AN INTELLIGENT MODULE GENERATOR ENVIRONMENT**

P. Six, L. Claesen, J. Rabaey and H. De Man

VSDM division of IMEC, Kapeldreef 75, B-3030 Heverlee, Belgium.

** Research sponsored by the EEC under ESPRIT Project nr. 97

Abstract

An environment for the generation of modules is described. It includes tools for interactive design of parameterised procedures describing the structure as well as the topology. For the layout symbolic cells are used which are automatically fitted together as defined by the topology.
For the verification and characterization rule based expert tools were developed to recognize registers, check the clocking rules, find the critical path and the appropriate test patterns to calculate the accurate delay via simulation.

Introduction

In the last years, a new approach to the design of integrated circuits has been introduced to the market : the so-called module generators. In this approach the idea is to generate the layout of cells or blocks from parameterised software procedures.

In this paper we present the system we are developing at IMEC for the interactive design of parameterised generators of VLSI modules like data-paths, adders, multipliers, RAM and ROM. It provides immediate feedback in graphics form each time a command of the module generator is defined allowing mistakes to be corrected where they are made. The system is based on powerful constructs to describe the structure, the relative position of subblocks as well as the procedures to fit their layout together. Another new idea is to integrate into the system, tools for expert verification of clocking- and electrical rules as well as for generating the delay model, using automatically generated test patterns that activate the longest path(s). In this way the designer is relieved from most of the nitty-gritty of the detailed layout and the cumbersome search for worst case delay situations.

First we discuss the scope of module generators. Then the interactive creation of a module generator and the main features of the module design system are described. Next the automatic abutment of symbolic cells to generate the layout of the modules is discussed. In the last part the intelligent verification of modules, using rule based tools for the recognition of registers, checking clocking rules, pinpointing problems with electrical design rules like charge redistribution and finding the critical paths and their delays are presented.

1. MODULE GENERATORS

In traditional systems the design of a chip is done by composing larger cells out of smaller ones. In this approach each cell is built by interactively placing the smaller cells on specific coordinates.

In modern designs a lot of the blocks have some kind of structure, they are built from smaller cells or groups of cells according to a certain pattern (rows, columns...). Examples of this type of blocks are adders, multipliers, datapaths etc. It also happens that the same functional block is needed with different complexities, like an 8-, a 12- and a 16 bit adder. In the traditional design style we would create 3 different cells containing respectively 8, 12 and 16 full adder cells. The effort to create and check these functions is repeated over and over again.

A very simple procedure could be written to place and connect N full adder cells. This procedure once written and tested could then generate the netlist, the layout or the delay for any type of adder if the designer specifies what value of N he needs. This kind of procedure is in fact what is called a module generator.

The first reason for the introduction of module generators is that, once they have been created, they reduce the design time. A second reason is that, instead of designing at the gate level, system designers decompose their system specification in terms of functional building blocks (FBB) like counters, registers, PLA, RAM, ROM which are of MSI and even LSI complexity. The system designers can assemble the layout outline of these modules in nearly the same way as they did in PCB design but they lack the skills to design the internals of these blocks. This is the role of the silicon specialist. Since the functional blocks will be used in different applications, they must be designed in such a way that they can be adapted to the environment in which they are used. This means that silicon designs becomes intermixed with programming since the only way to reach the required flexibility is to make use of parameterized software procedures that generate FBB's.

A drawback of programming is that the designer has to wait for the compilation and linking steps before he can check the procedures he has defined. For this reason we implemented an interpretative system that shows the result of the designers action immediately on the graphics screen.

0738-100X/86/0000/0730$01.00 ©1986 IEEE

2. THE MEET-IN-THE-MIDDLE DESIGN METHODOLOGY

The scenario for chip design described above suggests that the system to support this methodology is organised in two parts as described in Fig. 1. The upper part is the area of the system designer. He has access to the higher level design tools such as register- and functional level simulators, controller synthesis tools and a floorplanner with which he can place and route the functional blocks. When he needs data, like the functional description, the delay or the bounding box of a functional block he requests this information from the procedures that contain the definition of that block.

