Tools for Verifying Integrated Circuit Designs

Clark M. Baker and Chris Terman, Massachusetts Institute of Technology

So, you've just completed the design and layout of a new architecture that will revolutionize computing, and you'd like to know if it will work before sending it off to be manufactured...

During the past 18 months, many designers at M.I.T. have found themselves in a similar situation. This paper describes a set of tools developed by the authors (Baker 1980) (Terman in preparation) to help designers make the best use of the occasional manufacturing opportunities. Our goal was to provide a variety of feedback on the different aspects of a design, and do it in a timely enough fashion that the designer would be encouraged to repeat the cycle until his design passed through unscathed. Many of the tools evolved from their original conception under pressure from our rather vocal community of users. The intent is to accumulate a repertoire of programs that incorporate all the checks that a designer might wish to do by hand.

Many of the aspects of design verification represented in our tools would be more appropriate in earlier phases of the design process; in the "ounce of prevention" versus "pound of cure" controversy, the tools fall mostly into the latter category because they are used so late in the design cycle. A lot of research is currently being directed toward the construction of ideal design environments. In a couple of years we would like to report that many of our tools have fallen into disuse as their functions were subsumed in new designer support systems. Until that time, however, these tools represent the only opportunity for most M.I.T. designers to run their designs through a gauntlet of tests that catch most ordinary design errors. We are not alone: many of the tools (or their descendants) are in use at Stanford, Carnegie-Mellon, Xerox PARC, and other institutions where small design teams (often one person) are responsible for seeing a design through from conception to final layout.

The organization of our family of verification tools is shown in Figure 1.

During the early stages of design, heavy use is made of checkplots and the design rule checker as subcomponents of the design are laid out and debugged. Once a complete design is ready for testing, the source file (usually stored in Caltech Intermediate Form) is expanded and scrutinized one last time by the design rule checker to ensure that no last-minute errors have crept into the layout. If the layout passes muster, an electrical network is derived from the mask information by the network extractor. This network can be checked for violations of certain electrical rules and then simulated with a variety of inputs to verify correct operation.

The next three sections describe the major components of the verification system in more detail. The last section reports the performance of the system components.

Design-Rule Checks

Due to physical limitations in the manufacturing process, chip designers must obey a set of geometrical constraints (design rules) when laying out their chips. Most of these constraints specify minimum line widths, minimum spacing between lines, and minimum extensions (the minimum amount by which one layer must overlap another). The basic design rules for a given process are usually straightforward, but complications occur when certain deviations from the basic rules are added to allow the designer to save space in special situations. The description of the full set of design rules for a given process is often many pages long. However, if the designer is willing to forego exploiting the details of the current manufacturing processes, it is possible to adopt a simplified set of design rules that is appropriate for a wide variety of current and future processes. In their book on VLSI design, Mead and Conway (Mead and Conway 1980) have specified one such set of simplified design rules for nMOS processes. Their set of lambda-based design rules is checked by the design rule checker described in this section.

Most design rule checkers are actually geometry engines. The input is a list of polygons (each associated with a specific mask layer) and a sequence of commands describing operations on all the objects of one or more layers. Typical commands include the merging of all intersecting polygons, intersections, unions, and differences of layers, along with width and spacing checks. Checking whole chips can involve manipulating quite a large data base of polygons. Care must be used to ensure that not too much time is spent. The design rule checker described below uses a different approach to improve the turnaround time and capacity of the checking process.

All the basic design rules are local, in that they specify only minimum widths and spacings. In the set of design rules we wish to check, the largest minimum width or spacing is three lambda. Therefore, there should be sufficient information in a four-lambda by four-lambda "window" to determine whether any design rules are violated in that window. If a checkplot of a chip is drawn on one-lambda graph paper, then successive windows can be examined by passing a four-

