Locating Functional Errors in Logic Circuits

Kensaburo Alfredo Tamura

NEC Corp.
C&C Systems Research Laboratories
Kawasaki, Japan

Abstract

In the verification phase of the design of logic circuits using the top-down approach, it is necessary not only to detect but also to locate the source of any inconsistencies that may exist between the functional-level description and its gate-level implementation. In this paper we present a method that determines the areas, within the gate-level circuit, that contain the functional errors. The indicated areas are shown to have sufficient resolution to allow the designer to quickly find the cause of the inconsistency and, therefore, reduce the time required for debugging.

1. Introduction

In the design of large digital systems, such as mainframe computers, a complete specification at the register-transfer level (the functional level) is written for the simulation of the entire system—a simulation of the whole system at the gate level is impractical in many cases. When this description is found to be correct, via simulation, a gate-level circuit is designed based upon this description. Although many advances have been made in logic synthesis, this design is still mostly manual, particularly for circuits in which the timing is critical. Consequently, it becomes necessary to detect, locate, and correct any inconsistencies that may exist between the functional-level representation and the gate-level representation. A major portion of the total logic design time is spent at this stage.

There are two approaches used in practice at this stage of the design process. The first and most common approach is simulation [SASA84, ABAD88]. In this approach the functional-level circuit and the gate-level circuit are both simulated with the same input patterns, and the outputs of these circuits are then compared to check for any inconsistencies. Although this approach is the quickest and most efficient, it has the disadvantage that the verification is not complete because of the impracticality of simulating the circuits with all possible input combinations. Therefore, it is possible that a logic error will not be detected until after the hardware has been made—a costly affair. In addition, the generation of the input test patterns is very time-consuming.

The second approach is Boolean comparison. The functional-level description is converted into a Boolean expression and then this is compared, using formal verification techniques, with the Boolean equation that corresponds to the manually-designed gate-level circuit. The two functions are tested for equivalence by proving the graph isomorphism of binary decision diagrams [AKER80, BRYA86], or by proving the tautology of the exclusive-or of the two functions [SMIT82, ODAW86]. Other techniques are described in [HACH88].

Both of the above approaches give a yes/no answer to equivalency but no useful information on the location of the logic error. By logic error or functional error we mean those errors that can not be detected just by checking the structure. To detect these errors it is necessary to analyze the function of the circuit. A logic error would occur, for example, when an inverter is missing or when there is an AND gate instead of an OR gate. There are tools and techniques for aiding the design engineer in finding the error, but it still remains mostly a manual process requiring considerable amounts of the engineer's valuable time.

The need to locate the source of the inconsistencies between the functional-level description and its gate-level implementation occurs frequently. The implementation of the gate-level circuit usually goes through several iterations. During each iteration the circuit needs to be verified, and if there are any inconsistencies then the errors need to be found and corrected. This search for the location of the errors will also occur during technology re-mapping.

To shorten the design cycle, this paper addresses the above problem of finding the location of functional errors within logic circuits. A method for finding such errors, to be of any practical use, must have *resolution* and be *deterministic*. By *resolution* we me mean that the area specified as containing the error must be small enough so that the designer can quickly determine the exact cause and proceed to make the proper corrections. It must also be *deterministic* because it is not of much use to say that "maybe" the error is contained in a specified area.

Although extensive research has been done in the area of verification, very little has been reported on how to locate the errors once the circuit has been shown to be incorrect. The technique described in [ODAW86] uses the patterns that yield an inconsistency. The circuit is simulated with each of these patterns to find the values of all the internal nets. The circuit is then traced from the output back to the

inputs along the paths that yield an incorrect value. The backtracking continues until a gate is reached in which it is no longer possible to determine a faulty path. The error is, thus, said to be found somewhere between this gate and the inputs. Often such a gate is reached close to the outputs, and, as a consequence, the area that is said to contain the errors occupies a large portion of the circuit; i.e., this method does not have *resolution*.

The method presented in this paper satisfies the above conditions of being *deterministic* and having *resolution*. It achieves this by partioning the functional description of the circuit into *sub-functions*, and then verifying the sub-circuits corresponding to these *sub-functions* to determine the location of the functional errors. In this procedure, no assumptions are made on the type of functional errors that may occur.

2. Verification

To be able to find *deterministically* the location of logic errors, it is first necessary to prove that the descriptions at the functional and gate levels are consistent for all regions not containing logic errors. This means that formal verification techniques need to be used. The Boolean comparison method described above is used at this stage of the location method, but not in the traditional manner. Because of the NP-complete nature of the the Boolean comparison problem [GARE79], it is not efficient to apply these methods directly to the verification of logic circuits.

