GateMaker: A Transistor to Gate Level Model Extractor for Simulation, Automatic Test Pattern Generation and Verification

Sandip Kundu*, Intel Corporation
2200 Mission College Boulevard
Santa Clara, CA 95052

Abstract

Hierarchy is key to managing design complexity. A hierarchical design system needs to maintain many views of the same design entity. Some of the examples might be physical view for placement, routing and extraction; transistor schematic view for circuit simulation, timing characterization and noise analysis; a gate level schematic view for timing, verification, logic simulation, fault simulation and automatic test pattern generation (ATPG); a register transfer level (RTL) view for specification and high level simulation etc. In order to achieve highest system performance, multiple design iterations are necessary, each iteration involving both forward and backward pass through hierarchy, with manual changes at any level of the hierarchy. This poses an essential challenge of keeping all views of same design entity in sync. In this paper we describe an automatic tool called GateMaker, that has been developed to extract a gate level schematic model from a transistor level schematic model for the purposes of logic simulation, fault simulation and automatic test pattern generation. This eliminates a manual process and offers manifold advantages that will be discussed in this paper.

1. Introduction

A cell or megacell in any custom or semi-custom digital design methodology is a manually manageable design piece that stands on multiple views. Each view (also called a model or book or library element in various terminologies) is created to support a level of abstraction suitable for a particular analysis. A gate level schematic view of a cell is necessary for supporting fault simulation, automatic test pattern generation and potentially other tasks such as logic simulation, verification, static timing analysis, cell sizing (adjusting transistor gate widths) etc. In this paper, we describe an automatic method of extracting a gate level schematic model from a transistor level schematic model for the purposes of fault simulation and test pattern generation. This method has been embodied in a tool called GateMaker. There are many advantages of having an automated process for this extraction. We describe some of the advantages below.

1.1. *Consistency of Gate Level Schematic Model*

Besides saving designers' time, an automatic tool provides several advantages. First of all, it provides a consistent modeling approach. In Figure 1, we show a custom implementation of an exclusive OR gate.

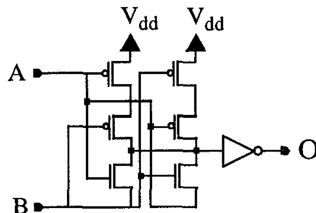


Figure 1 : *Custom implementation of an Exclusive-OR gate*

At gate level there are several possible ways of representing the transistor network shown in Figure 1. Some of these possible implementations are shown in Figure 2.

* This work was performed when the author was with IBM Corp.

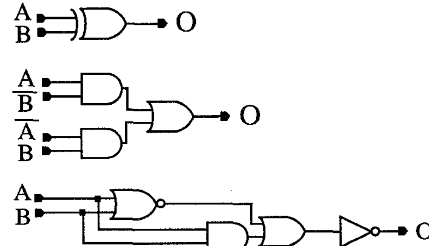


Figure 2 : *Some possible gate level realizations of Figure 1*

It is easy to see that the absolute number of stuck-at faults and the number of equivalent faults in each of the gate level realization is different. Once we attach stuck-at faults with internal nets of these models, it is easy to see that they have different number of stuck-at faults. This leads to an inadvertent weighting of the fault coverage. Such weighting is a serious concern because in the context of the larger design, improved fault coverage may mean a different gate level model rather than improved test quality.

An automatic tool brings consistency to the modeling process, so that the overall fault coverage is a more meaningful number

1.2. *Quality of Test Patterns*

There are many reasons why an ATPG tool runs on a gate level model for the design. Firstly, it provides a layer of abstraction and hides the test pattern generator from technology dependent information. Therefore, even if the underlying circuit techniques change, the test pattern generator does not have to be rewritten. Secondly, most switch level test pattern generators are not capable of handling complex circuits such as memory, clock regenerators etc. A layer of abstraction eases the level of complexity. Thirdly, the circuit data volume is reduced when transistor level schematics are translated to gate level models. Fourthly, faults at the transistor level tend to produce a large number of cases where the faults are possibly detected. From a practical standpoint this is not desirable, because the ultimate goal is to generate a set of vectors to supply to a tester, where the tester makes a pass/fail decision. Thus having too many possibly detected faults increases uncertainty about fault coverage number without providing any patterns. In contrast, test pattern generation at the gate level model runs the risk of producing irrelevant patterns. This may happen when the gate level model of a transistor level schematic is logically correct under fault-free situation but the faulty situations have no corresponding situation at the transistor level model.

Thus the extractor not only has to extract a gate level schematic that is valid under fault-free situation, but also 'valid' under faulty situation. On this issue it differs from most of the prior work in the area of functional extraction [1-7]. We shall revisit this point later in sections 3.3 and 3.5.

In section 2, we review the target CMOS circuit families and our extraction goal in context of these circuits.. In section 3 we describe the key steps of our extraction algorithm. In section 4 we present the implementation details and finally conclude in section 5.

0-7803-5092-8/98 $10.00 © 1998 IEEE

2. Review of Target Circuit Families

Generally speaking most logic circuits can be classified in the following four broad categories: Combinational static circuits, DCVS circuits (static or dynamic), pass gate logic and dynamic domino logic. In each category several permutations are possible. In this section, we will describe some of the realizations and devote some comments relevant to their modeling.

2.1. *Non-Clocked Logic*

Non-clocked logic describes any category of static CMOS circuit, where a valid input produces a valid output regardless of any prior condition.

2.1.1. *Complementary P & N Networks* [8]

This is the most popular CMOS design style. No matter what the input pattern is, there is a valid path from output node to Ground or output node to V_{dd}. The P and the N networks are usually dual of each other, meaning a series(parallel) connection of nFETs imply a parallel(series) connection of pFETs. In Figure 3 we give an example of such a circuit and describe how one might arrive at a gate level model for that.

