The Gemini Users Guide

Carl Ebeling
Neil McKenzie
Larry McMurchie

Northwest Laboratory for Integrated Systems
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

March 15, 1994

1 Introduction

Gemini is a program for comparing circuit wirelists. Normally it is used to verify circuit layout by comparing a wirelist extracted from a circuit layout to a specification wirelist that is assumed to be correct. This kind of verification is commonly known in industry parlance as LVS (layout versus schematic).

Gemini is able to determine very quickly whether or not the two circuits are equivalent. If the circuits are different, then Gemini must work somewhat harder to determine exactly where the differences occur. Gemini will not work well if the circuits are substantially different, but in most cases layout bugs cause only small changes in a circuit. In cases where Gemini cannot isolate the errors, names can be used to help out Gemini.

Although Gemini is used most frequently with MOS circuits, it can be used to compare any kind of circuit that can be represented by the Gemini input format (See Appendix A). Gemini accepts the Berkeley/MIT SIM wirelist format that is widely used to represent MOS circuits. Both the Magic and MEXTRA circuit extractors produce SIM format files. Specification circuits in SIM format may be produced using the schematic capture system WireC which is also distributed by the Laboratory for Integrated Systems.

Gemini does not normally need any naming information to do the circuit comparison, but names are useful annotation when Gemini reports errors. If the input file also contains location information for devices and nets, then Gemini can build a Magic file that marks the location of devices and nets in error. The MEXTRA circuit extractor includes both

label and location information in the circuit wirelist.

Gemini is written in C and runs under the Unix operating system. For comparing large circuits, we recommend running Gemini on a computer with virtual memory support and a large amount of physical memory (16 Mbytes or more).

This user guide is divided into two parts: the mechanics of getting Gemini up and running, and the theory of operation of the the graph isomorphism algorithm.

2 Getting Gemini Up and Running

2.1 Gemini Distribution

Gemini is distributed by the Laboratory for Integrated Systems at the University of Washington. Send requests for Gemini source code to Larry McMurchie, `larry@cs.washington.edu`.

2.2 Building Gemini

Gemini is built by running *make* in the Gemini source directory. Gemini can be compiled using either the portable C compiler `cc` or the Gnu C compiler `gcc`. The make file that is supplied with the Gemini source files uses `cc` by default. Compiling with `gcc` can be accomplished by typing

```
make CC=gcc
```

The optimizer flag `-O`*digit* may be used in `gcc` versions 2.1 and later. Example:

```
make CC='gcc -O3'
```

It may be necessary to include the preprocessor flags `-DNOLABS`[1] and/or `-DNO_RANDOM`[2] to compile Gemini successfully. Example:

```
make CC='gcc -DNOLABS -DNO_RANDOM'
```

[1] `labs()` is the absolute value function that returns a long integer. `Abs()` and `labs()` are interchangeable if the size of the default integer is the same as the size of a long integer. On many machines both are four bytes.

[2] Include this flag if and only if `random()` is not supported by your run-time system. Note that `rand()` and `random()` are **not** interchangeable! See the comments in the source file `gemini.h`.

Much attention has been paid to portability, to allow Gemini to be built using both ANSI and non-ANSI compliant compilers. Please keep us informed on compiler problems you encounter. Send bug reports to `mckenzie@cs.washington.edu`.

2.3 Command Line Syntax

Gemini takes two files in SIM format as its input. The two file names are specified on the Unix command line:

```
% gemini file1.sim file2.sim
```

