

# A CAD Tool for the Optimal State Assignment of Sequential Synchronous Circuits

M. M. Haji<sup>1</sup>, S. D. Katebi

*Department of Computer Science and Engineering, Shiraz University,  
Shiraz, Iran*

mehdi.haji@gmail.com, katebi@shirazu.ac.ir

## Abstract

Assigning proper binary codes to the states of sequential circuits is a long studied problem known as state assignment. The choice of the numbers assigned to the states determines the final hardware structure and implementation requirements of the circuits. Conventional state assignment techniques can not be applied when arbitrary optimality criteria are defined. The problem can actually be seen as a search problem with a huge non-linear space. The nature of the space makes it impossible to find optimal solutions by conducting exhaustive, random or conventional search techniques. In this paper, a CAD tool is introduced which solves the problem for sequential synchronous circuits by means of a Genetic Algorithm (GA). The main advantage of the tool is the ability to cope with large circuits and optimize with respect to different objective functions. Moreover, it is free, easy-to-use and cross-platform.

**Keywords:** CAD, state assignment, sequential synchronous circuits, genetic algorithms

## 1. Introduction

The problem of state assignment (coding) is a major concern in the efficient design of logic circuits [1, 2]. However, simple design methods do not mind which numbers assigned to which states and consequently an efficient implementation is unlikely to be obtained. These methods may use a default assignment in which typically 0 is assigned to the first state, 1 to the second and so on; or the gray code assignment in which the code assigned to each state differs in only one bit from the code assigned to the next state. Conventional state assignment techniques are applicable only when a simple optimality criterion is defined, such as minimizing the number of gates required for the gate-level implementation of the circuit. The optimal state assignment can be seen as a point in the huge non-linear space of all possible assignments, so it can be found by effectively exploring the space. But the nature of the space prohibits the use of exhaustive, random or conventional search (calculus-based) techniques such as hill-climbing. It has been shown that the size of the search space is exponential in the number of states [3]; exhaustively searching such a huge space even for circuits with 10 to 20 states needs months and years of the processor time to complete<sup>2</sup>.

## 2. Genetic Algorithms

Genetic Algorithms (GAs) are non-deterministic general search schemes for solving, or approximately solving, optimization problems. GAs belong to a larger category of algorithms called evolutionary algorithms (EAs) which are based on the idea of natural selection and evolution. They have been successfully applied to a large number of NP-hard problems from different domains [3-6]. The standard GA starts with a random population of candidate solutions to the problem at hand called individuals. A quality measure called fitness is calculated and assigned to each individual expressing how well it solves the problem. From the current population, the GA selects individuals proportionate to their fitness to be parents and uses them to produce children for the next generation. There are three main types of genetic operators to create the next generation from the current: crossover, reproduction and mutation. In the GA, the majority of the individuals of the next generation are produced by the crossover operator which randomly combines parts (partial solutions) of two individual parents to form a new child. A few individuals are produced by reproduction and mutation. To form a new child, the former clones a parent and the latter applies random changes to it. The mutation operation guarantees the genetic diversity of the population, it is thus necessary to apply mutation to prevent premature convergence. Over generations, the average fitness increases and the population evolve

---

<sup>1</sup> Corresponding author

<sup>2</sup> For a 20-state circuit, the size of the search space is exactly 143,055,170,278,100,720,640,000.

toward an optimal solution. The standard GA has a number of variants, of which perhaps the most widely used is the elitist simple generational GA whose pseudo-code is given in Figure 1.

```

nGenerations = 0; // The number of generations
pop = randomPopulation(); // Create a random population
Evaluate the fitness of each individual;
best_of_run = An individual of pop with the best fitness;

while( best_ind is not good enough AND nGenerations < MAX_GENERATIONS )
{
  Generate a new population from pop by applying genetic operators;
  Evaluate the fitness of each individual of the new population;

  if ( the best individual of the new population is better than
      best_of_run )
  {
    best_of_run = the best individual of the new population;
  }
  pop = the new population;
  nGenerations = nGenerations + 1;
}
return best_of_run;

