Using VHDL Neural Network Models for Automatic Test Generation

Morteza Fayyazi, Zainalabedin Navabi and Armita Peymandoust
Department of Electrical and Computer Engineering
Faculty of Engineering, Campus No. 2
University of Tehran
14399, Tehran IRAN
Tel : 98-21-877-8690; Fax : 98-21-802-7756
Email : navabi@khorshid.ece.ut.ac.ir, navabi@ece.neu.edu

ABSTRACT:

VHDL models for neural networks for automatic test generation of gate level circuits are presented in this paper. A program converts a gate netlist to its equivalent neural model. A good circuit and faultable bad circuits will be generated. A VHDL test bench has been developed to apply the faults to the neural network bad circuit model, and report tests that are generated for each injected fault.

1. Introduction

Test pattern generation of a circuit for a fault involves searching among possible input vectors of the circuit for a test that can activate and propagate the fault. This algorithm is NP-complete and application of massively parallel algorithms can reduce the timing cost of test generation process. If these algorithms are implemented in hardware, they can be expected to be more efficient than serial techniques. Neural networks are effective means of solving problems involving parallel computations. Neural networks are also suitable candidates for heuristic solutions.

In this paper we propose a massively parallel VHDL modeling for implementing test pattern generation algorithm that was described in Reference 2. We will also consider the implementation of this algorithm in hardware, in the form of a configurable array logic.

For the purpose of test generation, a netlist of a circuit will be mapped into a Hopfield neural network for justification, and into a Perceptron neural network for fault injection and propagation. Faults are sequentially injected to the circuit and if a test vector can be found for the fault, it will appear on the circuit primary inputs and will be reported by a test controller.

The paper first discusses neural network modeling of logic circuits, and then we will illustrate how such models can be used in test generation. VHDL modeling for this purpose will be discussed next. In this part, VHDL code for the good circuit, the bad circuit and the test controller will be discussed. Following this section a simple example will be presented.

2. Neural Network Modeling

For a gate level circuit, a neural network model can be obtained. For this purpose, each net (signal) of the circuit is represented by a neuron and the value on the net is the activation value (0 or 1) of the neuron. The neurons corresponding to the primary inputs (outputs) of the circuit are called primary input (output) neurons. Each gate is independently mapped onto a Hopfield neural network. Interconnections between the gates are used to combine the individual gate neural networks into a neural network representing the circuit. Neural network for 2-input AND, OR, NAND, NOR, XOR, XNOR, NOT gates constitute the basic set. Gates with more than two inputs are constructed from this basic set. Figure 1, shows a 2-input AND gate and its corresponding neural network.

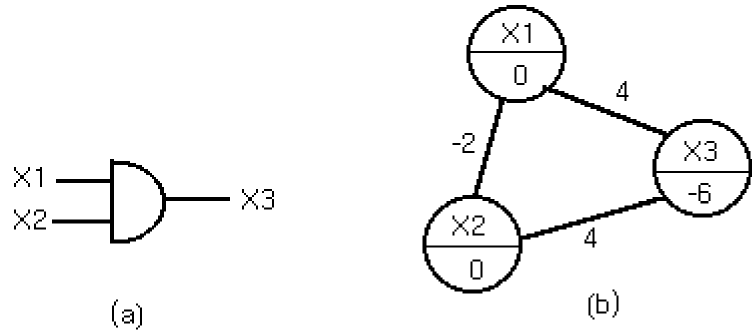


Fig.1. AND gate and the corresponding neural network

The advantage of this model is its forward and backward propagation capability. If the input neuron activations are forced on a state, the output neuron activations in the stable condition of the neural network will be consistent with the logic circuit functionality. In the opposite direction, by forcing the output neurons to a specific state, the input neuron activations will be justified by the neural network. The neural network for a digital circuit is characterized by an energy function E that has global minima only at the neuron states which are consistent with the function of all gates in the circuit. All other neuron states have higher energy. The summation of the energy functions of the individual gates yields the energy function E of the logic circuit. Since the individual gate energy functions can only assume non-negative values, E is non-negative. As an illustration, consider the logic circuit shown in Figure 2.