There are also a variety of options that can be specified on the command line. See the Unix *man* page entry for Gemini for a complete explanation. Here is a brief summary of the command line options. This information is available on-line by the command `gemini -h`.

```
% gemini -h
Gemini 2.7.2 1994/3/15
gemini {-[CFGIcfhmotvz]} {-[DEM]filename} {-[enpswy]number} file1 file2
-C : Collapse like sized devices (-w is implied)
-F : Do not collapse fingered transistors
-G : Use Gemini file format instead of SIM format
-I : Interactive mode
-c : Do not collapse transistor chains
-cw: Print warnings for out-of-order chains
-f : Case-fold net names (ABC==abc)
-h : Help: print this usage summary
-m : Do not use local matching
-o : Do not optimize labeling procedure
-t : Trace execution
-v : Verbose output
-z : Print nets with zero connections
-D<filename> : Output file of name equivalences
-E<filename> : Input file of name equivalences
-M<filename> : Output Magic file with error tiles at mismatched device locations
-e<number> : Set error limit
-n<number> : Set net size limit when printing connections
-p<number> : Set no-progress limit
-s<number> : Set suspect-node limit
-w<number> : Compare transistor sizes using number as tolerance percentage
-y<number> : Compare capacitance using number as tolerance percentage
```

2.4 SIM file syntax

The SIM file syntax is MOS-specific. SIM files have no hierarchy; they are simply a list of devices, capacitors and connections. Normally, designers create circuits using a hierarchical

tool such as WireC that generates SIM file output as a post-process. Gemini uses only a subset of full SIM file syntax. The SIM file entities recognized by Gemini are:

- Unit scale and SIM file format type

  This line appears as the first line in the file with the syntax:

  `| units:` *scale* `tech:` *tech* `format:` *fmt*

  Scale is an integer scale factor that is used with the `-w` option. Transistor lengths and widths in this file are multiplied by this factor and the result is in centimicrons. *Tech* is ignored. *Fmt* selects a format type from the set {MIT, UCB, LBL}. The format type is required if the `-w` option is selected. If no format type is specified, no property information is assumed on the transistor lines. If the entire line is absent then MIT format is assumed. Either UCB or LBL format is required if the `-M` option is selected.

- Transistors

  Transistors are of two types in CMOS: n and p. For backwards compatibility with NMOS, Gemini also accepts e as a synonym for n and d as a synonym for p. Transistor lines appear anywhere in the file. The syntax is:

  `n` *gate source drain* {*substrate*} *length width x y*

  `p` *gate source drain* {*substrate*} *length width x y*

  The arguments *gate, source* and *drain* are required. The argument *substrate*[3] is required if and only if the LBL format type was selected (see previous paragraph). These arguments name the nets (wires) in the circuit. Net names are a non-zero-length string of printable ASCII characters. Tab and space separate arguments. Numbers are neither truncated nor converted: net names `131` and `0131` are distinct. The other arguments are all integers denoting distance in centimicrons and are scaled by the unit scale factor. The arguments *length* and *width* are required if the `-w` option is selected; the arguments *length, width, x* and *y* are required if the `-M` option is selected (see the previous paragraph on unit scale). Total line length must be less than 4K bytes[4].

- User-defined device types

  Beginning with version 2.5, Gemini allows custom devices to be defined and instantiated in the SIM file. User-defined devices require a device definition with the syntax:

  `DEFINE` *usertype param1* {*param2 ... paramn*}

  Usertype must contain two or more alphanumeric characters. *Param1* through *paramn* denote the formal parameter list. Each parameter denotes a terminal in an instance of

[3] Implementation note: internally, all MOS transistors are represented using four terminals. If the LBL format type is not specified, all substrate contacts are connected to the pseudo-net '`No connect`'.

[4] Although schematic capture tools such as WireC sometimes generate very long names as a result of flattening the hierarchy, we believe that the 4K buffer is sufficient for even very deeply nested circuit descriptions.

the device type. Parameter identifiers are used to denote terminal classes and can be repeated in this declaration to indicate membership in the same class. For example, the inputs of a generic two-input NAND gate are symmetric and should be treated equivalently by Gemini's matching algorithm. The corresponding user-defined type may be declared: "`DEFINE nand in in out`".

Device definitions may appear anywhere in the file and may appear more than once as long as they are consistent. All device definitions must be present in the first input file, and are optional in the second input file. If Gemini detects a syntax error, it prints the line number where the error was detected and then aborts.

