Proceedings of MIDCON '84, 1984

HIERARCHICAL LOGIC COMPARISON

Ron Ellickson
J.D. Tygar
Valid Logic Systems, Inc.

INTRODUCTION

Logic comparison determines if two circuits are identical, and identifies the differences if they are not. It is most commonly used for comparing circuits derived from IC layouts with their intended schematics.

Historically layout to logic comparison was done by hand. Large plots were traced by hand and manually compared with the schematics. This method was tedious and error-prone. Several years ago programs were developed that could extract a netlist (in various forms) from an IC layout. This eliminated one source of error. Later, programs appeared that could compare the netlist extracted from the layout with a manually-entered netlist. This reduced the possibility of comparison errors, but depended on the quality of the translation from schematic to netlist. Now we have programs that capture the schematic of logic diagrams. Both the layout and the intended schematic are available in machine readable form; comparison programs should require no additional data.

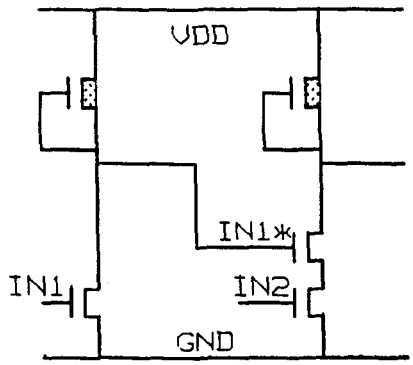
Early circuit comparison techniques compared "flat" representations of the circuitry, ignoring hierarchy in the design process. As logic designs and IC layouts became larger the comparison task became more difficult, since comparison was CPU intensive and the data required increasing amounts of disk space. Even when comparison was computationally feasible, the lack of hierarchy made it difficult to isolate even very simple errors such as a power-ground short on large circuits, since the short could have occured in any subcircuit using power and ground nodes. In addition, the presence of a single error could render many other error reports spurious, and propagate errors far out of proportion to the magnitude of the original problem. Hierarchical circuit comparison is a technique which solves these problems.

SYSTEM INTERFACE GUIDELINES

In older design verification systems, logic designs were entered as netlists. This was not the optimal method, since it is harder to read a netlist than a circuit schematic. Modern systems allow schematics to be entered directly with a graphics editor.

Schematic Input

As an example of schematic input, the logic diagram for a circuit could be entered like this:



Logic Diagram

This schematic shows a NMOS inverter connected to a NAND gate. Note that the schematic serves three purposes: it shows how the circuit works, it documents the circuit for future reference, and it specifies the desired circuit for purposes of comparison.

When the schematic is saved the schematic editor automatically produces a netlist that can be read by other programs. Here is the netlist for this circuit:

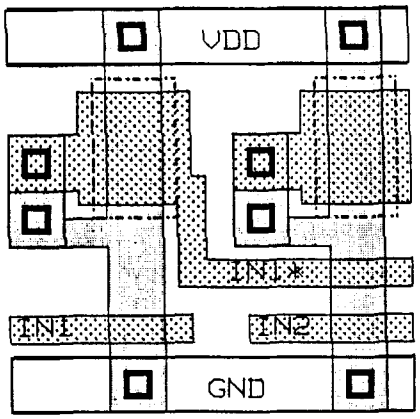
```
% NETS
1 "VDD"
2 "GND"
3 "IN1"
4 "IN1*"
5 "IN2"
6 "UNNAMED$1"
7 "UNNAMED$2"

% ELEMENTS
```

```
DEPL    "S" 1    "D" 4    "G" 4
DEPL    "S" 1    "D" 6    "G" 6
ENH     "S" 2    "D" 4    "G" 3
ENH     "S" 7    "D" 2    "G" 5
ENH     "S" 6    "D" 7    "G" 4
```

Note that this netlist is made up of a list of elements. These elements can be either primitives, such as FETS, or more complex subcircuits. One benefit of this type of netlist is that the format is the same whether it is created from a flat schematic, a hierarchical schematic, or several schematics compiled into one -- which is necessary if a single system is to be used with different design styles and methodologies.