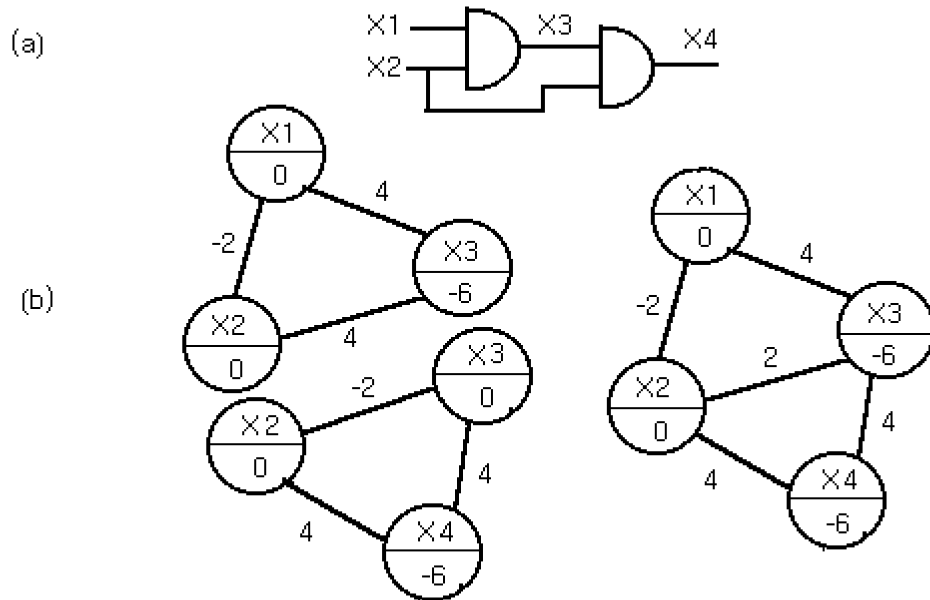


Fig. 2. Merging of two AND gates

The individual neural networks of the two input AND gates are merged together to result the neural network of the logic circuit as described below. Neurons with identical labels are replaced by a single neuron with a threshold equal to the sum of the original neuron thresholds. Similarly, identical edges are merged into a single edge with edge-weight equal to the sum of the original edge-weights. The energy of the neural network is calculated by the following expression:

$$E = (-1/2)\sum\sum T_{ij}V_iV_j - \sum I_iV_i + K;$$

where N, the range of Σ, is the number of neurons in the neural network, T_{ij} is the weight of the edge between neurons i and j, V_i is the activation value of neuron i, I_i is the threshold of neuron i, and K is a

constant. Obviously T_{ij} is equal to zero. The difference between energy of a neural network with its Kth neuron off ($V_k = 0$) and its energy with that neuron on ($V_k = 1$) is derived from the following expression:

$$\Delta E_k = E(V_k = 0) - E(V_k = 1) = I_k + \sum T_{jk} V_j$$

As it can be seen from the above expression, the following update rule reduces the total energy of the neural network:

$$V_k = (1, \text{ if } \Delta E_k > 0 \text{ and } 0, \text{ if } \Delta E_k < 0 \text{ and } V_k, \text{ otherwise}).$$

This updating rule is described as the gradient descent technique. The gradient descent algorithm terminates at a local minimum, due to the fact that it is a greedy algorithm. Greedy algorithms only accept moves towards reducing the energy of the neural network. To achieve a global minimum, probabilistic algorithms are devised that can accept moves towards increasing the neural network energy. One such algorithms will be described in Section 6.

3. Use in Test Generation

We consider stuck-at-fault model for fault simulation in a gate level circuit. This model consists of an acceptable coverage of physical errors. A test pattern for a given stuck-at-fault is a case of primary inputs that is able to control the faulty line to its inverse value, and make that fault observable at the primary outputs of the circuit. Therefore by assigning a test pattern to the primary inputs of the circuit under test, outputs of the faulty and the fault-free circuit will be different.

There are two blocks referred as the good circuit and the bad circuits in our model. The fault list is produced by fault collapsing process, faults from this list are sequentially injected to the bad circuit. The bad circuit performs fault simulation, and the good circuit performs justification of input values for any injected fault in the test generation process. There is a direct connection between primary inputs of two blocks. An interface is needed between the primary outputs of two blocks. If there is only one primary output for the circuit, the interface can be a NOT gate; otherwise the interface should make sure that there is at least one different bit between two primary outputs. If the primary inputs do not change after the good circuit justification, the values at the PIs are the generated test pattern. However if the PI values change, the above process should continue with the new primary input values.