The lower part of the system is more like the traditional chip design environment. It provedes the silicon engineer with tools to define the contents of the functional blocks, to simulate them at the logic and circuit level and to create its layout.

The main difference with the traditional design system is that the result of what we call the module creation is not the simple layout of one specific instance of a certain function but a procedure that can be used to generate a set of instances of that function.

The above described methodology is what we call the meet-in-the-middle design methodology. At IMEC we are developing a system called CATHEDRAL 2 (Ref. 1) based on this methodology. In the rest of this paper we present the tools involved in the module creation part of the system.

3. MODULE CREATION

Programs that create modules have been implemented using traditional programming languages like PASCAL and C (Ref. 2-4). The main disadvantage of these systems is the delay introduced by the compilation and linking steps, from the time the designer specifies the module and the moment he can check the results of certain actions or parameter assignments.

Our idea is to create an interactive environment, where the designer has immediate feedback in graphics form, each time a command of the module generator is defined. In this way mistakes are detected exactly where they are made and can be immediately corrected.

To remain flexible in adapting the input language we decided to implement the command parsing of the first version of the system in LISP while for some of the supporting algorithms, like the compactor and router we use PASCAL and C for reason of efficiency on the VAX.

When we want to describe a module there are essentially two types of information we have to capture :

- STRUCTURAL data describing how the module is built up from lower level modules, how they are interconnected and which are the input and output terminals.
- TOPOLOGICAL data describing how the sub-modules have to be placed relative to each other, how they fit together and which of the interconnections are realised by abutment or routing.

Furthermore since the generators must be able to describe a set of modules rather than only one instance, constructs like LOOPS and IF.. THEN..ELSE must be provided and we must be able to define structural parameters, like the word-length of an adder or the depth of a ROM. The structural and topological data must then be defined as a function of these parameters.

In the next part we present the main features of the interpretative procedural design capture system. We will highlight, by an example, the fact that the designer does not not need to bother about layout details but is able to place blocks relative to each other. The system then takes care of the detailed fitting of the blocks.

3.1. Main features

In a traditional design environment, the structural data is captured via a schematic editor and the topological data is entered on a graphics system like Calma or Applicon. The main difference between a schematic block diagram and a floorplan is that they present the design from a functional versus a layout point of view. Since in essence the same information of contained in both views, we have decided to combine these two views in the first version of our system.

To design a new module generator, the first thing that must be done is telling the system that we start with a new module. At the same time the parameters for this module are specified. For instance the command

```
(EDIT-COMP (RAM N M))
```

specifies the start of the definition of a block called "RAM" with two parameters N (number of address bits) and M (wordlength).

Lower level modules can be included in a module by the ADD-COMP command. This command provides a construct to specify the type of component to be added, where it must be located and what transformations (rotate, flip) must be performed on it. The location is given as a grid point or a series of grid points specifying the relative locations of the instances. The command

```
(ADD-COMP RAMCELL 1..(*2 M) 1..EXPT 2 (- N 1)) :ROT 90)
```

will place the component RAMCELL, rotated by 90 degrees on the gridpositions 1 to 2 times in M in the x direction and from 1 to 2 to the power N-1 in y, assuming that two columns of memory cells are multiplexed on the one sense amplifier. For N=4 and M=4 this command would result in the display shown in fig. 2a. Fig 2b shows the graphic response after adding the decoder to the right and the multiplexers and sense amplifiers on top of the array. Fig. 2c shows the final layout of the ram after the cells have been fitted together (see further).

If an instance of the component that we want to add WITH THE SPECIFIED SET OF PARAMETERS does exist in the library, its blackbox representation, generated from the compacted layout (see further), is loaded into the system and used for display. If only the definition of the component exists it is executed WITH THE SPECIFIED SET OF PARAMETERS, added to the library, loaded and used. If not even a definition of that component exists the system will give a warning and propose two options :
to start the definition of that component first or at least give a "dummy" blackbox it can use to temporary represent the undefined component.

