Synthesis of Synchronous Sequential Logic Circuits from Partial Input/Output Sequences

Chaiyasit Manovit, Chatchawit Aporntewan and Prabhas Chongstitvatana

Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University
Bangkok 10330 Thailand
prabhas@chula.ac.th

Abstract. This work takes a different approach to synthesize a synchronous sequential logic circuit. The input of the synthesizer is a partial input/output sequence. This type of specification is not suitable for conventional synthesis methods. Genetic Algorithm (GA) was applied to synthesize the desired circuit that performs according to the input/output sequences. GA searches for circuits that represent the desired state transition function. Additional combination circuits that map states to the corresponding outputs are synthesized by conventional methods. The target of our synthesis is a type of registered Programmable Array Logic which is commercially available as GAL. We are able to synthesize various types of synchronous sequential logic circuit such as counter, serial adder, frequency divider, modulo-5 detector and parity checker.

1 Introduction

The conventional method to synthesize a sequential logic circuit requires knowledge of the circuit's behavior in the form of a state diagram. A sequential network is a common starting point for sequential synthesis system such as Berkeley Synthesis System SIS [1]. We aim to realize an evolvable hardware that can "mimic" other sequential circuit by *observing its partial input/output sequences*. In this case, circuit specifications are in the form of partial input/output sequences which are not suitable to synthesize a circuit by the conventional method. Genetic Algorithm (GA) [2,3] is used to search for circuits that represent the desired state transition function. Additional combination circuits that map states to the corresponding outputs are synthesized by conventional methods. The simulated evolution has been used to synthesize a finite state machine (FSM) in [4,5] where the resulting FSM can predict the output symbol based on the sequence of input symbols observed. In contrast to representing circuits as FSMs [6] proposes the automated hardware design at the Hardware Description Language level using GA. [7] describes the evolution of hardware at function-level based on reconfigurable logic devices. [8,9] evolved circuits at the lowest level, in the actual logic devices, using real-time input/output. Our work is similar to [2,3] in the use of FSM but we use FSM as the *model* of the desired circuit behavior. We aim to evolve the circuit at the logic device level similar

to [9]. The following sections describe our synthesis method, the experiment in synthesizing various simple sequential circuits and the analysis of the result. Notably the analysis about the length of the input/output sequence which has implication on the efficiency of the synthesizer.

2 The synthesizer

We use Programmable Logic Device (PLD) as our target structure. For simplicity, we used GAL structure which is composed of rows of two-level sum-of-product Programmable Array Logic (PAL) connected to D flip-flops. The implemented model has four logic terms per one flip-flop, and has a total of four flip-flops. A circuit is specified by a linear bit-string representing all connection points, which is entirely 256 bits in length. We applied Genetic Algorithm to synthesize simple circuits, such as counter, serial adder, frequency divider, modulo-5 detector and parity checker. The specification of a desired circuit is in the form of a partial input/output sequence. The circuit acquired from the evolution process will realize only the state transition part. The outline of the synthesizer's steps is as follows:

1 sample a partial input/output sequence from the target circuit
2 use the sequence as inputs of the synthesizer program
3 verify the resulting circuit if the run yields a solution within 50,000 generations (because GA is a probabilistic algorithm, not all runs are successful in yielding a solution by the specific generation)

3.1 Genetic Operations

Each individual is represented by a 256-bit bit-string. We defined genetic operators as follows:

1 Reproduction: Ten new offsprings survive in the next generation by selecting the first 10 fittest individuals ordered by combined rank method [10] (calculated from each individual's fitness rank and its diversity rank).
2 Crossover: More individuals being added to the population are produced by uniform crossover [11]. All possible pairs among 10 already selected individuals are used to produce new offsprings. That is, we will have 90 new individuals.
3 Mutation: Last 10 individuals being added are mutated version of the first 10 selected individuals. The mutation process is controlled so that it changes exactly 5 bits of each individual.

We also add some more constraints. First, we avoid creating a product term which always be "0". Second, connection points to unused input signals are left unwired. This reduces the search space of the problem and lets the program concentrate on the connection points that do affect the function of the circuit.