The model described in Section 1 is appropriate for the good circuit since it is capable of input justification. Primary inputs are justified when the primary outputs are forced to specific values from the output interface and the network reaches a stable state. In order for a network to be able to do justification, it has to be a bi-directional network.

The bad circuit operates in the forward propagation mode and a unidirectional model will function properly. For this circuit, it is desirable to be able to inject faults easily and be able to propagate faults with minimum simulation time. The good circuit model will not be suitable for this purpose, since there is not a simple way for fault injection and the structure of network should be changed in order to inject a fault. The simulation time will also be long, due to two direction edges that are not required for the bad circuit. One-directional simulation insures the faster simulation and the properties of the Perceptron neural network insures the simple fault injection. Hopfield and Perceptron neural networks require the same number of neurons for representing a circuit. The difference is that edges between input neurons of a gate in a Hopfield neural network are disconnected; all other edges remain the same except that they operate in one direction only.

As an example, consider the circuit with two 2-input AND gates and its Perceptron neural network representation shown in Figure 3.

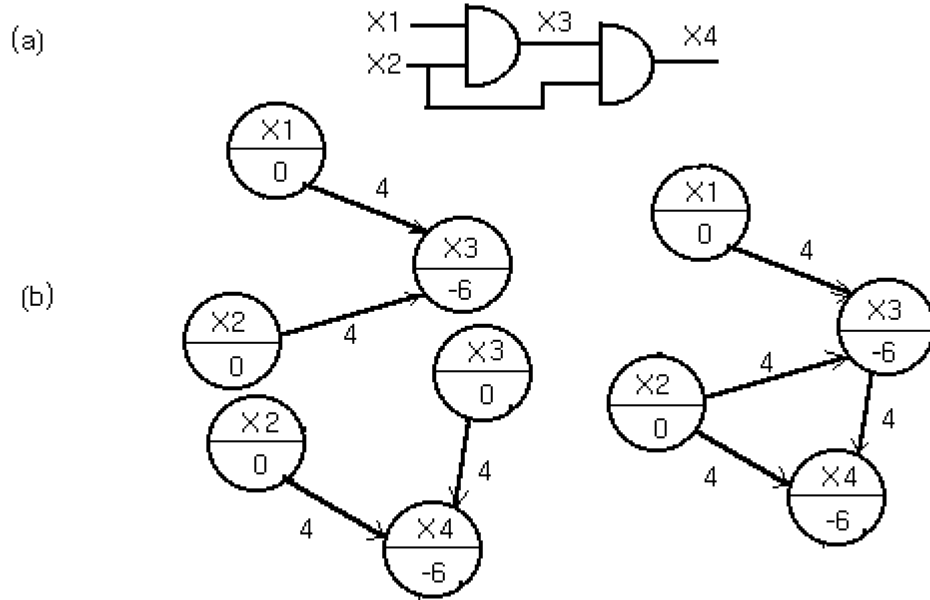


Fig. 3. Merging of two AND gates

The merge rules are the same as the rules for the good circuit model, but there is no connection between two input neurons. The threshold of an input neuron is zero.

4. VHDL Modeling

High level modeling capabilities of VHDL enable the use of this language for modeling for test generation. Component models that can operate concurrently are required for modeling a neural network. In addition to being a fully concurrent language, VHDL descriptions can be written with one-to-one hardware correspondence that will be useful in the hardware implementation of our neural network test generation methodology. Conventionally, neurons have been modeled as VHDL processes within a component description and a neural network is built by interconnecting such neuron components. However in our VHDL modeling of neural networks, an edge is modeled as a VHDL component and neurons are formed by resolution functions.

4.1. Modeling a Hopfield Neural Network

As mentioned above, the good circuit will be modeled as a Hopfield neural network. In such a network, edges are bi-directional and each edge is modeled as a VHDL component description. Neurons are modeled as resolution functions that compute neuron activation as a probabilistic function consistent with its inputs. Symbolic representation of an AND gate is shown in Figure 4.

