1986 VLSI Tools:

Still More Works by the Original Artists

Includes these smash hits:

Crystal	by John Ousterhout
Eqntott	by Bob Cmelik
Esim	by Chris Terman (MIT)
Espresso	by Richard Rudell
Magic	by Gordon Hamachi, Robert Mayo, John Ousterhout, Walter Scott, and George Taylor
Meg	by David Wood
Mpack	by Robert Mayo
Mpanda	by Grace Mah
Mpla	by Robert Mayo and Fred Obermeier
Mquilt	by Robert Mayo
Multibus Design Frame	by Gaetano Borriello
Peg	by Gordon Hamachi
Pleasure	by G. D. Micheli and Duksoon Kay
Spice2summary	by Fred Obermeier
Vlsifont	by Robert Mayo

Plus many others...

Walter S. Scott, Robert N. Mayo, Gordon Hamachi,
and John K. Ousterhout, editors

Computer Science Division
EECS Department
University of California at Berkeley

The development of these tools was supported in part by the Defense Advanced Research Projects Agency under contract N00039-84-C-0107, the Semiconductor Research Corporation under grant SRC-82-11-008, and the National Science Foundation under grant ECS-8351961.

TABLE OF CONTENTS

About this distribution.....

This manual describes the programs in the 1986 VLSI Tools Distribution put together by the CS Division of the Department of EECS, University of California, Berkeley. The distribution consists of about twenty programs for designing and analyzing VLSI circuits. The programs were designed to run on both VAXes under the Berkeley 4.3 distribution of Unix, and Suns under their Berkeley 4.2 distribution of Unix. They are available to the public on an internal-use-only basis. To find out how to obtain them, write to

John Ousterhout
CS Division, Dept. of EECS
University of California
Berkeley, CA 94720.

Several other design packages are available from other groups within the department. Inquiries about these programs may be addressed to

Industrial Liaison Program - ERL
Dept. of EECS
University of California
Berkeley, CA 94720.

Highlights

This distribution contains mostly the same programs as our previous distribution. The main change is in the Magic program, which has undergone major revisions. In addition, there are a few new programs. Here's an overview of some of the major tools:

Crystal A timing analyzer that helps the designer find performance problems in his design.

Eqntott Converts a set of logic equations into a truth table format for input to our PLA optimization and layout tools.

Esim An event driven logic-level simulator developed at MIT and distributed with their permission. The version on this tape handles CMOS as well as nMOS.

Espresso A fast new boolean equation minimizer.

Ext2sim Part of the Magic suite of programs. Used for converting the output of Magic's hierarchical extractor into a form usable by other tools on this tape, such as Esim and Crystal.

Magic The second release of our graphical layout editor. Magic has undergone many major changes since the 1985 distribution; they are described in a section below.

Mpack A new release of the *tpack* library for generating semi-regular modules. This version is compatible with Magic layout files. These routines allow module generators to generate layouts by assembling tiles (which are small chunks of layout designed with Magic). The end result is a module generator that can generate

different styles of modules depending upon what set of tiles is used.

Mpanda A technology-independent generator of split and folded PLAs built using Mpack. Used in conjunction with Pleasure.

Mpla A technology-independent generator of ordinary PLAs built using Mpack.

Mquilt A generator of personalized arrays built using Mpack.

Peg A tool that compiles a high-level description of a finite state machine into logic equations. These logic equations can be fed into the PLA tools for automatic layout and optimization of the FSM.

Pleasure Minimizes the area of a PLA by splitting and folding its **and** and **or** planes. Used in conjunction with Panda.

Several of the programs on this tape were developed by authors outside of the Computer Science Division. We wish to thank Prof. Alberto Sangiovanni-Vincentelli of Electrical Engineering and his students for allowing us to distribute their PLA optimization tools Espresso and Pleasure. Panda and Eqntott were developed by Prof. Richard Newton and his students, also in Electrical Engineering. Esim, the switch-level simulator, was developed by Prof. Chris Terman of MIT. Several programs are being distributed courtesy of Prof. Chuck Seitz of Caltech: Cifp, by Chris Kingsley, Magplot, by Wen-King Su, and Plamin by Don Speck. Finally, Shih-Lien Lu of MOSIS-ISI provided the SOS and SCMOS technology files for Magic. We are grateful for the authors' permission to distribute these tools.

Installation Instructions

The tape is written in Unix "tar" program format, 1600 BPI, 20 blocks per record. There are about 20 megabytes of stuff on the tape. Installing the tools on a VAX or Sun running Berkeley Unix is relatively easy. First, **create a new user, "cad".** Do not try to load the tape without a user ~cad: most of the programs will not run correctly. Then change your current directory to ~cad and load all the information from the tape into the ~cad area with the command **tar x**.