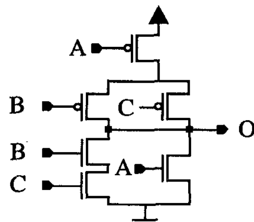
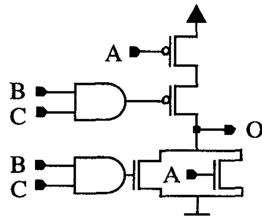
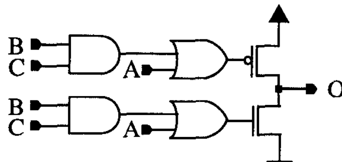


Figure 3 : *Example of a dual fully complementary gate*

Step 1: Compression of series transistors



Step 2: Compression of parallel transistors



Step 3: Drop the transistors for a functional model

Figure 4 : *Final model for Figure 3*

From the pictorial depiction of the steps, it should be apparent that we can perform series parallel compression of transistors by replacing them with just one transistor in successive steps. At the final step, a topology check can prove that the P and the N networks are complementary and then we can drop the transistors for a final model.

There may be a situation when P and N networks are functionally complementary but not dual of each other as described in Figure 5.

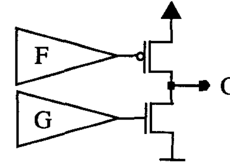


Figure 5 : *A P & N network after series parallel compression*

F and G are AND-OR networks derived after successive series-parallel compression steps. The issue of modeling here is a tricky one. One can take a safe approach and model it as shown in Figure 6 using tristate drivers.

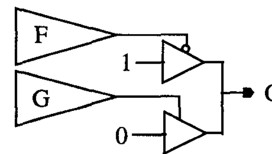


Figure 6 : *Model using tristate drivers*

However, usually one can run *boolean equivalence check* of function F and G and determine whether they are complementary or not. If F and G are proven to be complementary then the output O can be modeled as $\overline{F}$ or $\overline{G}$. Either network may be used. The advantages of using $\overline{F}$ or $\overline{G}$ is that:

a) we get a simpler model

b) simulation and test generation runs faster because X generation is suppressed. Tristate drivers tend to produce X under some faulty situations when enable input of both drivers are turned off. These faults can not be detected in purely combinational circuits (they may be possibly detected), and only potentially detected in sequential circuits where sequential test pattern generation is used.

The disadvantage in using a simpler model is that the structural equivalence is lost and the resultant gate level circuit may not represent the complete switch level model. GateMaker allows users to optionally choose one or the other. However, whatever approach is used, it should be used consistently.

2.1.2. *Static Differential Cascode Voltage Switch (DCVS)* [9]

Static Differential Cascode Voltage Switch circuits produce complementary outputs. The logic function is performed by the pull down network of nFETs and pFETs are used in a cross coupled manner for pull up as shown in Figure 7.

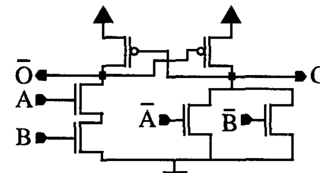


Figure 7 : *Example of a DCVS circuit*

Static DCVS circuit modeling can proceed with the first step of series parallel compression of transistors to yield a circuit shown in Figure 8 next.

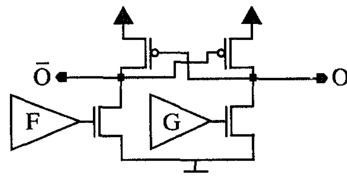


Figure 8 : *making the path conductingReduced static DCVS circuit*

However, at this stage it is important to know the *relationship between external inputs* to determine if F and G are complementary (such as between inputs A and $\bar{A}$ if they are supplied as external inputs). At this point, it is known that O is high when node $\bar{O}$ is low and node $\bar{O}$ is low when F is high. Therefore O = G and $\bar{O}$ = F. Thus we can reduce the circuit to F and G, provided:

a) relationship between inputs are known and

b) F and G are complementary

If F and G are not complementary an error should be flagged. In normal processing steps, Output O will be described as some function of $\bar{O}$ and other inputs and output $\bar{O}$ will be described as some function of O and other inputs. In such a case a logical loop in argument is detected. In this case the argument is resolved by looking past node $\bar{O}$ when the function O is being determined. It is however not very efficient to look beyond the input of the immediate node of the driving gate. Therefore, the program should back up and reason past when a loop is detected.

2.1.3. *Pass-Transistor Logic*

In complementary and static DCVS CMOS circuits we described so far, a transistor is used either for pull up or pull down of output node. In pass-transistor circuits the same transistor is used for both pull up as well as pull down of a node. Even though current may flow in both directions, usually signal flows is in one direction.

2.1.3.1. *Single Ended Pass-Transistor Logic* [10]

This is commonly used in non-differential logic. In Figure 1 we had given an example of a single ended pass-transistor logic. Here we describe the basic steps involved in deriving in a model for that circuit.

The key step involved in modeling such circuits is *enumeration of paths through channel connected components*. From the circuit it is apparent that output O is inverted function of N. To determine the function at N, all paths that traverse through source to drain or drain to source of a transistor are explored beginning at N. Such paths may terminate either at V_{dd} or at Ground or at a primary input node such as A. Once the condition for conduction of path is established it is captured in a logical form using AND gates. The output of the AND gate enables the appropriate terminal node for the path. This is shown next

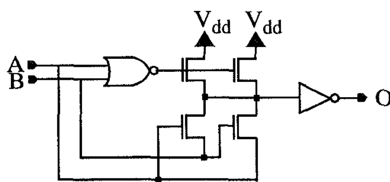


Figure 9 : *After series-parallel compression of transistors in Figure 1*

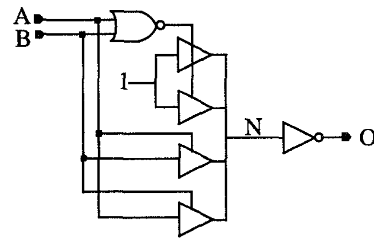


Figure 10 : *After enumerating 4 possible paths from node N of Figure 9*

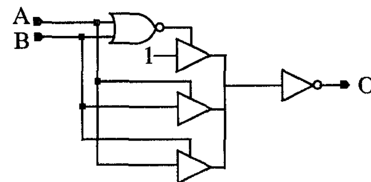


Figure 11 : *After eliminating duplicate tristate driver of Figure 10*

Once a tristate model has been derived, the next goal is possible *simplification of tristate drivers*. The tristate drivers are checked to see if further simplification is possible. This step will be described in detail in Section 3.7.5. It turns out that the tristate drivers (TSDs) can be replaced by AND gates and the dotting of the drivers can be replaced by a OR gate. The result is shown in Figure 12.