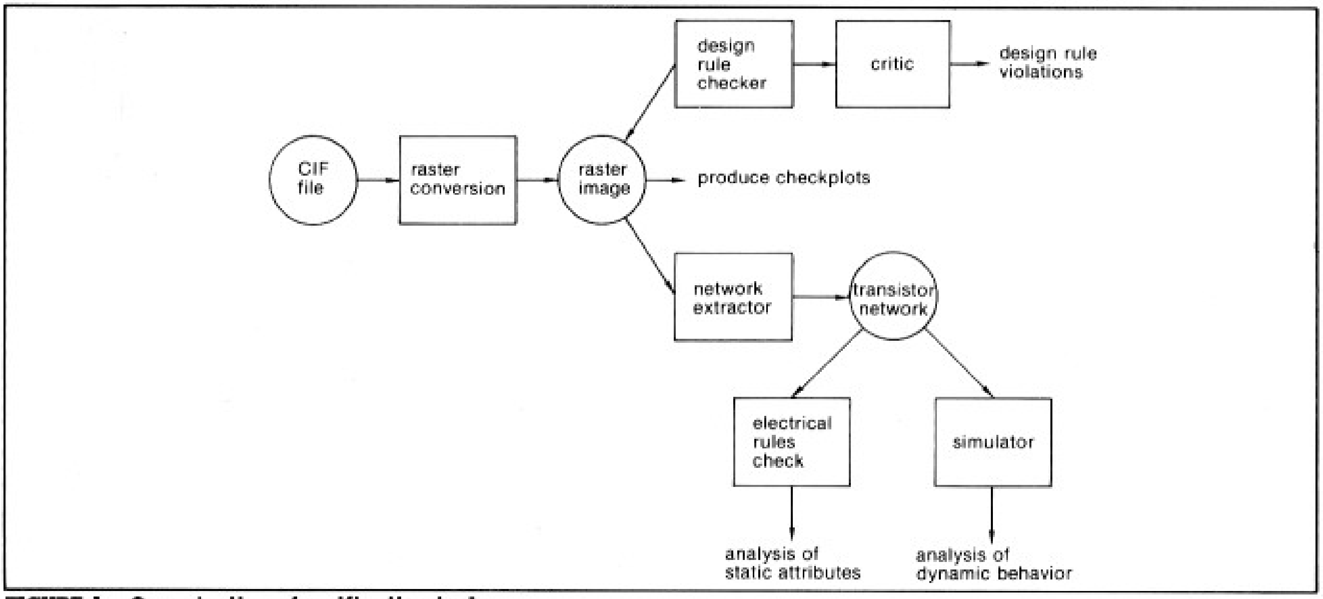

FIGURE 1. Organization of verification tools.

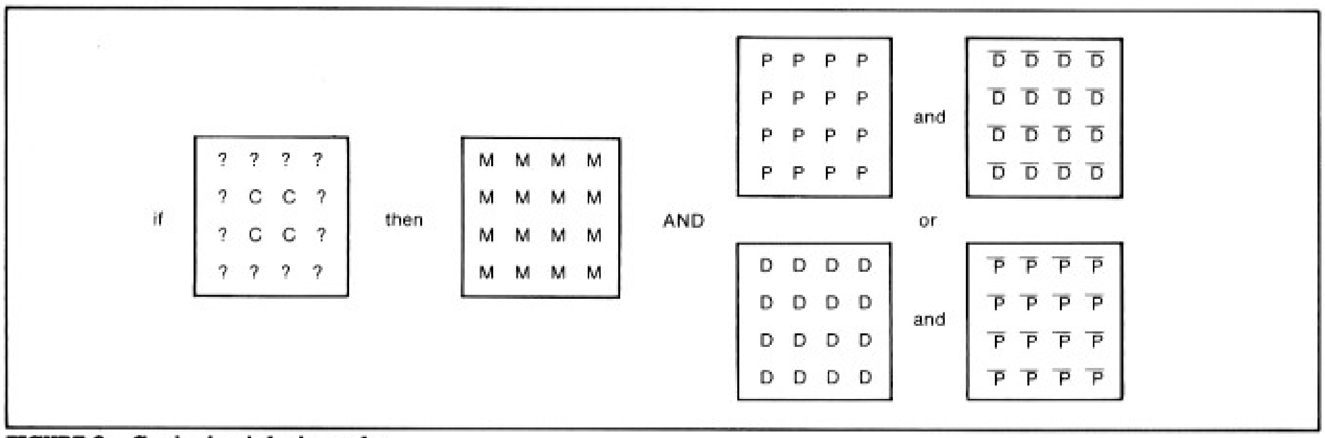

FIGURE 2. Contact cut design rule.

by-four aperture over the plot, moving one square at a time, left to right, top to bottom. If every four-by-four glimpse of the chip obeys the design rules, then the chip as a whole cannot contain any design rule violations.

For example, consider the possible windows for performing a minimum width check for the metal layer (minimum width = 3 lambda). First, look at the problem in one dimension. There are 16 possible four-by-one windows, eight of which are legal (M=metal, W=white space): WWWW, WWWM, WWMM, WMMM, MMMM, MMMW, MMWW, and MWWW. Any other combination represents a width error. It might seem that checking a chip twice, one in the X direction, and then in the Y direction, using a four-by-one window, would be sufficient; however, that method does not check diagonal width or spacing. A four-by-four window is necessary for complete checking.

In addition to the single-layer checks described above, some more complicated multi-layer checks are needed. One typical check pertains to contact cuts. Contact cuts must be surrounded by at least one lambda of metal. If the center four squares of the window are all contact cut, then the whole window must contain metal. If it does not, then an error should be reported. Design rules governing transistors can be checked in a similar way.

The heart of the design rule checker is implemented as a procedure that examines a four-by-four window and reports any errors it discovers. This function is called once for every position on the chip. The errors are accumulated in a file showing which rule was violated, the window containing the error and the coordinates of the window.

The design rules are embodied by the procedure in two ways: tabulation and special purpose code. Two tables are used to perform all width checks (spacing checks are simply width checks of white space). One table indicates which four-by-four windows contain violations of a 3-lambda width rule; the other indicates which three-by-three windows contain violations of a 2-lambda width rule. The first table is for checking metal spacing/width, and the second table is for poly and diffusion checks. Only a single bit (valid/not valid) needs to be stored for each possible window, so the tables are fairly compact. For example, there are 2^{16} possible four-by-four windows, so the corresponding table occupies 2^{16} bits or 8k bytes. Special-purpose code is used to check rules that involve multiple layers because the number of possible windows in these cases prohibits the use of tables.