Device instances are declared by:

usertype arg1 {*arg2 ... argn*} ; *attribute-list*

Arg1 through *argn* are the names of nets in the circuit that are bound to the respective terminal classes in the corresponding device definition. The number of arguments in the instance must match the number of formal parameters in the definition. A semicolon may appear after the last argument and must be separated by whitespace from the last argument. A device attribute list may appear after the semicolon. In the current version of Gemini, the attribute list is simply ignored, but future versions of Gemini may be capable of using the attribute list to assist Gemini's matching algorithm.

- Capacitors

  Capacitors are attributes of nets and are always declared relative to GND. By convention, capacitance is assumed to be in femtofarads. Capacitor lines appear anywhere in the file. The syntax is:

  `C` *net* `GND` *capacitance*

- Net aliases

  Net aliases appear anywhere in the file. The effect is to connect the two nets *net1* and *net2* together. The name *net1* replaces *net2* in all subsequent error information. Syntax:

  `=` *net1 net2*

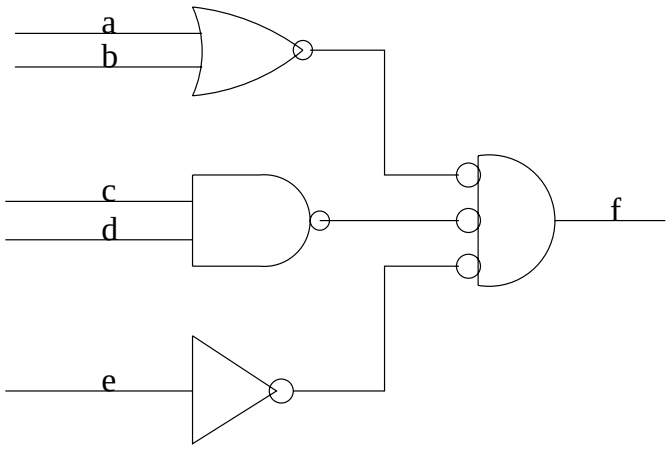
Gemini ignores all lines in the SIM file that start with `A, B, R, v` and semicolon.

For further information on the SIM file format, see the Unix *man* page entry for SIM (section 5, file formats). For reference, Appendix A has a description of the (obsolete) Gemini file format.