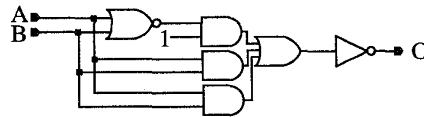


Figure 12 : *After converting the tristate circuits into AND-OR gates*

Next step involves *elimination of parallel duplicate gates* as shown in Figure 13 below.

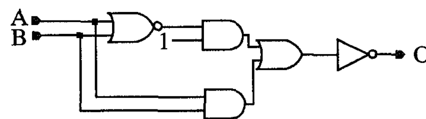


Figure 13 : *Result of removing duplicate parallel gate*

Constants are propagated for further simplification of model and *one-input buffers are elimated* next. The resultant circuit is the final model for this circuit as shown in Figure 14 below.

Figure 14 : *Final gate level model of circuit in Figure 1*

2.1.3.2. *Complementary Pass-Transistor Logic*

Complementary Pass-Transistor Logic (CPL) is a hybrid of static DCVS logic with pass transistors [11]. In Figure 15, we show an example of CPL. It should be apparent that modeling proceeds as in Single Ended Pass Transistor Logic with *complementarity detec-*

tion and *loop breaking* approaches of static DCVS modeling technique

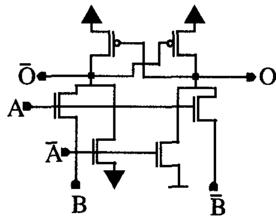


Figure 15 : *Example of a CPL circuit*

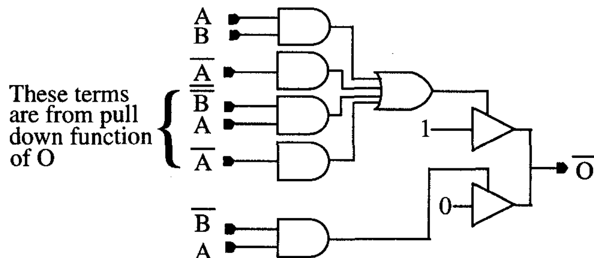


Figure 16 : *After path enumeration and loop breaking of Figure 15*

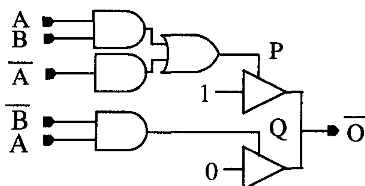


Figure 17 : *After inverter chain substitution and duplicate removal*

Figure 18 : *After boolean check to determine if P & Q are complementary*

2.1.3.3. *Double Pass-Transistor Logic (DPL) [12]*

DPL is a differential pass gate logic family that uses pFET and nFET in parallel. It makes the cross coupling unnecessary and does not suffer from any Vt drop. With DPL, modeling is easy because there is no loop breaking involved. However, boolean relationships are very important for simplification. Example of a DPL circuit is shown in Figure 19.

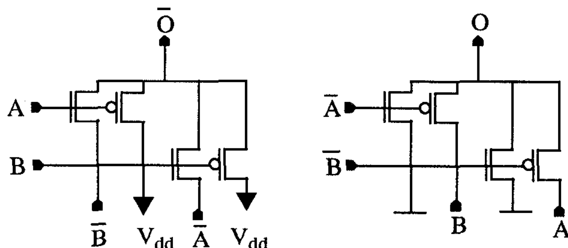


Figure 19 : *Example of DPL circuit*

2.2. *Clocked Logic*

Clocked logic circuits have two phases of operation. The first phase is called *precharge phase* during which the precharge clock is active and charge is stored on a node. The second phase is called *evaluate phase* during which the precharge clock is inactive and the stored charge may or may not be removed. An interesting case is when the precharge is conditional. In such cases, the behavior of the logic is sequential. Such circuits are beyond the scope of this paper. In this paper, we are purely focussed on circuits that are precharged in every cycle and evaluated in every cycle.

If we enumerate all channel connected paths, we can tag them as belonging to one of four following categories:

a) The path conducts to V_{dd} during precharge phase.

b) The path conducts to Ground during precharge phase.

c) The path is definitely off during precharge phase.

d) The path may potentially conduct to V_{dd} during evaluate phase.

e) The path may potentially conduct to Ground during evaluate phase.

Cases (a) and (b) are mutually exclusive, otherwise we have a conditional precharge situation which is explicitly disallowed and a warning message may be produced. Cases (d) and (e) need not be mutually exclusive because the path may terminate in a input node. If (c) is not true, the path may potentially turn on during precharge phase.

Once the paths are tagged, following scenarios are possible:

1. Output node is precharged to 1 (recorded).
2. Output node is precharged to 0 (recorded).
3. Output node has a path to V_{dd} as well as Ground during precharge phase (error message is produced).
4. The paths that are not on during precharge are not definitely off. In which case it may conflict with precharge activities and a error message is produced.
5. If output node is precharged to V_{dd} (Ground), there is no potentially conducting path to Ground(V_{dd}) during evaluate phase, indicating that the output node has a constant value. An error message is produced in such a scenario.

Once the required checks are performed and the circuit meets the validity criteria a gate level model is produced. The gate level model need not include the precharge signal and it is controlled optionally. Once again, the modeling process is explained below with aid of examples.