The computation time required for comparing two Boolean functions, in the worst case, grows exponentially with the number of variables. In the proposed method, to reduce the number of variables contained in the functions that need to be compared, each statement in the functional-level description is partitioned into *sub-functions*. The corresponding *sub-functions* of the circuit are extracted by performing a hybrid symbolic simulation in which symbols as well as Boolean values can be processed [BARR84, SRIN86]. This partitioning, as described in section 3, also serves as the basis for locating the functional errors.

2.1 Partitioning of Functional Descriptions

The functional-level description language used within NEC, is a low-level register-transfer language called FDL [KATO83](an example of an FDL description is shown in Fig. 1 and the corresponding gate-level circuit is shown in Fig. 2). It is a non-procedural language that describes hardware by dividing it into functional blocks, each one containing combinational logic and a register. Each statement describes one of these functional blocks by defining the next state of the register (`NOC` stands for "No Change"). If, for the signal names of the primary inputs, primary outputs, and registers, there is a one-to-one correspondence between those of the FDL description and those of the gate-level circuit, then each of these statements can be analyzed separately. The problem is then reduced to verifying and debugging combinational circuits.

FDL statements use basically two structures to describe behavior: the `IF... THEN... ELSE...` and `CASE... OF...` structures (both of which can be used recursively). These statements can, therefore, be thought of as having input data signals, input control signals and output data signals. Those signals appearing in the conditional parts (the `IF` and `CASE` parts) represent the control signals. The `THEN`, `ELSE` and `OF` parts consist of expressions that describe specific *sub-functions* in terms of the input data signals. For example, in the following expression

```
REG MATON = IF TMRST
            THEN 0
            ELSE IF BCK .UP.
                 THEN CASE TSMC OF
                      /0/ IF TSC8EP
                          THEN NOC
                          ELSE IF MTRCRT
                               THEN 0
                               ELSE IF MTRCST
                                    THEN NNSNOT
                                    ELSE NOC,
                      /1/ NNSNOT
                 ELSE NOC;
```

Figure 1: Example of Circuit Description in FDL

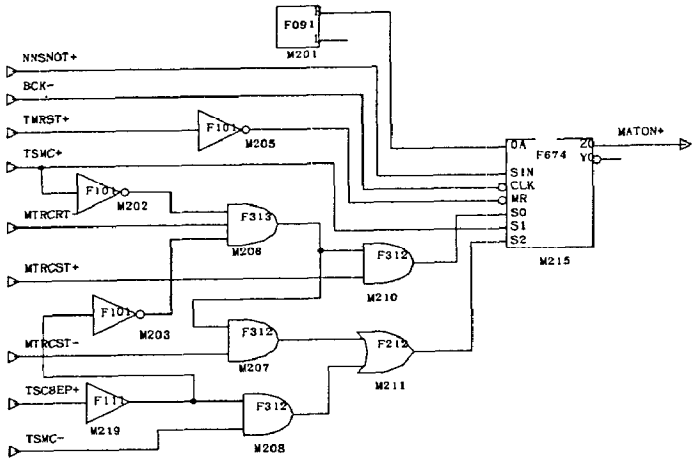

Figure 2: Gate-level Circuit

`OF` parts consist of expressions that describe specific *sub-functions* in terms of the input data signals. For example, in the following expression

```
Z = IF G THEN T.OR.U
          ELSE IF H THEN V
                    ELSE W;
```

Example 1

the control signals are `G` and `H`, the input data signals are `T`, `U`, `V`, and `W`, the output signal is `Z`, and the *sub-functions* are `T.OR.U`, `V`, and `W`. Likewise, in the following expression

```
K = CASE M-N OF
          /0 0/ P.AND.Q
          /0 1/ 1
          /1 0/ R
          /1 1/ P
```

Example 2

the control signals are `M` and `N` (the "-" denotes concatenation), the input data signals `P`, `Q` and `R`, and the *sub-functions* `P.AND.Q`, `1`, `R`, and `P`.

2.2 Extraction of Sub-Functions

To extract the corresponding sub-functions from the gate-level circuit, the combinational circuit is modeled as a black box that has the same input data signals, input control signals, and output data signals as the functional-level

description. Depending on the input pattern of the control signals, the black box will have a specific sub-function—similar to a complex ALU. If a hybrid symbolic simulation (in which symbols as well as Boolean values are allowed) is performed by applying Boolean values at the control inputs and signal names at the data inputs, the respective sub-function will appear at the output (Fig. 3). The equivalence problem is then reduced to determining if each of the sub-functions that are described in the functional-level description is equivalent to the corresponding sub-functions of the gate-level circuit.