2.5 Examples of Running Gemini

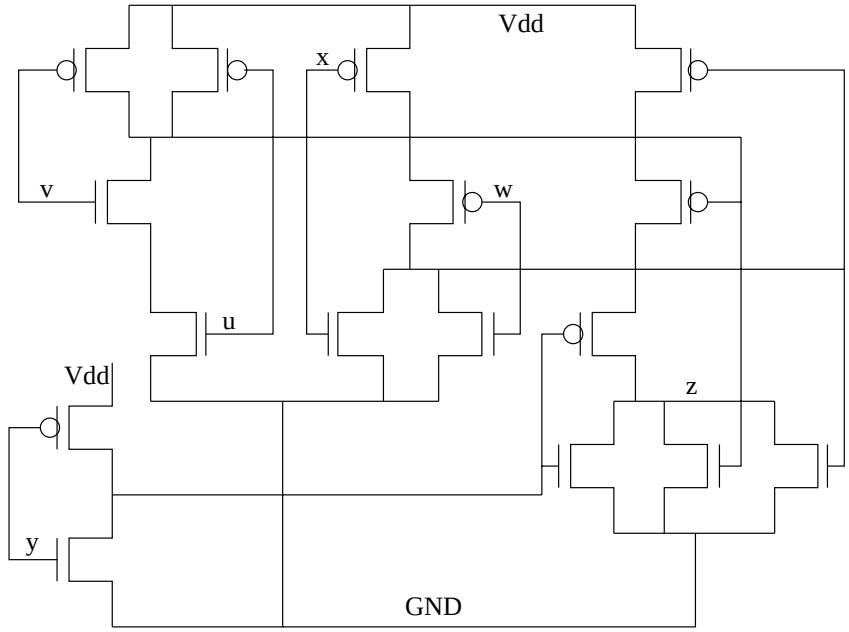
2.5.1 Isomorphic circuits

Here is a CMOS circuit and its SIM equivalent:




```
| units: 100 tech: cmos-s   format: UCB
= GND Gnd
= Vdd VDD
= GND gnd
= Vdd vdd
p a Vdd /cnor_0/_CNOR1_0 2 3 0 0
p b /cnor_0/_CNOR1_0 _CIRCUIT12_1 2 3 0 0
e a GND _CIRCUIT12_1 2 3 0 0
e b _CIRCUIT12_1 GND 2 3 0 0
p _CIRCUIT12_1 Vdd /cnor3_1/_CNOR31_0 2 10 0 0
p _CIRCUIT13_2 /cnor3_1/_CNOR31_0 /cnor3_1/_CNOR32_1 2 3 0 0
p _CIRCUIT11_0 /cnor3_1/_CNOR32_1 f 2 3 0 0
e _CIRCUIT12_1 f GND 2 3 0 0
e _CIRCUIT11_0 f GND 2 3 0 0
e _CIRCUIT13_2 f GND 2 3 0 0
p d Vdd _CIRCUIT13_2 2 3 0 0
p c _CIRCUIT13_2 Vdd 2 3 0 0
e d _CIRCUIT13_2 /cnand_2/_CNAND1_0 2 3 0 0
e c /cnand_2/_CNAND1_0 GND 2 3 0 0
p e Vdd _CIRCUIT11_0 2 3 0 0
e e _CIRCUIT11_0 GND 2 3 0 0
```

Here is a second CMOS circuit and its SIM equivalent:




```
| units: 100 tech: cmos-s   format: UCB
= GND Gnd
= Vdd VDD
= GND gnd
= Vdd vdd
p v Vdd _CIRCUIT22_1 2 3 0 0
p u Vdd _CIRCUIT22_1 2 3 0 0
p y Vdd _CIRCUIT23_2 2 3 0 0
e v _CIRCUIT22_1 _CIRCUIT27_6 2 3 0 0
e y _CIRCUIT23_2 GND 2 3 0 0
e u _CIRCUIT27_6 GND 2 3 0 0
p _CIRCUIT21_0 Vdd _CIRCUIT24_3 2 20 0 0
p _CIRCUIT22_1 _CIRCUIT24_3 _CIRCUIT26_5 2 3 0 0
p x Vdd _CIRCUIT25_4 2 3 0 0
p w _CIRCUIT25_4 _CIRCUIT21_0 2 3 0 0
p _CIRCUIT23_2 _CIRCUIT26_5 z 2 3 0 0
e _CIRCUIT22_1 GND z 2 3 0 0
e w _CIRCUIT21_0 GND 2 3 0 0
e x GND _CIRCUIT21_0 2 3 0 0
e _CIRCUIT21_0 GND z 2 3 0 0
e _CIRCUIT23_2 z GND 2 3 0 0
```

Gemini verifies that the two circuits are isomorphic:

```
% gemini circuit1.sim circuit2.sim
Gemini 2.7.2    1994/3/15

Graph "circuit1.sim": unit scale = 100, format = UCB
        Number of devices: 12
        Number of nets: 11

Graph "circuit2.sim": unit scale = 100, format = UCB
        Number of devices: 12
        Number of nets: 11

These circuits contain some symmetry (33% nodes not yet matched).
Gemini will attempt to find a valid match for symmetrical nodes.
####
19 (79%) matches were found by local matching
All nodes were matched in 10 passes
```

For every thousand nodes matched, Gemini prints the number. Special characters indicate the relative amount of difficulty Gemini is having in resolving symmetrical circuits. `x` indicates that an extra relabeling step was needed. `#` indicates that Gemini was forced to guess a match. `M` indicates a match made by comparing string suffixes.

