LOGIC SIMULATOR FOR BENCH FORMAT

By

Jie Qin

Dept. of Electrical and Computer Engineering

Auburn University

(Final Project for ELEC7250)

1. Introduction

In the final project, we need to design a logic simulator with support for the hierarchical bench format which was proposed in our previous project. And the implemented logic simulator need support the standard bench format too since it is widely used in the benchmark circuit description, such as, ISCAS'85, ISCAS'89, and ISCAS'99. Because the hierarchical bench format I proposed has the different grammatical structure from the standard one, the logic simulator should provide the option to accept the circuit description in different format.

Another goal of the project is to try to detect the error which may happen in the provided design and roughly locate the place where the error happens. Working in such a mode, the user need provide the design file and expected response of the system. If the simulated result is different from the expected response, the simulator should be able to diagnose the error automatically and give out a suspect area where the error is most likely to happen.

This report is organized as follows. Section 2 gives a detailed introduction to the Hierarchical Bench Format I used in the final project. This is followed by a brief introduction to the logic simulator in Section 3. Then the working mechanism of the compiler and the simulator are well presented in Section 4 and 5 respectively. An algorithm to perform the fault diagnosis is proposed in Section 6. Section 7 provides some experimental results. And finally the section 8 concludes the report.

2. Hierarchical Bench Format

In order to support the hierarchical description, a circuit description language need provide the mechanism to define a system and to reuse the system to build another system. So a hierarchical language must be able to describe the following stuffs:

- system's name
- system's ports(input, output, or bidirectional inout)
- system's function
- instantiation of a system in another system

The following block is the hierarchy *Bench* format I proposed to define a system:

SYSTEM (*oport_name, … , oport_name, ioport_name, … , ioport_anme*) = *system_name* (*iport_name, … ,iport_name*)

INPUT(*iport_name*)

...

INPUT(*iport_name*)

OUTPUT(*oport_name*)

...

OUTPUT(*oport_name*)

INOUT(*ioport_name*)

...

INOUT(*ioport_name*)

instantiation of another user-defined system or a basic gate

```
    ...
ENDSYSTEM
```

system_name is the unique name used to identify the system going to be defined; *iport_name, oport_name,* and *ioport_name* are the input, output and inout ports of the system "*system_name*" respectively. A port can be declared as a bus by giving the bus width. A certain line of the bus can be identified using a subscript. Here is an example. Suppose a system "**SYS_A**" which has a *4-bit* input bus and one output port. Its definition could be written as follows:

```
SYSTEM (output_port) = SYS_A (input_bus)
    INPUT(input_bus[4])
    OUTPUT(output_port)

    …
ENDSYSTEM
```

Then the first, second, third, and fourth line of the "input_bus" could be referenced as input_bus[0], input_bus[1], input_bus[2], and input_bus[3] respectively in the above block.

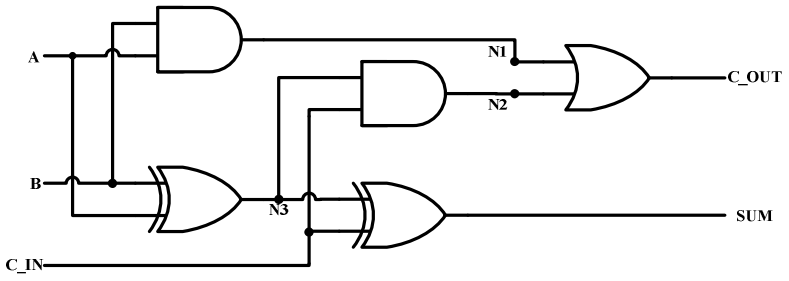
The instantiation of a user-defined system or a basic gate could also be clearly illustrated by an example. Suppose a system "**SYS_B**" which has three input ports and two output ports. Its instantiation should be like this:

(*output_port1, output_port2*) = *SYS_B* (*input_port1, input_port2, input_port3*)

When a system has only one output port, the parentheses at the left side of the equal sign can be deleted for simplicity. Using the format mentioned above, *Bench* provides the hierarchical ability to describe the four stuffs listed at the beginning of this section.

In order to show how to describe a circuit in the hierarchical bench format, a *4-bit* ripple-carry adder will be given as an example below. Firstly, A *1-bit* full adder is defined; then four *1-bit* full adders are cascaded together to construct the *4-bit* ripple-carry adder.