This tape contains binaries for either version 4.3 of Berkeley Unix for a VAX, or for Sun's release 2.0 of its version of 4.2 BSD. If you have a VAX running version 4.2 of Berkeley Unix, or one of the new Sun-3's, the installation process involves some recompilation. The details of installation on a particular system are handled by the **INSTALL** script. To install the tools on a VAX running 4.3 Berkeley Unix, type **INSTALL VAX4.3**. To install them on a VAX running 4.2 Berkeley Unix, type **INSTALL VAX4.2**. To install the tools on a Sun-2 workstation, type **INSTALL SUN2**. We don't have an automatic script for installing the tools on a Sun-3 workstation yet; you'll have to recompile the binaries by hand.

The above instructions may be inconvenient if you already have ~cad directories that you've been using. One alternative is to save all the current

~cad information (**mv** the subdirectories to other names), then load the tools, then **mv** any programs that you want back from the old subdirectories. Another alternative is to tar the tape into another area, "cad86" for example, then move individual programs over to try them. Be careful if you use this approach, since many of the programs (like Magic, Cifplot, Mpack, Mquilt, and Mpla) use library information in ~cad/lib.

There's very little machine-specific configuration needed for the tape besides what's done by the INSTALL script. If the normal troff -man macros are not in the standard place (/usr/lib/tmac), the cadman macros in ~cad/man/tmac/tmac.anc will need to be modified to reflect the location of these standard macros. Also, vlsifont expects the Berkeley fonts to be present in the standard place (/usr/lib/vfont). Finally, if you are installing the tools on a VAX, you should set up the file ~cad/lib/displays to reflect the locations and types of graphics terminals in your system.

If you're interested in playing around with any of the other programs, most of their source directories have files called README or ReadMe or something similar. See these files for information on installation and maintenance.

Using the Tools

To use the various tools, have each would-be user add "~cad/bin" to the path in his or her .login or .cshrc file. To get information about the programs, use "cadman": it works just like "man" except that it deals with VLSI CAD tools. In addition to manual pages, many of the tools have longer tutorial-style user manuals in ~cad/doc and its subdirectories. These manuals can be printed with *ditroff* and/or *deqn* and/or *dtbl*. Some of them contain *gremlin* files, however, and so require *grn* to print them. Along with the tape you should get a printed copy of the manuals, in order to save you time running them off.

While ~cad/bin is the only directory that contains binaries, there are several other directories in the ~cad area:

lib	Contains library files used by the various programs.
man	Contains manual pages.
src	Contains the sources for all of the programs.
doc	Contains tutorial-style user manuals for a few of the more complicated programs.
contrib	Contains programs contributed by outside authors to this distribution. Most of these arrived very shortly before this tape was made, so we have neither had time to test them nor integrate them with the rest of the directory structure above.

At Berkeley we create a ~cad/new area that is just like ~cad/bin except that it holds the latest experimental binaries of programs. If you also follow this convention then be sure to include ~cad/new in your search path before ~cad/bin.

Support

These tools work fine for us at Berkeley, but they almost certainly have bugs. The programs are distributed on an as-is basis. We are busy building the next generation of tools, so we can't provide installation assistance, tutorial help, or other support. We continue to welcome comments from our test sites, and we will listen to reports of major bugs discovered by other sites.

Other Machines and Displays

Magic as distributed runs only on VAXes with AED displays and Suns. However, Magic has been ported to various machines by groups outside of U. C. Berkeley. We have not tested or evaluated any of these ports.

Our tape includes modifications for these machines:

- Apollo under DOMAIN/IX (ported by Paul Anderson and Prof. Ronald Lomax at the University of Michigan),
- MassComp (ported by Clem Cole of MassComp),
- IRIS 1400 (ported by Doug Pan and Prof. Mark Linton at Stanford).

Magic also runs on Pyramid machines under their version of 4.2 Unix, and on Gould machines under UTX/32 (Gould port by Prof. Yuval Tamir, UCLA). We appreciate the work that other people have done in porting Magic.

Graphics drivers have also been developed by other groups. The tape contains contributed drivers for these graphics terminals:

- Omega 440 (ported by Warren Jessop, University of Washington),
- Lexidata 3700 (ported by Wm. F. Hargrove, North Carolina Educational Computing Service),
- Jupiter (ported by MITRE-Bedford).