IC Layout Input

The layout shown below corresponds to the logic diagram shown above.



NMOS Layout of NAND and Inverter

A netlist can be extracted from this layout by a circuit extractor. Note that the extractor must be hierarchical to support hierarchical compare. (Also, hierarchical extract is faster and uses less disk space than a flat extractor.) If the subcircuits are extracted with pins then the extractor can generate netlists in the same format as the logic diagram.

However, the use of hierarchy introduces some restrictions on a user's design style. Once a cell has been completed and independently verified, the geometry placed over the cell should only include items which do not change the netlist of the cell. Otherwise, the cell will need to be re-extracted.

ALLOWABLE NON-ISOMORPHISM

We sometimes want to report netlists as equivalent even when they are not identical. For example, one might wish to specify that two pins on a device can be swapped. These features can not be supported without a hierarchical design system.

Swapping Pins

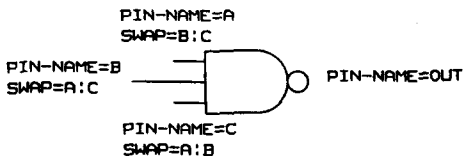
In most MOS devices the drain and souce are interchangeable. In fact the IC layout extractor that the authors use labels the first FET terminal it encounters the drain and labels the second terminal the source. The extractor may pick an assignment of labels different from the MOSFET in the schematic. To handle this situation correctly, the compare program must allow drain and source pins of FETs to be matched.

Our compare program allows this by attaching swap-properties to the pins of the device. For example, on a FET the drain pin would have two properties: the first property is SIG-NAME=DRAIN and the second property is SWAP=SOURCE. Likewise the source pin would have the following properties: PIN-NAME=SOURCE, SWAP=DRAIN.

We generalized the swap of drain and source to swapping of arbitrary pins on any element. For example, suppose a designer wishes to allow swapping the inputs on a 3 input AND gate which has inputs A, B, and C. Then he would attach the following properties to the pins:

INPUT PIN A:	INPUT PIN B:	INPUT PIN C:
PIN-NAME=A;	PIN-NAME=B;	PIN-NAME=C;
SWAP=B|C;	SWAP=A|C;	SWAP=A|B;

The logic representation of this nand gate could look like this:

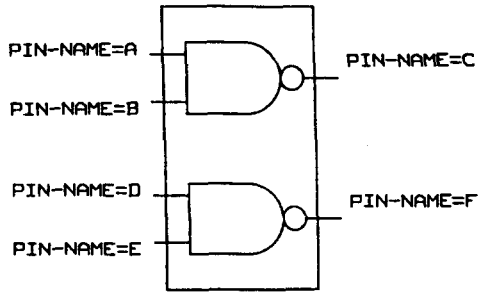


Three Input NAND Gate

Note that it is necessary to have a hierarchical netlist if you want to allow swapping of pins in anything other than the lowest level primitives.

Swapping Sections

Another kind of swapping that is commonly used in design is called section swapping. In the example below there is a subcircuit that has two identical 'sections' in it. Since the sections are identical, it should not matter which one is wired. Section swapping is implemented with properties in much the same way as pin swapping.



SECTION-GROUP=(A,B,C)(D,E,F)

Swappable Sections

This property states that the first section in the subcircuit has pins: A, B, and C. The second section has pins D, E, and F. It also implies that if the two sections were to be swapped then pin A would be swapped with pin D, pin B would be swapped with pin D and pin C would be swapped with pin F. This technique can also used for subcircuits that have more than two swappable sections.

In the current implementation of the program it is not possible to use both pin swap and section swap on the same level of the hierarchy. If you wish to use both then one must be used on one level and the other on a different level. Then the circuit can be flattened with a netlist compiler and compared.