A neuron is activated by a binary value but its inputs can be any real number. In this model after forcing a neuron activation to a value, other neurons will be justified to be consistent with the gate functionality. This ability is used in test pattern generation process. The following probabilistic P function is computed in the neuron resolution function:

$$P=1/(1+\exp(-\Delta E/T));$$

where ΔE is the sum of neuron inputs and T is the temperature value needed for the probabilistic algorithm described in Section V. The neuron input values come form the edge components and the T value is generated in the *test controller*. The test controller is described in Section 3.3.

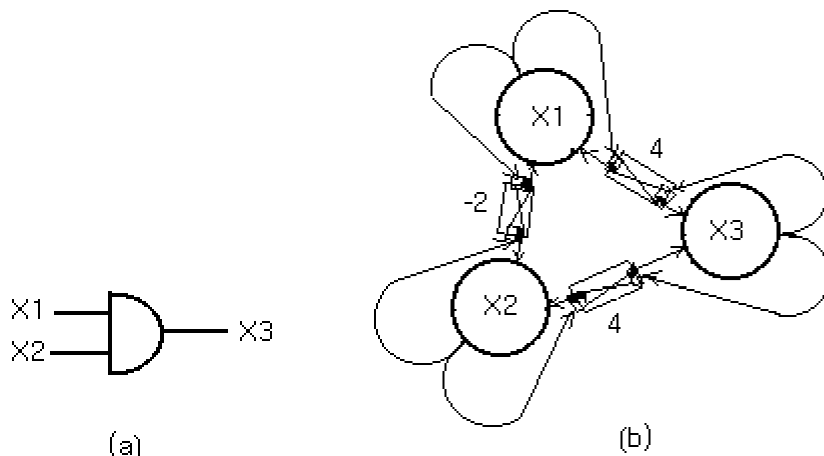


Fig.4. AND gate VHDL model

A random number, *r*, is generated by a 16-bit LFSR. If *P* is greater than *r* then the neuron output will be on, and if *P* is less than *r* then the neuron output will be off. The VHDL code for the edge component and resolution function for good circuit are shown below:

```
FUNCTION res_prob_act (drivers:rec_array) RETURN rec_type IS
….
VARIABLE  rout : rec_type :=(0.0,inp) ;
BEGIN
        FOR i IN drivers'RANGE LOOP
                IF driver(i) is of input mode THEN
                        add drivers(i).amount to the sum amount;
                ELSIF driver(i) is of temperature mode THEN
                        t:=drivers(i).amount;
                ELSIF driver(i) is of itself mode THEN
                        keep drivers(i).amount in the self variable;
                ELSIF driver(i) is of random mode THEN
                        r := drivers(i).amount;
                ELSIF driver(i) is of fault mode THEN
                        sfault:=drivers(i).amount ;
                END IF;
        END LOOP ;
        IF there was a driver of fault mode THEN
                value of the output is equal to the value of the diver with fault mode;
        ELSIF t>1.0 THEN  --perform the probabilistic algorithm
                p := 1.0/(1.0+exp(-sum/t));
                IF p>r THEN
                    rout.amount:=1.0 ;
                ELSIF p = 0.5 THEN
                    rout.amount:=self ;
                ELSE  -- p<r
                    rout.amount:=0.0 ;
                END IF;
        ELSIF sum>0.0 THEN  -- perform the greedy algorithm
                rout.amount:=1.0;
        ELSIF sum<0.0 THEN
                rout.amount:=0.0;
        ELSE
              rout.amount:=self;
        END IF;
        RETURN rout;
END res_prob_act ;
```

```
ENTITY edge IS
      GENERIC(w:INTEGER := 0);
      PORT(r,l:INOUT rec_type:=(0.0,inp));
END  edge;

ARCHITECTURE dataflow OF edge IS
BEGIN
PROCESS
BEGIN
                IF r'EVENT THEN
                          l.amount<=r.amount*REAL(w);
                END IF;
                IF l'EVENT THEN
                          WAIT FOR 0 NS;
                          r.amount<=l.amount*REAL(w);
                END IF;
                WAIT ON r,l ;
END PROCESS;
END dataflow;
```

The inputs of resolution function are records with two parts: *amount* and *mode*. *Mode* part shows the kind of resolution function input. It can be equal to *Random* mode as an input random number used in the probabilistic algorithm, *Temperature* mode also used in probabilistic algorithm, *Fault* mode for forcing a neuron activation at a given value, *Itself* mode as a feedback for a neuron, and *Inp* mode as the ordinary input numbers.