Acknowledgements

In order to turn our programs from academic exercises into useful tools, we depend on designers to use the systems, explain to us their problems, and put up with the problems until we can fix them. We've been very fortunate to have a large and very cooperative group of users. It's impossible to thank them all here, but some of the most helpful early users have been:

Randy Katz and the Fall 1985 CS250 class (UCB).
Peng Ang, Jonathan Greene, and the rest of the LSI Logic Palo Alto Research Laboratory
Norman Jouppi (Digital Equipment Corporation)
Shih-Lien Lu (Information Sciences Institute)
Chuck Seitz (Caltech)
Mark Horowitz (Stanford)
MITRE Integrated Electronics Group (MITRE-Bedford)
The July 1985 MOSIS teacher's course (Information Sciences Institute)
Richard Kenner (NYU)
The Computer Sciences Division of Bolt, Beranek, and Newman

NAME

cadman − run off section of UNIX manual

SYNOPSIS

cadman [−] [**−t**] [section] title ...

DESCRIPTION

Cadman is a program which prints sections of the cad manual. *Section* is an optional arabic section number, i.e. 3, which may be followed by a single letter classifier, i.e. 1m indicating a maintenance type program in section 1. If a section specifier is given *cadman* looks in the that section of the cad manual for the given *titles*. If *section* is omitted, *cadman* searches all sections of the cad manual, giving preference to commands over subroutines in system libraries, and printing the first section it finds, if any.

If the standard output is a teletype, or if the flag − is given, then *cadman* pipes its output through *ul*(1) to create proper underlines for different terminals, and through *more*(1) to crush out needless blank lines and to stop after each page on the screen. Hit space to continue, or a control-D to scroll 12 more lines when the output stops.

The **−t** flag causes *cadman* to arrange for the specified section to be *troff'ed* to the default typsetting output device.

FILES

~cad/man/man?/*

NAME

chksum – checksum a file

SYNOPSIS

chksum

DESCRIPTION

Chksum reads from standard input and prints the character count and checksum on standard output. This checksum and character count matches that of the program used by MOSIS and is useful for checking large CIF files transmitted from on site to another.

AUTHOR

John Foderaro

NAME

cifplot − CIF interpreter and plotter

SYNOPSIS

cifplot [*options*] file1.cif [file2.cif ...]

DESCRIPTION

Cifplot takes a description in CalTech Intermediate Form (CIF) and produces a plot. CIF is a low-level graphics language suitable for describing integrated circuit layouts. Although CIF can be used for other graphics applications, for ease of discussion it will be assumed that CIF is used to describe integrated circuit designs. *Cifplot* interprets any legal CIF 2.0 description including symbol renaming and Delete Definition commands. In addition, a number of local extensions have been added to CIF, including text on plots and include files. These are discussed later. Care has been taken to avoid any arbitrary restrictions on the CIF files that can be plotted.

To get a plot call *cifplot* with the name of the CIF file to be plotted. If the CIF description is divided among several files call *cifplot* with the names of all files to be used. *Cifplot* reads the CIF description from the files in the order that they appear on the command line. Therefore the CIF *End* command should be only in the last file since *cifplot* ignores everything after the *End* command. After reading the CIF description but before plotting, *cifplot* will print a estimate of the size of the plot and then ask if it should continue to produce a plot. Type **y** to proceed and **n** to abort. A typical run might look as follows:

```
% cifplot lib.cif sorter.cif
Window -5700 174000 -76500 168900
Scale: 1 micron is 0.004075 inches
The plot will be 0.610833 feet
Do you want a plot? y
```

After typing **y** *cifplot* will produce a plot on the Benson-Varian plotter.

Cifplot recognizes several command line options. These can be used to change the size and scale of the plot, change default plot options, and to select the output device. Several options may be selected. A dash(-) must precede each option specifier. The following is a list of options that may be included on the command line:

−w *xmin xmax ymin ymax*

(**window**) The -w options specifies the window; by default the window is set to be large enough to contain the entire plot. The windowing commands lets you plot just a small section of your chip, enabling you to see it in better detail. *Xmin*, *xmax*, *ymin*, and *ymax* should be specified in CIF coordinates.

−s *float*

(**scale**) The -s option sets the scale of the plot. By default the scale is set so that the window will fill the whole page. *Float* is a floating point number specifying the number of inches which represents 1 micron. A recommended size is 0.02.