2.2.1. *Single Ended Domino with Foot Device* [8]

A foot device ensures that there are no sneak paths during precharge by definitely turning of all paths that do not participate in the precharge process. In Figure 20 we show a simple domino circuit with foot device.

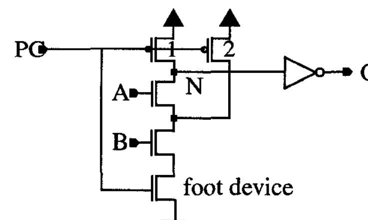


Figure 20 : *A footed domino circuit*

Processing this circuit requires additional input from the user. The exact value of the precharge clock (PC) during precharge and evaluate phases need to be known. This information may be supplied by tagging the schematic itself or with overrides from an assistant file. In this circuit PC is tagged to have a value 0 during precharge phase and 1 during evaluate phase.

In the first step of processing this circuit, we perform series parallel compression of transistors excluding all transistors that are fed by a precharge clock. In this case there is no compression possible because of the intermediate precharge point.

Next, N is recognized to be responsible for O and all paths from N are enumerated. There are 3 possible paths:

1. Path 1 goes through transistor 1 to V_{dd}.
2. The second path goes to V_{dd} via the transistor fed by signal A and transistor 2.
3. The third path goes to Ground, via stacked nFETs.

It is recognized that when PC=0, Path 1 conducts, Path 2 potentially conducts to same value and Path 3 does not conduct. So there is no interference with precharge process and it passes the checks. Thus node N is precharged to 1.

When PC=1, Path 1 and 2 are definitely off and Path 3 may potentially conduct depending on the value of A and B. Node N is 0 if and only if, A & B are 1 in this phase. Thus the output function at node N can be written as $N = \overline{A \cdot B \cdot PC}$ or $N = \overline{A \cdot B}$. Therefore the resultant model is as shown in Figure 21.

Figure 21 : *Gate level model of circuit in Figure 20*

2.2.2. *Single Ended Domino without Foot Device*

This case [8] is very similar to the previous case except we need more analysis for modeling. In Figure 22 we describe the circuit of Figure 20 without a foot device.

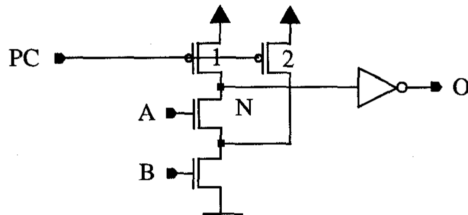


Figure 22 : *A domino circuit without foot device*

As before there are 3 possible paths in this circuit. However, when PC=0, unless A or B or both are turned off, there is a conflict during precharge. A and B may be output of domino circuits and may have value = 0 during precharge. If the circuits driving inputs A and B are known, the analysis proceeds *recursively backward* until we reach primary inputs or until we encounter circuits with footed device that are guaranteed to be off during precharge. When primary inputs are reached, they are checked for their precharge status, which are optionally specified by the user. If the precharge status is unknown, then an error message is produced and the analysis stops. Once non-interference during precharge is established, the rest proceeds as before.

2.2.3. *Dual-Rail Domino* [9]

We have discussed static DCVS circuits before. Dual rail domino looks pretty much like static DCVS circuits as shown in Figure 23. Dual-Rail Domino may or may not have a footed device. In absence of a footed device the *check* described in section 2.2.2 needs to be performed. *Loop detection* as described in 2.1.2 is also necessary. Dual-Rail domino circuits are popular because of noise immunity. Avoidance of floating state is a very attractive feature in a noisy environment.

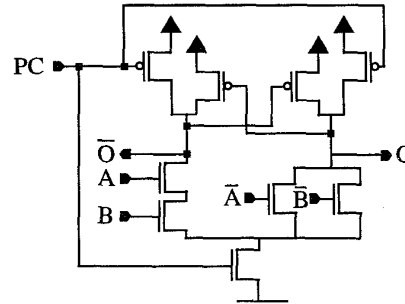


Figure 23 : *An example of a Dual-Rail Domino circuit*

2.2.4. *Zipper Domino* [13]

Zipper domino logic eliminates the inverter normally associate with single rail domino logic. It features cascaded logic blocks alternately executing logic with pFETs and nFETs and requires inversion of PC as well. Zipper domino is not very popular because drive through pFET compromises performance, noise immunity and load capacity. However, we mention this family because GateMaker can handle members of this circuit family with ease.

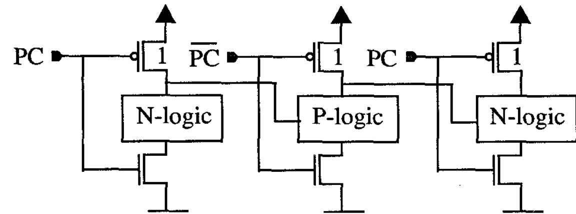


Figure 24 : *An example of a Zipper Domino circuit*

2.2.5. *Pseudo Clocked*

In pseudo static circuit, the precharge is driven by a signal rather than a clock. Therefore a recursive analysis is necessary to determine the precharge status of driving nodes.

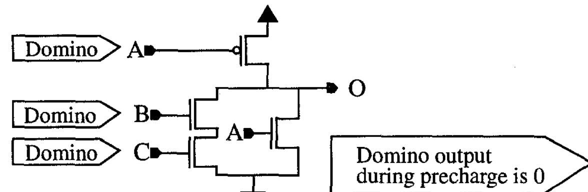


Figure 25 : *Example of a pseudo clocked circuit*

2.2.6. *Keeper Devices*

A floating node may discharge due to a sudden noise pulse and therefore, sometimes a feedback device is attached to increase noise immunity at the expense of performance. Such devices are called keeper devices. Keeper devices may help just one value (pFET for keeping 1, or nFET for keeping 0) or they may help both values (more common with pass logic or transparent latches or pulse to DC

converter circuits). In most instances keepers can be readily recognized and ignored as helper devices. Though semantically there is a difference between keeping and latching, circuit wise there is no such distinction. Both look the same. Therefore it is wise to annotate schematic as to which is a keeper and which is a latch and treat them appropriately.

