Modeling of VHDL Design Errors and Methods for their Correctability

Mohammad R. Movahedin+[1], Peter Kindsmüller*, Walter Stechele*
+Electrical and Computer Engineering Department
Faculty of Engineering, University of Tehran
North Karegar, 14399 Tehran, Iran
*Institute for Integrated Circuits
Technical University of Munich
Arcisstr. 21, D80290 Munich, Germany

Abstract

Design Errors are all of the non-equalities between the desired functionality and the implemented design. They lead in many cases to a new IC fabrication that means higher NRE (Non Recurring Engineering) costs and delay in the time to market.

This paper deals particularly with VHDL Design Errors. They are classified and modeled, and a new general methodology for correctability of design errors (i.e. the ability to correct them even after the fabrication) is presented. Afterwards, this method is applied for a subset of error groups and discussed in detail. Some guidelines for making a good and reasonable trade-off between the correctability and its resultant overheads (such as area and speed) are presented too.

1) Motivation

Nowadays, considering the high complexity of the available ASICs, including Gate Arrays and standard cell based ICs, the designers are obliged to develope and specify their designs no longer in conventional schematic methods, but in a high level hardware description language (HDL) like VHDL or Verilog-HDL and generate the required net-list automatically by HDL synthesizers.

Meanwhile, Design Errors, that are mistakes of a designer made during the specification or development of a design, are always problematic and must be taken into account. These errors always lead to non-equalities between the desired functionality and the fabricated circuit and in many cases the flawed circuit is not usable in the system. Accordingly, in order to avoid causes of these errors, each design team must develope its individual design strategy so that they would not appear at all. This strategy may include specific styles of code writing, restrictions on the complexity of each module, or even some human relevant rules such as limited working hours. Furthermore, there exist many opportunities to check the functionality of a description and its compatibility to the desired specification, like computer based or hardware accelerated simulation and emulation. Nevertheless, statistics show that particularly in HDL descriptions, one design error (bug) per 125 to 10000 code lines, based on the spent costs and the quality of the designer, is a very common fact [1]. Assuming each code line, if the design is described in the RT (Register Transfer) level, translates averagely to 10 physical gates [1], one can hardly claim that a design with more than 100K gates is absolutely free of any design errors.

The doubt about error freeness of a design is a result of various facts:

---

1. Currently works at the Institute for Integrated Circuits, TU Munich, supported by the German Academic Exchange Services (DAAD).

- Simulation of a design description, either in RT or in gate level, is a very time consuming task, so that it is practically impossible to try all available states of a complex ASIC in a reasonable time, though there are methods such as hardware accelerated simulation [2] or logic emulation [3] in order to speed up the verification phases. An uncompleted simulation that has not considered carefully each possible state, means the fact that the not verified states are potentially faulty.
- A huge amount of data is used and produced for the verification and specially simulation of an ASIC. The evaluation of these data and the achievement of an abstract result, that is whether a specific HDL description does or does not match the desired functionality (i.e. go/no-go decision), is usually done at a relative low level (e.g. text or waveform), and it is the duty of the designer to check the simulation results manually and decide if the behavior of the data meets the appropriate specification. Hence, design errors may be overlooked easily.
- Last but not least important is the fact, that the specifications are very often written and converted to a HDL description manually in an informal manner. Therefore if the specification itself contains an error or does not consider some seldom happening situations, detection of these kinds of faults is possible only after the fabrication and system integration phase.

After detection of an error that can not be compensated by a workaround [4] (e.g. software or firmware changes), there exist some very restricted physical methods to correct the design bugs [5], [6]. However, in many cases, these errors lead to a re-fabrication of the corrected design, which results in additional NRE costs and delays the launch date of the system.

Design errors in the logic level description and implementation are considered previously [7], [8], and some methods for their correction are already available [9], [10], but this paper deals with design errors that appear particularly in VHDL description of an ASIC design, followed by logic synthesis.

Next section introduces a new design methodology that enables the designer to correct design errors even after the fabrication. The correctability presented in that section (i.e. second section) strongly depends on the different possible error classes in a VHDL description that are presented deeply in the third one. The subsequent section deals with the application of the design error tolerant methods to make a design robust against some groups of faults.

2) Design Error Tolerant Methodology

Assuming that the design is described and specified at the RT level by VHDL, the goal of this new methodology is to achieve the ability to correct design errors without the need of re-fabrication (= correctability). This is realized through proper usage and fitting of Programmable Logic Elements (PLEs) distributed in the gate level implementation of a design. Fig. 1 shows the ASIC design flow of this new design error tolerant approach.

The original VHDL description, which may contain a design error, is read by a special VHDL compiler belonging to the synthesis tool. This VHDL compiler produces a logic level structure that is a mixture of conventional standard cell library elements and PLEs. The usage and nature of PLEs depends on defined VHDL error classes (described in the next section) that must be specified first by the designer to be correctable.