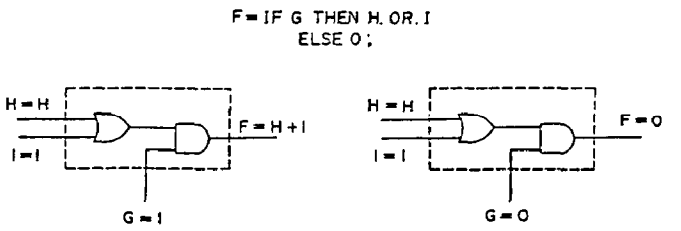

Figure 3: Hybrid Symbolic Simulation

Although it may not be immediately apparent, partitioning the-functional description reduces the computation time. Earlier systems [SMIT82] applied the Boolean comparison method to the whole function describing the combinational circuits between the latches. In other words, each statement in the functional description was converted into a Boolean expression, and this was compared to the logic function corresponding to the combinational circuit. For Example 2, the Boolean expression would be as follows:

$$K = \overline{M}\,\overline{N}PQ + \overline{M}N + M\overline{N}R + MNP$$

The computation time required, however, for comparing two boolean functions grows, in the worst case, exponentially in the number of variables. Therefore, the computation time can be reduced considerably by decreasing the variables that appear in the functions to be compared. In most statements in functional-level descriptions, each input data signal appears only in some of the sub-functions. Therefore, partitioning the statement reduces the number of variables. Let's consider the same `CASE` example. The sub-functions are as follows:

$$\begin{array}{llll} K = PQ & \text{when} & M = 0, N = 0 \\ K = 1 & \text{when} & M = 0, N = 1 \\ K = R & \text{when} & M = 1, N = 0 \\ K = P & \text{when} & M = 1, N = 1 \end{array}$$

In this case three Boolean comparisons, each with at most two variables, would be required instead of one comparison with five variables. In order to get an idea of how these two approaches compare, let's assume that this description is to be verified using a truth table. Without partitioning, the table would have 32 ($= 2^5$) entries. With partitioning, the table would have 8 ($= 2^2 + 2 + 2$) entries. Of course, if each of the input data signals `P`, `Q`, and `R` appeared in each of the four sub-functions, then the table would have 32 ($= 2^3 + 2^3 + 2^3 + 2^3$) entries. The partioning would then be meaningless, but such cases are rare. Even if, on the average, only one input data signal does not appear in each of the sub-functions of a given statement, the computation would still be reduced by half ($= 2^2 + 2^2 + 2^2 + 2^2 = 16$).

The computation is further reduced by the fact that often there are fewer than 2^n sub-functions, where n is the number of control bits. This occurs whenever one of the control variables is a "don't care" for a sub-function. For example, in the nested `IF-THEN-ELSE` structure shown in Example 1, `H` is a "don't care" for obtaining the sub-function `T.OR.U`. As a consequence, there are in total three sub-functions instead of the four sub-functions that could possibly be obtained from the two control variables `G` and `H`.

By partitioning the statements in the functional-level description, however, an overhead is incurred; i.e., it is necessary to determine the values of the conditional variables for each of the sub-functions. This is straightforward for a sub-function inside a `CASE` statement, but for a sub-function in a nested `IF-THEN-ELSE` structure it is necessary to take the intersection of all the conditions in the outer `IF-THEN-ELSE` structures and then minimize this expression. In the following example,

```
Y = IF condition1
        THEN sub-function1
        ELSE IF condition2
                THEN sub-function2
                ELSE sub-function3
```

the conditions for obtaining *sub-function2* are when

$$(\overline{condition1}) \cdot (condition2) = 1$$

It was found that this minimization could be efficiently computed because the control signals constitute only a fraction of all the input signals, and because each of the conditional variables tend to appear only in one of the *conditions*.

What the above partitioning method is doing, in effect, is to substitute logic values for some of the variables to reduce the complexity of the Boolean comparison of two functions. The variables selected for this substitution were the control signals, but it is possible that a different selection would give more effective results. However, searching for a better selection would be inefficient if the total number of variables (for control and data signals) is large. The effectiveness, of the proposed approach can be explained by the fact that designers tend to describe the circuit functions as simply as possible. In other words they prefer to use the expressions such as

```
Y = IF A THEN B
          ELSE C;
```

to describe the function of a two-input multiplexor instead of

```
Y = IF C THEN A'.OR.B
          ELSE A.AND.B;
```

(`A'` denotes the negation of `A`). The first expression would benefit from partioning but not the second.