The initial implementation of this scheme reported spacing errors between mask features that were electrically connected. In the simplest case, these are caused by notches in wires (though some people consider these notches to be real errors); see Figure 3.

Unfortunately, connectivity analysis during design rule checking can become quite complicated, as shown by the example in Figure 4.

In order to eliminate all such false errors, a complete connectivity analysis, like the one performed by the circuit extractor (described below), would be required. A simpler solution that eliminates almost all the false error reports is to include a critic subroutine that performs a connectivity check on a small region (10 lambda by 10 lambda) surrounding any window reported by the checker to contain an error. The additional overhead incurred by the critic is not noticeable, because the number of windows that must be examined by the critic is quite small compared to the total number of windows examined by the checker.

When design rule checks are performed on a whole chip, many of the errors that are reported are actually due to only a small number of errors in symbols that were replicated many times. The checker has a post-processor that attempts to group all errors resulting from the replication of a symbol into a single error message. For each error type (e.g., diffusion spacing), the coordinates of all such reported errors are examined to see if some subset could correspond to replication of a symbol in one or two dimensions. If so, a single error is reported for that subset with information about replication counts, thus reducing the amount of output the designer must examine.

The raster image is generated on the fly and passed to the design rule checker. Since the design rule checker needs only four raster scan lines to perform its checks, older portions of the raster image can be discarded as checking progresses. The current algorithms for raster image generation and design rule checking are reasonably fast, but an increase in speed by a factor of 50 could be gained by implementing the inner loops of both in hardware. The algorithms in both cases are simple enough to be candidates for implementation using only a modest number of MSI chips (or perhaps a single VLSI chip!).

A few limitations of the design rule checker should be mentioned here. The algorithms outlined above are applicable only to dimensional checks. Other design rules, e.g., those involving electrical considerations such as current through a via, cannot be handled. The algorithms are also sensitive to the size of the minimum feature that must be checked. The use of a half-lambda grid would quadruple the number of elements in each window, for example, and cause a corresponding increase in the window processing time. Finally, reducing the mask information to a raster image requires approximating diagonal lines as a "staircase" of raster elements, thus making it impossible to do certain width and spacing checks correctly. Despite these limitations, the programs have been very useful in locating many design rule violations.

Circuit Extraction

The next step is extraction of the actual electrical circuit from the mask descriptions. A major goal is to make this process completely automatic. The designer should not be required to indicate where the transistors are, or where connections exist. The final output is an electrical network consisting of an unordered list of transistors, each one consisting of gate, source, and drain nodes. Other information, such as the length/width ratio for each transistor and the area by layer for each node, is also accumulated during this phase.

The simple extraction algorithm is based on a raster scan approach. The chip is examined in raster scan order (left to right, top to bottom), looking through an L-shaped window containing three raster elements: the current cell, the cell to the left, and the cell above. Using only this information, it is possible to follow connectivity and locate transistors.

There are four layers for which connectivity has to be followed: the original metal layer, M (metal wires); the original polysilicon layer, P (polysilicon wires); a derived layer, D, formed by subtracting polysilicon from diffusion (diffusion wires); and a derived layer, T, formed by intersecting diffusion and polysilicon (transistor gate regions). To follow the connectivity of these layers, the program examines the L-shaped window for each layer in turn and decides between four courses of action:

(1) The current cell is empty. Do nothing.

(2) The layer is present in the current cell, but not in the cells to the left or above. The upper left corner of a new electrical node for the layer has been located and is assigned a new (unique) node number.

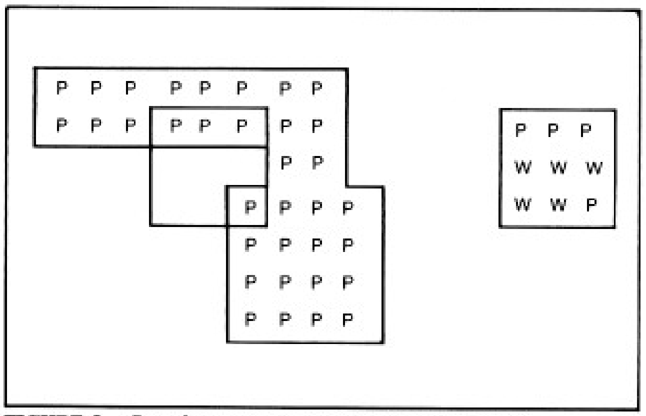

FIGURE 3. Spacing error that should be reported.