Examples for PLEs are lookup tables (LUT), which are widely used in Field Programmable Gate Arrays (FPGA) [11], and programmable operators (e.g. switchable between four comparators: <, <=, >, >=). The incorporation of PLEs into a gate level circuit consisting of conventional standard cell library elements *must* be likewise considered in all subsequent steps (e.g. logic optimization, technology mapping) within the synthesis tool. Parallel to the gate level circuit, the synthesis tool produces necessary programming data that must be loaded into the fabricated circuit by the user in order to achieve the desired functionality.

Consequently, if a design error is detected in the system integration phase, changes can be applied without any new expensive fabrication. New necessary programming data for the PLEs in the ASIC is produced by the synthesis tool according to the changes in the VHDL description, provided that the changes belong to the same VHDL error class(es) that must be defined to be correctable before the first synthesis step. After the new programming data is loaded into the ASIC, it performs the functionality as specified in the changed VHDL description.

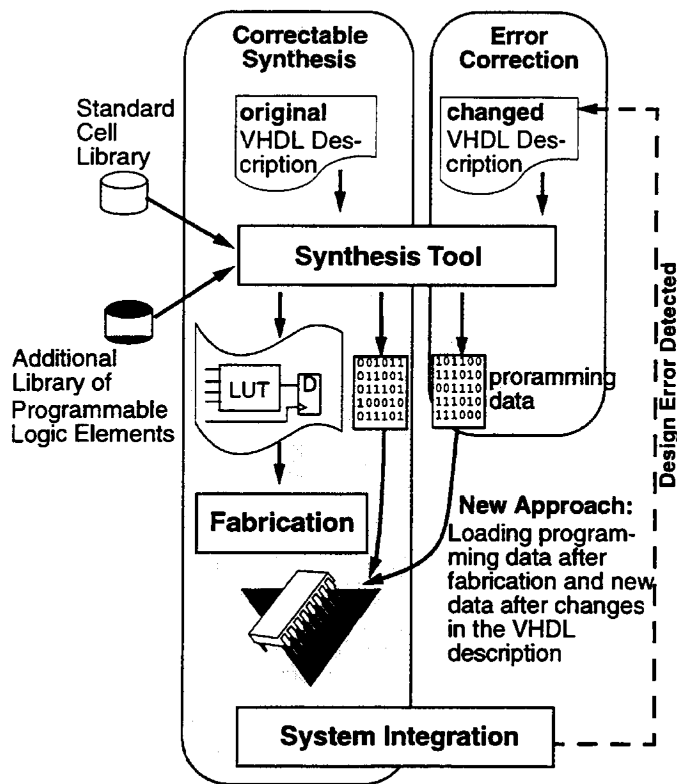

Figure 1: Design Error Tolerant Design Flow

3) Modelling and Classification of VHDL Design Errors

As mentioned before, it is assumed that a design is described by VHDL and a synthesis tool is used to generate a net-list in the gate level. The VHDL synthesizer should not make any extra scheduling between the clock periods of the design, and it needs the designer to describe the circuit in the RT level.

This section deals with classifying all of the errors that can occur in a VHDL description. In some cases, the concept of the Behavioral Fault Modelling (BFM) is used to promote this classification [12], [13], [14], but the model described here is quite different from the BFM, considering their dissimilarities in their natures. The BFM is a representation of fabrication failings in the abstracted high level delineation, while the following one models the mistakes, made by designers in the circuit developing stages.

A *VHDL Design Error* is defined as a deviation in the source code of the circuit description from the right form of the desired specification that does lead to a different functionality of the circuit after its synthesis. It can be an *exchange, absence, extra existence* or *misplacement* of some names (like variable's or signal's), keywords and/or other language elements (such as operators). These changes neither cause any syntax errors in the applied subsequent compilation, nor make the design non-synthesizable, based on the VHDL subset in use.

Since the compilers, and particularly VHDL synthesis tools translate the design after the compilation of the entity declaration and other declarative parts, it is assumed here that no fault exists in these portions in any way. This assumption is a very important limitation which means that the design changes caused by errors in declaration parts are not studied by this model at all. Examples of this sort of errors can lead to a great change in the size of circuit busses, the number and size of the registers, differences in the input and output ports or even changes in the nature of the instantiated or called sub-modules. The reason of not investigating these errors is almost clear: the errors in declarative parts often lead to redesign and the probability of their correction without any re-fabrication is small. Besides, we understand implicitly from *design errors* only some *small* changes in the design.

The outcome of this presumption is that one can concentrate only on errors in the *concurrent* and *sequential* statement section of a VHDL description, which are classified in the following six categories. Each category has a relevant example (apart from the last one, which is clear enough) showing the errors between V_0 and V_F, correct and faulty codes respectively.