The proposed partitioning method has a dual purpose. In addition to reducing the computation time, the partitioning of the circuit is used for searching efficiently for the logic errors. The reasons are described in the next section.

3. Locating Errors

When debugging, determining which statements are inconsistent with the corresponding combinational circuit is only half the problem. The other half is to find the location

of the error and then to make the appropriate corrections. Presently, searching for the error is done manually. What designers commonly do is to visually inspect the circuit diagrams and trace each path that exists between the inputs and the outputs that give an inconsistent result. As the designer traces each path he does a "mental" simulation to try to determine if the path is correct or not. This can be a time-consuming and error-prone task especially if several IF-THEN-ELSE structures are nested. If the circuit is complicated the designer may resort to simulation and set "probe" points within the circuit. But doing so, requires that the designer determine beforehand the expected values at the probe points for a set of input patterns.

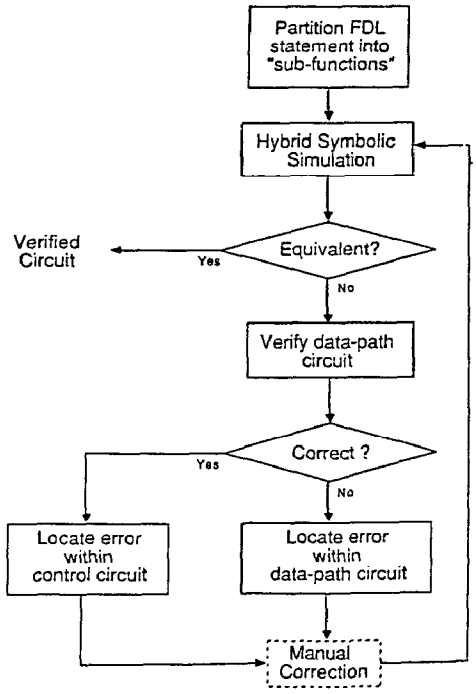

Figure 4: Approach for Locating Functional Errors

Figure 4 shows the general flow of the method that will indicate which sub-functions of a statement in a functional-level description are inconsistent with the circuit and which parts of the circuit contain the errors. After it has been determined that an error exists, the combinational circuit is divided into two regions: the data-path circuit and the control circuit. The data-path circuit is defined as that portion of the combinational circuit that depends on the input data signals. In other words, it is the area that includes all the paths between the input data signals and the output. The remaining portion is referred to as the control circuit. (It should be noted that these definitions are not the usual definitions for these two terms, and in this paper they refer only to parts within a combinational circuit.)

Figure 5 shows this partioning. The triangular shape of the data-path circuit indicates that there are several input data signals, but only one output data signal (the thick lines denote data signals that may be several bits wide). The small open circles in the boundary between the data-path circuit and the control circuit denote the lines that connect these two circuits, and will be referred to as the "connection points" of the combinational circuit.

The combinational circuits could be partitioned in

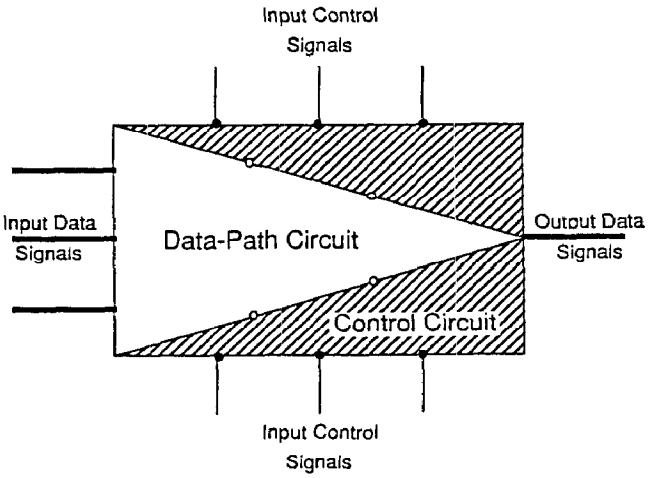

Figure 5: Division of Combinational Circuits

other ways, but the above partioning method has several advantages. As mentioned earlier, this partitioning reduces the computation time required in the verification phase. In addition, during this phase the functional-level description is partitioned into sub-functions and the conditions required for obtaining them are computed. This information is reused in the locating phase, where only the corresponding sub-circuits need to be extracted and compared with the sub-functions. This partitioning also separates naturally two types of logic primitives: gates in the data path tend to be complex function blocks such as multiplexors and adders while gates in the control circuit tend to be low-level gates such as AND's and OR's. This separation makes it easier to find and correct the logic errors.

