# 5

# Evolutionary Synthesis of Synchronous Finite State Machines

Nadia Nedjah and Luiza de Macedo Mourelle

Department of System Engineering and Computation,
Engineering Faculty,
State University of Rio de Janeiro,
Rua São Francisco Xavier, 524, Sala 5022-D,
Maracanã, Rio de Janeiro, Brazil
(nadia | ldmm)@eng.uerj.br
www.eng.uerj.br

Synchronous finite state machines are very important for digital sequential designs. Among other important aspects, they represent a powerful way for synchronising hardware components so that these components may cooperate adequately in the fulfilment of the main objective of the hardware design. In this chapter, we propose an evolutionary methodology synthesise finite state machines. First, we optimally solve the state assignment $NP$-complete problem, which is inherent to designing any synchronous finite state machines using genetic algorithms. This is motivated by the fact that with an optimal state assignment one can physically implement the state machine in question using a minimal hardware area and response time. Second, with the optimal state assignment provided, we propose to use the evolutionary methodology to yield optimal evolvable hardware that implement the state machine control component. The evolved hardware requires a minimal hardware area and introduces a minimal propagation delay of the machine output signals.

## 5.1 Introduction

Sequential digital systems or simply finite state machines have two main characteristics: there is at least one feedback path from the system output signal to the system input signals; and there is a memory capability that allows the system to determine current and future output signal values based on the previous input and output signal values [15].

Traditionally, the design process of a state machine passes through five main steps, wherein the second and third steps may be bypassed as shown in Fig. 5.1:

1. the specification of the sequential system, which should determine the next states and outputs of every present state of the machine. This is done using state tables and state diagrams;
2. the state reduction, which should reduce the number of present states using equivalence and output class grouping;
3. the state assignment, which should assign a distinct combination to every present state. This may be done using Armstrong-Humphrey heuristics [15];
4. the minimisation of the control combinational logic using K-maps and transition maps;
5. finally, the implementation of the state machine, using gates and flip-flops.

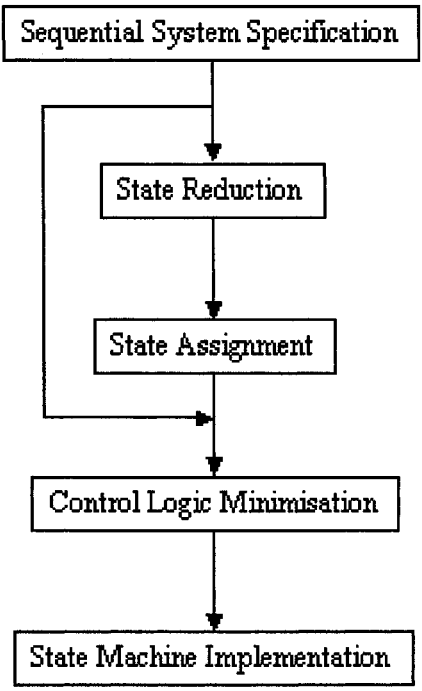



**Fig. 5.1.** The structural description of a finite synchronous state machine

In this chapter, we concentrate on the third and forth steps of the design process, i.e. the state assignment problem and the control logic minimisation. We present a genetic algorithm designed for finding a state assignment of a

given synchronous finite state machine, which attempts to minimise the cost related to the state transitions. Then, we use genetic programming to evolve the circuit that controls the machine current and next states.

The remainder of this chapter is organised into seven sections. In Section 5.2, we introduce the problems that face the designer of finite state machine, which are mainly the state assignment problem and the control logic. We show that a better assignment improves considerably the cost of the control logic. In Section 5.3, we give a thourough overview on the principles of evolutionary computations and genetic algorithms and their application to solve NP-problems. In Section 5.4, we design a genetic algorithm for evolving best state assignment for a given state machine specification. We describe the genetic operators used as well as the fitness function, which determines whether a state assignment is better that another and how much. In Section 5.5, we present results evolved through our genetic algorithm for some well-known benchmarks. Then we compare the obtained results with those obtained by another genetic algorithm described in [1] as well as with NOVA, which is uses well established but non-evolutionary method [16]. In Section 5.6, we briefly introduce the genetic programming concepts and their applications to engineer evolvable hardware. Subsequently, we present a genetic programming-based synthesiser for evolving minimal control logic circuit provided the state assignment for the specification of the state machine in question. We describe the circuit encoding, genetic operators used as well as the fitness function, which determines whether a control logic design is better than another and how much. In Section 5.7, we compare the are and time requirements of the designs evolved through our evolutionary synthesiser for some well-known benchmarks and compare the obtained results with those obtained using the traditional method to design state machine, i.e. using Karnaugh maps and flip-flop transition maps. In Section 5.8, we summarise the ideas presented throughout the chapter and draw some conclusions.

## 5.2 Synchronous Finite State Machines

Once the specification and the state reduction step have been completed, the next step consists then of assigning a code to each state present in the machine. It is clear that if the machine has $N$ distinct states then one needs $N$ distinct combinations of 0s and 1s. So one needs $K$ flip-flops to store the machine current state, wherein $K$ is the smallest positive integer such that $2^K \geq N$. The state assignment problem consists of finding the best assignment of the flip-flop combinations to the machine states. Since a machine state is nothing but a counting device, combinational control logic is necessary to activate the flip-flops in the desired sequence. This is shown in Fig. 5.2, wherein the feedback signals constitute the machine state, the control logic is a combinational circuit that computes the state machine output signals (also called *primary output signals*) from the state signals (also called *current state*)

and the input signals (also called *primary input signals*). It also produces the signals of new machine state (also called *next state*).

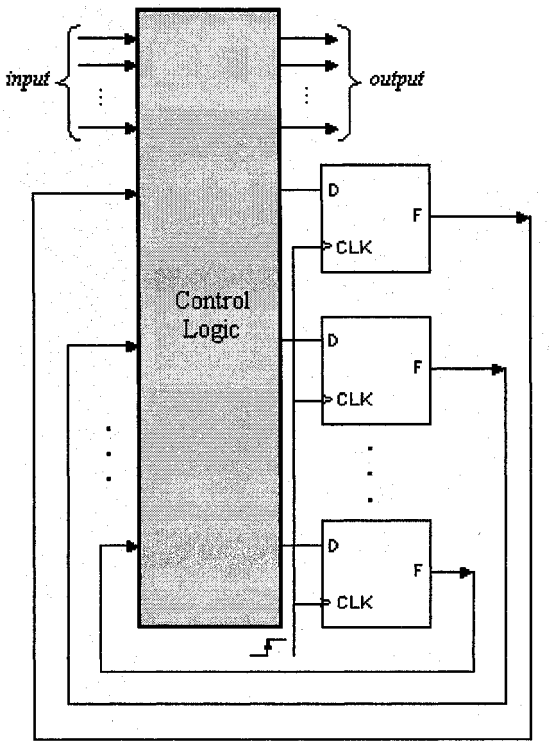



**Fig. 5.2.** The structural description of a finite synchronous state machine

The control logic component in a state machine is responsible of generating the primary output signals as well as the signal that form the next state. It does so using the primary input signals and the signals that constitute the current state (see Fig. 5.2). Traditionally, the combinational circuit of the control logic is obtained using the transition maps of the flip-flops [15]. Given a state transition function, it is expected that the complexity (area and time) and so the cost of the control logic will vary for different assignments of flip-flop combinations to allowed states. Consequently, the designer should seek the assignment that minimises the complexity and so the cost of the combinational logic required to control the state transitions.