Also shown above is the VHDL code for the Edge component. This model computes the multiplication of the weight and value of the edge and outputs this value in both directions. There is a Delta time difference in changing the left and right values which is implemented by the WAIT statement in the process. This is needed so that the network reaches a stable state avoiding simultaneous changes of values. In the test controller, there is an energy shared-variable that is equal to the network energy. The network reaches a stable state when the energy is zero.

4.2. Modeling a Perceptron Neural Network

The bad circuit model is used for fault injecting and is modeled as a Perceptron neural network. As in the previous model, edges are components and neurons are resolution function. Edges are one-directional and the resolution function is simpler than the one for the good circuit.

The updating rule depends only on the sum of inputs. If the sum is greater than zero, the output will be assumed one, and if the sum is less than zero the output will be assumed zero; otherwise it will not change. In order to inject a stuck at 0 or 1 fault, a record with 0 or 1 value and the *fault* mode will be sent to resolution function by the test controller. This will cause the neuron activation to be forced to the faulty value. The edge component describes a one directional edge and outputs the multiplication of the edge weight and value to the next neuron. As an illustration, the VHDL code of edge component and resolution function for the bad circuit are shown below:

```
FUNCTION res_grad_act (drivers:rec_array) RETURN rec_type IS
…
VARIABLE  rout : rec_type :=(0.0,inp) ;
BEGIN
FOR i IN drivers'RANGE LOOP
             IF driver(i) is of input mode THEN               add drivers(i).amount to the sum amount;
             ELSIF driver(i) is of itself mode THEN           keep drivers(i).amount in the self variable;
             ELSIF driver(i) is of fault mode THEN            sfault:=drivers(i).amount ;
             END IF;
END LOOP ;
IF there was a driver of fault mode THEN     value of the output is equal to the value of the diver with fault mode;
ELSIF sum>0.0 THEN  -- perform the greedy algorithm
             rout.amount:=1.0;
ELSIF sum<0.0 THEN              rout.amount:=0.0;
ELSE                            rout.amount:=self;
END IF;
RETURN rout;
END res_gard_act ;
```

```
ENTITY edge2 IS
GENERIC(w:INTEGER := 0);
PORT(l : IN     rec_type:=(0.0,inp);
        r : OUT rec_type:=(0.0,inp));
END  edge2;

ARCHITECTURE dataflow OF edge2 IS
BEGIN
r.amount<=l.amount*REAL(w);
END dataflow;
```

4.3. Test Controller

It is assumed that first, a C program computes weights and thresholds of the Hopfield neural network. Then it outputs the good circuit component. By eliminating extra edges and modifying threshold values, the bad circuit component is generated. Primary outputs of two components are connected to each other by an interface. The circuit starts with zero values assigned to the primary inputs and fault is injected to the bad circuit. Primary outputs of the bad circuit will become available after several delta times. The interface circuit transmits values of the primary outputs of bad circuit with at least one toggled bit to the good circuit primary outputs. The values of good circuit will be justified until the network reaches to a stable condition. If in this case the new primary inputs are the same as before, they will be the test vector for injected fault; otherwise the above process will be repeated with the new primary inputs. When the test vector is obtained, other faults will sequentially be injected. Fig. 5. Shows the test pattern generation process.