If the user chooses to define the missing component, an EDIT-COMP command is automatically issued.

To provide connection points to the module at a higher level of the hierarchy terminals can be defined. In the module itself the interconnection of the sub-modules is defined by specifying a net and the terminals of the sub-modules and of the module itself that are connected to it. As with the ADD-COMP command facilities to express repetitive connections are provided in the ADD-CON command.

(ADD-CON (CARRY 1...(- N 1)) (COUT 1..(- N 1) 1) (CIN 2...N 1)) would connect all the carry signals in an N bit adder when the cells are placed from left to right. The first loop defines the names of the nets (CARRY 1 to N-1) and the other loops define the pinnames and the component position of the pins connected to those nets (eg. pin COUT of the components on gridpoints [1,1] to [N-1,1]).
Note that no assumption is made at this stage of how the connections will be realised. The physical connection will be realised later by the ABUT command or when the router is called to generate the wiring between terminals of the same net. At this moment a river router has been implemented and we are working on the integration of a channel router.

The final step of the module creation phase is to save the steps of the interactive session into a procedure that can later on generate the instances of the module on request of the system designer.

3.2. Leaf cell design

By using the commands described above the module generator procedure can be specified and can be hierarchically refined until the leaf cells are reached. For the leaf cells we must give the full layout information. For this task we use CAMELEON our symbolic layout editor (Ref. 5). This program can be started as a subprocess from within the module generator environment.

CAMELEON allows the user to design the layout of a cell using point elements like transistors and contracts, wires to interconnect the point elements and areas to define for example the n- or p-well in CMOS technology. After the topology (the relative positions of the elements) is entered a compaction step, based on a constraint graph and a longest path algorithm (Ref. 6), is performed to minimise the area of the cell. The design rules and parameter values for the particular technology used are read in from the technology description file (Ref. 7) and the compaction makes sure that the layout is correct with respect to these design rules, also taking into account extra constraints the user may have specified.

3.2. Abutment of cells

A technique to interconnect cells that is often used in the design of structured modules is abutment. This means that the layout of the cells is done in such a way that the connections are made by just placing them side by side.

In a traditional layout environment abutment of cells can only be achieved by taking into account, while designing the layout of a cell, the requirement to abut to the other cells. In our system the designer can express this requirement by the command :

(ABUT XORY (CELL-LIST)).

The module generator environment will then call the compactor, load the cells specified in CELL-LIST and put the required horizontal or vertical constrains (dependent on the value of XORY), between the terminals of adjacent cells.
CAMELEON will then compact the internals of the cells, taking into account the constraints imposed by their neighbours. Fig. 3a and b show a set of cells before and after the ABUT command has been applied.

The way we solve the abutment of the cells is as follows : each of the cells is compacted separately. From the constraint graph we extract what we call a "substitute graph" that contains the constraints imposed on the terminals by the internal components of the cells. The substitute graphs of abutting cells are combined by putting constraints between connecting terminals. Then the larger graph is solved by the longest path algorithm resulting in constraints to be imposed on the individual cells. The final step is to recalculate the longest paths for the individual cells but taking into account the constraints imposed from solving the combined substitute graphs.

4. VERIFICATION OF THE MODULES

One of the difficult problems with module generators is to prove their correctness over the validity range of the parameters. The traditional way of doing this, is by simulation. However simulation is a subjective test method which depends entirely on the patterns defined by the designer to detect expected problems. It is costly in CPU time and does not guarantee to capture unexpected problems.

To overcome these drawbacks we have developed DIALOG (Ref. 8) a set of rule based analysis tools that allows to detect violations against basic design principles. These tools are based on the LEXTOC language which provides powerful facilities to express rules about MOS circuits. Fig. 4 gives an overview of the verification system.