3.2 Fitness Evaluation

An individual is evaluated by the following steps:

1 feed one input to the circuit and clock the circuit
2 next state of the circuit would be mapped with the corresponding output, record the number of times the state has been mapped to output "0" and "1", independently
3 repeat steps 1 and 2 until the end of the sequence

After the sequence is completed, the fitness value of the individual, F, is computed by:

$$F = \sum_{i=0}^{i=S-1} f_i \quad \text{where} \quad f_i = \begin{cases} \max(p_i, q_i) & ;\ p_i = 0 \text{ or } q_i = 0 \\ -\min(p_i, q_i) & ;\ \text{otherwise} \end{cases} . \tag{1}$$

where

f_i is the fitness value of state i
p_i is the number of times in which state i has to be mapped with output "0"
q_i is the number of times in which state i has to be mapped with output "1"
S is the number of states, equal to 16 for the GAL structure

Based on Moore's model, a state of an FSM must be mapped to only one output value. Therefore, any state that is mapped to both "0" and "1" will cause a penalty in the fitness value as shown in the formula. For Mealy's model, the evaluation is similar. The difference is that output values are mapped to transition paths instead of states.

3.3 Size of Input/Output Sequences

A partial input/output sequence, which is used as a circuit's specification, is attained by generating a sequence of inputs, feeding each one to the target machine and then recording the corresponding output that the machine gives. To be a general approach and yield a simple analysis, the input sequence is created at random with uniform distribution (i.e. at any time, the probability that the input bit be "0" and "1" are equal).

The input sequence should be long enough to exercise all aspects of the circuit's function. In other words, it should be able to test all paths of the state diagram of the circuit. As mentioned earlier in this paper, we use the GAL structure which can be programmed as a 16-state state machine, larger than the desired circuit's need. Therefore, the input sequence should also be long enough to exercise all paths of any 16-state circuit. If the sequence is too short, it may cause an ambiguity in describing the desired circuit. In this paper, we will call the length of the input sequence which is long enough to describe (exercise) only the desired circuit as *lowerbound length* and call the length of the input sequence which is long enough to exercise any 16-state circuit as *upperbound length*.

We can find the proper size of the input sequence by making an analogy to a dice rolling problem. Consider rolling a dice, how many times do we have to roll it, until

all of its faces appear? This problem is called *waiting times in sampling* [12], in which we can find the *expected value* of the number of times by the following formula:

$$E(n) = n\left\{\frac{1}{1}+\frac{1}{2}+\cdots+\frac{1}{n}\right\} \quad ; \text{ n is the number of faces .} \tag{2}$$

Consider a circuit that has i bit inputs, giving possible $I = 2^i$ input patterns, and has S states. We assume that at any time, the probability that the circuit be in any state is equal. So, from the starting state, we expect the number of state transitions to be $E(S)$ to traverse through all states of the state diagram. And at any state, we expect the number of inputs to be $E(I)$ to traverse through all paths from that state. Therefore, to traverse through all paths of the state diagram, we would expect the number of inputs, the length of the input sequence, to be:

$$L = E(S) \times E(I) . \tag{3}$$

The lowerbound length is computed by using S equal to the number of states of the circuit. While the upperbound length is computed by using S equal to the maximum number of states which the GAL structure can represent, in this case, 16.

Table 1. Description of circuits in the experiment

Circuit	# of inpu bits	# of states		lower bound length		upper bound length
		Moore	Mealy	Moore	Mealy	
Frequency Divider	0	8	8	22	22	55
Odd Parity Detector	1	2	2	9	9	163
Modulo-5 Detector	1	6	5	45	35	163
Serial Adder	2	4	2	70	25	451

Note

Frequency Divider give square wave outputs of clock's frequency divided by 8
Odd Parity Detector give an output "1" when the number of "1" in inputs is odd
Modulo-5 Detector give an output "1" when the current input is the $5n^{th}$ "1" ; $n \in I^+$
Serial Adder give the sum of 2 inputs, inputs are feeding from LSB to MSB

The experiment was done with many input/output sequences of different lengths for each problem. The selected lengths are 10, 100, 1000, lowerbound and upperbound. For each length, at least 3 different random sequences of input/output are used. And the synthesizer was repeatedly run 50 times on each sequence. Thus, we had at least 3×50, equal to 150, independent runs for each length of the sequence for one problem. Table 1. shows the details of desired circuits and the calculated lengths of each problem.

3.4 Circuit Verification

In this experiment, the state diagrams of desired circuits are known. The state transition part of the state diagram of the resulting circuit can be derived by fixing the current state and feeding in all patterns of inputs and recording state transitions. We

then compare to check if it is equivalent to the known one. They can be considered to be equivalent if and only if each state of one diagram can be matched with one state of the other.

In a general case, when we do not know the state diagrams of desired circuits, we propose one way to verify it. The concept is similar to the determination of the size of input/output sequences. An upperbound size of input/output sequence should be used to test the circuit. Only one sequence, however, might not suffice to test the correctness, especially of the very first states. It is possible that the input sequence leads the circuit to pass one state and never go back to that state again. In this case, we cannot be sure that other out paths from that state lead to the correct next states. Consequently, we should use several test sequences to be more confident.