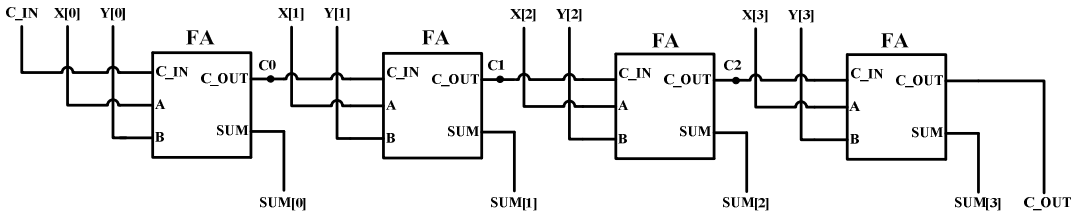
The *1-bit* full adder is described in *Bench* as follows:




```
SYSTEM (SUM, C_OUT) = FA (A, B ,C_IN)
    INPUT(A)
    INPUT(B)
    INPUT(C_IN)
    OUTPUT(SUM)
    OUTPUT(C_OUT)
```

```
        N3 = XOR(A, B)
        SUM = XOR(C_IN, N3)
        N1 = AND(A, B)
        N2 = AND(N3, C_IN)
        C_OUT = OR(N1, N2)
    ENDSYSTEM
```

The *4-bit* ripple-carry adder is described using the defined system "**FA**". The following is the source code.




```
SYSTEM (SUM, C_OUT) = FourBitAdder (X, Y ,C_IN)
    INPUT(X[4])
    INPUT(Y[4])
    OUTPUT(SUM[4])
    OUTPUT(C_OUT)

    (SUM[0], C0) = FA(X[0], Y[0], C_IN)
    (SUM[1], C1) = FA(X[1], Y[1], C0)
    (SUM[2], C2) = FA(X[2], Y[2], C1)
    (SUM[3], C_OUT) = FA(X[3], Y[3], C2)
ENDSYSTEM
```

3. Brief Introduction to the Logic Simulator

The logic simulator I implemented consists of a compiler and a simulator. The main job of the compiler is reads in the circuit description and translates the information in the description into the "gate records" which are stored in memory. Actually there are two built-in compilers in the logic simulator. One is the hierarchical compiler used for hierarchical bench format; the other is the standard compiler used for the standard format. In a hierarchical circuit description file, there may be several systems defined inside. In order to generate a flatten list of "gate records" which is required by the simulator, the hierarchical compiler needs to analyze the circuit description in two rounds. In the first round, the compiler just reads in the circuit description line by line and builds a system list just depending on the system declaration in the description. In the second round, the compiler will try to expand the systems included in the designated system according to the system list obtained in the first round and generate a flatten list of "gate records". However, in the standard compiler, there is no such a problem associated with the hierarchical one, so the compiling job can be done in one round. The detailed information about how the compiler works will be given in Section 4 using the hierarchical compiler as an

example.

The simulator utilizes the flatten list of "gate records" and maintains two lists in memory to propagate the input vectors at PIs to POs. The detailed working process will be given in Section 5.

4. Compiler

The basic architecture of the compiler is based on a well-known "syntax-directed" design technique. It breaks the compiler into three modules, as shown in Fig. 1. The function of the modules (in hierarchical compiler) and how they work in the first round are described as follows:

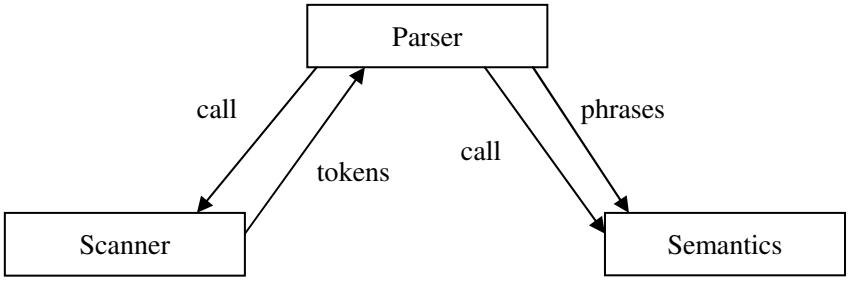


Figure 1. General Model for Compiler

- The scanner reads in the source file line by line and breaks each line into several tokens using a set of predefined separators. Then the tokens are delivered back to parser.
- The parser gets the tokens from the scanner and utilizes several built-in finite state machines to generate phrases depending on the tokens (Note: each line of tokens generates a phrase). The phrase can be classified into three categories: system declaration, ports declaration, and circuit connection. So the parser utilizes three corresponding finite state machines to process the tokens. The parser works according to the following procedure: Firstly the parse needs to tell which category the tokens in a line belongs to according to the reserved word used in this line. Then the parser will call the corresponding finite state machine to process this line of tokens. Using finite state machines, the parser is capable of checking the syntax and transforming the source code to the phrases stored in the memory. Finally the parser will transfer phrases to the semantics module to do the rest of the job.
- The semantics module gets the phrases from the parser and processes them block by block using another finite state machine. Here the semantics module supposes that every system described in the source file consists of four parts in order. The first one is the beginning of the system which gives the system's name and the name and the order of the ports the system has. The second is the ports declaration which gives the bus width of the ports. The third part is the circuit description. The last one is the end of the system. The finite state machine will process the phrases according to the four orderly

parts. In this stage, the semantics module will also generate and maintain an internal wire (also called as internal node) list automatically for future use. Using the finite state machine, the semantics module can check the consistency of the ports description and the circuit description to some degree. Finally, all the information are organized as a system list and stored in memory.

Then it will be very easy for the compiler to generate a flatten list of "gate records", which is also called as simulation table, in the second round according to the system list obtained in the first round. The "gate record" is a data structure which is organized in memory as the table shown below.

ID
Name
Class
Fanin_List (FIL)
Fanout_List (FOL)
Fanin_Value (FIV)
Gate_Value (GV)

Table 1. Data Structure of a Gate Record

In a simulation table, all the PIs, gates, and POs in a given circuit are going to be represented by the "gate record". A constructed simulation table built by the compiler for the circuit in Figure 2 is shown in Table 2.

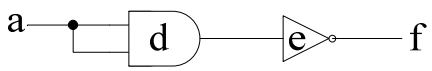


Figure 2 A Simple Circuit

Gate "d"		**Gate "e"**		**PI "a"**		**PO "f"**
ID: 1		ID: 2		ID: 3		ID: 4
Name: "d"		Name: "e"		Name: "a"		Name: "f"
Class: "AND"		Class: "NOT"		Class: "PI"		Class: "PO"
FIL: {3, 3}		FIL: {1}		FIL: {3}		FIL: {2}
FOL: {2}		FOL: {4}		FOL: {1}		FOL: {4}
FIV: {-2, -2}		FIV: {-2}		FIV: {-2}		FIV: {-2}
GV: {-2}		GV: {-2}		GV: {-2}		GV: {-2}

Table 2 Illustration of Simulation Table

It should be noted that the possible values for the elements in FIV and GV are -2, -1, 0, and 1. The states that these values represent are summarized in Table 3.

Value	States
-2	NULL
-1	Logic “X”
0	Logic “0”
1	Logic “1”

Table 3 Logic Value Table

5. Simulator

The simulator must be able to propagate the values applied at PIs through the whole circuit and obtain the value at POs. This process is called as logic propagation in this report. Before a vector is applied to the simulated circuit, the simulator will initialize all the FIVs and GVs in simulation table to value “-2”. After the vector is applied to PIs, the logic propagation begins. During the logic propagation, the circuit can be divided into two parts. One part includes all the “gate records” in known state (known state means that the output of the “gate record” is not equal to “-2”), while the other one includes all the “gate records” in unknown state (unknown state means that the output of the “gate record” is equal to “-2”). If the simulator can find the border between these two parts, it would be easy to propagate the known value to the area in the unknown state. In order to do this job, two lists are maintained in the simulator.

- Active List: All the “gate records” in this list are in known state. However, once all the fanouts of a “gate record” enters into the known state, the “gate record” will be removed from this list.
- Passive List: This list includes all the “gate records” in unknown state. Once a “gate records” in this list changes from the unknown state to known, it will be moved from this list to the active list.

Using these two lists, what the simulator needs to do is just to pick up a “gate record” from the active list and propagate the known value to the fanouts of the “gate record”. It will accelerate the simulation process a lot. The detailed working process can be shown in the flow chart in Figure 3.