4.1. MOS network extraction

To verify the design we first have to extract from the layout a model for the cell on the circuit level. This can be done very easily since the stick diagram contains the devices and the interconnection wires. The internode capacitances can be found by detecting the points where elements cross. Starting from the extracted MOS transistor network the user can perform the following tests :

4.2. Decompilation

First it is checked if the network belongs to the class of valid circuit configurations, eg. static CMOS using passive multiplexing trees and prescribed registers. All parts of the circuit violating these rules are flagged as errors.

Next, a high level check on the clocking can be done. Starting from the primary clock information the drived clocks are found, latches are detected and separated from combinatorial logic. By assigning the appropriate properties to the nodes

in the network illegal combinations can be detected and reported. Examples are :

A DERIVED CLOCK CAN ONLY BE FORMED WITH SIGNALS LATCHED ON THE SAME CLOCK PHASE.

A LATCH DATA INPUT MUST NOT BE A PRIMARY CLOCK.

Fig. 5 shows the response of DIALOG to a clocking rule verification request. The system has identified and highlighted that a loop exists containing only a single clock phase which could cause a race problem.

4.3. Electrical rule checking

Starting from the primitive circuits found in the previous step we search for illegal configurations like open, short, odd n- p-MOS combinations etc... For all acceptable circuits, it is checked if they can guarantee correct logic levels. This check is not restricted to static situations but includes dynamic effects like charge redistribution.

An example of a possible problem circuit, and the rule involved in the checking procedure is given in Fig. 6. It must be noted that to diagnose the occurence of an error on node B caused by charge sharing, we have to find the combination of input signals that creates the worst case configuration. This is done by the SETUP-INPUTS procedure. Using these inputs, the part of the circuit under investigation is simulated on the circuit level using the SIMMY module (Ref. 9). The output waveforms of the simulation are then checked to see if the voltage node on node N2 drops below 3.0 volts. If so, a message is issued telling the designer charge sharing problems can be expected on node N2. If the designer does not believe this diagnosis, a facility can be called to display the conditions creating the problem situation and explain by what successive steps it was detected.

4.4. Timing verification

If the circuit has passed al the general rules we must still check that it does not violate the timing specifications for the module. From the primary clock specifications, the edges of the clock inputs for the latches are computed and the maximum allowed delay of the combinatorial blocks can be calculated.

To check that a block satisfies this restriction, we locate the critical path using a gradual refinement technique. First we eliminate the bulk of the shorter paths using simple RC models. For the remaining paths test patterns, generated at the gate level, are applied to the inputs in a circuit simulation step (again using SIMMY) to automatically calculate the gate delays. Finally, logical impossible paths are eliminated, the most critical paths are cut out and sent to a circuit simulator, together with the appropriate test patterns to calculate the accurate delays. These can then be compared with the allowed delay of the block.

Fig. 7 highlights some of the power of the timing verifier. The program has identified the critical path in the circuit and generated the pattern required on the other nodes to be able to activate it. If the "thick line" nodes are forced to logic 1 and the "thin line" ones at logic 0 a "0 to 1" transition at the upper left input causes "1 to 0" changes on the "dashed"nodes and "0 to 1" transitions on the "dotted" nodes. The dashed and dots have been used for clarity in black and white reproductions, these states are represented by colors in the real system. Using these patterns a circuit simulator is activated to find the accurate delay for the critical "dot-dash" path. This delay can then be stored and used by the system designer as a characteristic of this block.

We want to stress that in the complete process of defining this delay the designer does not need to define a critical path or specify any simulation pattern. All these time consuming tasks are automatically handled by the program.

CONCLUSIONS

In this paper we presented a set of tools to create an intelligent environment for the interactive design and verification of parameterised modules to be used in the VLSI design. The main features of the system are an interactive tool to design the structure, the topology and the layout of the modules including automatic abutment of symbolic cells and a set of rule based verification tools providing facilities for the recognition of registers, checking clocking rules, pinpointing problems with electrical design rules like charge redistribution and finding the critical paths and their delays.