To illustrate how the location of functional errors are found, a simple example will be described using the circuit shown in Fig. 6a. To verify this circuit, the FDL statement is first partitioned into sub-functions and the corresponding conditions for the control signal are determined as shown below:

$$Z = X \quad \text{when} \quad A = 1$$
$$Z = Y \quad \text{when} \quad A = 0$$

The circuit is then simulated with symbolic values at the X and Y inputs and logic values at the A input. The first sub-function X is verified by applying a 1 at the A input. The output of this simulation, instead of being X, is 0 and, therefore, an error exists within the circuit. To search for the error the circuit is partitioned into a data-path circuit and a control circuit. Since X and Y are the input data signals, the two AND gates and the OR gate constitute the data-path circuit (Fig. 6b). The remaining circuit shown in Fig. 6c is the control circuit.

After partitioning the circuit the general approach for finding the location of the errors is to first make a data-flow analysis to verify the data-path circuit (Fig. 4). If any logic errors exist, these are located and corrected. The whole circuit is then verified once more. If any inconsistencies are still detected, the data-path is verified once more to make sure no errors remain. If there are no logic errors in the data-path circuit, the control circuitry is then examined to locate and correct any errors that may remain. Since the corrections are manual, the whole circuit is verified once

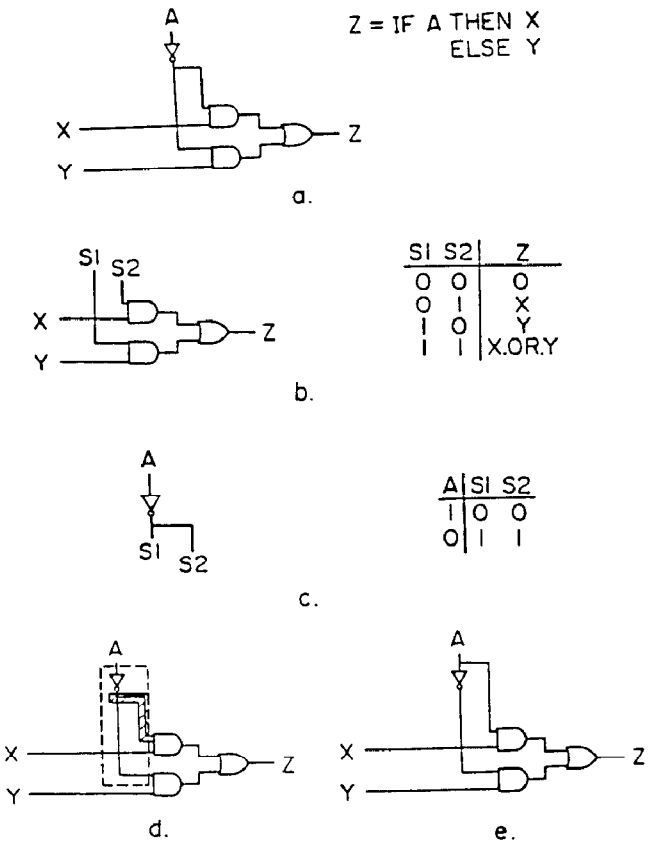

Figure 6: Example

more to make sure the circuit was properly corrected. If no inconsistencies are detected, then the final result will be a fully verified and correct circuit.

This process for locating the errors within the data-path circuit and the control circuit will be described in more detail in the next two sections.

3.1 Data-Path Circuit

The same techniques (symbolic simulation and Boolean comparison) used for verifying the entire combinational circuit are used for verifying the data-path circuit. The hybrid symbolic simulation is performed by applying Boolean values at the connection points (S1 and S2 in Fig. 6b) and signal names at the input data signals. The output (shown in the table in Fig. 6b) is all the sub-functions that can be obtained from the data-path circuit. The sub-functions (X and Y) that appear in the functional-level description are then compared with these sub-functions. If all of the sub-functions, that appear in the functional-level description, can be obtained from the data-path circuit, this implies that the logic errors are located within the control circuit. Otherwise, an error exists within the data-path circuit. (In Fig. 6b, sub-function X is obtained with S1 = 0 and S2 = 1, and sub-function Y is obtained with S1 = 1 and S2 = 0; therefore, the data-path is said to be error-free.)