### 5.2.1 Example of State Machine

Consider the state machine of one input signal ($I$), one output signal ($O$) and four states whose state transition function is given in tabular form in Table

5.1 and assume that we use D-flip-flops to store the machine current state. Then the state assignment $A_0 = \{s_0 \equiv 00, s_1 \equiv 11, s_2 \equiv 01, s_3 \equiv 10\}$ requires a control logic that consists of three AND gates, five AND gates and three OR gates while the assignments $A_1 = \{s_0 \equiv 00, s_1 \equiv 10, s_2 \equiv 01, s_3 \equiv 11\}$ requires a control logic that consists of only two NOT gates, five AND gates and two OR gates. The schematics of the state machines that encode the state according to state assignments $A_0$ and $A_1$ are given in Fig. 5.3 and Fig. 5.4 respectively.

**Table 5.1.** Example of state transition function

| Present State | Next State | | Output ($O$) | |
|---|---|---|---|---|
| | $I = 0$ | $I = 0$ | $I = 1$ | $I = 1$ |
| $q_0$ | $q_0$ | $q_0$ | 0 | 0 |
| $q_1$ | $q_2$ | $q_2$ | 0 | 1 |
| $q_2$ | $q_0$ | $q_0$ | 1 | 0 |
| $q_3$ | $q_2$ | $q_2$ | 1 | 1 |

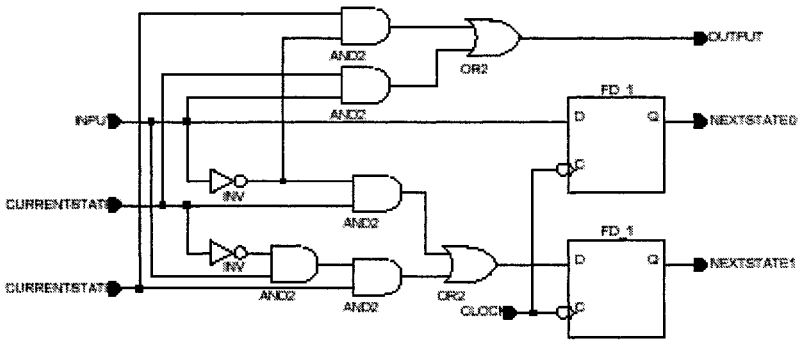



**Fig. 5.3.** The machine state schematics for state assignment $A_0$

In Section 5.3, we concentrate on the third step of the design process, i.e. the state assignment problem. We present a genetic algorithm designed for finding a state assignment of a given synchronous finite state machine, which attempts to minimise the cost related to the state transitions. In Section 5.5, we focus on evolving minimal control logic for state machines, provided the state assignment.

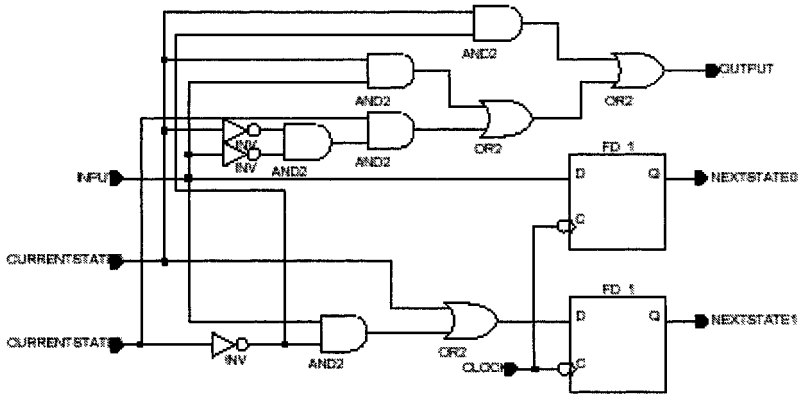



**Fig. 5.4.** The machine state schematics for state assignment $A_1$

## 5.3 Principles of Genetic Algorithms

Evolutionary algorithms are computer-based solving systems, which use the evolutionary computational models as key element in their design and implementation. A variety of evolutionary algorithms have been proposed. The most popular ones are genetic algorithms [13]. They have a conceptual base of simulating the evolution of individual structures via the Darwinian natural selection process. The process depends on the adherence of the individual structures as defined by its environment to the problem pre-determined constraints. *Genetic algorithms* are well suited to provide an efficient solution of $NP$-hard problems [4].

Genetic algorithms maintain a population of individuals that evolve according to selection rules and other genetic operators, such as mutation and recombination. Each individual receives a measure of fitness. Selection focuses on individuals, which shows high fitness. *Mutation* and *crossover* provide general heuristics that simulate the recombination process. Those operators attempt to perturb the characteristics of the parent individuals as to generate distinct offspring individuals.

Genetic algorithms are implemented through the following generic algorithm described by Algorithm 5.1, wherein parameters $ps$, $f$ and $gn$ are the *population size*, *fitness* of the expected individual and the number of *generation* allowed respectively.

In Algorithm 5.1, function *intialPopulation* returns a valid random set of individuals that compose the population of the first generation, function *evaluate* returns the fitness of a given population. Function *select* chooses according to some criterion that privileges *fitter* individuals, the individuals that will be used to generate the population of the next generation and function reproduction implements the crossover and mutation process to yield the

**Algorithm 5.1** Genetic Algorithms

**input:** population size ($ps$), expected fitness ($f$), last generation number ($gn$);
**output:** fittest individual ($fit$);
1. $generation := 0$;
2. $population :=$ initialPopulation();
3. $fitness :=$ evaluate($population$);
4. do
5. $\quad parents :=$ select($population$);
6. $\quad population :=$ reproduce($parents$);
7. $\quad fitness :=$ evaluate($population$);
8. $\quad generation := generation + 1$;
9. $\quad fit :=$ fittestIndividual($population$);
10. while($fit < f$) and ($generation < gn$);

End.

new population. The main genetic operators will be described in the following sections.

### 5.3.1 Assignment Encoding

Encoding of individuals is one of the implementation decisions one has to make in order to use genetic algorithms. It very depends on the nature of the problem to be solved. There are several representations that have been used with success [13]: *binary encoding* which is the most common mainly because it was used in the first works on genetic algorithms, represents an individual as a string of bits; *permutation encoding* mainly used in ordering problem, encodes an individual as a sequence of integer; *value encoding* represents an individual as a sequence of values that are some evaluation of some aspect of the problem; *tree encoding* represents an individual as a tree. This encoding is generally used to represent structured individuals such as computer programs, mathmatical expressions and circuits.

### 5.3.2 Individual Reproduction

Besides the parameters which represent the population size, the fitness of the expected result and the maximal number of generation allowed, the genetic algorithm has several other parameters, which can be adjust by the user so that the result is up to his or her expectation. The *selection* is performed using some selection probabilities and the *recombination*, as it is subdivided into *crossover* and *mutation* processes, depends on the kind of crossover and the mutation *rate* and *degree* to be used.

#### Selection

The selection problem consists of how to select the individuals that should yield the new population. According to Darwins evolution theory the best ones

should survive longer and create more new offspring. There are many selection methods [6], [9]. These methods include *roulette wheel* selection or *fitness proportionate* reproduction and *rank* selection. In the following, we describe the idea behind each of these selection methods. In our implementation, we use fitness proportionate reproduction.

