeometryMutationRate
0.6 BreedingRate
1 SelectionPressure
1 Elitism
GATypeOfAlgorithm
Percentage RateType
GeneUniform CrossoverType
Tournament SelectionType
Gene MutationType
No MutationTypeLowLevel
TwoInputOneOutput CellRepresentation
ParallelDynamic FitnessStatus
    
```

---

## Extrinsic EHW tools

0 FunctionalityWeight  
0 NumCompleteInOutWeight  
0 NumActiveGatesWeight  
No TypeOfExperiment  
1 gate\_distribution(1-proportional  
0 history  
1 num\_rows  
10 num\_cols  
10 levels\_back  
2 MaxNumCellInputs  
1 MaxNumCellOutputs  
0 flex\_geometry  
0 flex\_levels\_back  
DK27a.PLAfile  
16 num\_basis\_gates  
0 0  
1 0  
2 1  
3 0  
4 0  
5 0  
6 1  
7 1  
8 1  
9 1  
10 1  
11 1  
12 0  
13 0  
14 0  
15 0  
16 0

---

### \*.chr file example

---

\*.chr file contains the chromosome genotype and initial data required to download this genotype in the memory.  
"The\_population of 5" defines that 5 chromosome will be loaded in the memory. If this parameter is not specified, the chromosomes are not loaded in the memory.

---

5 population\_size  
1 #\_runs\_total  
500 #\_generations  
0.03 ParameterMutationRate  
0.05 GeometryMutationRate  
0.6 BreedingRate  
1 SelectionPressure  
1 Elitism  
GA TypeOfAlgorithm  
Percentage RateType  
GeneUniform CrossoverType



## Extrinsic EHW tools

Tournament SelectionType  
Gene MutationType  
No MutationTypeLowLevel  
No GA\_control  
TWOInputOneOutput CellRepresentation  
No TypeOfInvertedInput  
No FitnessDisplay  
No DisplayDistribution  
Yes PopulationDisplay  
No BestRunChromosome  
No BestChangedChromosomeDisplay  
No Best100ChromosomesDisplay  
Yes ResultFileDisplay  
ParallelFitnessF1F2 FitnessStatus  
0 FunctionalityWeight  
0 NumCompleteInOutWeight  
0 NumActiveGatesWeight  
No TypeOfExperiment  
1 gate\_distribution(1-proportional  
0 history  
1 num\_rows  
10 num\_cols  
10 levels\_back  
2 MaxNumCellInputs  
1 MaxNumCellOutputs  
0 flex\_geometry  
0 flex\_levels\_back  
DK27.PLA PLAfile  
DK27 DataFileName  
16 num\_basis\_gates  
0 0  
1 0  
2 1  
3 0  
4 0  
5 0  
6 0  
7 1  
8 1  
9 1  
10 0  
11 0  
12 0  
13 0  
14 0  
15 0  
16 0

## TwoInputOneOutput Rectangular Array Chromosome Representation

The TwoInputOneOutput rectangular array chromosome representation should be represented by the following way:

The cell data are the following:

CellI: <FunctionalGene input1 input2 >

The chromosome data are

```
//*****
```

**The\_chromosome** <ChromosomeIndex> <NumColumns> x <NumRows>

**4:** <Cell0>    **6:** <Cell2>    **8:** <Cell4>    **10:** <Cell6>

**5:** <Cell1>    **7:** <Cell3>    **9:** <Cell5>    **11:** <Cell7>

**Outputs:** <Output1> <Output2> .. <OutputM>

**Fitness1** <Fitness1>

```
//*****
```

All chromosome data is given in rectangular array representation.

An example of \*.chr file is given below.