Gemini is able to compensate for permutations of inputs to NAND and NOR logic gates by collapsing chains of same-type transistors linked by source-drain connections. Chains are processed when the files are read into Gemini, before any matching takes place. The chain is lumped into a single device with one source, one drain and multiple gates. In the example above, both circuits have 16 transistors and three chains: a chain of two n transistors, a chain of two p transistors and a chain of three p transistors. These seven transistors are replaced by three lumped devices, which accounts for Gemini reporting 12 devices instead of 16 when the file is read into memory. Chain collapsing can be disabled using the `-c` option. Example:

```
% gemini -c circuit1.sim circuit2.sim
Gemini 2.7.2    1994/3/15

Graph "circuit1.sim": unit scale = 100, format = UCB
        Number of devices: 16
        Number of nets: 15

Graph "circuit2.sim": unit scale = 100, format = UCB
        Number of devices: 16
        Number of nets: 15

24 (75%) matches were found by local matching
All nodes were matched in 6 passes
```

Gemini may report chains as out-of-order when there is in fact a possible in-order matching. The `-E` option can be used to force a set of matches and change Gemini's conclusion about the two circuits. For instance we can bind `d` in the first circuit to `u` in the second in the equivalence file `equiv`. This will lead Gemini to believe that the circuits match with out-of-order chains.

```
% more equiv
= d u
% gemini circuit1.sim circuit2.sim -Eequiv
Gemini 2.7.2    1994/3/15

Graph "circuit1.sim": unit scale = 100, format = UCB
        Number of devices: 12
        Number of nets: 11

Graph "circuit2.sim": unit scale = 100, format = UCB
        Number of devices: 12
        Number of nets: 11

These circuits contain some symmetry (16% nodes not yet matched).
Gemini will attempt to find a valid match for symmetrical nodes.
##
A total of 4 transistor chains were out of order
20 (83%) matches were found by local matching
All nodes were matched in 10 passes
```

Using an equivalence file may be necessary to match highly symmetrical circuits. False negatives may occur: incorrect information may cause Gemini to conclude that the circuits are different when they are actually isomorphic. However, false positives should be impossible.

A dictionary file, which is the set of net name equivalences deduced, can be created using `-D`*dictfile*. The dictionary file may be generated by one run and used as the equivalence file on a later run.

2.5.2 Properties and Warnings

Gemini allows secondary information, called properties, to be associated with devices or nets and these can be checked when nodes are matched. Examples are the actual size of a transistor and the capacitance of a net. This information is technology dependent and thus must be checked specially for different types of circuits. At present, only CMOS is handled.

The device properties for CMOS are the device width and length and the device location. The net property is the capacitance of the net with respect to ground. Since nets are not explicit in SIM format files, there is no location information for them. If the `-w` flag is

specified, then Gemini checks the properties of devices that are found to match. If `-y` is selected, the properties of nets are checked. For CMOS, the width and length of the devices must agree to within 10% or a warning message is printed. The location information is used in connection with the `-M` flag. For each device that cannot be matched, a 400 x 400 rectangle is placed in the Magic output file at the location of the device.

If device properties are mismatched, Gemini prints out the names of the matching nets and the corresponding property information in a columnar format. Names that are wider than the columns are truncated. Here is an example based on the same SIM files:

```
% gemini -w circuit1.sim circuit2.sim
Gemini 2.7.2    1994/3/15

Graph "circuit1.sim": unit scale = 100, format = UCB
        Number of devices: 12
        Number of nets: 11

Graph "circuit2.sim": unit scale = 100, format = UCB
        Number of devices: 12
        Number of nets: 11

These circuits contain some symmetry (33% nodes not yet matched).
Gemini will attempt to find a valid match for symmetrical nodes.
####
The following transistors do not match in size:
                          circuit1.sim :                          circuit2.sim
(1) Device type p:
s,d:              Vdd                 f :                Vdd                 z
g:       _CIRCUIT12_1  l/w:   200/ 1000 :       _CIRCUIT21_0  l/w:   200/ 2000
g:       _CIRCUIT13_2  l/w:   200/  300 :       _CIRCUIT22_1  l/w:   200/  300
g:       _CIRCUIT11_0  l/w:   200/  300 :       _CIRCUIT23_2  l/w:   200/  300

19 (79%) matches were found by local matching
All nodes were matched in 10 passes
```

2.5.3 Graphs that are not isomorphic

Here we show what Gemini prints when it discovers that the graphs are not isomorphic. To demonstrate, we changed one of the `p` transistors into a `e` transistor and called the new file `circuit1bad.sim`.