In fitness proportionate reproduction, parents are selected according to their fitness. The better the fitness the individuals have, the higher their chances to be selected are. Imagine a roulette wheel where are placed all individuals of the population, wherein every individual has portion proportionate to its fitness, as it is shown in Fig. 5.5.

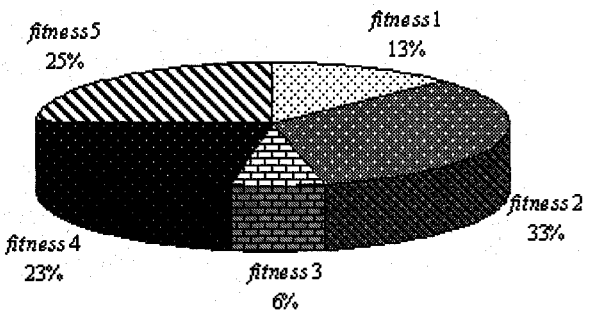


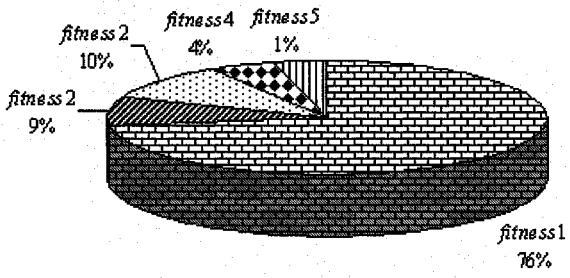

**Fig. 5.5.** Representation with the roulette wheel selection

Then a marble is thrown into the roulette and selects an individual. It is clear that individuals with bigger portion in the wheel will be selected more times. The selection process can be simulated by following steps:

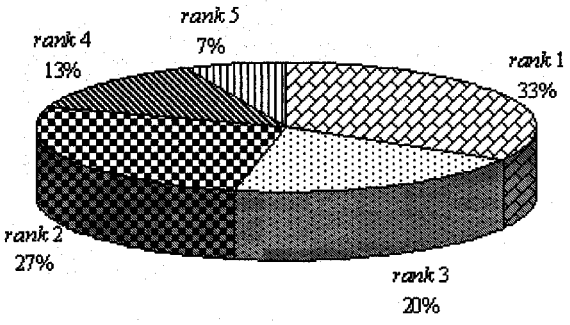
1. first, sum up the fitness of all individuals in the population and let S be the obtained sum;
2. then generate a random number from the $[0, S]$, and let $f$ be this number;
3. subsequently, go through the individuals of the population, summing up the fitness of the next one. Let $\sigma$ be this partial sum;
4. if $\sigma \geq f$, then stop the selection process and choose the current individual otherwise return to second step.

The fitness proportionate reproduction selection presents some limitations when the individual fitnesses differ too much from one another. For instance, if the best individual has a fitness of 95% of the entire roulette wheel then the other individuals will have very few, if any, chances to be selected. To get round this limitation, the rank selection method first ranks the individuals of the population according to their corresponding fitnesses. The individual with the worst fitness receives $rank_1$ and that with the best fitness receives $rank_N$, which is the number of individuals in the population. The impact of the ranking process is shown in Fig. 5.6, which represents the roulette

wheel before and after the ranking process. Rank selction may yield a slower convergence as the fittest individuals and those that are less fit have much closer ranks.




(a) before individual ranking




(b) after individual ranking

**Fig. 5.6.** Representation of the roulette wheel selection before and after ranking the individuals according to their fitnesses

### Reproduction

Given the parents populations, the reproduction can proceed using different schemes [6], [9]: a *total replacement*, *steady-state replacement* and *elitism*. In the first scheme, offspring replace their parents in the population of the next generation. That is only offspring are used to form the population of the next generation. The steady-state replacement exploits the idea that only few low-fitness individuals should be discarded in the next generation and should then be replaced by offspring. Finally, elitism exploits the idea that the best solution might be the fittest individual of the current population and so transports it unchanged into the population of the next generation. In our implementation we use the total replacement reproduction scheme as well as elitism.

Obtaining offspring that share some traits with their corresponding parents is performed by the crossover function. There are several types of crossover operators. These will be presented shortly. The newly obtained population can then suffer some mutation, i.e. some of the individuals of some of the genes. The crossover type, the number of individuals that should be mutated and how far these individuals should be altered are set up during the initialisation process of the genetic algorithm.

*Crossover*

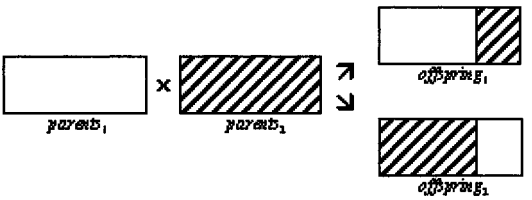
There are many ways on how to perform crossover and these may depend on the individual encoding used [13]. We present some of these techniques crossover techniques. *Single-point* crossover consists of choosing randomly one *crossover point* then, the part of the individual from the beginning of the offspring till the crossover point is copied from one parent, the rest is copied from the second parent as depicted in Fig. 5.7(a). *Double-point crossover* consists of selecting randomly two crossover points, the part of the individual from beginning of offspring to the first crossover point is copied from one parent, the part from the first to the second crossover point is copied from the second parent and the rest is copied from the first parent as depicted in Fig. 5.7(b). *Uniform crossover* copies parts randomly from the first or from the second parent. Finally, *arithmetic crossover* consists of applying some arithmetic operation to yield a new offspring.

The single-point and double-point crossover may use randomly selected crossover points to allow variation in the generated offspring and to contribute in the avoidance of premature convergence on a local optimum [5]. In our implementation, we tested all four-crossover strategies.

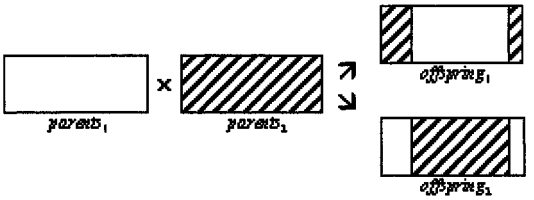
*Mutation*

Mutation consists of altering some genes of some individuals of the population obtained after crossover. The number of individuals that should be mutated is given by the parameter *mutation rate* while the parameter *mutation degree* states how many genes of a selected individual should be changed. The mutation parameters have to be chosen carefully as if mutation occurs very often then the genetic algorithm would in fact change to random search [5]. When either of the mutation rate or mutation degree is null, the population is then kept unchanged, i.e. the population obtained from the crossover procedure represents actually the next generation population.

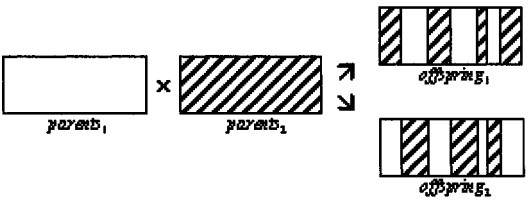
The essence of the mutation process depends on the encoding type used. When binary encoding is used, the mutation is nothing but a bit inversion of those bit genes that were randomised. When permutation encoding is used, the mutation is reduced to a permutation of some randomly selected integer genes. When value encoding is used, a very small value is added or subtracted from the randomised genes. When tree encoding is used, a content of a tree node is altered.