```
//*****
```

eh.dat

The\_population of 5

The\_chromosome 0 3 x 5

8: 7 6 7    13: 2 8 3    18: 14 17 10

9: 4 7 4    14: 11 10 0    19: 4 8 13

10: 3 7 5    15: 5 1 0    20: 11 8 16

11: 12 4 6    16: 4 12 6    21: 11 9 10

12: 8 4 2    17: 3 10 1    22: 12 9 12

Outputs: 18 19

Fitness 50

The\_chromosome 1 5 x 3

8: 8 5 6    11: 1 10 7    14: 12 9 11    17: 2 16 12    20: 10 16 18

9: 7 2 7    12: 12 8 6    15: 3 8 9    18: 5 16 12    21: 5 18 14

10: 4 3 2    13: 11 5 2    16: 1 11 13    19: 8 13 15    22: 2 17 18

Outputs: 21 21

Fitness 37.5

The\_chromosome 2 2 x 3

8: 6 5 4    11: 7 4 2

9: 10 2 4    12: 4 5 10

10: 3 6 5    13: 12 5 4

Outputs: 12 13

Fitness 62.5

The\_chromosome 3 3 x 2

8: 8 4 7    10: 12 3 4    12: 6 8 9

9: 12 4 1    11: 9 9 7    13: 12 11 8

Outputs: 10 10

Fitness 50

## Extrinsic EHW tools

The\_chromosome 4 5 x 2

8: 10 7 5    10: 9 5 4    12: 5 11 9    14: 2 13 11    16: 2 12 13  
9: 4 6 7    11: 2 4 6    13: 3 11 10    15: 4 12 10    17: 11 13 14

Outputs: 17 16

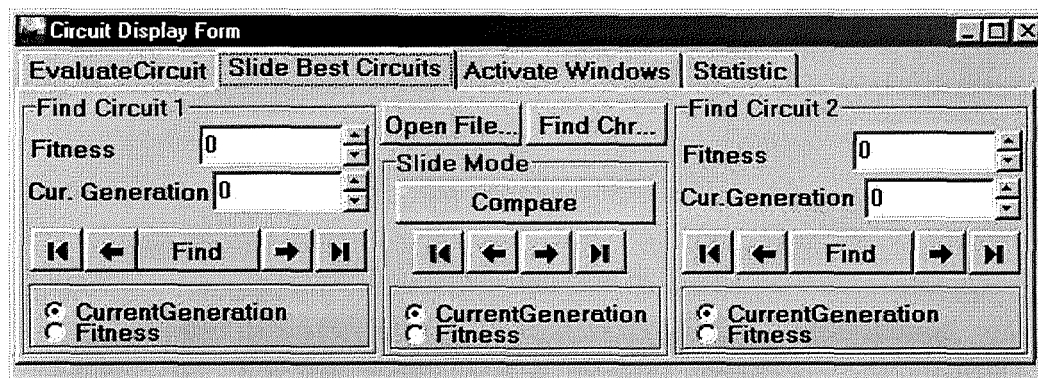
Fitness 50

## How to display data

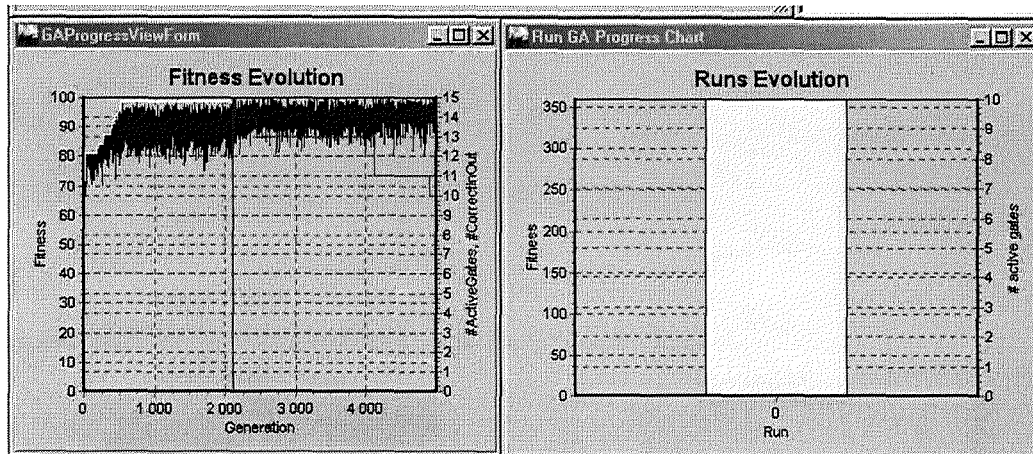
The chromosome genotype can be displayed in graphical form. In this case click "Draw Circuit", go to "Evaluate Chromosome", chose the number of chromosome required to be displayed and click "Draw Chromosome". The graphical form of chromosome genotype will be displayed in the new window. As an alternative way the chromosome can be downloaded from the file and, then, the required chromosome can be displayed by clicking "Find".