Gemini now finds that the two circuit graphs are different. Mismatched devices are indicated by printing the device type and all nets that connect to the device. Mismatched nets are indicated by printing all nets of all devices that connect to the net. Because the output from reporting mismatched nets can be verbose, Gemini prints nothing if the number of

devices is greater than 10. This limit can be changed using the `-n` command line option (see section 2.3). For brevity the example sets this limit to 4. The transistor gates are denoted by [g] and the source/drain pair by [s,d].

```
% gemini circuit1.sim circuit1bad.sim -n 4
Gemini 2.7.2    1994/3/15

Graph "circuit1.sim": unit scale = 100, format = UCB
        Number of devices: 12
        Number of nets: 11

Graph "circuit1bad.sim": unit scale = 100, format = UCB
        Number of devices: 12
        Number of nets: 11
The circuits are different.

These circuits contain some symmetry (33% nodes not yet matched).
Gemini will attempt to find a valid match for symmetrical nodes.
M##
17 (70%) matches were found by local matching

        Graph number 1: circuit1.sim
        ----------------------------
2 NETS could not be matched, possibly because of other unmatched nets:
NET "d" (index 12) 2 connections
  n: (index 13) [g] d, c :: [s,d] _CIRCUIT13_2, GND
  p: (index 10) [g] d :: [s,d] Vdd, _CIRCUIT13_2
NET "c" (index 13) 2 connections
  n: (index 13) [g] d, c :: [s,d] _CIRCUIT13_2, GND
  p: (index 11) [g] c :: [s,d] _CIRCUIT13_2, Vdd
2 DEVICES could not be matched, possibly because of other unmatched devices:
DEVICE p:     connections: (index 10) [g] d :: [s,d] Vdd, _CIRCUIT13_2
DEVICE p:     connections: (index 11) [g] c :: [s,d] _CIRCUIT13_2, Vdd

        Graph number 2: circuit1bad.sim
        -------------------------------
2 NETS definitely do not match:
NET "c" (index 13) 2 connections
  n: (index 13) [g] d, c :: [s,d] _CIRCUIT13_2, GND
  n: (index 11) [g] c :: [s,d] _CIRCUIT13_2, Vdd
NET "d" (index 12) 2 connections
  n: (index 13) [g] d, c :: [s,d] _CIRCUIT13_2, GND
  p: (index 10) [g] d :: [s,d] Vdd, _CIRCUIT13_2

2 DEVICES definitely do not match:
DEVICE n:     connections: (index 11) [g] c :: [s,d] _CIRCUIT13_2, Vdd
DEVICE p:     connections: (index 10) [g] d :: [s,d] Vdd, _CIRCUIT13_2
```

3 Theory of Operation

Here we give a synopsis of the graph isomorphism algorithm. See [Ebel88] for further discussion of the algorithm. See also [EZ83] and [Fitz81].

3.1 Labeling and partitioning

Gemini compares circuits using a graph isomorphism algorithm that works well for circuit graphs. This algorithm partitions graphs using vertex invariants and then iteratively refines the partitioning until all partitions consist of a single node. Since the partitions in the two graphs must correspond, the nodes between the two graphs can be matched.

Gemini partitions the graphs by labeling the nodes: nodes with the same label are in the same partition. The two graphs being compared are labeled in parallel and if the graphs are equivalent, then the number of partitions and their size and labels must be the same. Each time the graphs are relabeled, the partitions in the two graphs are sorted by label and matched. Nodes are matched when singleton partitions are created.