To determine the location within the data-path circuit that contains the logic error, a data-flow tree is made (Fig. 7). The root represents the output, the nodes represent gates, and the leaves represent input data signals (A, B, C, D, and E in the Figure). From the sub-functions that were unobtainable from the data-path circuit, a list is made of all the input data signals that appear in these sub-functions. The error is said to be located somewhere along the paths leading from these signals to the output. If, for example, C and E are on this list, then the error would be located in the area indicated in Fig. 7.

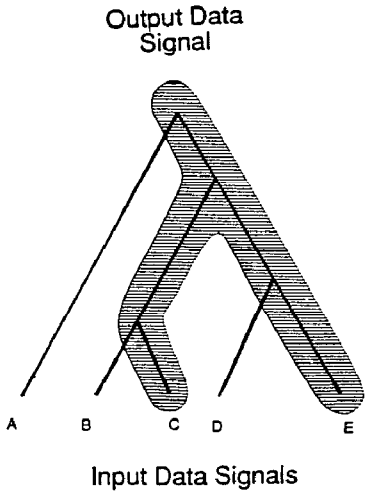

Figure 7: Data-Path Tree

3.2 Control Circuit

After the data-path circuit is found to be correct, but not the overall circuit, the control circuit is then examined. From the verification of the data-path circuit, the required values at the connection points are known for each of the sub-functions. A simulation is then performed of the control circuit by applying at the control inputs the conditions that are necessary to obtain each of the sub-functions. The outputs at the connection points are then checked. For each of the connection points that give an inconsistent output, a list is made of all the gates that have outputs that lead to those points. If any of the gates have outputs leading to any of the correct connection points, then they are eliminated from the list. The error is, thus, said to be located somewhere in the gates that appear on the list.

For the example in Figure 6, the data-path circuit was shown in section 3.1 to be correct. From the FDL description in Fig. 6a we know that sub-functions X and Y are obtained when A is 1 and 0, respectively. We also know from the table in Fig. 6b that these sub-functions can be obtained from the data-path circuit by having S1 = 0, S2 = 1 and S1 = 1, S2 = 0, respectively. Putting this together means that when A = 1, the expected function of the circuit is X, and, therefore S1 must be 0 and S2 must be 1. Likewise, when A = 0, the expected function of the circuit is Y, and, therefore, the value of S1 must be 1 and S2 must be 0. Simulating the control circuit under the conditions A = 0 and A = 1, we get the results shown in the table in Fig. 6c. We immediately see that S1 is correct and S2 is not. This implies that the circuitry between the inputs and S2 (the area within the dashed-line box in Fig. 6d) is faulty. However, the fact that the values at S1 are correct implies that the path between the input A and S1 is correct. Consequently, the error is actually located within the shaded area. A properly designed circuit would either have another inverter between the inverter and the AND gate, or a direct

connection from the A input to the AND gate as shown in Fig. 6e.

The method described above for finding the location of logic errors provides sufficient conditions, but not necessary conditions, for correcting the circuit. It may be possible that a data-path circuit, even though it was shown to be correct, could be modified in such a way that it would work properly with the existing control circuit. However, if this was a possibility for a certain circuit, it is most likely that the circuit that the designer intended to build was one that would be obtained by modifying the control circuit. This is explained by the fact that designers implement the functional-level descriptions with efficient circuits that have only the essential sub-functions, and therefore, if a given combinational circuit has the required sub-functions, it is most probable that its data-path circuit is correct and an error is located in the control circuit.

4. Implementation and Experimental Results

A system, called *Cóndor*, based upon the method described in this paper was implemented on an NEC EWS4800 workstation, a 68020-based Unix machine. The inputs to this system are the FDL description, the net-list of the gate-level implementation, and the library of primitives used by the symbolic simulator. The net-list extraction program and the translation program that converts the FDL descriptions into a LISP-readable format were written in C (6K lines). The verification and error locating programs were written in LISP (3K lines).

```
(REG (!  'MATON 0 1)
     ((!-OUT 'MATON 0 1 '+)
      (F674 '(2 M215 9 1)
          1
          (F091 '(2 M201 7 1)
              1)
          (!-IN 'NNSNOT 0 1 '+)
          (!-IN 'BCK 0 1 '-)
          (F101 '(2 M205 3 1)
              1
              (!-IN 'TMRST 0 1 '+)...))))
```

Figure 8: Extracted Circuit Function

The net-list information of the logic circuit is extracted from the output data files of an internally-developed schematic-capture program. Included in this program is a function that checks the logic circuit for any syntactic errors, such as shorted lines, unconnected gate inputs, and input/output signals without labels. Once a circuit passes this check, it can be assumed that only logic errors remain in the circuit.