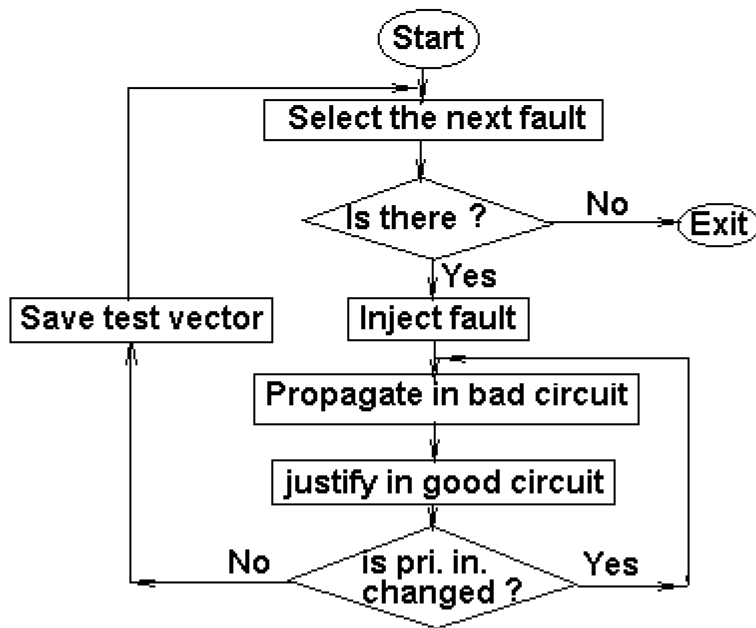
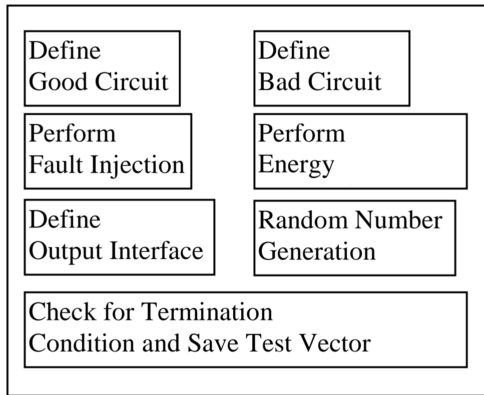


Fig. 5. Test pattern generation system

A test controller is in charge of handling the tasks described above, for which a block diagram is shown below, is a generic VHDL description. Using several GENERATE statements, the good circuit and bad circuit are defined. The edge weights and neural threshold are passed to the controller via GENERIC MAPS.



Fault injection is modeled as a PROCESS statement that sequentially reads the fault list, passes it as a constant to the controller, and forces the neurons to the faulty value. Random number generation is done by instantiating a 16-bit LFSR is the test controller. The random number is *r,* needed for the probabilistic algorithm used in the good circuit. Output interface is also defined in the test controller.

Energy calculation is performed in a PROCESS statement. The total energy of the Hopfield network is calculated using the following expression, as previously explained:

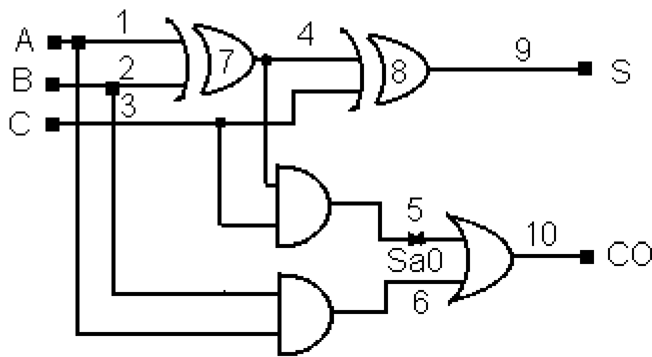
$$E = (-1/2)\sum\sum T_{ij}V_iV_j - \sum I_iV_i + K.$$

The energy value is passed to another PROCESS statement by a shared-variable. If this energy is equal to zero the test generation process has been terminated with the given primary inputs. At this point, it is checked to see if the primary input values have remained unchanged. If they have not been changed, the value will be saved as a test vector generate for the injected fault; otherwise the process with continue with new value of the primary inputs. If the energy has not reached zero, the temperature is decreased and the neuron values are calculated for the new temperature. This is continued until the energy reaches zero or the temperature value reaches 0.1; whichever happens first.