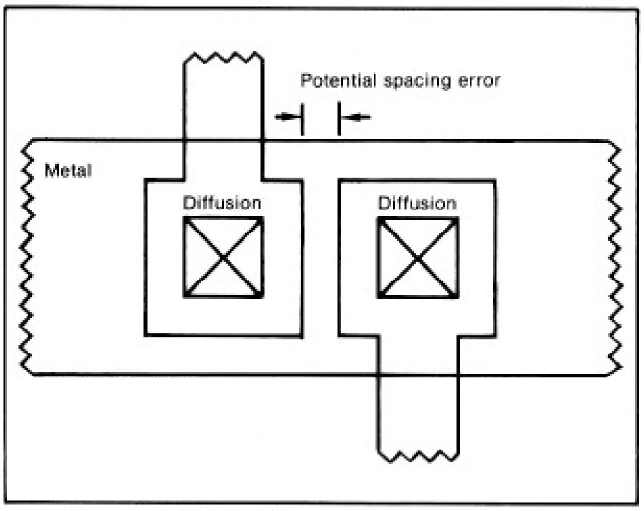

FIGURE 4. Complicated false error.

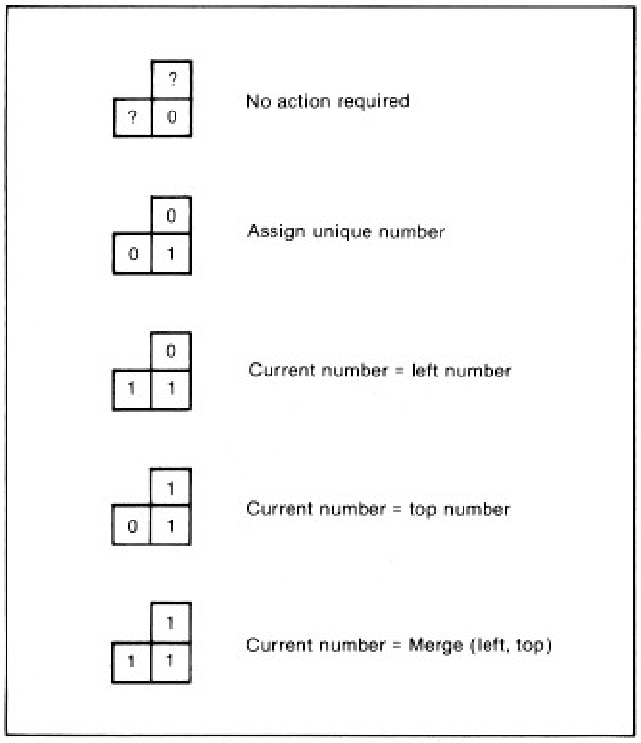
?
? 0
No action required
0
0 1
Assign unique number
0
1 1
Current number = left number
1
0 1
Current number = top number
1
1 1
Current number = Merge (left, top)

FIGURE 5. Basic raster-scan connectivity following algorithm.

(3) The layer is present in the current cell and in one (but not both) of the cells to the left or above. The layer in the current cell is just an extension of the electrical node already found in the neighboring cell, and therefore is given the same node number.

(4) The layer is present in all three cells. If the node number of the layer in the left cell is the same as the node number of the layer in the cell above, then that number is also the number of the current cell. If the two node numbers are different, then two nodes that appeared distinct previously are really part of the same node. In this case, the two node numbers are merged, and the layer in the current node is assigned the merged value.

Merging two nodes requires picking one node as the result and replacing all references to the other node with references to the result. A separate file of obsolete node numbers and the corresponding correct numbers is kept for later use.

In order to account for vias between layers, an additional check of the current cell is made to determine whether contact cut, M, and either P or D is present. If so, the nodes for the M and the P or D layers are merged as in step (4) above. A similar method can be used for buried contacts.

The problem of finding transistors is split into two phases. Transistors are non-local in that the source can be arbitrarily far from the drain, and a whole transistor may not fit into a single window. Therefore, pieces of transistors (i.e., source and drain connections) are located during the first phase, and then all the pieces are combined into whole transistors during the second phase. A transistor piece contains the node numbers of the T, P, and D layers, along with a bit indicating whether the piece was implanted. Any one of four possible window configurations indicates that a transistor piece has

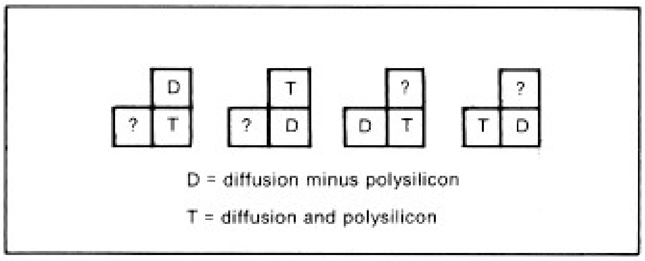
D
? T
T
? D
?
D T
?
T D
D = diffusion minus polysilicon
T = diffusion and polysilicon

FIGURE 6. Basic nMOS transistor finding algorithm.

been found. The current cell must contain either T or D (it cannot contain both), and either the left or top cell must contain the other (D or T). When this configuration is detected, a transistor record is written out to a file and saved for later processing.