From the net-list, a LISP function describing the behavior of the circuit is extracted. Figure 8 shows part of the function that would be extracted from the circuit shown in Fig. 2. This list is essentially a tree in which the root is the function for the gate at the output, in this case register F674, and the inputs to this function are the functions for other gates or input signals connected to the inputs. The definition of each gate function is kept in a library. To simulate the circuit, values (symbolic or boolean) are assigned to the input-signal variables and the function is evaluated. The result will be the output of the circuit.

The circuit in Fig. 2 actually contains an error. Figure 11 shows what the output of the *Cóndor* System looks like for this case. The output shows the result for the sub-function **NNSNOT**. An inconsistency was detected and the error was found to be in the control circuit. The input S2 of the Register was found to be incorrect, and therefore, all the gates connected to this input were listed as being in the area where the error is located. However, the input S0 of the register was found to be correct. Consequently, those gates connected to this input are correct and those that appeared in the previous list are marked with and asterisk. Even though six gates are listed, the actual area where the error is said to be found is in the area that contains the gates M207, M208, and M211.

```
Expected circuit function is

   (! 'NNSNOT 0 1)

  when
       TMRST-0-1  is  0
       BCK-0-1  is  1
       TSMC-0-1  is  0
       TSC8EP-0-1  is  0
       MTRCRT-0-1  is  0
       MTRCST-0-1  is  1

Actual circuit function is:

   1

 =====>  Equivalent?  NO !!!!!

                    Bug in CONTROL circuit

              ------------------------------
                Page  2      Gate  M208
                Page  2      Gate  M219  *
                Page  2      Gate  M203  *
                Page  2      Gate  M206  *
                Page  2      Gate  M207
                Page  2      Gate  M211
```

Figure 9: Partial Output of the *Cóndor* System

The *Cóndor* system was tested with circuits that were actually used in CMOS gate-arrays. A fault was introduced artificially into the circuits by randomly adding one inverter. Table 1 shows the computation times that were required for verifying these faulty circuits and for locating the fault areas that contained the extra inverter. The verification times indicate the amount of time that was required in the verification phase. This means that if the circuits had not contained any errors, this would be the total time that would be required for verifying the circuits. The location time indicates the additional time that was required for determining the area where the extra inverter was located. For the largest circuit, *dagc*, it took a total of 89 seconds to verify and to locate the error within an area that contained 5 primitive blocks.

Since each FDL statement is processed separately, the computation time grows linearly with the number of statements. However, the total computation time is heavily dependent on how the circuit is described in FDL. For example, by increasing the number of statements used to describe the same circuit, the computation time is generally decreased. The number of statements can be increased by using **TERMINAL** statements that describe internal signals not connected to registers. If each statement describes a smaller circuit, then fewer variables appear in the Boolean functions

Table 1: Verification and Fault Location Times

Circuit	FDL Statmnt.	Gates †	Primitive Blocks	Blocks in Fault Area	CPU Time ‡	
					Verification	Location
clk	2	112	73	2	1 sec	1 sec
erst	2	133	29	2	1	1
atmc	7	59	35	3	4	1
dbcc	21	128	28	1	9	2
brcc	31	406	62	3	18	3
dnr	12	513	105	3	26	2
dagc	51	3455	627	5	87	2

† Equiv. two-input nand gates ‡ On a EWS4800/50

that are to be compared. Since the computation time of this comparison is highly dependent on the number of variables, and since the total computation time is dominated by these comparisons, the total computation time is reduced.

One important advantage of the method described in this paper is that no assumptions need to be made as to the type or number of functional errors that may exist within the circuit. There are many types of errors that may occur during the design of the circuit. For example, the type of gate may be wrong, gates might be missing or superfluous, wires may be exchanged, etc. But because of the fact that *sub-functions* are extracted from the circuit and compared to those in the functional-level description using formal verification techniques, it is guaranteed that all these errors will be detected and located. Experiments were done with these type of errors, and it was found that the type of error has an effect only on the size of area specified to contain the error. If multiple errors occur simultaneously, multiple areas are specified or only one large area containing all the errors is specified.