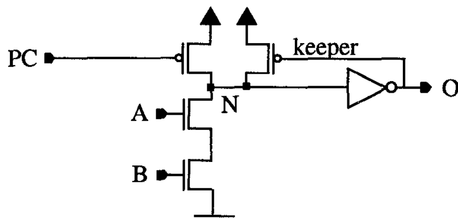


Figure 26 : *A domino circuit with a pFET keeper*

2.3. *Combinational Logic with Feedback*

Many combinational circuits have topological feedback in them that are logically open. We saw examples of them earlier in the context of DCVS circuits. One way to break these loops is to treat pull down functions and pull up functions separately. This was described earlier in section 2.1.2. However, this may increase complexity if the loop keeps getting bigger. A simple way to fix this complexity problem may be with user help where by users annotate a direction of signal flow through a transistor. While tracing paths all paths where signal flow direction is violated will be aborted. This solves a complicated problem with trivial help. Next we illustrate this with an example.

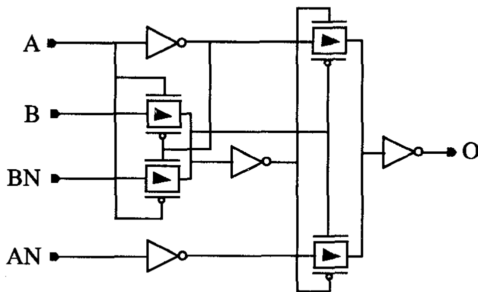


Figure 27 : *A circuit with annotated signal flow direction*

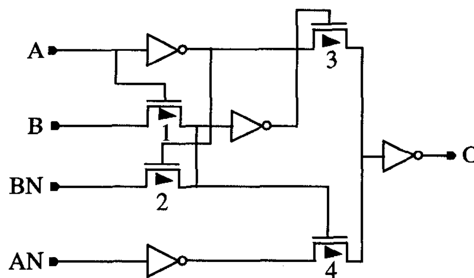


Figure 28 : *After parallel compression of transmission gate with boolean check*

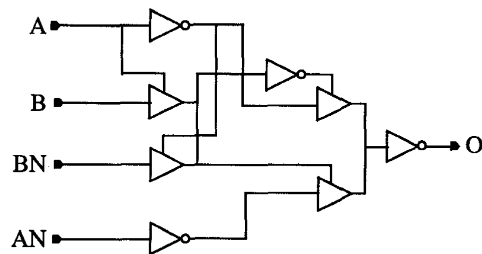


Figure 29 : *Transmission gates in Figure 28 are converted to TSDs; Loops broken by directions*

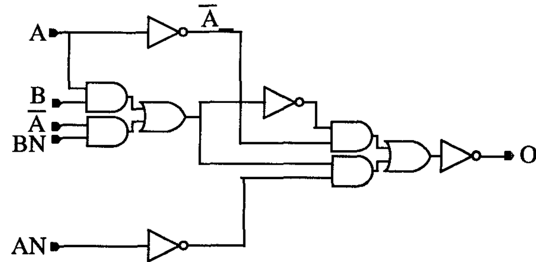


Figure 30 : *After boolean check TSDs are converted to AND-OR gates*

3. Key Ideas & Algorithms

In the preceding section we have described the major steps involved extracting a gate level model from a transistor level schematic. In this section we will describe a flow that integrates these major steps. There are many other nuances that we have not described yet. After a flow has been presented we will describe some of the subtleties that are need to be addressed as well.

In the introduction section a claim was made that all detectable switch level faults should be detected by patterns generated by running ATPG on these gate level models. We will highlight the feature of the algorithm that makes it so.

3.1. *Pre-Processing Steps*

Methodologically, GateMaker can be run after schematic entry of a transistor network or after layout extraction of a physical layout. A number of issues crop up when GateMaker is run after layout extraction. First of all, extracted layout typically has a lot of parasitic elements that are important for electrical simulation but irrelevant for gate level modeling. Secondly, post layout schematic may differ from pre layout schematic due to geometric reasons. In the subsequent sub-topics we discuss some of the steps that are run.

3.1.1. *Treat Split Transistors as One*

In a typical layout, metal 1 lines are run horizontally and polysilicon lines for transistors are run vertically. Typically, the cell height is fixed to realize maximum density. A cell height may typically be 10 to 20 metal 1 tracks in height.

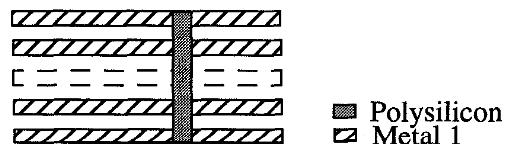


Figure 31 : *Typical layout*

It is easy to observe from Figure 31 that the maximum width of a transistor is limited by cell height. Suppose a cell is 20 tracks high. For a 0.35 μ technology, metal 1 lines will typically have pitch of the order of a μ. Therefore, any transistor more than 20 μ wide need to be split up into two or more parallel fingers with identical source drain and gate connections. If these transistors are not treated as a single transistor then GateMaker will produce different results depending on whether it is run on designed or extracted schematic. Therefore the first preprocessing step is to identify all parallel transistors and treat them as one. However, sometimes it may be wise to run it even afterwards. As an example, if this step was used on Figure 9 in section 2.1.3.1 after series parallel compression, then the subsequent steps would have been simpler. However, in GateMaker it is strictly done as a preprocessing step.

3.1.2. *Treat Resistances as Shorts*

Resistances are extracted to predict the delays and signal slew rate etc. Given enough time all signals will stabilize in a combinational circuit. Since, we are extracting a gate level model without a timing behavior all resistances are treated as shorts.

3.1.3. *Treat Capacitances as Opens*

At the steady state capacitors do not conduct and therefore for the same reasons mentioned before capacitances are treated as opens.