After one pass has been made through the raster image of the chip, two files have been created: a file of merged-out node numbers and a file of transistor pieces. Due to the merging process, some of the node numbers in the transistor pieces may have been subsequently merged into other node numbers. Therefore, an additional pass is made through the transistor pieces, updating any old node numbers to their final value. Next, the pieces are sorted by their T numbers (with duplicate records excluded), bringing all of the pieces with the same transistor gate region together. One pass through the transistor piece file is sufficient to generate a final file of transistors.

Some errors can be detected during circuit extraction. Node numbers that were created but never merged, and which do not appear in the list of transistors, are the result of unnecessary polygons on the chip. These nodes are usually found to be part of the designer's logo, but they may also result from layout errors. If the designer can supply a list of signal names along with X and Y coordinates and the associated layer, the program can perform two additional important checks. First, it can ensure that signal paths with different names are not connected together, and second, it can guarantee that signal paths with the same name are connected together. The signal names are also passed along to subsequent programs so that the designer can refer to nodes using his names instead of the automatically generated node numbers. The extra checking provided by the use of signal names led many designers to include these names in their mask files (e.g., by use of the CIF user extension facility).

A few simple extensions should be noted. It is useful to extract the areas, lengths, and widths of each layer for each node in the circuit, for later use by the static evaluator and the simulator. The static evaluator uses the length and width to calculate pullup/pulldown ratios, and the simulator uses the area and perimeter information to calculate node capacitances (used in charge sharing and timing calculations). It would also be nice if node resistances could be calculated, but the raster approach does not help solve this difficult problem.

An obvious speed improvement in the circuit extraction process would be to intersect the polygon data directly to obtain connectivity information. A simple version (allowing only orthogonal rectangles) of the extractor was constructed that processed designs in this manner. For large designs the speed increase was quite marked (from 4 hours to 15 minutes). However, the implementation has lingering bugs that have proven quite difficult to track down. Moreover, accommodating more exotic mask geometries (diagonal lines, polygons) would complicate an already complex algorithm even further. On the other hand, the raster approach was easily expanded to allow polygons because of the modular nature of the algorithm: geometries need only be handled by the rasterizer, and the remainder of the algorithms use only the resulting raster elements. Choosing to implement circuit extraction using raster elements resulted in a timely implementation, if for no other reason than it kept us out of the quagmire of general-purpose geometry engines.

Like rasterization and design rule checking, circuit extraction is also a prime candidate for hardware implementation. A modest piece of hardware should be able to extract even the largest chips in under an hour.

Static Checks

A variety of checks can be made using the extracted circuit before proceeding to simulation. If the designer has previously entered the circuit schematic, perhaps at an earlier stage in the design process, it is straightforward to compare the extracted circuit with the original. This sort of check is appropriate at development centers where the layout is done (often by a separate team) only after a circuit schematic has been completed by the designers. At M.I.T., most projects are the product of small groups of designers responsible for both design and layout. In this case, layout often proceeds directly from preliminary architectural specifications without creating a complete circuit schematic and entering it into the computer. Thus, with no canonical schematic for comparison, simulation is the only way to verify that the circuit is correct.

However, a collection of "trivial" errors can be discovered without simulation. Because designers in the past often spent much time tracking down errors that could have been detected automatically, a static checker was developed, to be run once over the whole circuit.

These basic checks ensure that each node in the circuit can be potentially pulled up and pulled down. Each depletion-mode transistor is checked to be sure it is used in an appropriate manner. The possibilities are limited (e.g., simple pullup, superbuffer, etc.). A check is also made for threshold drops, and occurrences of two or more threshold drops are flagged as potential errors. Transistors that connect V_{dd} and GND together are located. Using the length and width information, all of the pullup/pulldown ratios are calculated and checked against a range of legal values (taking into account any threshold drops on the gates).

The program that performs these checks is not very complicated, and some checks are dependent on the manufacturing process. It would be reasonable to have a different static checker for each process to detect the specific conditions known to be true and to flag any violations.

Simulation

(Note: The simulator description given here is a shortened version of what was received. Most of what is omitted, including the basic transistor model, is similar to the simulator described in Bryant's article—Ed.)

Nodes are categorized according to how much current they can source or sink, i.e., how they affect neighboring nodes to which they are connected by transistor switches:

input Node is designated input node (e.g., V_{dd} or GND). The value of input nodes can only be changed by explicit simulator commands—the assumption is that they supply enough current to be unaffected by connections (possibly shorts to other inputs) made by transistor switches.

driven Node is connected by closed switches to other driven or input nodes. Driven nodes can affect the value of weak and charged nodes without being affected themselves, but may be forced to an X state if shorted to a driven or input node that has a different logic level.

weak Node is connected to an input node by depletion-mode (implanted) transistor. Weak nodes can affect charged nodes without being affected themselves, but are forced to a driven state when connected to another driven or input node. A weak node returns to the appropriate weak state when completely disconnected from driven or input nodes (i.e., a weak node can never enter the charged state).

charged Node is connected, if at all, only to other charged nodes. Until reconnected to some other part of the network, charged nodes will maintain their current logic state indefinitely (charge storage with no decay). This is the default state of all non-weak nodes.