1. Errors in Signal and/or Variable Names: If the name of a signal or a variable is false in the faulty VHDL code and must be changed to another signal or variable name, definitely based on its declaration and without causing any syntax error, the VHDL code has a fault in its signal and/or variable names. This exchange can occur just directly, or indirectly in various ways, such as a fault in array indexes or ranges, record fields or when using array aggregates. These are shown in the different parts of the Example 1 successively. Since usage of parentheses defines new signals and/or variables implicitly, any changes in them can be modelled as an error in the signal names too. For instance, the last two parts of the above pointed example, show the implied by parentheses variables explicitly. It demonstrates how the misplacements of the parentheses can be modelled by a group of faults in the temporary defined signals. Though, the other form of modeling of this misplacement is discussed later.

Example 1: V_0: V_F:

V_0	V_F
`... X <= A; ...`	`... X <= B; ...`
`... C( 5 ) ...`	`... C( 8 ) ...`
`... D(7 DOWNTO 0) ...`	`... D(15 DOWNTO 8) ...`
`...F_FIELD.RECORD_1...`	`...F_FIELD.RECORD_2...`
`...E_VECTOR(2 => '1'...`	`...E_VECTOR(3 => '1'...`
`g<= x AND (y OR z);`	`g<= (x AND y) OR z;`
`....`	`....`
`t <= y OR z ;`	`g <= t OR z ;`
`g <= x AND t ;`	`t <= x AND y ;`

2. Errors in Constants: Each VHDL code usually contains some constants. False constants are considered as an error class. This can be in integer or bit (generally numeric) literals used directly in the code or declared CONSTANT, and indirectly through attributes or user defined enumeration types.

Example 2:

V_0	V_F
`CONSTANT C:BIT_VECTOR:="101100";`	`CONSTANT C:BIT_VECTOR:="110100";`
`... B:= B + 4; ...`	`... B:= B + 9; ...`
`... defined_type'LEFT ...`	`... defined_type'RIGHT ...`
`... defined_state_0 ...`	`... defined_state_1 ...`

3. Operator Errors: Each interchange of operators forms an operator error. Generally, this exchange must be between the operators of a common group, like logic (AND, OR, XOR, NAND, NOR, XNOR) or relational operators. Since the meaning of the operators can be overloaded in VHDL, the presumption about operator groups can be changed too. It means one can simply gather logical and arithmetic operators on vectored signals together. However, some operators can not be grouped with any other ones in any way, like the concatenation operator (&) considering the dissimilarity in the word length of two concatenated signals and the result of any other operator on them. To recover this difference and allowing change of the concatenation operator with any other one, it needs a change in the declaration parts, which is forbidden in the assumptions discussed before.

4. Errors in Assignments: An additional assignment, absence of a necessary one, or false or uncompleted assignments are three different types of faults in assignments.

5. Conditional Statement Faults: Sequential conditional expressions, such as expressions using IF THEN ELSE or CASE WHEN statements (**if_statement** and **case_statement**), or concurrent conditional or selected signal assignments allow the designer to control the data-flow of the design. Controlling an expression with an unneeded condition or absence of a necessary control for an expression, while the condition expression itself exists somewhere in the description, or forgetting of having a needed conditional expression at all, are considered as different conditional statement errors.

6. Errors in Component Instantiating: Faults in the name of an instantiated component, in ports binding or in passed generic parameters are three different faults grouped in this title. It must be mentioned that if one generalizes the

Example 3:

V_0	V_F
`... a XOR b ...`	`... a OR b ...`
`... ( C >= D ) ...`	`... ( C > D ) ...`
`... ( E + F ) ...`	`... ( E - F ) ...`
`... ( G + H ) ...`	`... -( G + H ) ...`
`... K MOD L ...`	`... K REM L ...`
`... x AND y ...`	`... x AND NOT y ...`

Example 4:

V_0	V_F
`----`	`A<= B AND C ;`
`D<=E + F ;`	`----`
`G<=(K XOR L);`	`G<=(K XOR L)OR M;`

Example 5: V_0: V_F:

```
V0:                          VF:
IF (condition) THEN          IF (condition) THEN
  statements1                statements1
  ---                        statements3
ELSE                         ELSE
  statements2                statements2
END IF;                      END IF;
statements3                  ---

IF (condition) THEN          IF (condition) THEN
  statements1                  statements1
  statements3                  ---
ELSE                         ELSE
  statements2                  statements2
END IF;                      END IF;
---                          statements3

IF (condition) THEN          ---
  statements1                statements1
ELSE                         statements2
  statements2                ---
END IF;                      ---
```

faults in the signal and variable names to all of the identifiers, no matter whether they are a signal's or another object's, the faults in the names of instantiated components can be grouped with them too. But they are characterized separately, because almost all identifiers are used in the declaration parts, which are assumed to be without any fault in this modelling.