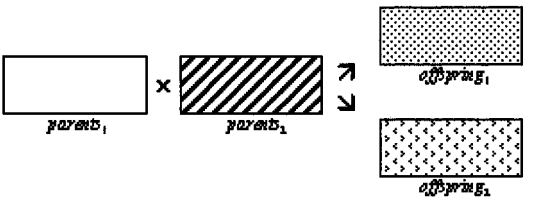

(a) single-point crossover




(b) double-point crossover




(c) uniform crossover




(d) arthmetic crossover

**Fig. 5.7.** Different types of crossover

## 5.4 Application to the State Assignment Problem

The identification of a good state assignment has been thoroughly studied over the years. In particular, Armstrong [2] and Humphrey [11] have pointed out that an assignment is good if it respects two rules, which consist of the following:

- two or more states that have the same next state should be given adjacent assignments;

- two or more states that are the next states of the same state should be given adjacent assignment. State adjacency means that the states appear next to each other in the mapped representation. In other terms, the combination assigned to the states should differ in only one position;
- the first rule should be given more important the second. For instance, state codes 0101 and 1101 are adjacent while state codes 1100 and 1111 are not adjacent.

Now we concentrate on the assignment encoding, genetic operators as well as the fitness function, which given two different assignment allows one to decide which is fitter.

### 5.4.1 State Assignment Encoding

In this case, an individual represents a state assignment. We use the integer encoding. Each chromosome consists of an array of $N$ entries, wherein entry $i$ is the code assigned to *ith.* machine state. For instance, the chromosome in Fig. 5.5 represents a possible assignment for a machine with 6 states.

| $S_0$ | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $S_5$ |
|---|---|---|---|---|---|
| 4 | 2 | 1 | 0 | 7 | 6 |

**Fig. 5.8.** Example of state assignment encoding

Note that if the considered machine has stores its state in $K$ flip-flops, then the state codes can be only chosen from the integer interval $[0, 2^K - 1]$. Otherwise, the code is not considered valid as it can be kept in the machine memory.

### 5.4.2 Genetic Operators for State Assignments

As state assignments are represented using integer encoding, we could use single-point, double-point and uniform crossovers (see Section 5.3 for details). The mutation is implemented by altering a state code by another valid state. Note that when mutation occurs, a code might be used to represent two or more distinct states. Such a state assignment is not possible. In order to discourage the selection of such assignment, we apply a penalty every time a code is used more than once within the considered assignment. This will be further discussed in next section.

### 5.4.3 State Assinment Fitness Evaluation

This step of the genetic algorithm allows us to classify the individuals of a population so that fitter individuals are selected more often to contribute in

the constitution of a new population. The fitness evaluation of state assignments is performed with respect to two rules of Armstrong [2] and Humphrey [11]:

- how much a given state assignment adheres to the first rule, i.e. how many states in the assignment, which have the same next state, have no adjacent state codes;
- how much a given state in the assignment adheres to the second rule, i.e. how many states in the assignment, which are the next states of the same state, have no adjacent state codes.

In order to efficiently compute the fitness of a given state assignment, we use an $N \times N$ *adjacency matrix*, wherein $N$ is the number of the machine states. The triangular bottom part of the matrix holds the expected adjacency of the states with respect o the first rule while the triangular top part of it holds the expected adjacency of the states with respect to the second rule. The matrix entries are calculated as in Equation 5.1, wherein $AM$ stands for the adjacency matrix, functions $next(\sigma)$ and $prev(\sigma)$ yield the set of states that are next and previous to state $\sigma$ respectively. For instance, for the state machine in Table 5.2, we get the $4 \times 4$ adjacency matrix in Fig. 5.9.

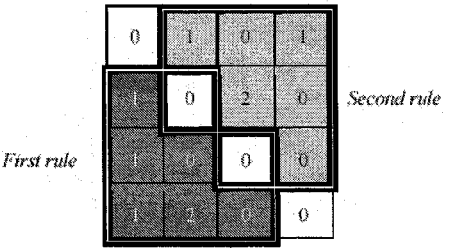



**Fig. 5.9.** Adjacency matrix for the machine state specified in Table 5.1

$$AM_{i,j} = \begin{cases} \#\left(next(q_i) \cup (next(q_j)\right) & \text{if } i > j \\ \#\left(prev(q_i) \cup (prev(q_j)\right) & \text{if } i < j \\ 0 & \text{if } i = j \end{cases} \tag{5.1}$$

Using the adjacency matrix $AM$, the fitness function applies a penalty of 2, respectively 1, every time the first rule, respectively the second rule, is broken. Equation 5.2 states the details of the fitness function applied to a state assignment $\sigma$, wherein function $na(q,p)$ returns 0 if the codes representing states $q$ and $p$ are adjacent and 1 otherwise. Note that state assignments that encode two distincts states using the same codes are penalised. Note that $\psi$ represents the penalty.

$$fitness(\sigma) = \sum_{i \neq j \& \sigma_i = \sigma_j} \psi + \sum_{i=0}^{N-1} \sum_{j=i+1}^{N-1} (AM_{i,j} + 2 \times AM_{i,j}) \times na(\sigma_i, \sigma_j) \quad (5.2)$$

For instance, considering the state machine whose state transition function is described in Table 5.1, the state assignment $\{s_0 \equiv 00, s_1 \equiv 10, s_2 \equiv 01, s_3 \equiv 11\}$ has a fitness of 5 as the codes of states $s_0$ and $s_3$ are not adjacent but $AM - 0, 3 = 1$ and $AM_{3,0} = 1$ and the codes of states $s_1$ and $s_2$ are not adjacent but $AM_{1,2} = 2$ while the assignments $\{s_0 \equiv 00, s_1 \equiv 11, s_2 \equiv 01, s_3 \equiv 10\}$ has a fitness of 3 as the codes of states $s_0$ and $s_1$ are not adjacent but $AM_{0,1} = 1$ and $AM_{1,0} = 1$.

The objective of the genetic algorithm is to find the assignment that minimise the fitness function as described in Equation 5.2. Assignments with fitness 0 satisfy all the adjacency constraints. Such an assignment does not always exist.

## 5.5 Comparative Results

In this section, we compare the assignment evolved by our genetic algorithm to those yield by another genetic algorithm [5] and to those obtained using the non-evolutionary assignment system called NOVA [16]. The examples are well-known benchmarks for testing synchronous finite state machines [3]. Table 5.2 shows the best state assignment generated by the compared systems. The size column shows the total number of states/transitions of the machine.

Table 5.3 gives the fitness of the best state assignment produced by our genetic algorithm, the genetic algorithm from [1] and the two versions of NOVA system [16]. The $\#AdjRes$ stands for the number of expected adjacency restrictions. Each adjacency according to rule 1 is counted twice and that with respect to rule 2 is counted just once. For instance, in the case of the $Shiftreg$ state machine, all 24 expected restrictions were fulfilled in the state assignment yielded by the compared systems. However, the state assignment obtained the first version of the NOVA system does not fulfil 8 of the expected adjacency restrictions of the state machine.

The chart of Fig. 5.10 compares graphically the degree of fulfilment of the adjacency restrictions expected in the state machines used as benchmarks. The chart shows clearly that our genetic algorithm always evolves a better state assignment.

## 5.6 Evolvable Hardware for the Control Logic

Genetic programming [10], [12] is way of producing a program using genetic evolution. The individuals within the evolutionary process are programs.