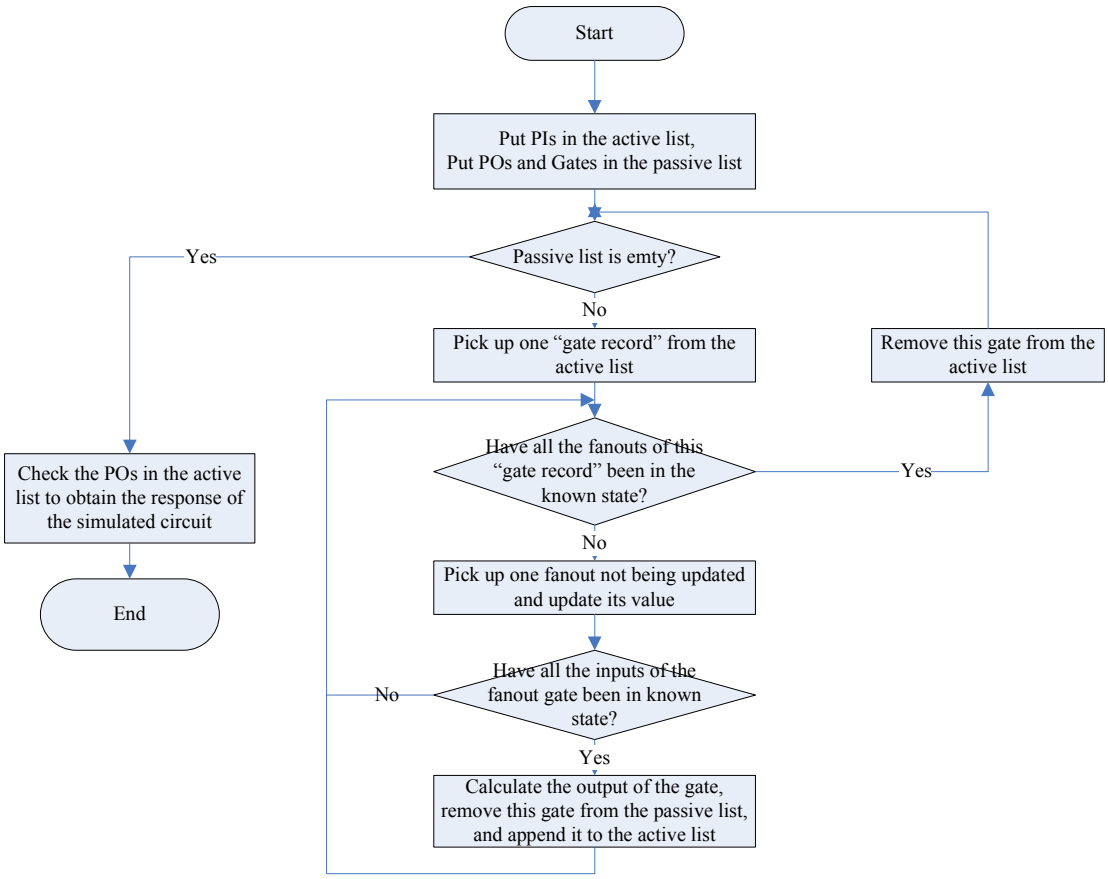


Figure 3 Flow Chart of the Logic Simulator

6. Fault Diagnosis

Fault diagnosis is a very difficult problem which is still not well understood by people. So before I started my attempt on this problem, I made three assumptions to make the problem easier.

- Assumption #1: the internal state cannot be observed from the outside of the circuit.
- Assumption #2: the interconnection information is known beforehand and no fault related to the interconnection happens.
- Assumption #3: the possibility of a single fault is much larger than multiple faults.

According to these three assumptions, the fault diagnosis must be able to find out the area where a single fault may happen based on the input vectors and output response of the circuit. Also the interconnection information could be used to diagnose the fault. Therefore, the following diagnosis procedure is proposed:

- Step #1: for each faulty PO, obtain a reversed logic cone from PO;
- Step #2: intersect the logic cones obtained in Step #1 to generate a suspect list.
- Step #3: try more test vectors, intersect the suspect lists obtained from each test vectors, and generate a suspect list of gates as small as possible.

The fault diagnosis algorithm still needs to be improved because of a couple of reasons. First, it can only give very rough estimation where the fault may happen. Second, the assumption #2 does not meet the real situation because the possibility of an interconnection fault is very high. Third, a module may be reused for many times in a hierarchical design. So if a fault happens in a module and this module is reused by another system several times, the fault would repeat multiple times in the second system. All these issues would seriously lower the accuracy and resolution of the proposed diagnosis algorithm.

7. Experimental Results

In order to the check the performance of the logic simulator, the execution time was measured for a 4-bit adder with different number of test vectors. The result is shown in the Figure 4 as follows. Observing the figure, we can find out that the execution time almost linearly increases with the number of the input vectors.