```

Figure 1. Pseudo-code of elitist simple generational GA

### 3. Genetic State Assigner

For the optimization of sequential synchronous circuits we developed a tool called Genetic State Assigner (GSA) which solves the state assignment problem by a GA (Figure 1). It should be emphasized that GSA currently solves the problem only for synchronous circuits; the problem becomes more complicated for asynchronous circuits because state codes must be chosen carefully to ensure the avoidance of critical races and logic hazards [1]. GSA is freely available from the author's webpage at <http://www.mhaji.com/gsa>. It has an easy-to-use dynamic GUI and is implemented in Java making it cross-platform. In the current version, GSA generates the gate-level implementation of a circuit using only the three basic types of gates: AND, OR, NOT. In other words, it can not use other common types of gates such as NAND, XOR, XNOR. In the following section, it will be shown that how a circuit description can quickly and efficiently be converted to the implementation by GSA.

### 4. Comparison of Two Design Methods

In this section a 7-state synchronous circuit is designed both by GSA and the manual method using a default coding. It is shown that GSA finds an assignment which results in an implementation with three times less logic than that of the manual method. An exhaustive search is also conducted to find the best possible solution which is then seen to be the same one produced by the genetic search of GSA.

PS	NS / Z	
	X = 0	X = 1
S <sub>0</sub>	S <sub>1</sub> / 0	S <sub>4</sub> / 0
S <sub>1</sub>	S <sub>5</sub> / 0	S <sub>2</sub> / 0
S <sub>2</sub>	S <sub>3</sub> / 0	S <sub>6</sub> / 0
S <sub>3</sub>	S <sub>0</sub> / 0	S <sub>0</sub> / 1
S <sub>4</sub>	S <sub>5</sub> / 0	S <sub>5</sub> / 0
S <sub>5</sub>	S <sub>6</sub> / 0	S <sub>6</sub> / 0
S <sub>6</sub>	S <sub>0</sub> / 0	S <sub>0</sub> / 0

X: Input, Z: Output, PS: Present State, NS: Next State

Figure 2. State transition table for 0101 sequence detector

Since an exhaustive search is to be performed, a 7-state circuit for which the search space size is small (exactly 840) is considered: a non-overlapping sequence detector for 0101; by non-overlapping we mean that, for example, if 01010101 is the input sequence, the detector should outputs 00010001, not

00010101. In the first step of design, the circuit description is stated in the form of a Finite State Machine (FSM), and then it can be expressed by a state transition table which is shown in Figure 2.

#### 4.1. Manual Design

Having specified the state transition table, unique binary codes should be assigned to all of the state. Since the machine has 7 states, at least 3 bits (state variables), denoted by  $y_0$ ,  $y_1$  and  $y_2$ , are necessary to assign unique numbers to the states, so there are 8 numbers (0 to 7) that can be used for coding. A simple manual design typically uses the following assignment:

$$S_0 \leftarrow 000, S_1 \leftarrow 001, S_2 \leftarrow 010, S_3 \leftarrow 011, S_4 \leftarrow 100, S_5 \leftarrow 101, S_6 \leftarrow 110.$$

For the final implementation, the memory elements are chosen to be of type T flip flop. The input equations of the three flip flops, named  $T_0$ ,  $T_1$  and  $T_2$ , and the equation of the output, named  $Z_0$ , are finally obtained as:

$$T_0 = y_0' \cdot y_2' \cdot x_0 + y_0' \cdot y_1' \cdot y_2 \cdot x_0' + y_0 \cdot y_1$$

$$T_1 = y_2 \cdot x_0 + y_1 \cdot y_2 + y_0 \cdot y_2 + y_0 \cdot y_1$$

$$T_2 = y_2 \cdot x_0 + y_0' \cdot y_2' \cdot x_0' + y_0' \cdot y_1 \cdot x_0' + y_0 \cdot y_1'$$

$$Z_0 = y_1 \cdot y_2 \cdot x_0$$

where the input is denoted by  $x_0$ .