To complete the classification of the VHDL Design Errors, other important points must be thought over too:

- Some designers' mistakes can be described by a set of faults, instead of a singular one. The parentheses misplacement in the Example 1 is a typical sample which is classified with four faults in the signal names. (Exchange of the t, y, g, t to the g, t, t, y respectively)

  Sometimes this classification of these mistakes can be done with multiple sets of faults instead of a unique one, which lead necessarily to an identical circuit after the synthesis. For example, the previously mentioned parentheses misplacement can be represented by two errors in the signal names and two ones in the operators, too. (Exchange of the OR, z, AND, x to AND, x, OR, z, when comparing the assignments with the similar targets)

- Some changes in the source code of the VHDL description that produce a faulty circuit individually after its synthesis can neutralize the effect of each other when they are grouped and appeared together. Definitely, this group of individual changes is not a design error, considering that they do not lead to any undesired functionality in the design. Example 6 shows five changes in the V_0, making the absolutely equivalent description as V_1, even though these changes are five different faults, if they appear lonely.

Example 6: V_0: / V_1:

V_0:	V_1:
`IF a AND b THEN`	`IF` **`NOT`** `a` **`OR NOT`** `b THEN`
`  statements1`	**`  statements2`**
`ELSE`	`ELSE`
`  statements2`	**`  statements1`**
`END IF;`	`END IF;`

- As shown previously in the parentheses misplacement case, many designers' mistakes must be classified indirectly, particularly faults in procedures, generated statements, loop expressions, etc. In some cases, one simple fault can result a long set of different ones. For example, an operator error in a function that is used many times is not *only one* single fault in it, but *some* faults propagated whenever the function used. In the following example, a fault in the binding of an instantiated component is repeated 16 times, in account of the GENERATE statement. Therefore, this error must not be considered as only a simple one, but sixteen design errors.

Example 7:

V_0:	V_F:
`label_1:FOR i IN 0 TO 15`	`label_1:FOR i IN 0 TO 15`
`GENERATE`	`GENERATE`
`  label_2:ANY_COMPONENT`	`  label_2:ANY_COMPONENT`
`    PORT MAP(A(i),B(i),C(i));`	`    PORT MAP(A(i),B(i),`**`D`**`(i));`
`END GENERATE;`	`END GENERATE;`

4) Methods for Correctability of Design Errors

For investigating different aspects of programmable implementation of designs, an experimental compiler is developed that produces programmable circuit, robust against defined groups of design errors. The synthesized design needs some programming informations to operate equivalent to the original VHDL code, or the code after some variations. Thus, a proper amount of Programming Memory Cells (PMC) holding these informations are distributed overall the design and must be loaded with the relevant data at the beginning of the circuit operation. This is exactly like the operation of the SRAM based FPGAs, but the method of the design programmability is totally different: using PLEs like configurable operators and LUTs, without having any programmable interconnection resources.

Specification and operation of these PMCs are quite identic to the Configuration Memory Cells used in the SRAM based FPGAs [11]. Therefore, the experiences of their design can help well for an optimal development of the PMCs in an ASIC. Besides, these cells must match other specifications, particularly geometrical ones of the ASIC library in use, so that the other necessary steps of the design, like Place

and Route or timing back annotations can be done regularly without any change in their algorithms or tools. Otherwise, along with the synthesizer, other afterward necessary tools must be designed newly in addition, which means a high expenditure. However, the exact definition of PMC structure and their development for an ASIC library as an example is not yet done.

The behavior of the two parts of the compiler is being discussed in the following two subsections. The current compiler accepts only sequentially described designs in VHDL, which is used widely in the RT level description and synthesis, but its enhancement from pure sequential description to a mixture of sequential and concurrent statements is not an important issue in this stage.

Design Error Tolerant Implementation of Expressions

Each conditional statement has a boolean expression whose trueness or falseness controls the flow of data. This expression, with a considerable role in the circuit, might be faulty and hence must be implemented appropriately programmable.

Primarily, the below shown brief BNF (Backus-Naur Form) is used to describe the boolean expressions [15]. This BNF is selected such as to cover the major parts of the practically used subset of VHDL expressions. The quite specific forms of the boolean expressions that infer some flip-flops in the synthesized design (using 'EVENT or 'STABLE attributes) are behind of this short BNF. A design error in these forms either is not synthesizable and therefore is not imaginable as a design error (like using `clk OR clk'EVENT` instead of `clk AND clk'EVENT`), or causes some great changes in the design, like inferring fewer or more than required flip-flops, changing the polarity of their clocks or even defining some new ones, that are not considered yet by the compiler and hence are not repairable now.

```
expression ::= relation {logic_operator_1 relation} | relation [logic_operator_2 relation]
logic_operator_1 ::= AND | OR | XOR
logic_operator_2 ::= NAND | NOR | XNOR
relation ::= simple_expression [ relational_operator simple_expression ]
relational_operator ::= > | < | = | /= | >= | <=
simple_expression ::= term { adding_operator term}
adding_operator ::= + | -
term ::= factor
factor ::= primary | NOT primary
primary ::= signal_name | constant | ( expression )
```

Further, with the following techniques, the synthesized expression is made robust against the relevant faults.