ACKNOWLEDGEMENT

The tools in this system have been developed through the cooperation of many people. We want to thank more in particular I. Bolsens, J. Cockx, K. Croes, L. Rijnders, E. Vanden Meersch, I. Vandeweerd and all those who contributed to the concepts and implementation with their creative inputs.

REFERENCES

[1] H. De Man : "Evolution of CAD tools towards Third Generation Custom VLSI Design", Invited Paper ESSCIRC 1985.

[2] M. Buric et al. : "Silicon Compilation Environments", Proceedings IEEE Custom Integrated Circuits Conference, pp. 208-212, May 1985.

[3] B. Muller et al. : The Chipgenerator Concept - A New Approach to Full Custom CMOS IC Design", Proceedings of the 11th European Solid-State Circuits Conference, pp. 186-192, Sept. 1985.

[4] T.G. Matheson et al. : "Embedding Electrical and Geometrical Constraints in Hierarchical Circuit-layout Generators", Proceedings IEEE International Conference on Computer Aided Design, pp. 3-5, Sept. 1983.

[5] L. Rijnders et al. : "CAMELEON Version 1.1, Users Guide", Report nr. 5-C1-3 of EEC project MR-03-KUL, available from IMEC.

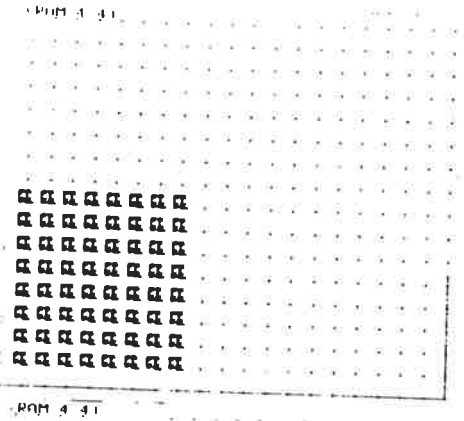
[6] M.Y. Hsueh et al. : "Computer Aided Layout of LSI Circuit Building Blocks", Proceedings of the International Symposium on Circuits and Systems Conference, pp. 474-477, July 1979.

[7] R. Zinsner et al. : "Technology Independent Symbolic Layout tools", Proceedings of the

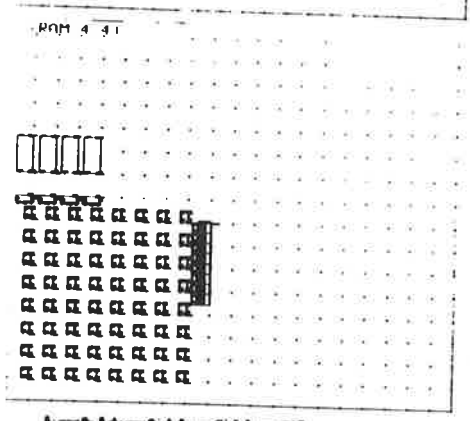
IEEE Internat. Conference on Computer-Aided-Design, Sept. 1982.

[8] H. De Man et al. : "DIALOG : An Expert Debugging system for MOS VLSI Design", IEEE Transactions on Computer Aided Design, Vol. CAD-4, No. 3, pp. 303-311, July 1985.

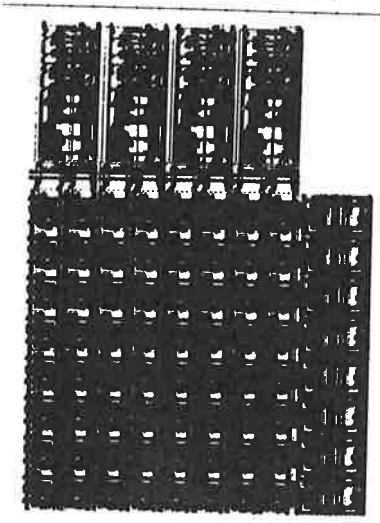
[9] J. Cockx : "SIMMY Users Manual", Internal Report.