4 Effort

The experiment is done on a 200 MHz workstation. In the worst case, one run uses 40 minutes and terminates at generation 50,000 with no solution. We will calculate and compare empirical computational effort using a method which is slightly adapted from Koza [13] as follows:

i	generation number
M	population size
P(M, i)	cumulative probability of yielding a correct solution by generation *i*
W(M, i)	cumulative probability of yielding a wrong solution by generation *i*
X(M, i)	instantaneous probability of yielding a wrong solution by generation *i*
I(M, i, z)	individuals that must be processed by generation *i* with probability *z*
R(z)	number of independent runs required to yield at least one successful run with probability *z*
z	probability of satisfying the success predicated by generation *i* at least once in *R* independent runs = 0.99

$$z = 1 - \left[1 - P(M,i)\right]^{R} = 0.99 \tag{4}$$

$$R = R(M,i,z) = \left\lceil \frac{\log(1-z)}{\log(1-P(M,i)} \right\rceil \tag{5}$$

$$I(M,i,z) = M \times (i+1) \times R(z) \ . \tag{6}$$

We pay attention only to the correct solutions. We could not count the run yielding a wrong solution as a normal unsuccessful run. If we do count it, the calculated effort will be over-estimated. We have to subtract *I(M,i,z)* by the number of individuals which are not produced after the run stopped. Empirically, there are *X(M, j)*×*R* runs that yield wrong solutions at generation *j*. Therefore, if we let the program run till generation *i*, *i* > *j*, the number of individuals which are not produced is equal to

$$M \sum_{j=0}^{j=i-1} (i-j) X(M,j) R = M \left\{ \sum_{j=0}^{j=i-1} iX(M,j)R - \sum_{j=0}^{j=i-1} jX(M,j)R \right\} \tag{7}$$

$$M\sum_{j=0}^{j=i-1}(i-j)X(M,j)R = M\left\{R\times i\times W(M,i) - R\sum_{j=0}^{j=i-1} jX(M,j)\right\}. \quad (8)$$

Consequently,

$$I(M,i,z) = M\times(i+1)\times R - M\left\{R\times i\times W(M,i) - R\sum_{j=0}^{j=i-1} jX(M,j)\right\}. \quad (9)$$

The computational effort value, E, is the minimum $I(M,i,z)$, varying value of i. E is the minimum number of individuals needed to produced to yield a correct solution with a satisfactorily high probability (z = 0.99). However, the time in evaluating a circuit also depends on the length of input/output sequences. Thus, it is noted that the length of input/output sequences should be considered along with the computational effort.

5 Experiment and the Results

Fig. 1 shows the evolution of a serial adder. The circuit that performs accordingly to the input/output sequence appears in the 53rd generation. We know it is a correct serial adder after we verified it. We could observe many redundant states in the resulting circuit because the given space (16-state machine) is larger than the solution.

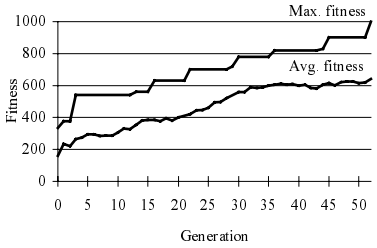

Fig. 1. Evolution of a Serial Adder (Mealy, size of input sequence = 1000)

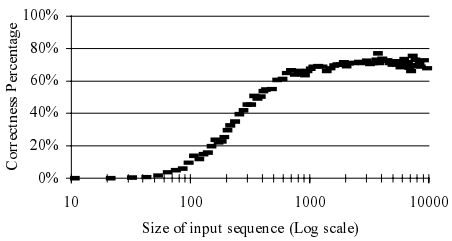

Fig. 2. Correctness and size of sequence (Serial Adder, Mealy's model)

We define the correctness percentage as

$$\text{Correctness Percentage} = \frac{\text{number of runs yielding correct solutions}}{\text{number of runs yielding solutions}}. \quad (10)$$

Fig. 2 shows the relation of the size of input sequence to the percentage of correct results of synthesis from the sequence of that size. We used five random input sequences for each point and run each input sequence for 100 times to make the average correctness. The longer input sequence increases the correctness. However, the correctness percentage becomes saturated at the large size of input sequence.