OVERVIEW OF THE ALGORITHM

Hierarchical organization of the circuit speeds up the action of the compare program significantly. But at each level of hierarchy, the compare program still needs to check that the two designs are the same. Moreover, in some cases the circuits may be the same while their hierarchical descriptions are different; in these cases the designs must be flattened and then compared.

The compare program won't be fast unless we can compare flat levels of hierarchy quickly. This is very similar to the famous open problem in computer science: to find an algorithm to quickly check whether two graphs are isomorphic, i.e., whether two sets of abstract objects called vertices are connected in the same way. Much effort has been expended but the general case of this problem has net yet been solved.

The Solution to an "Impossible" Problem.

Fortunately, circuits don't display the same generality that mathematical graphs have. There are only a small number of pins on every component compared to the number of pins in the whole circuit. With this general restriction in mind, we have developed a fast isomorphism test for circuits.

The method depends on random hashing functions. To see how this works, imagine someone gave you two netlists and asked you if they are identical. The first thing you might do is count the transistors per node, and discover there is exactly one node in each circuit with 17 transistors attached to it. Since there is only one such node in each circuit, these two nodes must correspond to each other if the circuits are identical.

The same is true of any other structural property of nodes. If the property depends only on the connections and if only one node in each circuit has the property, then the nodes in the two circuits must match. It does not matter what the property is, so long as it only depends on the connections of the network. We therefore pick a function (called a "hash" function) that gives a wide range of values for typical circuits. If the circuits are identical we get the same values out of both circuits, and every number that appears once in each circuit identifies another match. If the circuits are not identical then some unmatched numbers will appear.

This process is fast, but it does not find all matches. If four nodes in each circuit come up with the same value (say 7) then we don't know which nodes match. You can solve this problem by picking another hash function at random. This new function will probably give different results for the nodes that had the same result on the previous step. If not, you can try again with yet

another randomly chosen hash function. If two nodes ever get distinct hash values, you know they can't match. If several applications of this process do not distinguish the pair of nodes you can guess that the two nodes will match. If you guess correctly, it is easy to verify your answer by constructing the matching relation based on the results of the hash function.

How many distinct hash functions must one try before one is guarranteed that the guess will be good? The answer is provided by a famous conjecture in number theory, the Extended Riemann Hypothesis (ERH). This is the same conjecture that allows computer scientists to quickly determine whether a given integer is prime or composite. ERH states that it is sufficient to test 2 log N different hash functions where N is the number of nets plus the number of nodes. ERH has been verified for its first 250,000,000 instances; so regardless of its truth it is applicable for all conceivable designs.

CONCLUSION

You can compare flat netlists; but if you use hierarchy then you win the following benefits:

Execution is faster.

Incomplete designs can be compared.

Error reporting is localized.

Advanced constructs such as swappable pins and swappable sections can be used.

BIBLIOGRAPHY

[1] E. Bach. "What to Do Until the Witness Comes: Explicit Bounds for Primality Testing and Related Problems." UC Berkeley TR. 1983

[1] C. Hoffman. "Group-Theoretic Algorithms and Graph Isomorphism." Springer-Verlag: New York. 1982

[2] R.Karp, M.Rabin. "Randomized Algorithms." 1976

[3] E. Luks. "Isomorphism of Bounded Valence can be tested in Polynomial Time." Proceedings 21st IEEE FOCS. 1980

[4] R. Mathon. "A Note on the Graph Isomorphism Problem." Information Processing Letters. 1979

[5] G. Miller. "Isomorphism Testing for Graphs of Bounded Genus." Proceeedings 12th ACM STOC. 1980

[6] G. Miller. "On the n log n Isomorphism Technique." Proceedings 10th ACM STOC. 1978

[7] "Random Hashing." 7th IEEE FOCS. 1976

[8] R. Read, D. Corneil. "The Graph Isomorphism Disease." Journal of Graph Theory. 1977

[9] J. Tygar, R. Ellickson. "Graph Isomorphism, The Extended Riemann Hypothesis, and Netlist Comparators." To be published.