Table 1 lists the possible node values, each of which has an associated logic state and drive capacity. Note that input and driven nodes are actually assigned similar values; input nodes are recognized by the simulator through a different mechanism.

Four pseudo-values are introduced in this table: DXH, DXL, WXH, and WXL. These values are similar in meaning to DX and WX with regard to the electrical properties of

TABLE 1. Possible node values.

Mnemonic	Logic State	Drive Capacity	Description
DH	1	driven	node is an input, or is connected to an input, specified to be at logic high.
DL	0	driven	node is an input, or is connected to an input, specified to be at logic low.
DX	X	driven	node is part of a possible short circuit path between V_{dd} and GND.
DXH	X	driven	node is connected to DH by a series of transistor switches at least one of which was in an unknown state.
DXL	X	driven	node is connected to DL by a series of transistor switches at least one of which was in an unknown state.
WH	1	weak	node is pulled-up by depletion-mode transistor connected to V_{dd}.
WL	0	weak	node is pulled-down by depletion-mode transistor connected to GND.
WX	X	weak	result of shorting WH and WL, or connecting to either state through a transistor switch in an unknown state.
WXH	X	weak	node is connected to WH by a series of transistor switches at least one of which was in an unknown state.
WXL	X	weak	node is connected to WL by a series of transistor switches at least one of which was in an unknown state.
CH	1	charged	charge nodes maintain state because of associated capacitance.
CL	0	charged	similar to CH.
CX	X	charged	last driven logic state of node was X, or charge sharing failed due to bad capacitance ratio.

a node, and are included only to make the incremental simulation computation easier to implement (see the following section).

What remains to be described is how two or more nodes interact when connected by transistor switches. Rather than treat groups of connected nodes *en masse*, the model specifies interactions of pairs of nodes. The resulting specification is easy to understand, because it is based on only local interactions and can be used to model non-local interactions in a natural way by means of a relaxation computation. For efficiency, the simulator optimizes this calculation when it detects specific simple cases (e.g., when all nodes are connected only by closed switches). The iterative computation that happens during relaxation (when a particular node's value is calculated several times) needs to be done only in complex situations involving the X state. Modelling these situations by relaxation seems far more elegant and simpler to implement than other schemes.

If the transistor switch is closed (gate = 1), then both nodes will eventually have the same electrical potential (the resistance of a closed switch is assumed to be O), i.e., both nodes will be given the same value by the simulator when it is discovered that they are shorted together. What value that should be depends on the initial value and driving capacity of the connected nodes as shown in Table 2 below:

Because the transistor switch is symmetrical, the table is also symmetrical. If more than two nodes are connected by closed transistor switches, the final value for all the nodes can be computed by looking up the first two nodes in the table, taking the result and looking it up with the third node, etc., until all the nodes have been combined. As one might hope, the final answer does not depend on the order in which the nodes were examined.

If two or more charged nodes in different logic states are connected, then we have a situation called charge-sharing (indicated in Table 2 by **). In this case, the final value of the nodes depends on their relative capacitances. For example, if a large (high-capacitance) node such as a data bus were connected by a pass transistor to a small node such as the input to a register cell, then the small node would "share" the charge of the large node as its final value, regardless of the charge it had initially. The simulator runs through the connected nodes, adding up the capacitances for each logic level. If the ratio of the total capacitance for one level to sum of the others is greater than 3:1, then that level becomes the final state; otherwise, the result is CX. The choice of 3:1 is arbitrary, and was intended to be conservative in situations in which nodes of more or less equal size might become connected.

If two nodes are connected by a transistor switch in an unknown state where the switch may be open, closed, or resistive, then the two nodes may have different final values if they have different driving capacities. This is reflected in Table 3, which specifies the final value of the source node given initial values for the source and drain (source is NODE1 and drain is NODE2).

To determine the final value for the drain node, use the initial value of the drain as NODE1 and the final value of the source node just read from the table as NODE2.

The simulator uses similar tables and computations (with a few optimizations) when performing a basic simulation step. The details are described in the following section.

The Simulation Algorithm

A basic simulation step starts with a list, called the "event list", of nodes whose values have been changed by the designer since the last simulation step. The first node is taken from the event list, its new value is calculated using the model described in the previous section, and any neighboring nodes affected by the new value are added to the end of the list. This simple calculation is repeated until the event list is empty, indicating that the network has settled.