The initial label for nodes is the device type for devices and the number of connections for nets. Each subsequent label is a function of the previous label and the labels of neighboring nodes. The terminal classes are used by the labeling function to distinguish neighbors connected through non-equivalent terminals. Each time the graphs are relabeled, some nodes will usually have unique labels, that is, that they are members of singleton partitions. Using the heuristic that neighbors of uniquely labeled nodes are the most likely to be labeled uniquely next, Gemini uses a local matching algorithm. It tries to deduce matches using strictly local information and avoids the expensive relabeling step. If there are no frontier nodes, then all nodes are relabeled. The `-m` flag disables the local matching feature. The `-o` flag can be used to cause all nodes to be relabeled on every pass. Since circuit graphs are bipartite with nets and devices forming the two parts, labeling alternates between nets and devices.

If two circuit graphs are equivalent, the labeling process very quickly labels each node uniquely. If there are differences, however, this algorithm finds out that the graphs are different but does not do well finding the differences. This happens because nodes that are labeled differently as a result of differences between the graphs affect the labels of neighboring nodes. A small discrepancy may cause many nodes to be labeled differently after only a few passes. (If the node happens to be Vdd, GND or a clock signal, then almost all devices will have different labels in the next pass.) Gemini deals with this problem by removing nodes that may have wrong labels from the labeling process. These nodes are marked so that they will not be included when labels are calculated for neighboring nodes. Gemini detects these suspect nodes when it compares the partitions in the two graphs after each relabeling. All the nodes in partitions that do not correspond are marked suspect.

Labeling continues normally for the other nodes, but if there are too many suspect nodes, then there may not be sufficient labeling information for these nodes to be labeled uniquely. For sufficiently small differences, however, some progress in the labeling process can be made. When no more progress is detected, then the suspect nodes are 'redeemed' and the labeling starts over again. If there is still no progress, Gemini must give up and all remaining unmatched nodes are printed as error nodes.

Progress is measured as the number of unique nodes created in each pass. The number of passes that are made with no progress before Gemini gives up can be set by the `-p` flag which is 2 by default. The number of passes without progress that suspects are 'redeemed' can be set by the `-s` flag which is 1 by default. Changing these parameters will affect how quickly and how well Gemini will pinpoint differences between differing circuits, but the default values usually work pretty well. The `-t` flag can be used to trace the progress of the labeling.

3.2 Ambiguous Circuit Graphs

If Gemini ceases making progress in labeling the graphs before all nodes have been labeled uniquely, it is either because the graphs are ambiguous or because they are different enough that there is not enough information to label all nodes properly as described in the previous section. Gemini handles this situation by making an educated guess as to which of the remaining nodes in the two graphs match. One pair of nodes is arbitrarily picked and labeled with a new value, in effect placing them in a new singleton partition. If Gemini is run in interactive mode, then the user is asked to confirm Gemini's guess. If the guess is not confirmed, Gemini will make another guess. (If the user answers with '!', Gemini will not ask for further confirmation). Labeling then continues and other nodes can usually be distinguished based on this arbitrary matching. If the graphs are equivalent and the labeling has produced an automorphism partitioning (meaning that all partitions contain equivalent nodes), then this procedure will always match equivalent nodes and the subsequent labeling will find the graphs equivalent (subject to other disambiguating matches). In some rare cases, however, Gemini's educated guess will be wrong, and Gemini will think that the circuits are different even though they are not. Gemini informs the user when this happens and the only recourse is for the user to give more information about the circuits by matching labels in the two circuits using the -E flag.

When graphs are different, Gemini will often end up with ambiguous partitions and be forced to make a guess at a pair of nodes that match. Eventually Gemini will either make the wrong guess or find no candidates that can be matched and all the remaining nodes that have not been matched between the two graphs will be printed. If the two circuits are sufficiently different, this set of nodes may contain nodes that are not really different between the two graphs, but usually they will give a good indication of where the problem is.

Although Gemini does not use names when comparing graphs (and does not do much better with them if the graphs are equivalent), names can be quite useful when the graphs are different enough that Gemini has problems identifying the source of error. The user can tell Gemini which nodes in the two circuits must match by entering their names in an equivalence file. This can be used to good effect to reduce the number of nodes in ambiguous partitions.