Assuming the complements are present and no factorization is used, the realization of the above equations needs 8 2-input OR and 18 2-input AND gates.

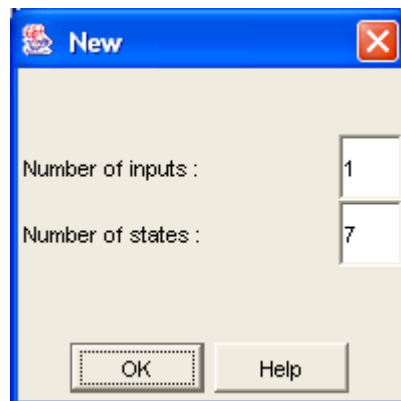


Figure3. New dialog to specify number of inputs and states

#### 4.2. GSA Design

The main GUI of GSA allows the user to enter/open a new/an existing state transition table by selecting the New/Open command from the File menu. After selecting the New command, a dialog is shown (Figure 3) in which the number of inputs and states of a new circuit can be specified. For this example, the number of inputs is set to 1 and the number of states to 7. After clicking the Ok button, the program dynamically creates a dialog for the state transition table to be entered (Figure 4). The Save As button allows the table to be saved into a text file (which can be loaded at a later time by the Open command from the File menu). After clicking the Ok button, another dialog is shown in which the user can specify the type of each memory element separately. Four common types of flip flops have been implemented in GSA: D, T, JK and RS. Here, the three memory elements are chosen to be of type T in order to be the same as what was done in the manual design.

Present State	Next State / Output	
	0	1
S0	S1/0	S4/0
S1	S5/0	S2/0
S2	S3/0	S6/0
S3	S0/0	S0/1
S4	S5/0	S5/0
S5	S6/0	S6/0
S6	S0/0	S0/0

Buttons: OK, Save As, Help

Figure 4. State table dialog to enter circuit description

Memory element # 0  D flip flop  T flip flop  JK flip flop  RS flip flop

Memory element # 1  D flip flop  T flip flop  JK flip flop  RS flip flop

Memory element # 2  D flip flop  T flip flop  JK flip flop  RS flip flop

Buttons: OK, Help

Figure 5. Dialog to specify type of memory elements

In the next dialog, the user defines the optimality criterion that will be used in the search. Currently, GSA can minimize either total delay, or total hardware or dependency to inputs. For teaching or experimental purposes, GSA also allows the user to do state assignment manually. Minimizing total hardware is used here as the objective (Figure 6). After clicking the Ok button, the exact size of the search space is shown to user (Figure 7). After clicking the Ok button, GSA starts exploring the space. When the search space is small, it is wise to conduct an exhaustive search so that the best solution is guaranteed to be found.