5. Summary and Conclusions

In the design of logic circuits, the time spent debugging the circuit is comparable or greater than the time spent in the actual design of the circuit. The purpose of the method presented in this paper is to reduce the debugging time by finding the locations of the logic errors. The method is *deterministic* in the sense that all errors are detected and the circuit can be corrected if the indicated areas are modified appropriately. The indicated areas were shown to have sufficient resolution to allow the designer to quickly find the cause of the inconsistency. In addition, information such as the actual and expected input/output relationship of the faulty area can facilitate the fixing of the circuit. The computation time required during the verification phase was reduced by partitioning the functional-level into sub-functions. This same partitioning is used for locating the logic errors. The big advantage of this debugging approach is that no input patterns need to be specified by the user.

The method described in this paper exploits the `IF-THEN-ELSE` and `CASE-OF` structures for verification and locating errors. This method can also be applied to other functional-level description languages, such as DDL and CDL [DULE68, CHU65], that have these structures explicitly or implicitly. For example in DDL, for each terminal and memory element all the "connections" and "transfers" as well as their conditions can be gathered in a table. The signals appearing in the conditions would be the control signals and the expression describing what is transferred would be the sub-functions. It is necessary to have, however, Boolean declarations defining all the states.

In this paper, it was assumed that the description at the functional level was correct and the errors were in the circuit. This assumption is really not necessary in the sense that the results will only indicate which parts of the functional-level description are inconsistent with specific regions in the circuit. In other words, if a circuit is known to be correct then the errors in the functional-level description can be located, as it would be necessary in a bottom-up design.

A possibility exists for extending this research to actually correcting the circuit. For some simple circuits, particularly in the control circuitry, this may not be too difficult. If a solution is not found, then a two-level circuit could be synthesized since sufficient information can be obtained about the input-output relations of the faulty areas. However, this approach may not be of much help for errors within the data-path where high-level primitives such as multiplexors and adders are frequently used. For such cases the issues involved are similar to those encountered in logic synthesis.

6. Acknowledgements

The author wishes to thank Dr. Satoshi Goto, Mr. Yoshihiro Nagai, Mr. Takeshi Yoshimura and Mr. Tomoyuki Fujita for their helpful comments and suggestions. The author would also like to thank Mr. Hidetoshi Tanaka and Mr. Hiroshi Ichiryu, both from the Computer Engineering Division, for providing the test circuits.

7. References

[ABAD88] M. S. Abadir, J. Ferguson, T. E. Kirkland, "Logic Design Verification Via Test Generation," *IEEE Trans. Computer-Aided Design*, vol. 7, no. 1, pp. 138–148, Jan. 1988.

[AKER80] S. B. Akers, "A Procedure for Functional Design Verification," *Proc. 10th International Symposium on Fault-Tolerant Computing*, pp. 65-67, 1980.

[BARR84] H. G. Barrow, "Verify: A Program for Proving Correctness of Digital Hardware Designs," *Artificial Intelligence*, vol. 24, pp. 437-491, Dec. 1984.

[BRYA86] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.

[CHU65] Y. Chu, "An ALGOL-like Computer Design Language," *Commun. ACM*, vol. 8, no. 10, pp. 607-615, Oct. 1965.

[DALE68] J. R. Duley and D. L. Dietmeyer, "A Digital System Design Language (DDL)," *IEEE Trans. Computers*, vol. C-17, no. 9, pp. 850-861, Sept. 1968.

[GARE79] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1979.

[HACH88] G. D. Hachtel and R. M. Jacoby, "Verification Algorithms for VLSI Synthesis," *IEEE Trans. Computer-Aided Design*, vol. 7, no. 5, pp. 616–640, May 1988.

[KATO83] S. Kato and T. Sasaki, "FDL: A Structural Behavior Description Language," *CHDL 83*, pp. 137–152, 1983.

[ODAW86] G. Odawara, M. Tomita, O. Okuzawa, T. Ohta, and Z. Zhuang, "A Logic Verifier Based on Boolean Comparison," *Proc. 23rd DAC*, pp. 208–214, 1986.

[SASA84] T. Sasaki, S. Kato, N. Nomizu, and H. Tanaka, "Logic Design Verification Using Automated Test Generation," *Proc. 1984 International Test Conference*, pp. 88-94, 1984.

[SMIT82] G. L. Smith, R. J. Bahnsen, and H. Halliwell, "Boolean Comparison of Hardware and Flowcharts," *IBM J. Res. Dev.*, vol. 26, no. 1, pp. 106–116, Jan. 1982.

[SRIN88] N. C. Srinivas and V. D. Agrawal, "Formal Verification of Digital Circuits Using Hybrid Simulation," *IEEE Circuits Device Mag.*, pp. 19–27, Jan. 1988.