**Table 5.2.** Best state assignment yield by the compared systems for the benchmarks

| FSM | System | State Assignment |
|---|---|---|
| **Shiftreg** | GA [1] | [0,2,5,7,4,6,1,3] |
| 8/16 | NOVA1 | [0,4,2,6,3,7,1,5] |
| | NOVA2 | [0,2,4,6,1,3,5,7] |
| | Our GA | [5,7,4,6,1,3,0,2] |
| **Lion9** | GA [1] | [0,4,12,13,15,1,3,7,5] |
| 9/25 | NOVA1 | [2,0,4,6,7,5,3,1,11] |
| | NOVA2 | [0,4,12,14,6,11,15,13,7] |
| | Our GA | [10,8,12,9,13,15,7,3,11] |
| **Train11** | GA [1] | [0,8,2,9,13,12,4,7,5,3,1] |
| 11/25 | NOVA1 | [0,8,2,9,1,10,4,6,5,3,7] |
| | NOVA2 | [0,13,11,5,4,7,6,10,14,15,12] |
| | Our GA | [2,6,1,4,0,14,10,9,8,11,3] |
| **Bbarra** | GA [1] | [0,6,2,14,4,5,13,7,3,1] |
| 10/60 | NOVA1 | [4,0,2,3,1,13,12,7,6,5] |
| | NOVA2 | [9,0,2,13,3,8,15,5,4,1] |
| | Our GA | [3,0,8,12,1,9,13,11,10,2] |
| **Dk14** | GA [1] | [0,4,2,1,5,7,3] |
| 7/56 | NOVA1 | [5,7,1,4,3,2,0] |
| | NOVA2 | [7,2,6,3,0,5,4] |
| | Our GA | [3,7,1,0,5,6,2] |
| **Bbsse** | GA [1] | [0,4,10,5,12,13,11,14,15,8,9,2,6,7,3,1] |
| 16/56 | NOVA1 | [12,0,6,1,7,3,5,4,11,10,2,13,9,8,15,14] |
| | NOVA2 | [2,3,6,15,1,13,7,8,12,4,9,0,5,10,11,14] |
| | Our GA | [15,14,9,12,1,4,3,7,6,10,2,11,13,0,5,8] |
| **Donfile** | GA [1] | [0,12,9,1,6,7,2,14,11,17,20,23,8,15,10,16,21,19,4,5,22,18,13,3] |
| 24/96 | NOVA1 | [12,14,13,5,23,7,15,31,10,8,29,25,28,6,3,2,4,0,30,21,9,17,12,1] |
| | NOVA2 | [6,30,11,28,25,19,0,26,1,2,14,10,31,24,27,15,12,8,29,23,13,9,7,3] |
| | Our GA | [2,18,17,1,29,21,6,22,7,0,4,20,19,3,23,16,9,8,13,5,12,28,25,24] |

**Table 5.3.** Fitness of best assignments yield by the compared systems

| State machine | *#AdjRes* | Our GA | GA [5] | NOVA1 | NOVA2 |
|---|---|---|---|---|---|
| **Shiftreg** | 24 | 0 | 0 | 8 | 0 |
| **Lion9** | 69 | 21 | 27 | 25 | 30 |
| **Train11** | 57 | 18 | 19 | 23 | 28 |
| **Bbara** | 225 | 127 | 130 | 135 | 149 |
| **Dk14** | 137 | 68 | 75 | 72 | 76 |
| **Bbsse** | 305 | 203 | 215 | 220 | 220 |
| **Donfile** | 408 | 241 | 267 | 326 | 291 |

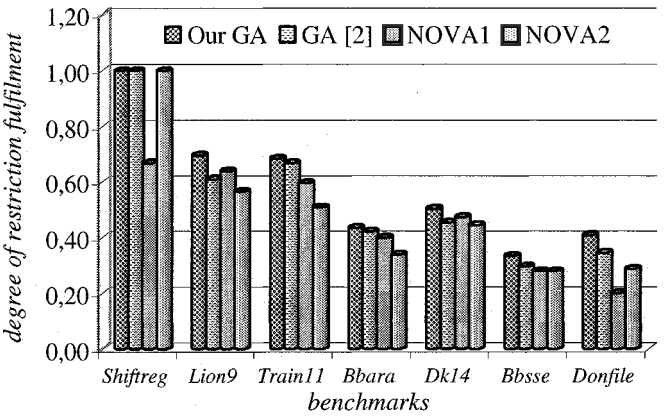



**Fig. 5.10.** Graphical comparison of the degree of fulfilment of rule 1 and 2 reached by the systems

The main goal of genetic programming is to provide a domain-independent problem-solving method that automatically yields computer programs from expected input/output behaviours. Exploiting genetic programming, we automatically generate novel control logic circuits that are *minimal* with respect to area and time requirements.

A circuit design may be specified using register-transfer level equations. Each instruction in the specification is an output signal assignment. A signal is assigned the result of an expression wherein the operators are those that represent basic gates in CMOS technology of VLSI circuit implementation and the operands are the input signals of the design. The allowed operators are shown in Table 5.4. Note that all gates introduce a minimal propagation delay as the number of input signal is minimal, which is 2.

**Table 5.4.** Gate name, symbol, gate-equivalent and propagation delay

| Name | Symbol | Gate Code | Gate Equiv. | Delay |
|---|---|---|---|---|
| NOT | | 0 | 1 | 0.0625 |
| AND | | 1 | 2 | 0.209 |
| OR | | 2 | 2 | 0.216 |
| XOR | | 3 | 3 | 0.212 |
| NAND | | 4 | 1 | 0.13 |
| NOR | | 5 | 1 | 0.156 |
| XNOR | | 6 | 3 | 0.211 |
| MUX | | 7 | 3 | 0.212 |

### 5.6.1 Circuit Encoding

We encode circuit designs using a matrix of cells that may be interconnected. A cell may or may not be involved in the circuit schematics. A cell consists of two inputs or three in the case of a MUX, a logical gate and a single output. A cell may draw its input signals from the output signals of gates of previous rows. The gates include in the first row draw their inputs from the circuit global input signal or their complements. The circuit global output signals are the output signals of the gates in the last raw of the matrix. An example of chromosome with respect to this encoding is given in Table 5.5. It represents the circuit of Fig. 5.11. Note that the input signals are numbered 0 to 3, their negated signals are numbered 4 to 7 and the output signals are numbered 16 to 19. If the circuit has $n$ outputs with $n < 4$, then the signals numbered 16 to n are the actual output signals of the circuit.

**Table 5.5.** Chromosome for the circuit of Fig. 5.11

| | | |
|---|---|---|
| $\langle 1, 0, 2, 8 \rangle$ | $\langle 5, 10, 9, 12 \rangle$ | $\langle 7, 13, 14, 11, 16 \rangle$ |
| $\langle 2, 4, 3, 9 \rangle$ | $\langle 1, 8, 10, 13 \rangle$ | $\langle 3, 11, 12, 17 \rangle$ |
| $\langle 3, 1, 6, 10 \rangle$ | $\langle 4, 9, 8, 14 \rangle$ | $\langle 7, 15, 14, 15, 18 \rangle$ |
| $\langle 7, 5, 7, 7, 11 \rangle$ | $\langle 4, 10, 11, 15 \rangle$ | $\langle 1, 11, 15, 19 \rangle$ |

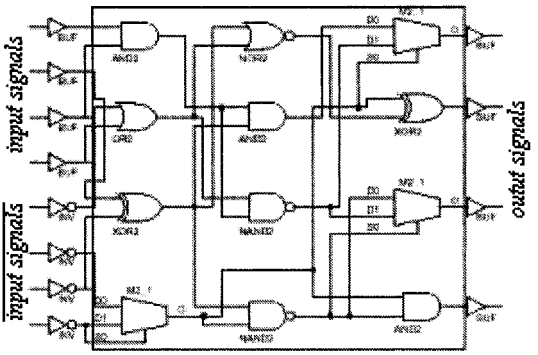



**Fig. 5.11.** Encoded circuit schematics

### 5.6.2 Circuit Reproduction

Crossover recombines two randomly selected circuits into two fresh offsprings. It may be single-point or double-point or uniform crossover as explained earlier. Crossover of circuit specification is implemented using a variable four-point crossover as described in Fig. 5.12.