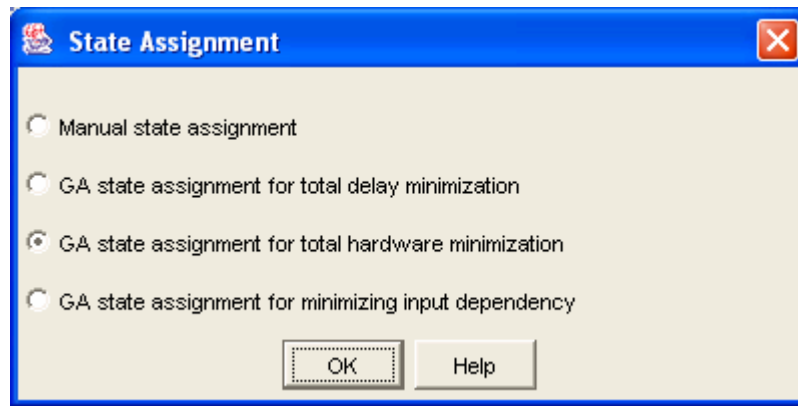


Figure 6. Dialog to specify type of state assignment

GSA searches exhaustively when the number of states is 6 or less by default; the user can change the default value in the Exhaustive Search dialog from the Settings menu. Since here the number of states is 7, GSA uses the GA to search for a solution. The GA has a number of parameters which can be set in the GA Settings dialog from the Settings menu. By default, the population size is 200, the number of generations is 100, the probability of crossover is 0.7 and the initial probability of mutation is 0.1. As regards the selection mechanism, a classical tournament of size 10% of the population size is employed. All parameters remain constant during a run except the mutation rate which is gradually reduced as the number of generations increases. A GA can usually find good solutions when it uses default values for the parameters, but if they are set properly the GA may also find the best solution (global optimum).

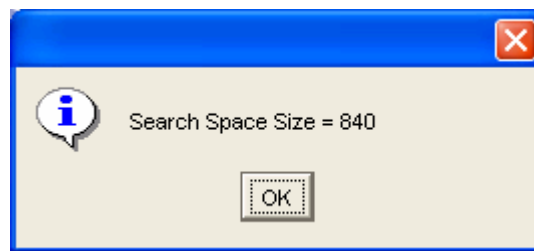


Figure 7. Dialog shows search space size

Since the GA does not know the best solution, it continues evolving until the maximum number of generations (100) is reached. The whole process takes about 20 seconds running on a PC with Windows XP and the AMD Athlon XP 2800+ processor. The gate-level implementation is then shown in a dialog (Figure 8). The implementation equations and the corresponding state assignment given below can be saved into a text file by the Save button:

$$S_0 \leftarrow 101, S_1 \leftarrow 100, S_2 \leftarrow 011, S_3 \leftarrow 010, S_4 \leftarrow 110, S_5 \leftarrow 001, S_6 \leftarrow 000$$

$$T_0 = 1$$

$$T_1 = y_2' \cdot x_0 + y_1 \cdot x_0 + y_0 \cdot y_1$$

$$T_2 = y_0$$

$$Z_0 = y_0 \cdot y_1 \cdot y_2 \cdot x_0$$

Assuming the complements are present and no factorization is used, the realization of the above equations needs 2 2-input OR and 6 2-input AND gates. In order to know the quality of the solution, an exhaustive search is also conducted, and it is seen that the same solution is produced, meaning that the GA has obtained the global optimum.

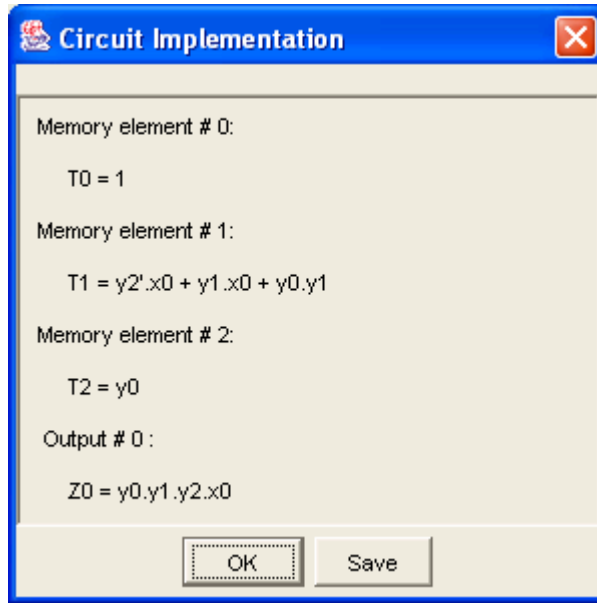


Figure 8. Dialog shows gate-level implementation

As seen, the state assignment obtained by GSA results in an implementation with about three times less logic than that of the manual method. Of course, the number of memory elements is the same in the two implementations because it is determined by the number of states, not by the coding. Simple realizations of the 0101 sequence detector by using the two state assignment methods are shown in Figures 9 and 10 respectively.