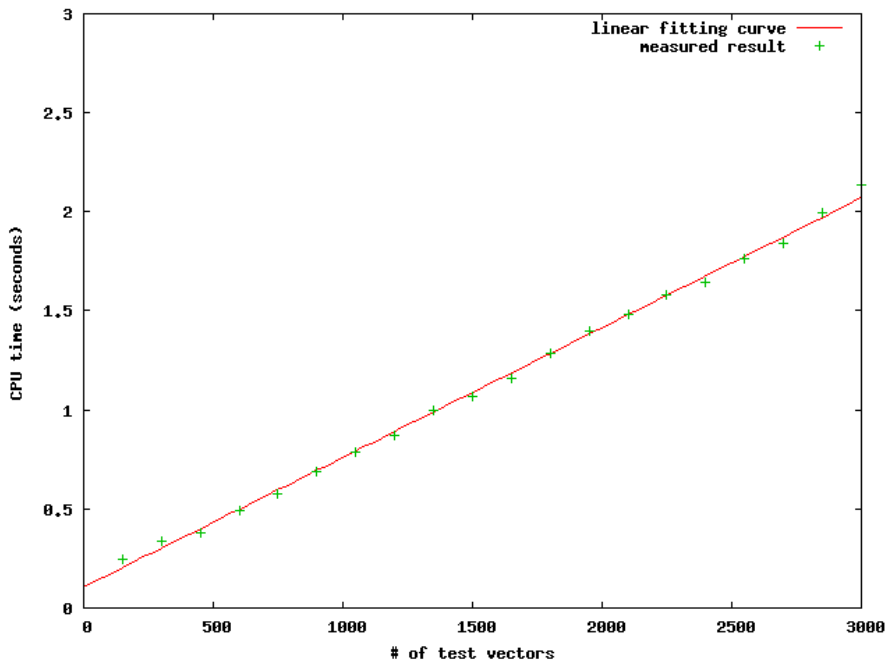


Figure 4. Execution Time vs. Number of Test Vectors

The ISCAS'85 benchmark circuits were also simulated with 1000 pseudo-random vectors using the implemented logic simulator and the execution time is recorded for each circuit. The result is shown in Figure 5. Two curves at 1st-order and 2nd-order are tried to fit the measured result. According to the comparison between these three curves, we can say that the execution time increases slightly faster than the linear function of the number of gates in the simulated circuit.

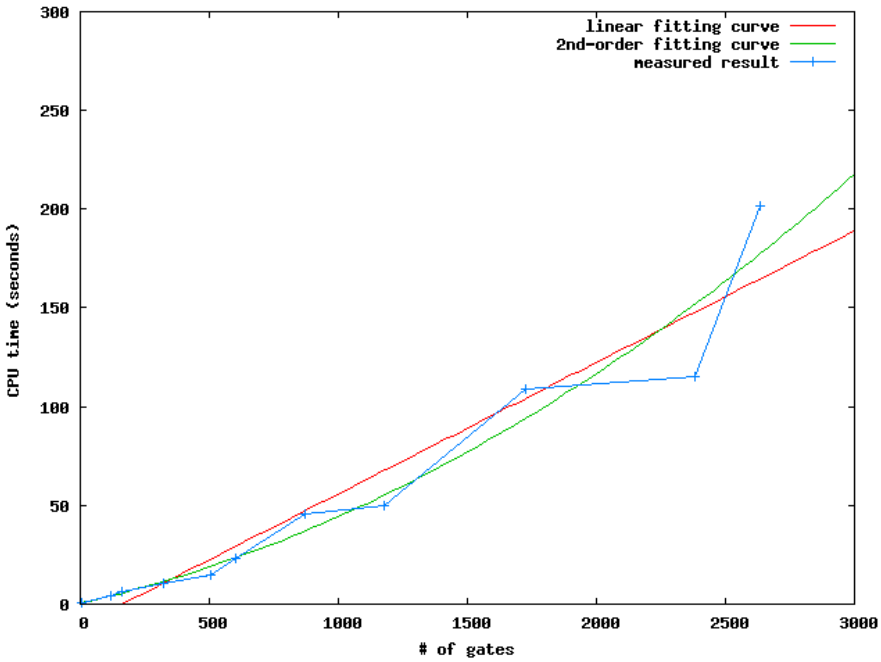


Figure 5 Execution Time vs. Number of Gates in Circuit

8. Conclusion

The logic simulator was implemented successfully. Its execution time increases almost as a linear function of the number of the test vectors and gates, which is what I expected. But the diagnosis is still under work. The algorithm should be improved and implemented to check its performance.