2a.



2b.



2c.

Fig. 2 : Interactive built up of a parameterized RAM.
a : The array of RAM cells
b : Relative positioning of decoder multiplexer and sense amplifiers.
c : Final RAM layout after automatic abutment.

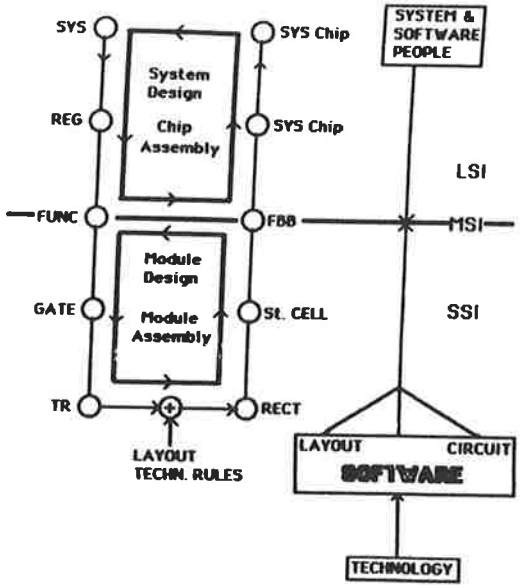


Fig. 1 : The meet-in-the-middle design methodology.

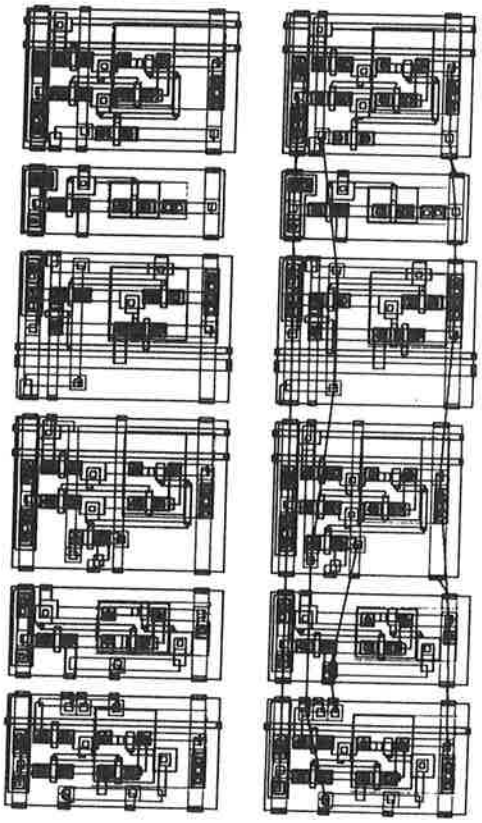


Fig. 3 : The automatic abutment of cells.

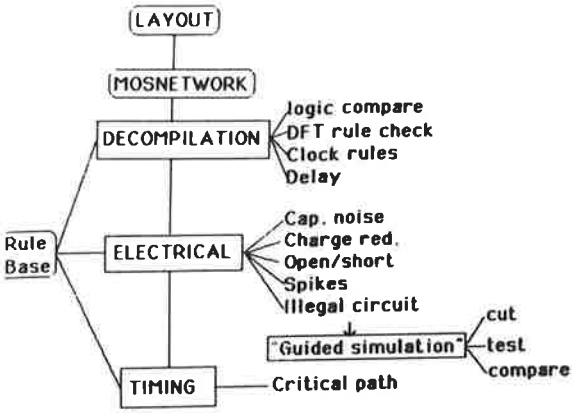


Fig. 4 : Overview of the DIALOG Verification system.