The calculation of a node's new value can be done simply by looking at the values of its neighbors and using the tables above. However, if a group of nodes is connected by closed switches, then eventually they will all reach the same final value. The simulator calculates this final value simultaneously for all the nodes in the group. Even though this optimization requires some extra programming effort, it is advantageous because connected groups of nodes are quite common. The optimized simulation calculation takes place in two phases: (1) determining the new value for the group,

		NODE2												
		DH	DL	DX	DXH	DXL	WH	XL	WX	WXH	WXL	CH	CL	CX
	DH	DH												
	DL	DX	DL											
	DX	DX	DX	DX										
	DXH	DH	DX	DX	DXH									
N	DXL	DX	DL	DX	DX	DXL								
O	WH	DH	DL	DX	$\boxed{\text{DXH}}$	DXL	WH							
D	WL	DH	DL	DX	DXH	$\boxed{\text{DXL}}$	WX	WL						
E	WX	DH	DL	DX	DXH	DXL	WX	WX	WX					
1	WXH	DH	DL	DX	DXH	DXL	WH	WX	WX	WXH				
	WXL	DH	DL	DX	DXH	DXL	WX	WL	WX	WX	WXL			
	CH	DH	DL	DX	DXH	DXL	WH	WL	WX	$\boxed{\text{WXH}}$	WXL	CH		
	CL	DH	DL	DX	DXH	DXL	WH	WL	WX	WXH	$\boxed{\text{WXL}}$	**	CL	
	CX	DH	DL	DX	DXH	DXL	WH	WL	WX	WXH	WXL	**	**	CX

□ more conservative than necessary **charge sharing (see text)

TABLE 2. Final value of two connected nodes.

		NODE2												
		DH	DL	DX	DXH	DXL	WH	WL	WX	WXH	WXL	CH	CL	CX
	DH	DH	DX	DX	DH	DX	DH	DH	DH	DH	DH	DH	DH	DH
	DL	DX	DL	DX	DX	DL	DL	DL	DL	DL	DL	DL	DL	DL
	DX	DX	DX	DX	DX	DX	DX	DX	DX	DX	DX	DX	DX	DX
	DXH	DXH	DX	DX	DXH	DX	DXH	DXH	DXH	DXH	DXH	DXH	DXH	DXH
N	DXL	DX	DX	DX	DX	DXL	DXL	DXL	DXL	DXL	DXL	DXL	DXL	DXL
O	WH	$\boxed{\text{DXH}}$	DXL	DX	$\boxed{\text{DXH}}$	DXL	WH	WX	WX	WH	WX	WH	WH	WH
D	WL	DXH	$\boxed{\text{DXL}}$	DX	DXH	$\boxed{\text{DXL}}$	WX	WL	WX	WX	WL	WL	WL	WL
E	WX	DXH	DXL	DX	DXH	DXL	WX	WX	WX	WX	WX	WX	WX	WX
1	WXH	DXH	DXL	DX	DXH	DXL	WXH	WX	WX	WXH	WX	WXH	WXH	WXH
	WXL	DXH	DXL	DX	DXH	DXL	WX	WXL	WX	WX	WXL	WXL	WXL	WXL
	CH	DXH	DXL	DX	DXH	DXL	$\boxed{\text{WXH}}$	WXL	WX	$\boxed{\text{WXH}}$	WXL	CH	CX	CX
	CL	DXH	DXL	DX	DXH	DXL	WXH	$\boxed{\text{WXL}}$	WX	WXH	$\boxed{\text{WXL}}$	CX	CL	CX
	CX	DXH	DXL	DX	DXH	DXL	WXH	WXL	WX	WXH	WXL	CX	CX	CX

□ more conservative than necessary

TABLE 3. Final value for partially connected nodes.

and (2) adding any nodes affected by the new value to the end of the event list.

Extensions to the Basic Algorithm

Several more recent additions to the basic simulation algorithm deserve brief mention. One of the more ticklish problems associated with logic-level debugging is the initialization of a complex circuit whose state is represented in a distributed fashion. If all the nodes representing state are initialized to X, this usually causes the nodes controlled by the state variables to also be set to X, and so on. Often the "all X" state is self-consistent and the simulator is unable to make any headway. The problem is especially severe in circuits with feedback, which require multiple cycles to initialize and for which arbitrary choices of values for state variables can be mutually inconsistent and as devastating to correct simulation as choosing X. A simple technique, which falls short of a general solution but has proven useful for initializing many circuits, is to locate all nodes whose current value is CX and set them to CL. A simulation step is performed, and the process repeated until no more CX nodes can be found. If this technique is going to result in a successful initialization, only a few iterations seem to be required. More than 3 or 4 iterations usually indicates that the designer will have to initialize certain nodes to a consistent state by hand.

Another extension to the basic algorithm was to accommodate different transition delays when a node changed value. Multiple-delay simulation is quite common in gate-level simulators, and our implementation borrowed many of the techniques they have developed for managing the event list, etc. The novelty of our approach lies in the automatic calculation of these delays based on the electrical properties of the current interconnections in the network. When one or more connected nodes change value, the delay is dynamically calculated using the impedances of the pullup and/or pulldown paths and the capacitances of the nodes. All effects of the value change are then scheduled at the appropriate time in the future. This timing calculation is crude by comparison to circuit analysis techniques, but offers a quick way to do comparative timing of different paths and thus select those paths deserving of more careful scrutiny.

Performance

Simulator peformance was observed to be independent of layout and circuit size; rather it depended on the amount of electrical activity at a particular point in simulated time. One measure of circuit activity is the number of nodes queued on the event list in a single simulation step. The amount of time to process a single node on the list was surprisingly uniform: the simulator processed between 1000 and 2000 events per second. Thus, simulation of one complete clock cycle (four separate phases) of the SCHEME chip required about 2000 events and 1.6 seconds. Simulating execution of a non-trivial program required about 1200 cycles or 30 minutes of CPU time.