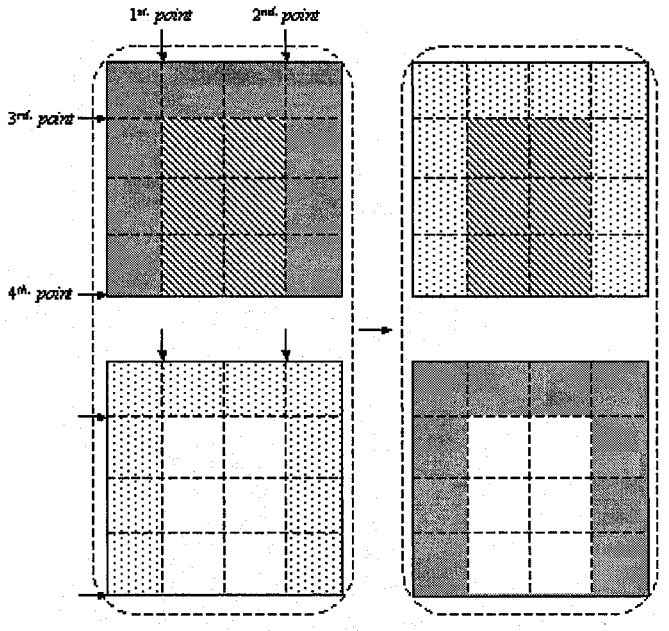



**Fig. 5.12.** Four-point crossover of circuit schematics

One of the important and complicated operators for genetic programming is the mutation. It consists of changing a gene of a selected individual. Here, a gene is the expression tree on the left hand side of a signal assignment symbol. Altering an expression can be done in two different ways depending the node that was randomised and so must be mutated. A node represents either an operand or operator. In the former case, the operand, which is a bit in the input signal, is substituted with either another input signal or simple expression that includes a single operator as depicted in Fig. 5.13 - top part. The decision is random. In the case of mutating an operand node to an operator node, we proceed as Fig. 5.13 - bottom part. The randomised operator node may be mutated to an operator node or to an operator of smaller (AND to NOT), the same (AND to XOR) or bigger arity (AND to MUX). In the last case, a new operand is randomised to fill in the new operand.

### 5.6.3 Circuit Evaluation

Another important aspect of genetic programming is to provide a way to evaluate the adherence of evolved computer programs to the imposed constraints. In our case, these constraints are of three kinds:

- First of all, the evolved specification must obey the input/output behaviour, which is given in a tabular form of expected results given the inputs. This is the truth table of the expected circuit.

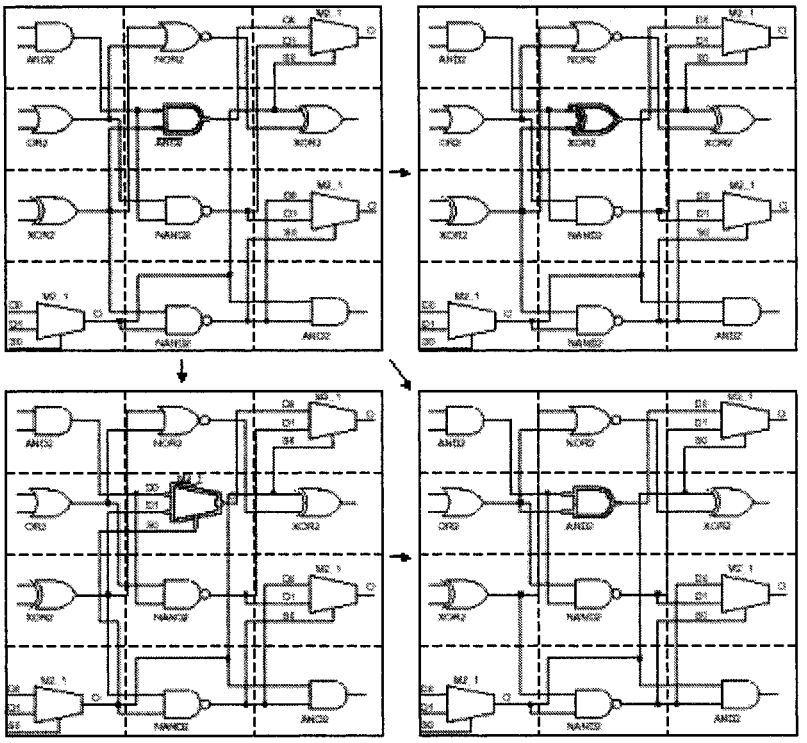


**Fig. 5.13.** Operand node mutation for circuit specification

- Second, the circuit must have a reduced size. This constraint allows us to yield compact digital circuits.
- Thirdly, the circuit must also reduce the signal propagation delay. This allows us to reduce the response time and so discover efficient circuits. In order to take into account both area and response time, we evaluate circuits using the weighted sum approach.

We estimate the necessary area for a given circuit using the concept of gate equivalent. This is the basic unit of measure for digital circuit complexity [7]. It is based upon the number of logic gates that should be interconnected to perform the same input/output behaviour. This measure is more accurate that the simple number of gates [7], [15].

When the input to an electronic gate changes, there is a finite time delay before the change in input is seen at the output terminal. This is called the propagation delay of the gate and it differs from one gate to another. Of primary concern is the path from input to output with the highest total propagation delay. We estimate the performance of a given circuit using the worst-case delay path. The number of gate equivalent and an average propa-

gation delay for each kind of gate are given in Table 5.4. The data were taken form [6].

Let $C$ be a digital circuit that uses a subset (or the complete set) of the gates given in Table 5.4. Let $Gates(C)$ be a function that returns the set of all gates of circuit $C$ and $Levels(C)$ be a function that returns the set of all the gates of $C$ grouped by level. Notice that the number of levels of a circuit coincides with the cardinality of the set expected from function Levels. On the other hand, let $Val(X)$ be the Boolean value that the considered circuit $C$ propagates for the input Boolean vector $X$ assuming that the size of $X$ coincides with the number of input signal required for circuit $C$. The fitness function, which allows us to determine how much an evolved circuit adheres to the specified constraints, is given as follows, wherein $X$ represents the input values of the input signals while $Y$ represents the expected output values of the output signals of circuit $C$, $n$ denotes the number of output signals that circuit $C$ has, function $Delay$ returns the propagation delay of a given gate as shown in Table 5.4 and $\Omega_1$ and $\Omega_2$ are the weighting coefficients [8] that allow us to consider both area and response time to evaluate the performance of an evolved circuit, with $\Omega_1 + \Omega_2 = 1$. Note that for each output signal error, the fitness function of Equation 5.3 sums up a penalty $\psi$. For implementation issue, we minimize the fitness function below for different values of $\Omega_1$ and $\Omega_2$.

$$\begin{aligned} Fitness(C) = & \sum_{i=0}^{n} \left( \sum_{i \mid Val(X_i) \neq Y_{i,j}} \psi \right) \\ & + \Omega_1 \sum_{g \in Gates(C)} GateEquiv(g) \\ & + \Omega_2 \sum_{l \in Levels(C)} \max_{g \in l} Delay(g) \end{aligned} \tag{5.3}$$

## 5.7 Comparative Results

In this section, we compare the evolved circuits to those obtained using the traditional methods, i.e. transition and Karnaugh maps. This is done for three different state machines that are generally used as benchmarks. These state machines are commonly called *shiftreg*, *lion*9 and *train*11. The detailed descriptions of these state machines can be found in [3]. The state assignments used are the best ones found so far. They also are the result of an evolutionary computation [14]. Theses state assignment are given in Table 5.2.