A new issue that crops up with extracted models is that of annotation of schematic. A precharge clock line may be extracted as a multi-segment RC network. After the resistances and capacitances are dealt with as opens and shorts, we have to propagate the fact that this is a precharge line all through out the circuit. This is also a part of the pre-processing step. It may be called *propagation of attributes.* A danger with blind propagation of attributes is that it may have been there as a delay line for phase inversion. In such a case, if attributes are propagated without taking phase inversion into account, the resulting models will be inaccurate. Therefore, the annotation needs to be explicit in stating whether there is any phase inversion involved.

3.2. *Series Parallel Compression*

Series parallel compression of transistors have already been explained. In actuality, it is implemented as an event driven routine that checks for possible parallel (serial) compression right after serial (parallel) compression. GateMaker treats transistors as switches and therefore pFETs and nFETs in series or parallel can be combined as long as there is no other connection incident on the junction. When dissimilar transistors are treated together, appropriate inversions are taken into account.

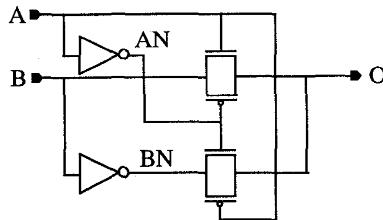


Figure 32 : *Example of a circuit with parallel connection of nFETs and pFETs*

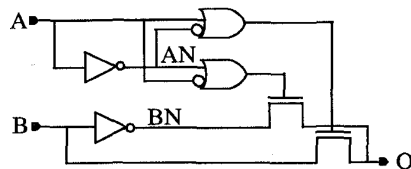


Figure 33 : *The resulting circuit after parallel compression of transistors in Figure 32*

3.3. *Path Tracing*

Path Tracing through Channel Connected Components (CCC) was briefly touched upon in section 2.2.1. It is a popular technique in switch level simulation, in electrical simulation of circuits and in function extraction.

Definition: A CCC is defined to be the maximal set of transistors and nets such that every net in the component is reachable from every other net by traversing source-drain connections of transistors within the component.

GateMaker uses an explicit path enumeration for extracting model for every channel connected component. The path traversal is done starting from each node that drives a primary output or gate input(s) of other CCCs. Thus a model is also created for every node that drives a gate input of a transistor. The path traversal is done in a recursive manner starting from primary outputs. The advantage of a recursive backward traversal is that *false paths* can be *eliminated* efficiently. We will visit how this is done in the next section.

Every path thus enumerated ends in a terminal node that is either V_{dd}, Ground or another primary input. All paths that lead to Ground are bunched together as AND-OR expression, where each AND gate represents the input conditions under which a path turns on. Similarly all paths that lead to V_{dd} are bunched together as AND-OR expression. For each primary input, all paths that are incident on it are bunched together as AND-OR expressions. These AND-OR expressions eventually controls TSDs that pass logic value 0 or 1 or specific primary input(s).

3.4. *Path Pruning* [7]

In general, a CCC can contain an exponential number of paths. Typical problematic structures are rotate operations that are implemented as flow *forest* (forests are collection of *trees*) of pass transistors. In Figure 34, we describe a small shifter to explain the problem and how it is solved.

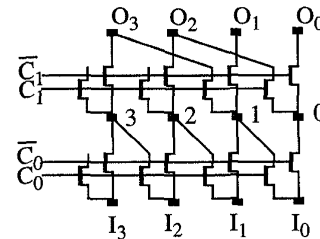


Figure 34 : *Barrel shifter circuit containing an exponential number of paths*

There are 4 inputs (I_0,...,I_0) that are shifted to produce 4 outputs (O_0,...O_3) and two control inputs (C_0,C_1). When the control inputs have value 00 there is no shift. If C_0=1 and C_1=0, inputs are shifted by one position to the left, C_1=1 produces additional shift by two positions. In the figure we show configuration for left shift only. However the principle extends to rotate left, shift and rotate right.

If path tracing begins at node O_0, then one possible path is O_0-Node 0-O_2-Node 2-Node 1-I_0. However it is easy to see that this path can never conduct because it requires $C_0 = \overline{C_0} = C_1 = \overline{C_1} = 1$ which is not possible to satisfy. This is called a *false path*. False paths should be avoided in this case because (1) they produce redundancy in the model and do not actually help produce patterns and (2) increases the model volume.

When the path is traversed from node O_0 to 0, we have two ways to expand further, and once we take a path to O_2 and reach node 2, we again have a choice of two ways. These possibilities multiply and for a 64 bit barrel shifter the number goes beyond any meaningful analysis. Thus false paths should be truncated early and any time we add a new segment, we should invoke boolean analysis to see if this

is feasible or not. This will prevent us from expanding path O_0-0 to O_2. This is called *path pruning*.

Path pruning requires that we have a boolean (gate level) model for all inputs. Thus processing can only proceed from input to output.

Boolean analysis proceeds by asking whether there exists a combination of input values required to turn on the path, (such as is $C_1 = \bar{C_1} = 1$ possible?). To answer that question we must know relationship between all primary inputs (are C_1 and $\bar{C_1}$ independent?). There are many ways to run the analysis. We use a branch and bound algorithm similar to one used in test pattern generation algorithms. We could also use BDDs. However, BDDs are built up from primary inputs and may not exist for all functions, on the other hand the type of information we are looking for are usually obtained by simple reasoning in the neighborhood and the analysis does not spread far.

3.5. *Analysis of Paths and Model Formulation*

In section 3.3 we briefly touched upon how the model is created. It is the most crucial part of the analysis. First of all, all switch level paths that can not be factored are turned into an AND clause in the model, assuring that the test pattern generator will exercise all paths in sensitization criteria. The paths are optimized only in specific cases where we can be certain that by throwing it away, we are not compromising the test quality as judged on the switch level circuit. Series-parallel compression ensures that all transistors are tested for conduction without creating unnecessarily large number of paths or AND gates in the model.