Good examples of nodes that should be named are the inputs and outputs of bit slices that are similar and clock or control signals. Care must be taken however. Consider the case where the outputs of a PLA have all been grounded (an actual case history from our files). Matching the ground nets for example causes every transistor in the OR plane of the PLA to be mislabeled and the resulting error report from Gemini will not be very elucidating. But leaving the grounds unmatched allows Gemini to throw away some information (the ground net) and still have enough to match all nodes except the PLA outputs, which are exactly the nodes in error.

References

[Fitz81] D. Fitzpatrick. Circuit Analysis from CIF Layouts. University of California, Berkeley, 1981.

[EZ83] C. Ebeling and O. Zajicek. Validating VLSI Circuit Layout by Wirelist Comparison. *Proceedings of ICCAD 1983,* pp. 172-173.

[Ebel88] C. Ebeling. GeminiII: A Second Generation Layout Validation Program. *Proceedings of ICCAD 1988,* pp. 322-325.

A Gemini Input Format

The Gemini input format is a simple but general wirelist syntax that can be used to represent almost any type of circuit. A circuit consists of devices of various types whose terminals (connection points) are connected by nets. A wirelist file simply lists the device types, devices and nets appearing in the circuit. Each type, device and net may be named but these names are usually used only for annotating the output listing. If there is no name for a node, then "*" is used by convention. Individual types, devices and nets are referenced by index (zero-based) according their position in each list.

The Gemini file syntax is:

```
<number-of-types>

<type-0-name> <number-of-terminals> <terminal-class-0> ... <terminal-cl
        ...
<type-n-name> <number-of-terminals> <terminal-class-0> ... <terminal-cl

<number-of-devices> <number-of-nets>

<device-0-name> <type> <net> ... <net> <properties>
        ...
<device-n-name> <type> <net> ... <net> <properties>

<net-0-name> <number-connects> <device>,<terminal> ... <device>,<termin
        ...
<net-n-name> <number-connects> <device>,<terminal> ... <device>,<termin
```

An example file appears at the end of this section.

The first part lists all the device types used in the circuit. Devices are the same type if and only if they are functionally equivalent and have the same number of terminals (connections). For example, a 2 input AND gate is different from a 2 input OR gate or a 3 input AND gate. The terminal classes are (arbitrary) numbers used to group device terminals that are equivalent. For example, an AND gate has two input terminals in one class (since these connections can be interchanged) and an output terminal in another class. NMOS circuits have only two device types: enhancement and depletion mode transistors. CMOS circuits have only two device types: n and p transistors.

Second is the list of circuit devices. Following the device name is the device's type index and a list of terminal connections in the same order as the terminals are given in the type entry. Each connection is given by the index of the net to which the terminal is connected. The remainder of the line is taken as the property string. This is an arbitrary string that

further describes the device but does not affect its type. The property string is technology dependent in general and can be used only if the particular technology is understood by Gemini. NMOS and CMOS are the only technologies specifically recognized. The property string contains the transistor length and width for devices.

Last comes the list of circuit nets. After the net name is the number of connections to the net followed by the list of connections. Each connection consists of a pair of numbers: the index of the device in the device list and the terminal number of the device. Following the connection list is the property string. For CMOS and NMOS technologies, this string is empty.

A complete file for a single NOR gate in NMOS technology is shown below by way of example.

```
2
dep     3       0       1       1
enh     3       0       1       1
3       5
*       0       0       0       1       8.0 2.0
*       1       2       3       0       2.0 2.0
*       1       4       3       0       2.0 2.0
c       4       2,2     1,2     0,1     0,0
vdd     1       0,2
a       1       1,0
gnd     2       2,1     1,1
b       1       2,0
```

This file is equivalent to the following SIM file:

```
d c c vdd 8.00 2.00 r 0 0 16.00
e a gnd c 2.00 2.00 r 0 0 4.00
e b gnd c 2.00 2.00 r 0 0 4.00
```