Performance

One important aspect of verification tools is their ability to process designs as expeditiously as possible. Considerable effort was expended to maximize the performance of the current tools. Over their lifetimes, many tools were sped up by factors of 2 or more as the algorithms and data representations improved. Most small projects (less than 1000 lambda square) can be processed from CIF to output files suitable for simulation in less than two hours (see Table 4).

Name	X	Y	Area	Rectangles	Nodes	Transistors
Padout	128	167	21376	144	8	10
Fipo	1192	1202	1432784	15887	888	1796
Scheme	3023	2374	7176602	108948	2412	9448

TABLE 4. Project sizes (dimensions in lambda).

For large projects, overnight turnaround is the norm.

As examples of the performance of the various tools, we present some timings for three projects of varying size.

The smallest project is a simple output pad; the FIPO is an 8-deep, 11-bit wide memory and sorting network; and the SCHEME chip (Holloway, et al. 1979) is a 32-bit LISP microprocessor. A variety of timings are given in the Table 5:

Name	T1	T2	T3	T4	T5	T6	T7	T8
Padout	6	1	1	9	15	2	7	9
Fipo	69	35	56	502	2111	92	438	281
Scheme	559	302	297	2527	16756	1380	7834	1951

T1 = Parse CIF and fully instantiate chip
T2 = Sort fully instantiated chip
T3 = Rasterization
T4 = Design rule check
T5 = Node extract
T6 = Rectangle-based node extract (requires no rasterization)
T7 = Post processing (includes additional sorting)
T8 = Make stipple plots (includes additional rasterization)

TABLE 5. Performance of verification tools.

The timings are CPU seconds on a PDP-11/70 running a UNIX timesharing system; all programs are written in the C language.

Conclusions

One of the most important measures of success for design verification tools is their use by the design community. By this standard, our efforts have been well rewarded: almost all M.I.T. designs submitted for inclusion in recent runs of the Xerox-sponsored multi-project chips have been checked using the tools described in this paper. With only one exception, all the designs had fatal flaws that would have passed unnoticed if not for some error message from one of the programs. Designers cheerfully invest additional time in running these tools. If two or three days will greatly increase the likelihood of success for a project in which three months has already been invested, then those days are not begrudged.

There are several observations worth making by way of conclusion. First, the choice to base the mask-specific algorithms on a raster scan data base has proven to be the correct choice. Using the intermediate files has insulated the design rule checker and network extractor from the vagaries of different input formats. It was easy to expand the number of input formats as the need arose, with no modification of the tools themselves. Raster based algorithms have also been straightforward to implement, and offer the prospect of efficient hardware realizations.

Second, as proposed designs become more complex, there is a greater need for simulation to serve as a test bed in which designers may evaluate their ideas. Many errors detected by the simulator are low-level ones that could have been avoided through the use of more comprehensive design systems, but simulation also uncovers conceptual errors with disturbing regularity. This problem is especially noticeable in an academic community, where many would-be architects are not digital designers.

Finally, it is all to easy to bury the designer under a mountain of error printout. The mountain is caused in part by repetitive layouts that lead to repetitive error reports. The problem can be alleviated by using a critic module to telescope these error reports into a manageable size. If designers know that each new line of the report contains new information, they will be much more conscientious in dealing with the errors. False errors can be another factor leading to designer *ennui* when presented with a stack of error messages; we know of no simple cure for false error messages, except, of course, improving the algorithms.

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0661.

References

Baker, C. May 1980. *Artwork Analysis Tools for VLSI Circuits*, TR-239, Laboratory for Computer Science, M.I.T.

Holloway, J.; Steele, G.; Sussman, G.; and Bell, A. January 1980. *The SCHEME-79 Chip*, Memo No. 599, Artificial Intelligence Laboratory, M.I.T.

Mead, C.A. and Conway, L.A. 1980. *Introduction to VLSI Systems*, Reading: Addison-Wesley.

Terman, C. (thesis in preparation). *Simulation Tools for VLSI Design*, Laboratory for Computer Science, M.I.T.

λ

About the Authors

Chris Terman received his B.A. (1973) degree in Physics from Wesleyan University, and the S.M., E.E. (1978) degrees in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology, where he is currently a PhD candidate. His research interests include computer-aided design tools for VLSI, system architectures, and compiler technology. Chris is currently working on techniques for rapid simulation of large digital circuits and on the architecture of a low-cost design station for VLSI design, layout, and verification. He is also one of the principal architects of the Nu, the personal computer system for the M.I.T. Laboratory for Computer Science.

Clark M. Baker earned his S.B. (1976), S.M. and E.E. (1980) degrees in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology. Clark is a member of the research staff at the M.I.T. Laboratory for Computer Science where he is working on a variety of tools for verifying VLSI designs. His current interests include tools for artwork analysis (design and electrical rules checks) and network extraction. Clark is also working on the software for a low-cost VLSI design station and on verification algorithms suitable for hardware implementation.