- Each **logic_operator** is implemented using LUTs instead of the normal inflexible gates. The maximum number of the LUT inputs is limited to a fixed number N (investigations towards the optimal value for N were made in [16]). If a **logic_operator** functions on more than N signals, a group of together connected LUTs is used. Using LUTs can not only make the design tolerant against all of the logic operation faults in the operator errors class, but also can implement some more functions, which are caused by other faults, i.e. false assignments. For example, using an LUT with 2 inputs for the synthesis of `(A AND B)`, can implement any new circuit with directly exchange of the **logic_operator,** like `(A OR B)`, `(A XOR B)`, `(A NAND B)`, `(A NOR B)`, `(A XNOR B)`, and some other functions as well, like `(A)`, `(NOT A)`, `(B)`, `(NOT B)`, `('0')`, `('1')`, `(A AND NOT B)`, `(NOT A AND B)`, `(A OR NOT B)` and `(NOT A OR B)`.

  An LUT with n inputs needs 2^n PMCs and can be simply implemented using a 2^n to 1 multiplexer with n control lines as LUT inputs [11]. Special and optimal development of 2, 3,... N input LUTs as complementary to the ASIC library, can increase the speed and decrease the area used by them, though the above mentioned structure of multiplexers is always usable in each ASIC library.

- A general comparator, which produces six different relational functions based on its three programming bits (controlled by three PMCs) is used to implement each **relational_operator**. With this technique, the circuit will tolerate all errors in the **relational_operator**.

- With passing each signal through a controlled inverting module, like an XOR gate whose one input is connected to a PMC, each **adding_operator** and inverter is implemented programmable. Controlling of the carry input of adders is necessary too. Principally these controlled inverting modules make adders function quite generally, adder or subtracter and therefore, each error in **adding_operator** and inverter (absence or extra existence of the NOT operator) is further repairable.

This module does not need to be implemented necessarily with XOR gates, despite that is the sole solution in many currently available ASIC libraries, because of two reasons. First, the difference of required timing restrictions between inputs and outputs. The delay between the controlled input and the output is not essential, because it changes so rarely (only when some new programming data should be loaded), but the other one is important and influences the operation of the whole system. Second, existence of the inverted version of the control line in the relevant PMC, which makes the implementation of the XOR function far easier. This fact can result in a very good efficiency when a number of controlled inverters are necessary and must be controlled with only one PMC. It means that one can develope some extra library elements, combining a PMC and a simplified XOR, or a PMC and an array of simple XOR gates for different vector signals, with an optimal area and speed.

- In order to prevent any faults in constants, each of them is synthesized using proper amount of PMCs. These PMCs can have any arbitrary value (within the same word length), based on the changes made by the designer in the VHDL source code.

  Unfortunately this type of implementation of constants has an absolutely huge overheads, because of the fact, that a circuit after the logic optimization does not usually have any cell holding constant values. It means that all constants, namely connections to power supply or ground, are absorbed by other gates during the logic minimization. Therefore, when the constants are implemented using this method, not only some extra cells (PMCs) are used to hold their values, but also any further area reduction by logic minimization is prevented too. For instance, a circuit that checks if a 32 bits wide signal is equal to a specific constant value (synthesized circuit of a VHDL code like: `A(31 DOWNTO 0)=(OTHERS => '0')`), needs approximately 5 times less gates and area than a circuit that checks if two 32 bits wide signals have equal values (VHDL code like: `A(31 DOWNTO 0)=B(31 DOWNTO 0)`). This is because the former is a simple function between 32 single bit signals that can be implemented easily using some AND and inverter gates (the nature of this function depends on the value of the constant side, but it is always an AND between the signals compared to '1' and the inverted value of the ones compared to '0'), but the later needs 32 XNOR gates to compare each individual signal first, and then an AND function between all of their outcomes.