The algorithm performance can be displayed in different way as well. In order to observe the progress of GA in graphical form, tick any of the following options in the main menu: "GA progress View" (It displays the GA progress during run as a function of the number of generations and the fitness parameters.), "GA Statistic View" (It displays the statistic parameters of GA.), "Run GA Progress View" (It displays the results of each run.). In order to manipulate by parameters displayed in the graph can be manipulated, tick "ChartAuto Paging". In order to save data from the chart into file, click "Save Chart to File".

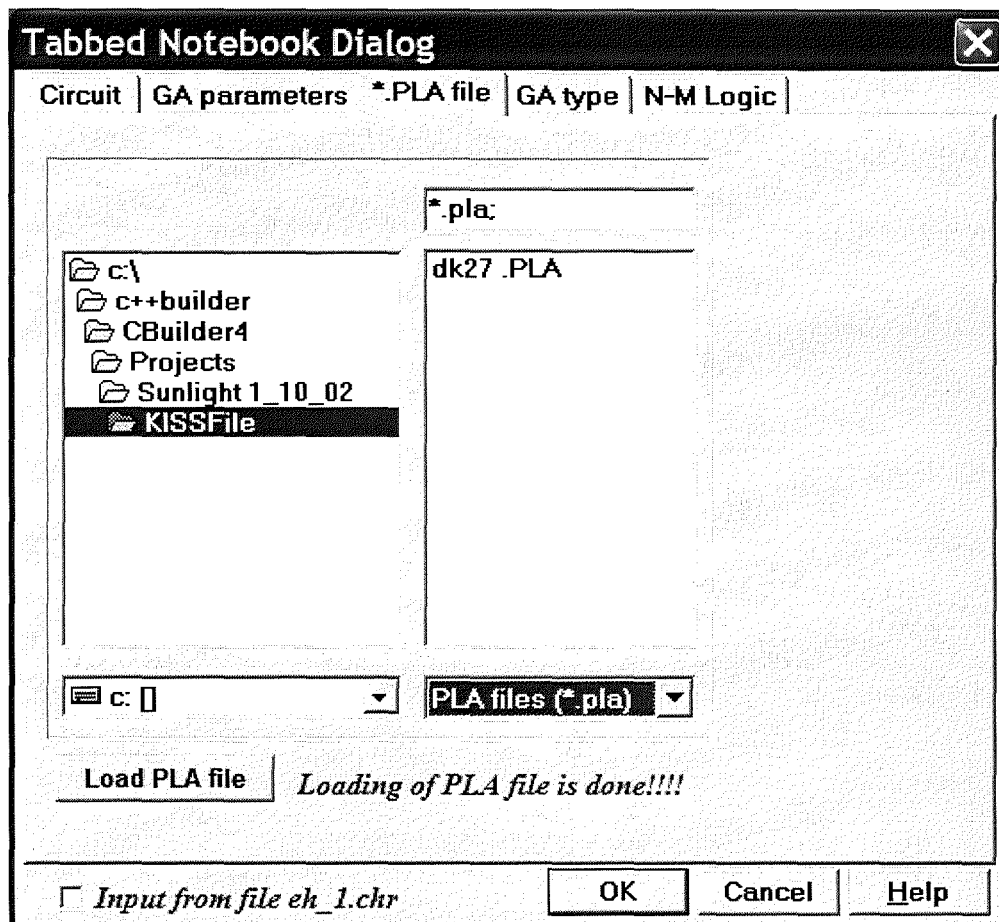
The data stored in the memory can be displayed in the window, if "Result View Window" is ticked. In this case depending on the data required to be checked, the buttons "Data", "F", "Struct", "Code", "Best", "Basis", "PLA file" can be clicked. In order to clear data in the "Result View" window, click "Clear