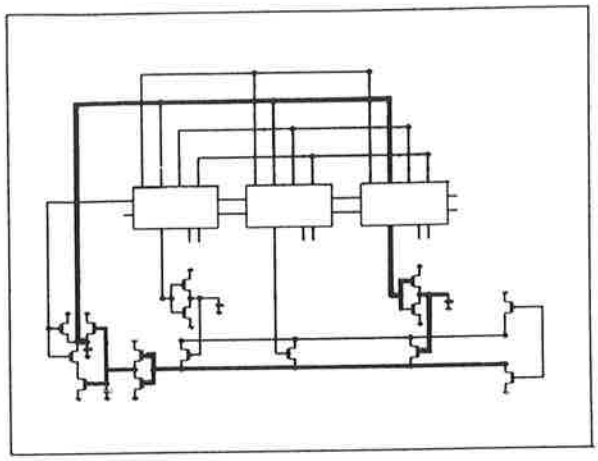
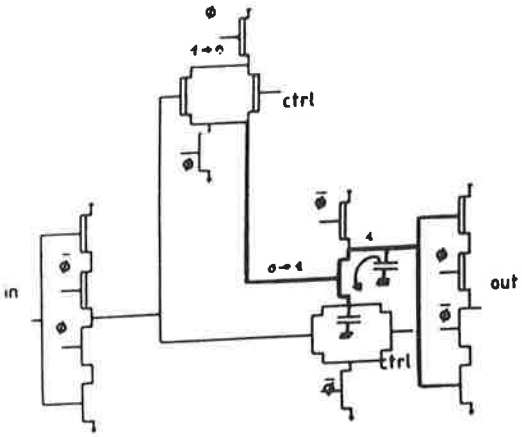


Fig. 5 : Example of a loop detected by DIALOG.




```
(RULE '(charge-sharing
*IF
 ((PROPERTY fibar-ndyn e1)
  (RELATION input  n1 e1)
  (RELATION output n1 e2)
  (PROPERTY fibar-pdyn e2)
  (RELATION output n2 e1)
  (RELATION component e3 e1)
  (RELATION input  n1 e3)
  (RELATION inout  n2 e3)
 )
*THEN
 ((ASSIGN-INFO possible-charge-sharing e3)
  (SETUP-INPUTS e1 1)
  (SIMULATE e1)
  (*IF
     ((DURING fibar ( > 3.0 ) T1)
      (ATTRIBUTE voltage N2 ( > 3.0 ) T1))
   *THEN
    (ASSIGN-INFO real-charge-sharing e3))))
```

Fig. 6 : Example of a circuit with charge sharing problem and the rules to detect it.

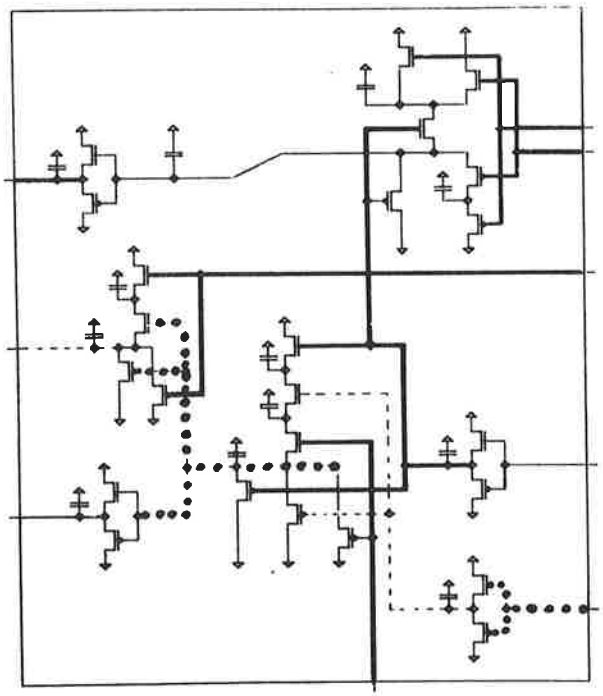


Fig. 7 : Automatic identification of critical path and required test pattern.

——— : zero
•••• : one to zero
- - - - - : zero to one
——— : zero