Design Error Tolerant Implementation of Assignment Controls

A sequential description, which is defined by a **process** keyword, with considering some restrictions explained later, can be seen as a tree of conditions, made by a group of **if_statements** and/or **case_statements**, and some assignments in each conditional branch. This is valid if the process does not have any **wait_statement**, and other sequential statements (such as **procedure_call_statements** and **loop_statements**) are expanded or translated to a plain sequential code. If the **process** defines only a pure combinational circuit, even with some logic loops defining latches, the description can be synthesized in the following four parts [17]:

1. A combinational circuit for each conditional expression, such as the argument of the **if_statement**. This can be synthesized robust against specific error classes using the methods presented in the previous subsection.
2. Regular combination of the conditional expressions to produce one and only one branch control signal, which controls all of the assignments and other conditional statements in each conditional branch. It can be done with two AND gates, one of whose inputs is the branch control signal of the conditional branch where the **if_statement** appears, and the other inputs are the value and the inverted value of the conditional expression output, building the branch control signal in the THEN and ELSE parts respectively. The first **if_statement** is controlled by an always '1' signal.
3. Combinational circuit necessary for the implementation of the right hand side of each assignment. The difference between signal and variable assignments and the role of the duplicated sequential assignments to a same target, which causes an overriding to the previous values (specifically the concept of the temporary and short-time variables), must be considered carefully in the synthesis of this part.
4. Some multiplexers, whose inputs come from the logic of the previous part (implementing the right hand side of the assignments), controlled by some signals from the second part, driving the common target of a group of assignments with the identic left hand side. The control lines of the multi-

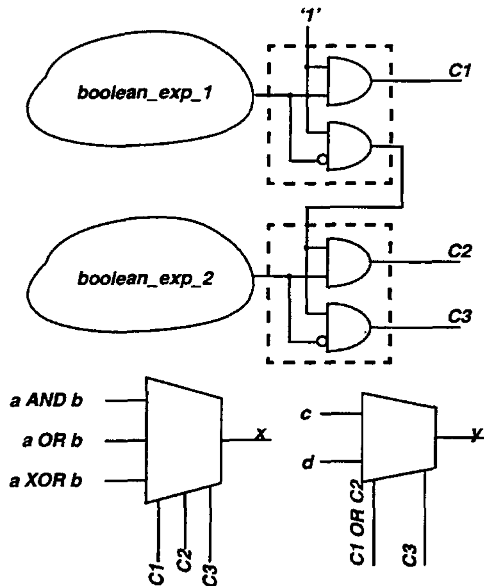

Figure 2: Four synthesized circuit parts of Example 8

```
PROCESS (a,b,c,d)
BEGIN
   IF boolean_exp_1 THEN
      -- this branch is
      -- controlled by C1
      x <= a AND b;
      y <= c;
   ELSIF boolean_exp_2 THEN
      -- this branch is
      -- controlled by C2
      x <= a OR b;
      y <= c;
   ELSE
      -- this branch is
      -- controlled by C3
      x <= a XOR b;
      y <= d;
   END IF;
END PROCESS;
```

Example 8: A simple sequential code

plexers are always either one of the branch control lines directly or the output of a multi-input OR gate that operates on a selected set of them.

These four parts for VHDL code of the Example 8 are shown in the Fig. 2.

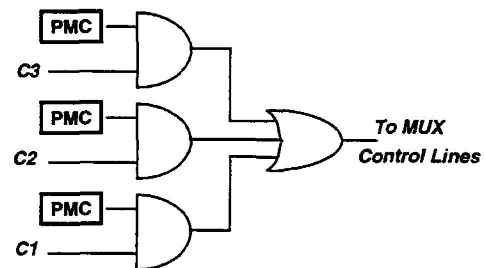

Figure 3: Controlled multi-input OR gate

In order to make the design robust against all of the designers' faults, which does not effect any change in the multiplexers and their inputs (3rd and 4th parts above), or saying more clearly, to make the assignments not to matter in which conditional branch appear (if the assignment does not have any temporary variable), a controlled multi-input OR gate is used between the branch control signals and the multiplexer control lines. It is a normal multi-input OR gate, whose inputs are a selection out of all of the branch control signals (C1, C2 and C3 in the above example). Some PMCs are needed to pick necessary control lines for each multiplexers control inputs. This structure is shown in Fig. 3. It is far better to encode the multiplexer control lines to reduce the number of them, and subsequently the number of the required PMCs. This encoding can simply use binary codes. The developed compiler benefits from this reduction, and for instance, generates for the multiplexers (driving X and Y signals) of the above example only 2 and 1 control lines, instead of 3 and 2, respectively. This does not influence the structure of the multi-input OR gate and the controlling PMCs.

Usage of this method of programmability in the multiplexer control lines can make two groups of faults in the VHDL code of the Example 8 repairable: 1- any exchange between AND, OR and XOR **logic_operator**, but not if one is changed to another operator, like NAND. 2- Y can be driven by C or D, no matter in which conditional branch, but not by other available sources like A or B.

If the compiler can somehow predict which alternatives, apart from the ones appeared in the original VHDL, are possible and probable for a target, it would be able to implement the needed circuits in the multiplexers part of the design so as to use them in the required cases, after some changes made by the designer. For instance, if the designer indicates the compiler, or the compiler guesses itself, that the Y in the previous example can be driven by A or B too, despite that it is not the fact now, it can implement a 4 to 1 multiplexer to drive Y from A,B,C and D, instead of the current one with only two inputs.