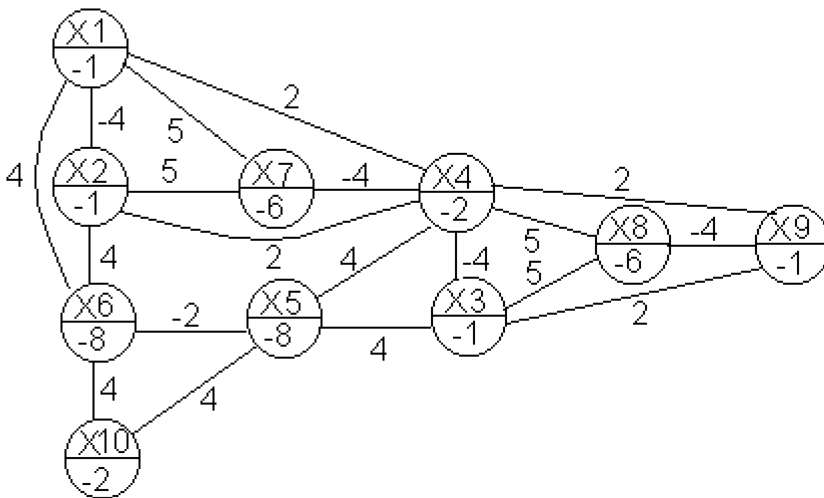
5. Example Run

As an example, we will use the neural network models to generate test vectors for a full-adder. The logic circuit is shown in Figure 6-a. This circuit is read by a C program and the two neural network models are generated. Figure 6-b shows the Hopfield neural network used to represent the good circuit and Figure 6-b shows the Perceptron neural network model that represents the bad circuit. These two neural networks are passed to the test controller via several arrays of constants and signals. The signal arrays represent the neurons of the network; since there are equal number of neurons in both networks, the signal arrays *ngood* and *nbad* are of the same length. The thresholds of the neurons are passed to the controller by the two constants *tgood* and *tbad*. The weight of edges between neurons are also passed to the controller by constants, *wgood* and *wbad*. Each member of the weight arrays show that there is an edge between *source* and *destination* with the weight equal to *w* in the format: (*source, destination, w*).

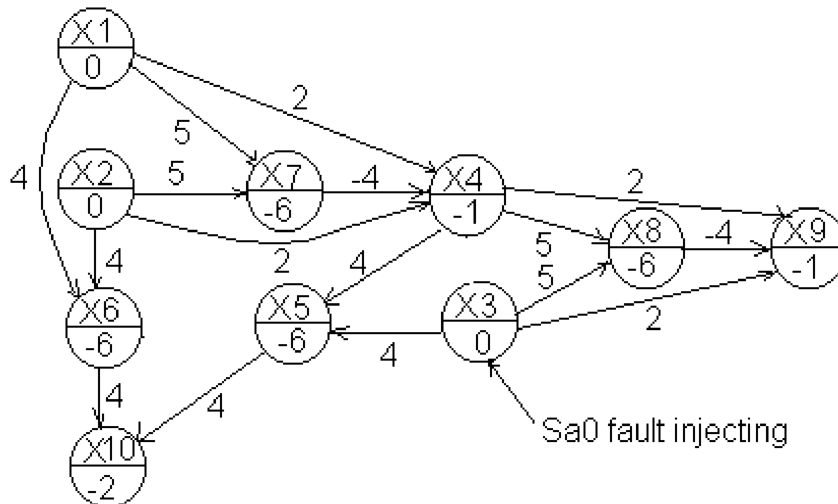
Fault list is also passed to the model as an array of constants, showing that neuron *i* is stuck-at -*v* in the format (*i, v*). By simulating the model, the algorithm described before will generate test vectors for the faults in the fault list.



(a) Logic circuit



(b) Good circuit



(c) Bad circuit

Fig. 6. Modeling of a Full-Adder

The following code shows an example VHDL architecture for test generation using the neuron models:

```
ARCHITECTURE structural OF . . .  IS
        SIGNAL ngood: rec_array (1 TO 10) := ((0.0,inp), (0.0,inp), (0.0,inp), …);
        SIGNAL nbad:rec_array (1 TO 10) :=  ((0.0,inp), (0.0,inp), (0.0,inp), …);
        CONSTANT tgood:threshold_array (1 TO 10):= (-1.0, -1.0, -1.0, -2.0,…);
        CONSTANT tbad:threshold_array (1 TO 10):= (0.0, 0.0, 0.0, -1.0, …);
        CONSTANT wgood:weight_array (1 TO 19):= ((1,2,-4), (1,7,5), (2,7,5), …);
        CONSTANT wbad:weight_array2 (1 TO 16):= ((1,6,4), (2,7,5), (7,4,-4), …);
CONSTANT fault_nodes : fault_array (1 TO 22):= ((1,0.0), (2,1.0), (7,0.0), …);
…
BEGIN
        ll: test_controller GENERIC MAP (tgood, tbad, wgood, wbad, k, Init_T, input_range, fault_nodes)
                PORT MAP (ngood, nbad);
END structural;
```