For each of these state machines, we evolved a minimal circuit that implements the required behaviour and compared it to the one engineered using the traditional method. Table 5.6 shows the details of this comparison. The schematics of the evolved circuit of state machines *shiftreg* are given in Fig. 5.14 and Fig. 5.15.

**Table 5.6.** Comparison of the traditional method vs. genetic programming

| State machine | Number of gate-Equivalent | | Response time | |
|---|---|---|---|---|
| | Traditional | GP | Traditional | GP |
| *Shiftreg* | 30 | 12 | 0.85 | 0.423 |
| *Lion9* | 102 | 33 | 2.513 | 0.9185 |
| *Train11* | 153 | 39 | 2.945 | 0.8665 |

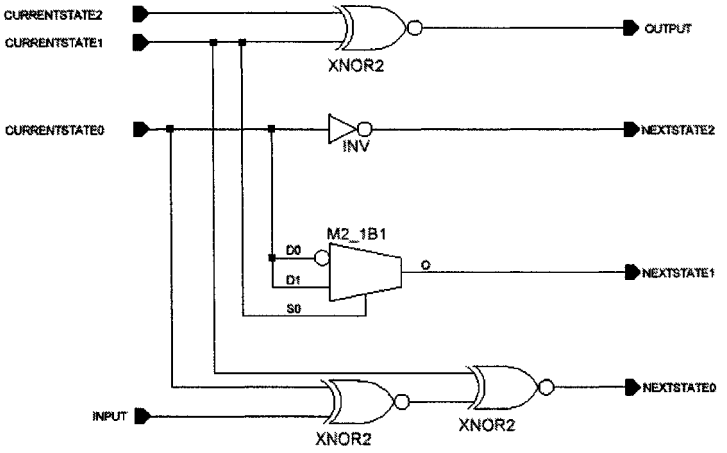



**Fig. 5.14.** First evolved control logic for state machine *shiftreg*

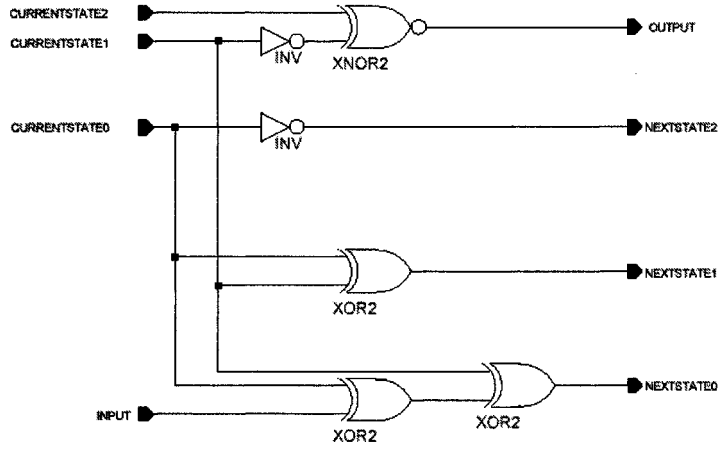



**Fig. 5.15.** Second evolved control logic for state machine *shiftreg*

The lookup table-based implementations of the *shiftreg* state machine for both control logics (i.e. of Fig. 5.14 and Fig. 5.15) exploits two 2-input, one 3-input and one 4-input lookup tables. The schematics are given in Fig. 5.10.

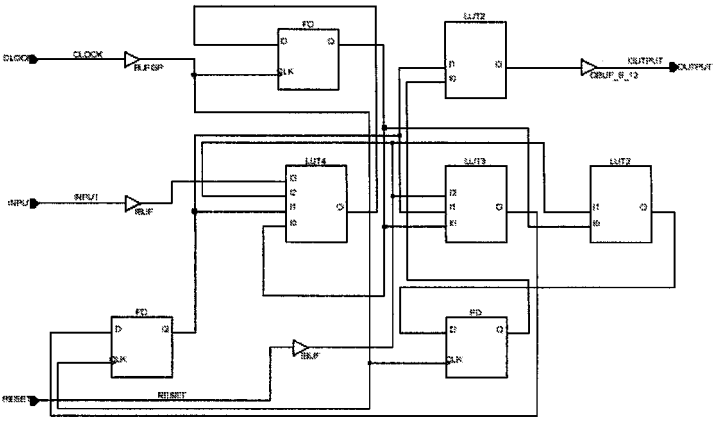


**Fig. 5.16.** Lookup table-based evolved architeture of *shiftreg*

The lookup table-based implementation of the *shiftreg* state machine as synthesised by the Xilinx$^{TM}$ [17] uses four 2-input, one 3-input and one 4-input lookup tables. The schematics are given in Fig. 5.16.

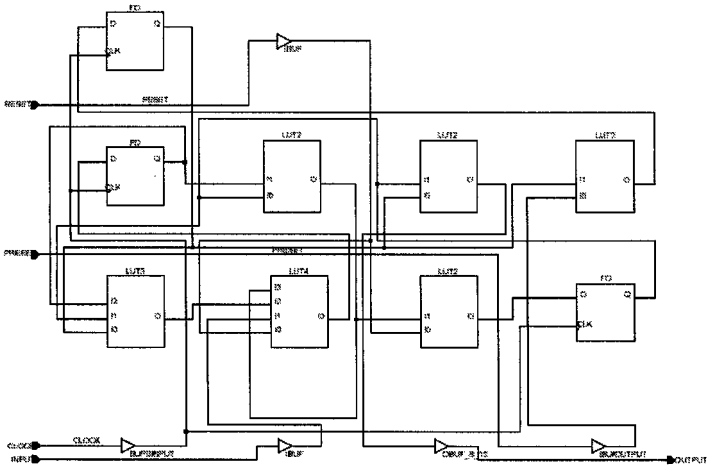


**Fig. 5.17.** Lookup table-based architeture of *shiftreg* as synthesised by Xilinx$^{TM}$

Fig. 5.18 and Fig. 5.19 show the evolved circuits for state machines *lion*9 and *train*11 respectively. It is clear that the evolved circuits are much better that those yield by the traditional methods in both terms hardware area and signal propagation delay.