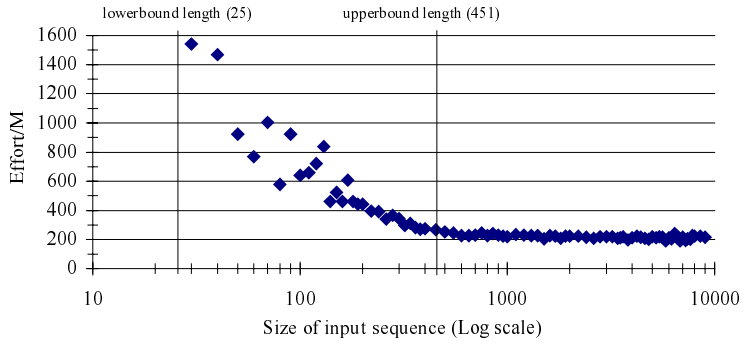

Fig. 3. Effort and size of input sequence (Serial adder, Mealy's model, M=110)

Fig. 3 shows the effort of evolving a serial adder using different sizes of sequences. This result confirms our lowerbound and upperbound length hypotheses. Any input sequence that is shorter than the lowerbound length can not yield any correct solution, consequently, the effort can not be computed. The input sequence longer than the upperbound length yields saturated effort value because of the approximately constant correctness percentage. It is possible to select the suitable size of input sequence to minimize effort and maximize correctness. The size of input sequence should be small to minimize the running time

Table 2. summarizes the computational effort of each problem using input/output sequences of upperbound length.

Table 2. The summary of computational effort

Circuit	Effort	
	Moore	Mealy
Frequency Divider	770	440
Odd Parity Detector	1,210	1,760
Modulo-5 Detector	87,967,440	7,018,000
Serial Adder	3,035,120	26,730

6 Conclusion

This paper described synthesis of synchronous sequential logic circuit from a partial input/output sequence. GA was applied to synthesize a circuit that based on Moore and Mealy's model on the GAL structure. We can approximate the suitable size of the input sequence to yield high correctness and low effort. This work can be extended in many ways. One major aspect is the enhancement of the evolutionary process in both effort and time. An implementation of this work in real hardware to realize an on-line evolware [14,15] is one interesting approach.

References

1. Sentovich, E., Singh, K., Moon, C., Savoj, H., Brayton, R. and Sangiovanni-Vincentelli, A.: Sequential circuit design using synthesis and optimization. Proc. of Int. Conf. on Computer Design (1992)
2. Holland, J.: Adaptation in natural and artificial systems. MIT Press (1992)
3. Goldberg, D.: Genetic Algorithm in search, optimization and machine learning. Addison-Wesley (1989)
4. Fogel, L.: Autonomous Automata. Industrial Research **4** (1962) 14-19
5. Angeline, P., Fogel, D., Fogel, L. : A comparison of self-adaptation methods for finite state machine in dynamic environment. Evolution Programming **V**. L. Fogel, P. Angeline, T. Back (eds). MIT Press (1996) 441-449
6. Mizoguchi, J., Hemmi, H., Shimohara K.: Production Genetic Algorithms for automated hardware design through an evolutionary process. Proc. of the first IEEE Int. Conf. on Evolutionary Computation (1994) 661-664
7. Higuchi, T., Murakawa, M., Iwata, M., Kajitani, I., Liu, E., Salami, M.: Evolvable hardware at function level. Proc. of IEEE Int. Conf. on Evolutionary Computation. (1997) 187-192
8. Thompson A., Harvey, I., Husbands, P.: The natural way to evolve hardware. Proc. of IEEE Int. Conf. on Evolutionary Computation (1996) 35-40
9. Thompson, A.: An evolved circuit, intrinsic in silicon, entwined with physics. Proc. of the First Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES96). Lecture Notes in Computer Sciences. Springer-Verlag (1997)
10. Winston, P.: Artificial Intelligence. Addison-Wesley (1992) 505-528
11. Syswerda, G.: Uniform crossover in Genetic Algorithms. Proc. of the Third Int. Conf. on Genetic Algorithms, J. D. Schaffer (ed). Morgan Kauffman (1989) 2-9
12. Feller, W.: An introduction to probability theory and its applications vol. **I**. Wiley (1968) 224-225
13. Koza, J.: Genetic Programming. MIT Press (1992)
14. Tomassini, M.: Evolutionary algorithms. Towards Evolvable Hardware. E. Sanchez and M. Tomassini (eds). vol. **1062** of Lecture Notes in Computer Science, Springer-Verlag, Heidelberg. (1996) 19-47
15. Sipper, M., Goeke, M., Mange, D., Stauffer, A., Sanchez, E., Tomassini, M.: The Firefly machine: Online evolware. Proc. of IEEE Int. Conf. on Evolutionary Computation. (1997) 181-186