Circuit display form



Graphic display during evolutionary algorithm execution



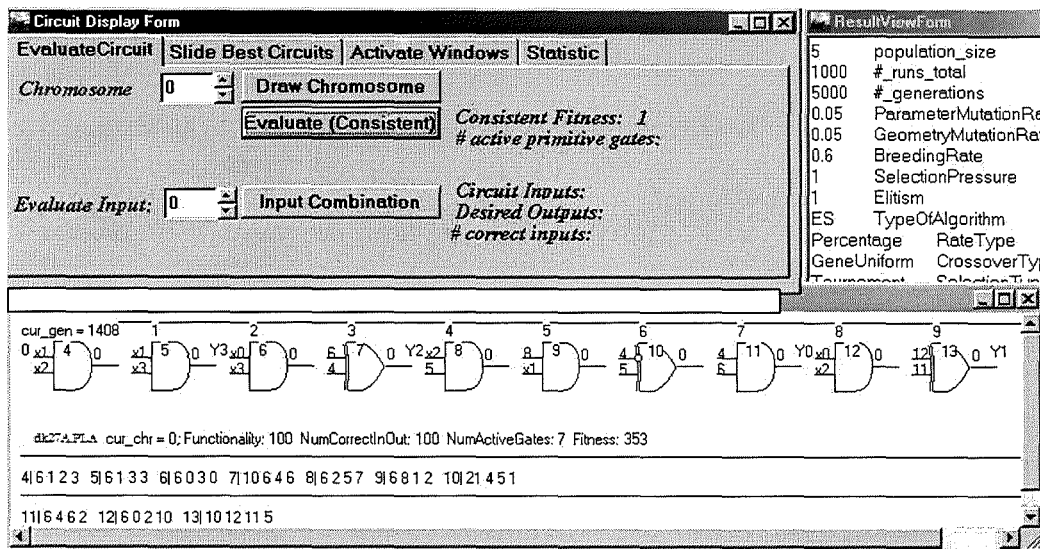
Input data from file

## How to evaluate data

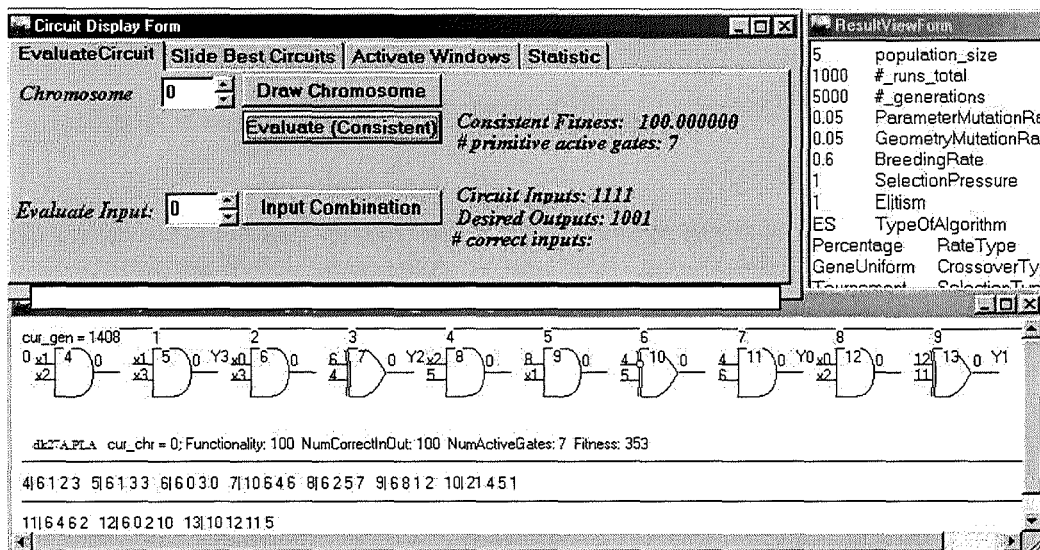
In order to evaluate data, load the data into memory, click "Draw Circuit", and then click "Evaluate Circuit".

In order to define the percentage of correct bits in evaluated circuit, click "Evaluate (Consistent)".

In order to evaluate the correctness of the circuit for the specific input combination, choose required input combination and click "Input Combination". The logic values for each logic gate will be displayed in the "Circuit View Form" and the percentage of correct bit and evaluated input combination will be displayed in the **"Evaluate Circuit"** window.