Furthermore, if this programmable OR function is used in the input of the two AND gates, associated to each conditional expression logic, one can make arrangements for the appearance of the **if_statement** in each conditional branch too. For instance, Fig. 4 shows a circuit which can implement the behavior of all of the six VHDL codes shown in the Example 9. Proper selection of the PMC values prevents

occurrence of any logical loop, which seems to appear in the circuit at the first look. (One of the two PMCs that are directly connected to the OR gates is always '1', which indicates the first boolean expression in the sequential VHDL code.)

This technique makes it possible too, to add a quite new **if_statement** block somewhere in the sequential description code, which is forgotten by the designer in the original description. For this purpose, a totally programmable circuit, e.g. some together connected LUTs, and additionally two previously mentioned regular AND gates are necessary. The common input of these two AND gates must be controlled by only one branch control signal out of all available ones. This control is done again by a multi-input OR gate and some PMCs. Thus, the reserved programmable logic, yet without any programming data, can become a new conditional expression in the branch, whose control signal drives the two AND gates common inputs. In this case, the structure of the reserved programmable section, which implements a conditional expression circuit, and its inputs must be predicted and implemented by the designer independently. This structure is shown in Fig. 5.

The physical structure of the above frequently mentioned controlled multi-input OR gates, whose inputs are selected by PMCs out of a set of signals, is rather similar to PLAs whose transistors are not controlled by (E)EPROM or bipolar antifuses, but by some PMCs, holding the necessary informations. They can be fabricated efficiently, small in the area and acceptable in the speed, using only NMOS transistors.

Area and Speed Overheads

Each so far presented method for synthesis of correctable designs can be performed using normal ASIC library elements. Even for PMCs, one can use serially connected D-type Flip Flops. However, this direct usage of library elements has an absolutely huge area overheads for the implementation of the methods. Specific layout design of necessary PLEs, such as PMCs, controlled inverting elements, LUTs and controlled multi-input OR gates can extremely reduce this overheads.

On the other side, in the optimal design of each PLE the speed losses must be additionally taken into account. Adding new programmable elements decreases generally the speed of the design, and this is more critical for some sort of PLEs, e.g. multi-input OR gates.

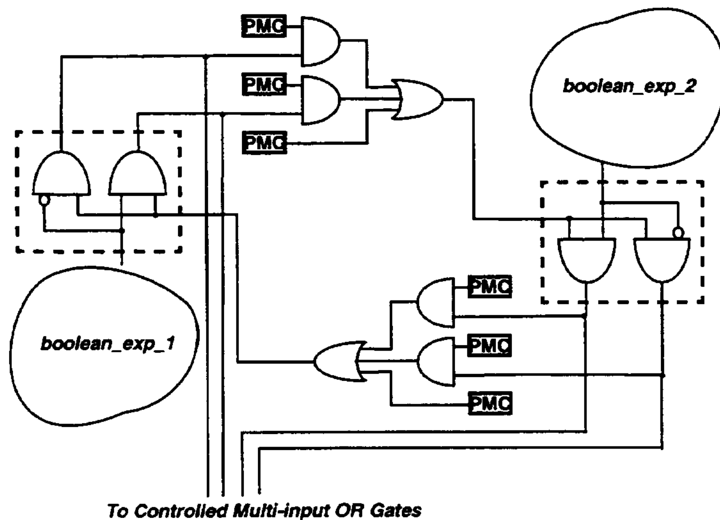

Figure 4: Controlling the if_statement appearance location

```
IF boolean_exp_1 THEN
   IF boolean_exp_2 THEN
      ...
   ELSE
      ...
   END IF;
ELSE
   ...
END IF;
------------
IF boolean_exp_2 THEN
   IF boolean_exp_1 THEN
      ...
   ELSE
      ...
   END IF;
ELSE
   ...
END IF;
------------
IF boolean_exp_1 THEN
   ...
ELSE
   IF boolean_exp_2 THEN
      ...
   ELSE
      ...
   END IF;
END IF;
------------
IF boolean_exp_2 THEN
   ...
ELSE
  IF boolean_exp_1 THEN
      ...
   ELSE
      ...
   END IF;
END IF;
------------
IF boolean_exp_1 THEN
   ...
ELSE
   ...
END IF;
------------
if boolean_exp_2 THEN
   ...
ELSE
   ...
END IF;
```

Example 9: Six different codes, that can be implemented using the circuit of Fig. 4.