3.5.1. *Model Formulation when the node is Precharged*

The simplest situation in this category is when the target node is precharged to 1 (0) and there is no path to V_{dd} (Ground) and no path to primary input. A check is performed to make sure that the precharge is conflict free (as explained in section 2.2.2) and a AND-NOR (OR) circuit is created.

If there are paths to primary inputs, tristate driver circuits are built. These circuits are later targeted for further simplification. Simplification steps will be discussed in section 3.7.

3.5.2. *Model Formulation when the node is not Precharged*

Two cases are possible: there is a path to primary input; there are no paths to primary input. In the first case a tristate driver model is produced which is later targeted for simplification.

In the second case a check is performed to make sure that there is at least one path to V_{dd} and one path to Ground. If there are none then an error is flagged. Now we subject the AND-OR function that turns on a path to V_{dd} to a check against the AND-OR function that turns on a path to Ground. If it is possible to turn on both at once, a error message is produced. If it is possible to turn off all paths to V_{dd} and Ground simultaneously, then a tristate driver model is produced else we subject the AND-OR functions to topology check and based on user option create a tristate model if topologies differ or produce a AND-NOR circuit. Outputs of gates (AND/OR etc.) are treated as if they are primary inputs.

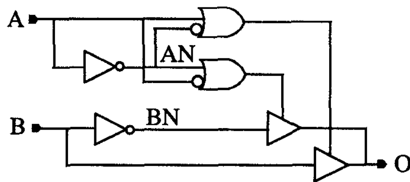


Figure 35 : *The resultant model from Figure 33 after model conversion*

3.6. *Complementarity Recognition*

In section 3.5.2 we described a check for complementarity without specifically stating it as such. Complementarity recognition deserves a separate mention because it forms an essential part of simplification. Simply stated two functions F and G are complementary iff there is no input pattern that can make $F = G = 0$ or $F = G = 1$.

Transmission gates can be easily identified if we perform a Complementarity Recognition of the respective inputs.

3.7. *Simplification*

It has been mentioned before that test pattern generators do not like to see too many tristate drivers because they end up producing too many possibly detecting faults without creating too many patterns. It has also been mentioned that we would not like to simplify the AND gate structures away because they represent electrically conducting paths that we are targeting for test pattern generation. We can still perform some simplifications without compromising either goal. They are described next. These transformations either remove redundancy created by intermediate steps in modeling process, or actual redundancies (in same spirit of split transistors) or eliminate gates that do not affect test patterns (but does affect fault coverage). All simplification steps are optional.

3.7.1. *Inverter Chain Elimination*

N (N > 1) inverters in a series can be simplified by removing them (N even) or replacing them with a single inverter (N odd) provided there are no fanouts from the internal nodes in the chain.

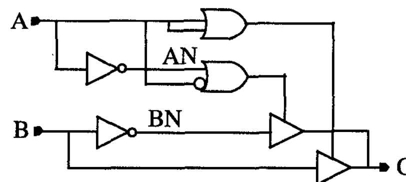


Figure 36 : *The resultant model from Figure 35 after inverter chain elimination*

3.7.2. *Input Dropping*

If there are multiple connections between output of one gate and inputs of another gate whose inputs are commutable (AND, OR, NAND, NOR etc. but not TSDs) then all but one of those connections can be dropped.

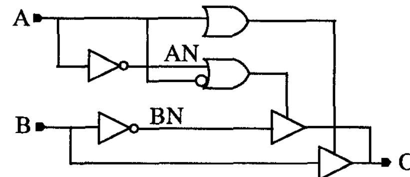


Figure 37 : *The resultant model from Figure 36 after input dropping*

3.7.3. *Gate Dropping*

If two or more gates have identical input output connections then all but one of them are dropped.

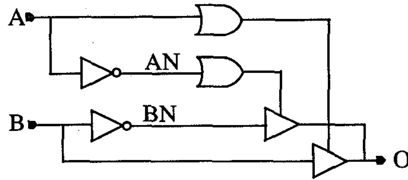


Figure 38 : *The resultant model from Figure 37 after gate dropping*

3.7.4. *Buffer Elimination*

All single input single output non-inverting gates can be eliminated by moving the output to input.

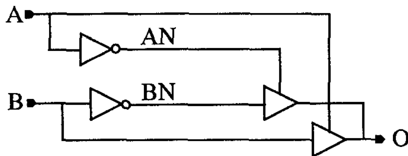


Figure 39 : *The resultant model from Figure 38 after buffer elimination*

3.7.5. *TSD to AND-OR conversion*

AND-OR conversion is a very important piece of simplification. This is also how MUX-es are recognized. Suppose we have a N (N>1) dotted tristate drivers with enable inputs $\{e_1,...,e_N\}$ and data inputs $\{d_1,...,d_N\}$.

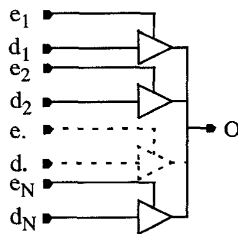


Figure 40 : *A TSD circuit*

The following two criteria are used for turning this TSD circuit to AND-OR circuit:

1. there should be no possibility of a floating state at the output (prove that $e_1 = e_2 = \ldots = e_n = 0$ is not possible) and,
2. if two or more TSDs are on they must drive the same value to the output ($\forall i, j, i \neq j$ show that $e_i = e_j = 1$ and $d_i = \overline{d_j}$ is not possible).

If above two conditions are met, the dotted TSD circuit can be turned into a AND-OR circuit as shown in Figure 41.