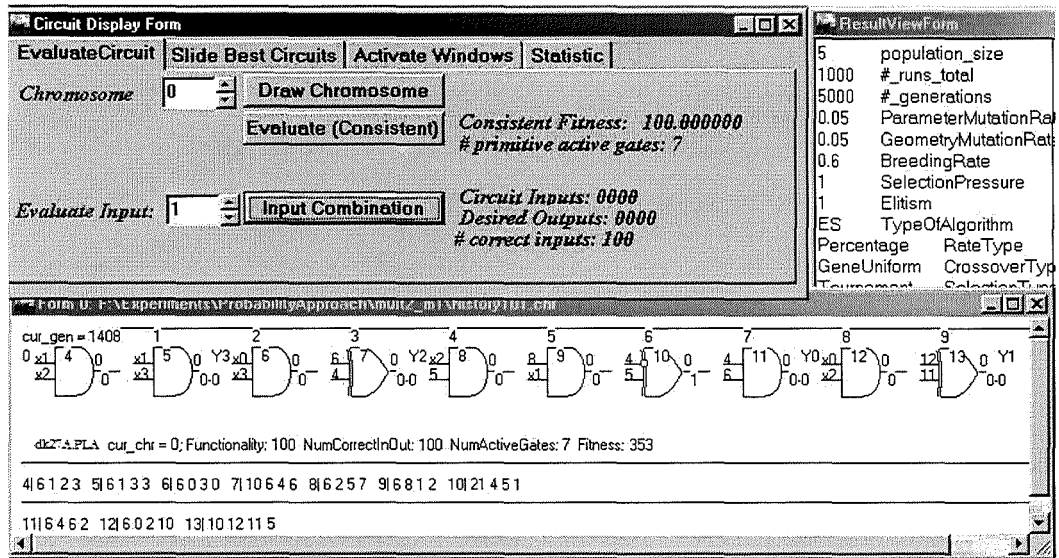


Before circuit evaluation

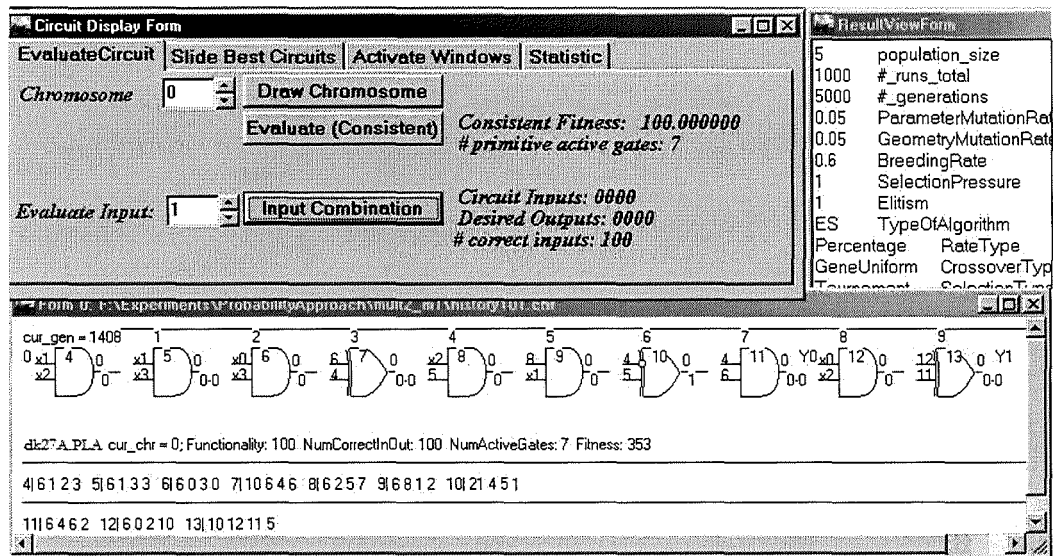


After Evaluation

## Extrinsic EHW tools



After input evaluation

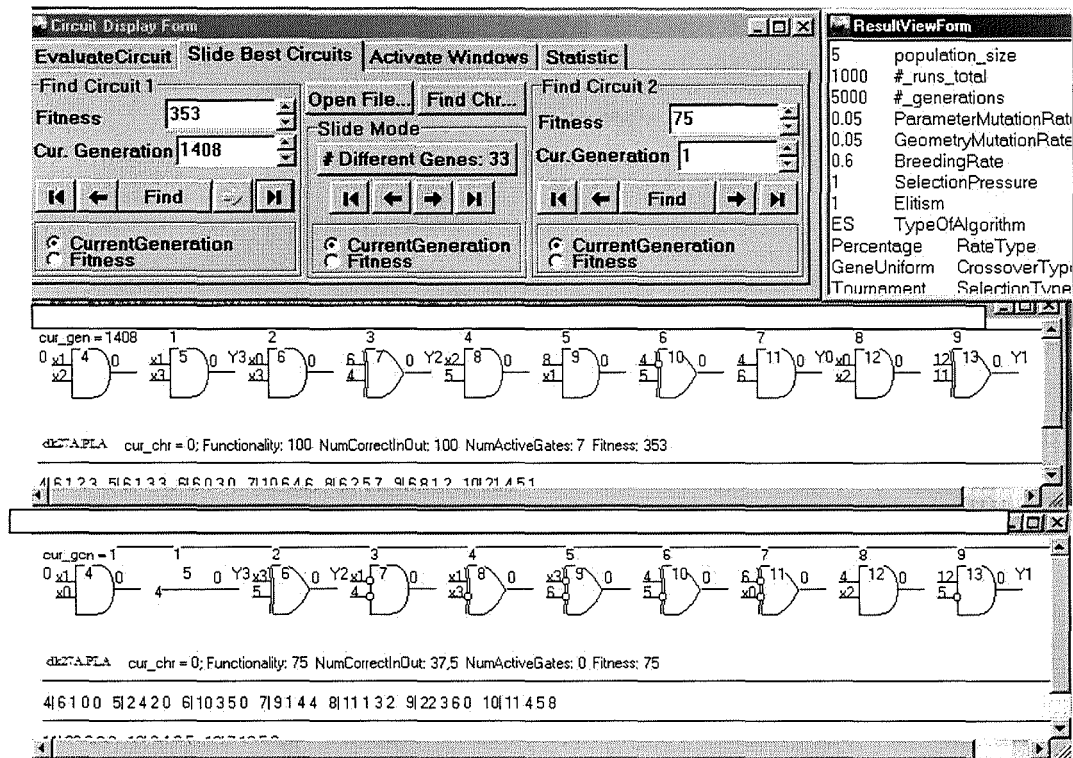


After input combination evaluation

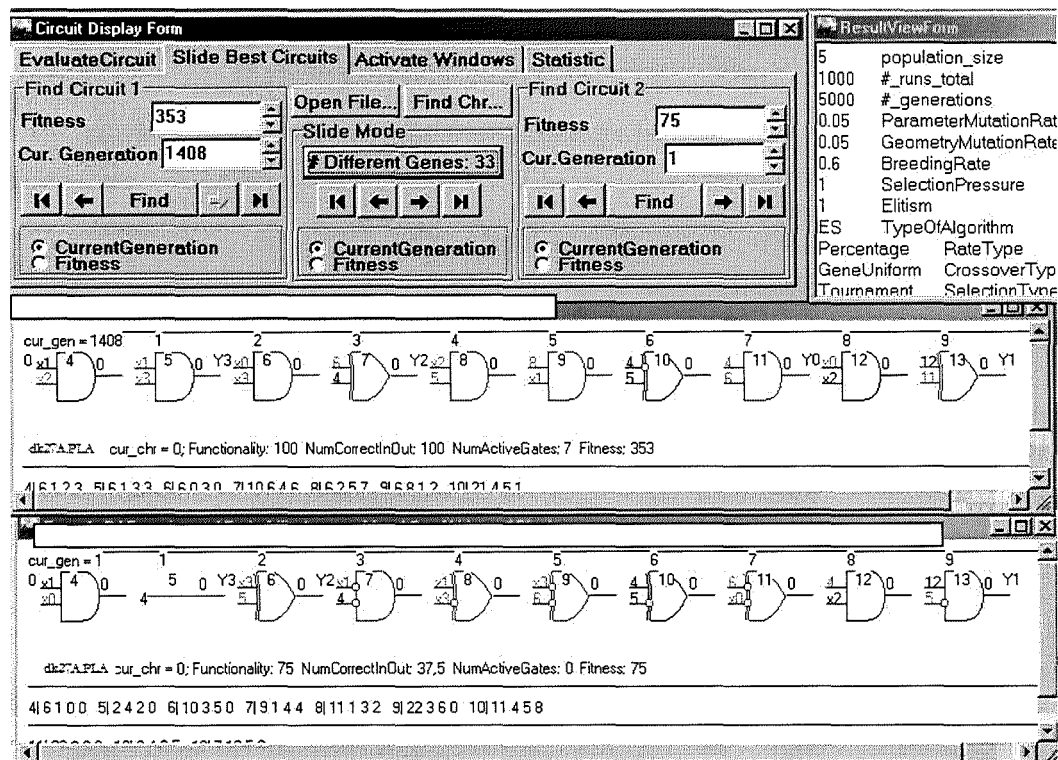
## How to compare data

In order to compare two genotypes, load into memory two chromosomes using "Find Circuit 1" and "Find Circuit 2" ("Draw Circuit" -> "Slide Best Circuits") and press "Compare". The different genes in the chromosome genotype are displayed in colour.

## Extrinsic EHW tools



Before comparison



After comparison



## **Appendix D**

### **Information about CD**

The attached CD contains the programs developed in the previous chapters.

The main menu of CD includes

- Developed software (GA for SAP &EHW)
- Source codes of the programs
- Electronic version of thesis