6. Probabilistic Algorithm

The probabilistic algorithm is based on the physics principle of annealing. This is a process whereby a material is heated and then cooled slowly to a freezing point. As a result, the crystal lattice will be highly ordered, without any impurities such that the system is in a state of very low energy. At the gradient descent algorithm, we described before, we use a deterministic update rule. And due to the fact that it is a greedy algorithm, it can terminate at a local minimum. But the probabilistic algorithm uses a stochastic update rule, in which a neuron becomes active with a probability P, $P(V_k \leftarrow 1)=1/(1+\exp(-\Delta E/T))$. Where T is a parameter comparable with the temperature of the system. At low temperature there is a strong bias in favor of states with low energy, but the time required to reach equilibrium may be long. At higher temperatures the bias is not so favorable but equilibrium is reached faster. A good way to beat this trade-off is to start at a high temperature and gradually reduce it. At high temperatures, the network will ignore small energy differences and will rapidly approach equilibrium. In doing so, it will perform a search of the coarse overall structure of the space of global states, and will find a good minimum at that coarse level. As the temperature is lowered, it will begin to respond to smaller energy differences and will find one of the better minima within the coarse-scale minimum it discovered at high temperature.

7. Output Interface

There is an interface between primary outputs of good circuit and bad circuit. When the circuit has only one primary output the interface is a NOT gate. Modeling a NOT gate between two blocks can be done with injecting a fault to the primary output of good circuit .This fault value is reverse of bad circuit primary output. If there is n primary outputs, the interface consists of n 2-input XOR gates and one n-input OR gate. And the output of the interface circuit that is OR function of the outputs of n XOR gates is assigned value 1 throughout test generation. because the model of this interface is a Hopfield neural network, it is a part of the good circuit.

8. Conclusion

A new algorithm for generating test patterns, based on Hopfield and Perceptron nets, is proposed. Due to special properties of such networks, the algorithm is suitable for massively parallel execution. We consider the hardware implementation of this model on FPGAs, because of their relaxation. In the process of implementing, it may be recognized that applying real circuit is more suitable than bad circuit model and also the primary output interface is better to replace by a neural network. Up to now, we have not used learning ability of neural networks. We can optimize number of neurons in the good circuit, if we replace its model with a Hopfield network that has learned the circuit functionality.

References

[1] Zainalabedin Navabi, "VHDL Analysis and Modeling of Digital Systems" MCGRAW HILL,1993

[2] S.T.Chakradhar,M.L.Bushnell,V.D.Agraval, "Toward Massively Parallel Automatic Test Generation " pp.981_994 ,IEEE TRAN.CAD 1990

[3] Richard.P.Lippmann, “ An Introduction to Computing with Neural Networks " IEEE ASSP MAGAZINE APRIL 1987

[4] E.L.Aarts and J.H.Korst,Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing. New York : Wiley , 1989

[5] G.L.Heileman M.Georgiopouplos,W.D.Room " A General Framework for Concurrent Simulation of Neural Network Models " IEEE TRAN.ON SOFTWARE ENGINEERING VOL. 18,No.7 JULY 1992.

[6] H.Fujiwara . " Three_Valued Neural Networks for Test Generation "In Proceedings of the 20th IEEE International Symposium on Fault Tolerant Computing , pp 64-71 ,June 1

[7] J.J.Hopfield ," Artificial Neural Networks ", IEEE 1988

[8] Bang W.LEE ,and Bing J.Sheu . " Modified Hopfield Neural Network for Retrieving the Optimal Solution ", IEEE Transaction on Neural Networks,Vol 2 ,No. 1,January 1991 .

[9] Assad Makki and Pepesity ," Hopfield Neural Networks for Optimal Solutions " IJCNN 1992 .

[10] S.T.Chakradhar,M.L.Bushnell,V.D.Agraval "An Energy Optimization for Delay Faults ", IEEE TRAN.CAD 1995

[11] G.Eldredge ,L.Hutchings ," Run-Time Reconfiguration : A Method for Enhancing the Functional Density of SRAM-based FPGAs", Kluwer Academic Publishers, Boston.