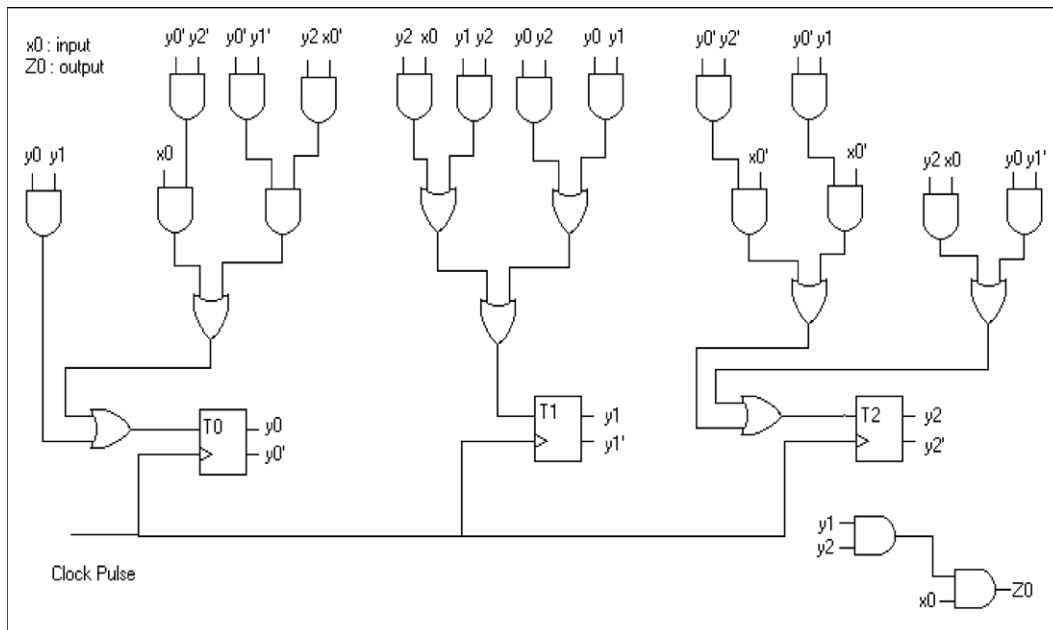


Figure 9. Realization of 0101 sequence detector obtained by manual state assignment

## 5. Conclusion

Over recent years, many methods have been proposed for the state assignment problem [3, 7-9]. In this paper, we did not aim at proposing a new one, but introducing a tool which can be really useful due to its distinctive features. It is free, easy-to-use and cross-platform. Moreover, since no assumption is made about the circuit size, and consequently many data structures and GUI components are designed to be dynamic, the program can cope with circuits of any size. However, there is an effective upper limit on the circuit size as well as the GA parameters which is imposed by the finite memory of the machine on which the program is running. Four common types of flip flops

and three different objective functions have been implemented in the current version. The object oriented design of the program makes it easily extendable and modifiable. In future versions, we plan to add an interface to allow defining arbitrary memory elements, implement more objective functions and use a multi-objective evolutionary algorithm (MOEA) [10] instead of the GA so that more than one objective can be optimized simultaneously.