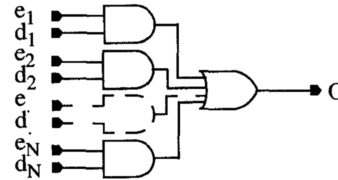


Figure 41 : *Final AND-OR circuit*

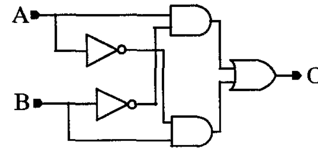


Figure 42 : *After AND-OR conversion of model in Figure 39*

3.8. *Pass-gate Factorization*

Path Tracing algorithm explicitly enumerates all paths to form AND-OR expressions. This approach works well in most cases. However, for pass transistor trees the number of paths become large and the model size increases. To contain the model size, we can keep the tree as it is and convert pass gates to TSDs. However, boolean analysis on TSDs is a little more complex than on AND-OR gates. We solve this dilemma by doing an explicit enumeration of all paths, but performing a factorization of paths to recover the original tree structure. This is explained next with an example.

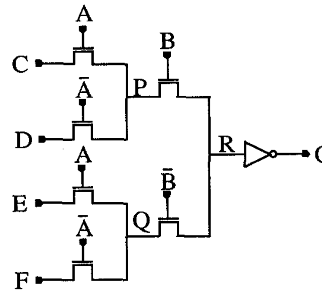


Figure 43 : *A pass transistor tree network*

The paths {R-P-C, R-P-D, R-Q-E, R-Q-F} are enumerated first and after boolean analysis is performed, they can be factorized as R-{P-{C,D},Q-{E,F}} and collapsed back to original structure. After the factorization step, the model is a tree structure of TSDs. Without this step it would have been a two level structure of AND-TSDs.

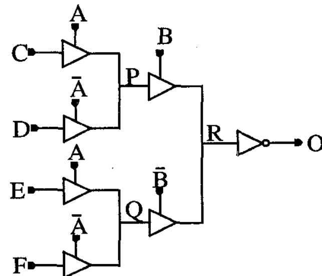


Figure 44 : *Coversion to gate level structure of Figure 43 after factorization*

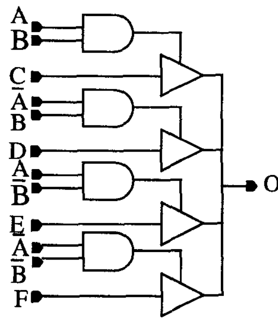


Figure 45 : *Coversion to gate level structure of Figure 43 without factorization*

4. Implementation

GateMaker was implemented in C. It has multiple source and destination interfaces and stands at approximately 15K lines of code. The most costly step in GateMaker is boolean analysis. Despite the frequency with which it is invoked, run time is still not an issue. Running on a RS6000 model 550, a fully custom 64 bit Fixed Point Unit has been extracted in less than 10 minutes of CPU time.

5. Conclusions

In high performance digital CMOS circuits, there are many custom components that are needed to be converted to gate level model for logic simulation, formal verification and most notably Automatic Test Pattern Generation. GateMaker addresses this problem and guarantees that the patterns created by ATPG on the gate level model exercise all switch level paths and makes their effect sensitized to an observable port.

GateMaker preserves the switch level structures as much as possible through series parallel compression of transistors and factorization of paths.

GateMaker imbeds a boolean analysis tool that can perform a variety of checks necessary to simplify tristate driver based designs to AND-OR circuits. The boolean analysis also helps in containing the exponential blow up in path enumeration in certain circumstances.

GateMaker also imbeds some transformations to simplify the model without compromising the test pattern quality. Simplification affects fault coverage computation and therefore all simplification stages are optional.

GateMaker brings uniformity in modeling process and the automated process saves several man months in a design cycle.

6. Acknowledgments

The author would like to thank Derek Beatty, Charlie Malley, Madhu Reddy of the PowerPC Development Center and Gill Vandling, Lori Smudde, Johnny Leblanc of IBM for their invaluable inputs and significant contributions in development of GateMaker. The author is also indebted to Andreas Kuehlmann and Vijay Iyengar for stimulating discussions.

7. References

[1] G. Ditlow, W. Donath and A. Ruehli, "Logic equations for MOSFET circuits", IEEE International Symposium on Circuits and Systems, pp. 752-755, May 1983

[2] Z. Barzilai, L. Huisman, G. M. Silberman, D. T. Tang and L. S. Woo, "Simulating pass transistor circuits using logic simulation machines", Design Automation Conference, pp. 157-163, June 1983

[3] R. E. Bryant, "Boolean analysis of MOS circuits", IEEE Transactions in Computer Aided Design, vol. 6, pp. 634-649, July 1987

[4] D. T. Blaauw, D. G. Saab, P. Banerjee and J. Abraham, "Functional abstraction of logic gates for switch level simulation", European Conference on Design Automation, pp. 329-333, February 1991

[5] R. E. Bryant, "Extraction of gate level models from transistor circuits by four valued symbolic analysis", International Conference in Computer-Aided Design, pp. 350-353, November 1991

[6] R. E. Bryant, D. Beatty and K. Brace, "COSMOS: A compiled code simulator for MOS circuits", Design Automation Conference, pp. 9-16, 1987

[7] A. Kuehlmann, D.I.Cheng, A. Srinivasan and D. P. Lapotin, "Error diagnosis for transistor level verification", Design Automation Conference, pp. 218-224, June 1994

[8] N. Weste and Kamran Eshraghian, "Principles of CMOS VLSI design", Addison Wesley Publishing Company, ISBN 0-201-53376-6

[9] L. G. Heller et. al., "Cascode Voltage Switch Logic: A differential CMOS family", International Symposium on Solid State Circuits, pp. 16-17, 1984

[10] K. Yano et. al. "Top-Down pass-transistor logic design", IEEE JSSC, June 1996, pp. 792-803

[11] K. Yano et. al, "A 3.8 ns CMOS 16x16 bit multiplier using CPL", IEEE JSSC, April 1990, pp. 388-395

[12] M. Suzuki et al., "A 1.5ns 32b CMOS ALU in double pass transistor logic", IEEE Journal of Solid State Circuits, pp. 1145-1151, Nov 1993

[13] V. Friedman et. al., "Dynamic logic CMOS circuits", IEEE Journal of Solid State Circuits, pp. 263-266, April 1994