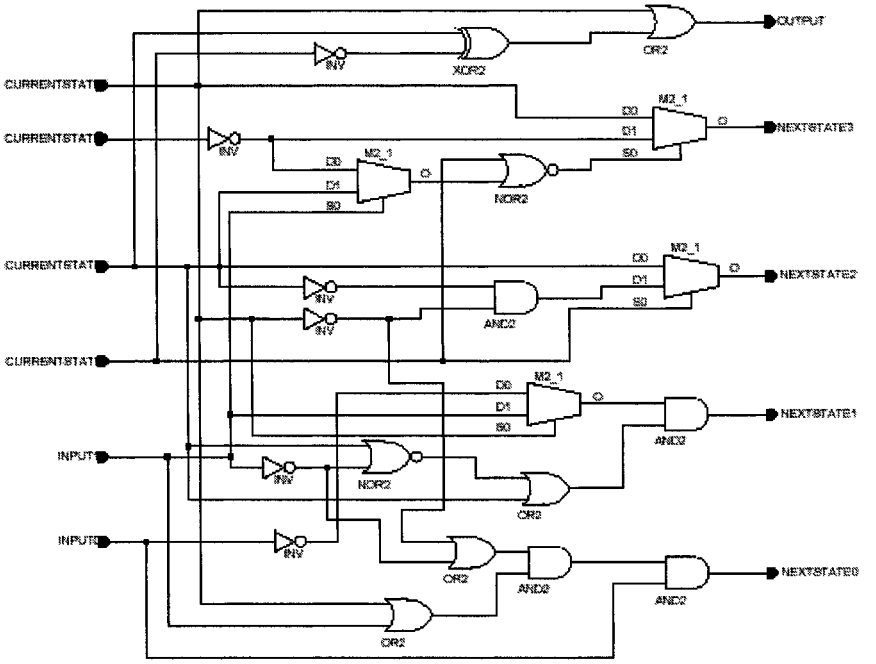



**Fig. 5.18.** The evolved control logic for state machine *lion*9

## 5.8 Summary

In this chapter, is divided into two main parts. In the first part, we exploited evolutionary computation to solve the $NP$-complete problem of state encoding in the design process of asynchronous finite state machines. We compared the state assignment evolved by our genetic algorithm for machine of different sizes evolved to existing systems. Our genetic algorithm always obtains better assignments. In the second part, we exploited genetic programming to synthesise the control logic used in asynchronous finite state machines. We compared the circuits evolved by our genetic programming-based synthesiser with that that would use the traditional method, i.e. using Karnaugh maps and transition maps. The state machine used as benchmarks are well known

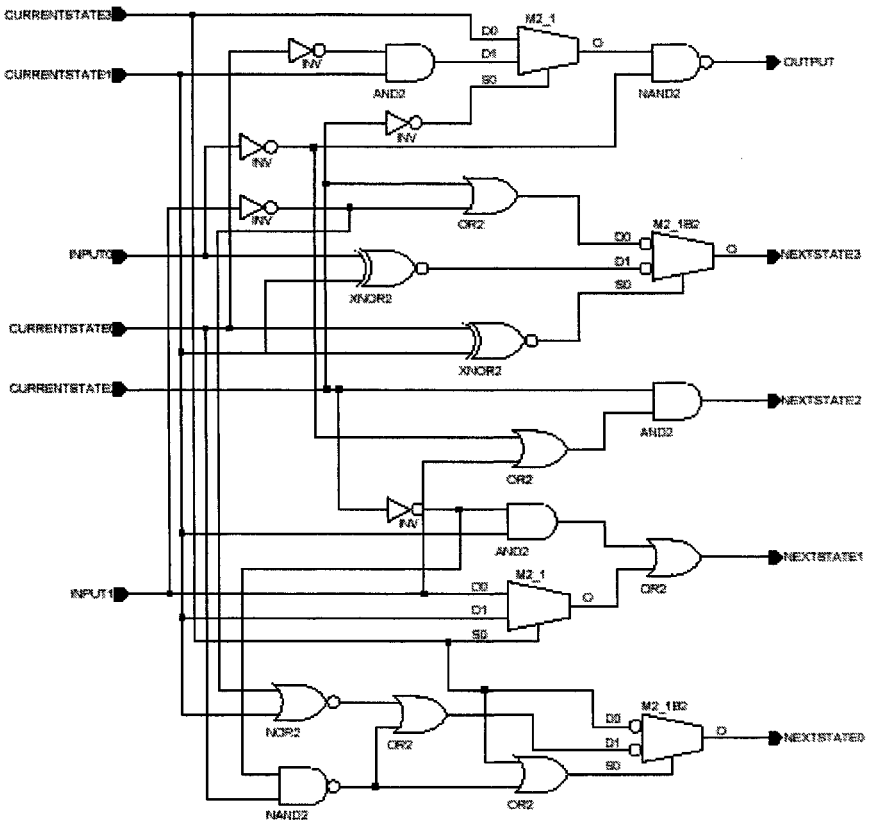



**Fig. 5.19.** The evolved control logic for state machine *train*11

and of different sizes. Our evolutionary synthesiser always obtains better control logic both in terms of hardware area required to implement the circuit and response time.

## References

1. Amaral, J.N., Tumer, K. and Gosh, J., Designing genetic algorithms for the State Assignment problem, IEEE Transactions on Systems Man and Cybernetics, vol., no. 1999.
2. Armstrong, D.B., A programmed algorithm for assigning internal codes to sequential machines, IRE Transactions on Electronic Computers, EC 11, no. 4, pp. 466-472, August 1962.
3. Collaborative Benchmarking Laboratory, North Carolina State University, `http://www.cbl.ncsu.edu/pub/ Benchmark_dirs/LGSynth89/fsmexamples`, November 27th. 2003.

4. DeJong, K. and Spears, W.M., Using genetic algorithms to solve NP-complete problems, Proceedings of the Third International Conference on Genetic Algorithms, pp. 124-132, Morgan Kaufmann, 1989.
5. DeJong, K. and Spears, W.M., An analysis of the interacting roles of the population size and crossover type in genetic algorithms, In Parallel problem solving from nature, pp. 38-47, Springer-Verlag, 1990.
6. Davis, L., Handbook of Genetic Algorithms, Van Nostrand Reinhold, New York, 1991.
7. Ercegovac, M. D., Lang, T. and Moreno, J.H., Introduction to digital systems, John Wiley, 1999.
8. Fonseca, C.M. and Fleming, P.J., An overview of evolutionary algorithms in multi-objective optimization, Evolutionary Computation, 3(1):1-16.
9. Goldberg, D. E., Genetic Algorithms in Search, Optimisation and Machine Learning, Addison-Wesley, Massachusetts, Reading, MA, 1989.
10. Haupt, R.L. and Haupt, S.E., Practical genetic algorithms, John Wiley and Sons, 1998.
11. Humphrey, W.S., Switching circuits with computer applications, New York: McGraw-Hill, 1958.
12. Koza, J.R., Genetic Programming. MIT Press, 1992.
13. Michalewics, Z., Genetic algorithms + data structures = evolution program, Springer-Verlag, USA, third edition, 1996.
14. Nedjah, N. and Mourelle, L.M, Evolutionary state assignment for synchronous finite state machine, Proceedings of International Conference on Computational Science, Lecture Notes in Computer Science, Springer-Verlag, 2004.
15. Rhyne, V.T., Fundmentals of digital systems design, Computer Applications in Electrical Engineering Series, Prentice-Hall, 1973.
16. Villa, T. and Sangiovanni-Vincentelli, A. Nova: state assignment of finite state machine for optimal two-level logic implementation, IEEE Transactions on Computer-Aided Design, vol. 9, pp. 905-924, September 1990.
17. Xilinx, Project Manager, ISE 6.1i, `http://www.xlinx.com`.