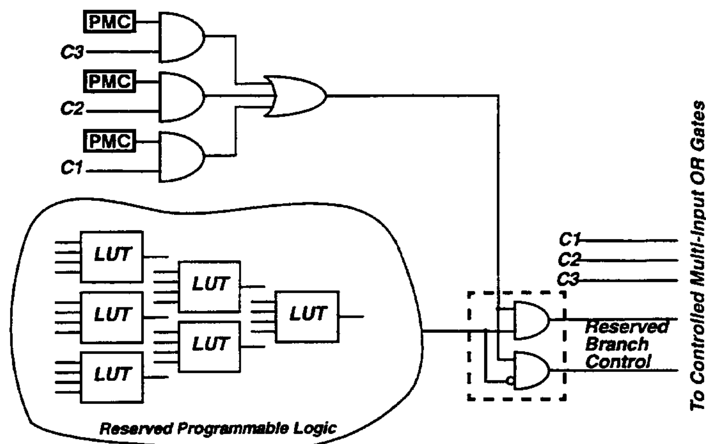

Figure 5: A structure that can implement a reserved if_statement

The speed and/or area overheads of the entire design depend highly on three major factors: First, individual overheads of each method. Second, the design error classes specified first to be correctable and third, number and type of PLEs that a design needs to be synthesized properly correctable.

5) Conclusion

Hitherto, designers have had only two alternatives for the implementation of their designs: ASICs and FPGAs. FPGAs do not have all the possibilities available by ASICs (e.g. analog components) and are very limited in capacity and speed. They offer about one-tenth and one-third to one-fifth of ASIC capacity and speed, respectively. This means, even though designers can develope their designs first using FPGAs and correct any design errors simply without spending any extra NRE costs (ASIC prototyping using FPGAs [3]), FPGAs can not achieve the required specification in many cases. ASICs on the contrary do not have currently any flexibility for the correction of design errors.

This new design error tolerant design methodology benefits from the advantages of both ASIC and FPGA: very high capacity and speed and meanwhile robust against specific classes of design errors. The merge of these two specific characteristics needs extended or even quite new synthesis tools and algorithms. In addition, specific solutions for other fault classes that can not be synthesized correctable yet (like errors in signal and variable names), considering the concept of programmable interconnections beyond PLEs and usage of other available programmable device technologies such as antifuses, (E)EPROM and laser are important points that must be studied in this regard. Finally, an accurate metric is necessary to help the designer to find an optimal trade-off between correctability and its overheads.

References

[1] High Density IC Design Technology Forum, Cadence, Nov.-Dec. 1995.
[2] R.T. Tets Maniwa, "Putting It Together: Prototyping Methods", Integrated System design, Sep. 1995, pp 18-27.
[3] P. Kindsmüller, W. Puffer, "Fast Prototyping and Emulation of ASICs using FPGAs", 5th Eurochip Workshop on VLSI Training, 1995, pp 369-374.
[4] P. Kindsmüller, "Investigations towards Avoiding and Correcting Design Errors", internal report, Institute for Integrated Circuits, TU Munich, 1994.
[5] D.C. Shaver, S.P. Doran, M. Rothschild, J.H.C. Sedlacek, "Laser-Induced Metal Deposition and Laser Cutting Techniques for Fixing IC Design Errors", SPIE vol. 1596, Metallization:Performance and Reliability Issues for VLSI and ULSI, '92, pp 46-50.
[6] R. Noone, "Modification of Logic on ASIC Devices", EuroASIC '92, pp 414-415.
[7] S. Kang, S.A. Szygenda, "New Design Error Modeling and Metrics for Design Validation", DAC '92, pp 472-477.
[8] A. Kuehlmann, D.I. Cheng, A. Srinivasan, D.P. LaPotin "Error Diagnosis for Transistor-Level Verification", DAC '94, pp 218-224.
[9] I. Pomeranz, S.M. Reddy, "On Diagnosis and Correction of Design Errors", ICCAD '93, pp 500-507.
[10] Y. Kukimoto, M Fujita, R.K. Brayton, "A Redesign Technique for Combinational Circuits Based on Gate Reconnections", ICCAD '94, pp 632-637.
[11] S. Trimberger, "Field-Programmable Gate Array Technology", Kluwer Academic Publisher, 1994.
[12] P.C. Ward, J.R. Armstrong, "Behavioral Fault Simulation in VHDL", DAC '90, pp 587-593.
[13] M.D. O'Neill, D.D. Jani, C.H. Cho, J.R. Armestrang, "BTG: A Behavioral Test Generator", CHDL '89, pp 347-361.
[14] S. Ghosh, T.J. Chakraborty, "On behavior Fault Modeling for Digital Designs", Journal of Electronic Testing, vol 2, Kluwer Academic Publishers, 1991.
[15] Z. Navabi, "VHDL Analysis and Modeling of Digital Systems", McGraw Hill, 1993.
[16] J. Rose, R.J Francis, D. Lewis, P. Chow, "Architecture of Field Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency", IEEE Journal of Solid-State Circuits, vol. 25, no.5, October 1990.
[17] A. Rushton, VHDL for Logic Synthesis, McGraw Hill, 1995.