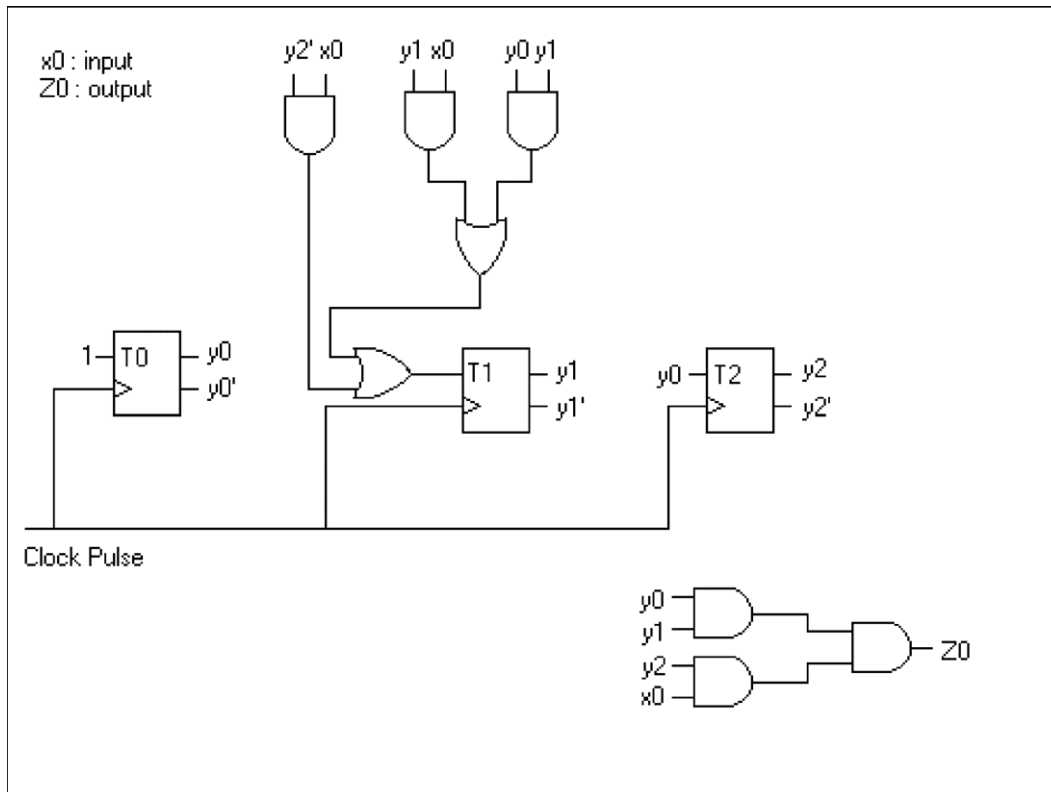


Figure 10. Realization of 0101 sequence detector obtained by GSA state assignment

## 6. References

- [1]. Z.Kohavi, "Switching and Finite State Automata Theory", McGrawHill, 1978.
- [2]. A.Almaini, "Electronic Logic Systems", Third Edition, Prentice Hall, 1994.
- [3]. A. E. A. Almaini, J. F. Miller, P. Thomson and S. Billina, "State Assignment of Finite State Machines using a Genetic Algorithm", IEE Proceedings on Computers and Digital Techniques, vol. 142(4), pp. 279-286, 1995.
- [4]. D. E. Goldberg, "Genetic Algorithms in Search, Optimization, and Machine Learning", Addison Wesley, 1989.
- [5]. Kenneth A. De Jong, William M. Spears and Diana F. Gordon, "Using Genetic Algorithms for Concept Learning", special issue on genetic algorithms for Machine Learning Journal, vol. 13, pp. 161-188. Kluwer Academic, 1993.
- [6]. M. Gudmundsson, E.A. El-Kwae and M.R. Kabuka, "Edge Detection in Medical Images Using a Genetic Algorithm", IEEE Trans. on Medical Imaging, vol. 17(3), pp. 469-474, Jun. 1998.
- [7]. T. Villa and A. Sangiovanni-Vincentelli, "NOVA: State Assignment of Finite State Machines for Optimal Two-Level Logic Implementations", IEEE Trans. on Computer-Aided Design, vol. 9, pp. 905-924, 1990.
- [8]. Imtiaz Ahmad and Raza Ul-Mustafa, "On State Assignment of Finite State Machines Using Hypercube Embedding Approach", IEEE International Conference on Computer Design (ICCD'99), 1999.
- [9]. G. Hasteer and P. Banerjee, "A Parallel Algorithm for State Assignment in Finite State Machines", IEEE Trans. on Computers, vol. 47, pp. 242-246, 1998.

[10]. E. Zitzler and L. Thiele. "Multiobjective Optimization Using Evolutionary Algorithms - A Comparative Case Study", Parallel Problem Solving from Nature - PPSN-V, pp. 292-301, Springer, Sep. 1998.