−l *layer_list*

(**layer**) Normally all layers are plotted. The -l option specifies which layers NOT to plot. The *layer_list* consists of the layer names separated by commas, no spaces. There are some reserved names: **allText**, **bbox**, **outline**, **text**, **pointName**, and **symbolName**. Including the layer name **allText** in the list suppresses the plotting of text; **bbox** suppresses the bounding box around symbols. **outline** suppresses the thin outline that borders each layer. The keywords **text**, **pointName**, and **symbolName** suppress the plotting of certain text created by local extension commands. **text** eliminates text created by user extension 2. **pointName** eliminates text created by user extension 94. **symbolName** eliminates text created

by user extension 9. **allText**, **pointName**, and **symbolName** may be abbreviated by **at**, **pn**, and **sn** repectively.

-c *n* (**copies**) Makes *n* copies of the plot. Works only for the Varian and Versatec. Default is 1 copy.

-d *n* (**depth**) This option lets you limit the amount of detail plotted in a hierarchically designed chip. It will only instanciate the plot down *n* levels of calls. Sometimes too much detail can hide important features in a circuit.

-g *n* (**grid**) Draw a grid over the plot with spacing every *n* CIF units.

-h (**half**) Plot at half normal resolution. *(Not yet implemented.)*

-e (**extensions**) Accept only standard CIF. User extensions produce warnings.

-I (**non-Interactive**) Do not ask for confirmation. Always plot.

-L (**List**) Produce a listing of the CIF file on standard output as it is parsed. Not recommended unless debugging hand-coded CIF since CIF code can be rather long.

-a *n* (**approximate**) Approximate a roundflash with an *n*-sided polygon. By default *n* equals 8. (I.e. roundflashes are approximated by octagons.) If *n* equals 0 then output circles for roundflashes. (It is best not to use full circles since they significantly slow down plotting.) *(Full circles not yet implemented.)*

-b *"text"*
(**banner**) Print the text at the top of the plot.

-C (**Comments**) Treat comments as though they were spaces. Sometimes CIF files created at other universities will have several errors due to syntactically incorrect comments. (I.e. the comments may appear in the middle of a CIF command or the comment does not end with a semi-colon.) Of course, CIF files should not have any errors and these comment related errors must be fixed before transmitting the file for fabrication. But many times fixing these errors seems to be more trouble than it is worth, especially if you just want to get a plot. This option is useful in getting rid of many of these comment related syntax errors.

-r (**rotate**) Rotate the plot 90 degrees.

-V (**Varian**) Send output to the varian. (This is the default option.)

-W (**Wide**) Send output directly to the versatec.

-S (**Spool**) Store the output in a temporary file then dump the output quickly onto the Versatec. Makes nice crisp plots; also takes up a lot of disk space.

-T (**Terminal**) Send output to the terminal. (Not yet fully implemented.)

-Gh

-Ga (**Graphics terminal**) Send output to terminal using it's graphics capablities. **-Gh** indicates that the terminal is an HP2648. **-Ga** indicates that the terminal is an AED 512.

-X *basename*
(**eXtractor**) From the CIF file create a circuit description suitable for switch level simulation. It creates two files: *basename*.**sim** which contains the circuit description, and *basename*.**node** which contains the node numbers and their location used in the circuit description.

When this option is invoked no plot is made. Therefore it is advisable not to use any of the other options that deal only with plotting. However, the *window*, *layer*, and *approximate* options are still appropriate. To get a plot of the circuit with the

node numbers call *cifplot* again, without the **-X** option, and include *basename*.**nodes** in the list of CIF files to be plotted. (This file must appear in the list of files before the file with the CIF End command.)

-c *n* (**copies**) The **-c** specifies the number of copies of the plot you would like. This allows you to get many copies of a plot with no extra computation.

-P *pattern_file*

(**Pattern**) The -P option lets you specify your own layers and stipple patterns. *Pattern_file* may contain an arbitrary number of layer descriptors. A layer descriptor is the layer name in double quotes, followed by 8 integers. Each integer specifies 32 bits where ones are black and zeroes are white. Thus the 8 integers specify a 32 by 8 bit stipple pattern. The integers may be in decimal, octal, or hex. Hex numbers start with '0x'; octal numbers start with '0'. The CIF syntax requires that layer names be made up of only uppercase letters and digits, and not longer than four characters. The following is example of a layer description for polysilicon:

```
"NP"  0x08080808 0x04040404 0x02020202 0x01010101
      0x80808080 0x40404040 0x20202020 0x10101010
```

-F *font_file*

(**Font**) The -F option indicates which font you want for your text. The file must be in the directory '/usr/lib/vfont'. The default font is Roman 6 point. Obviously, this option is only useful if you have text on your plot.

-O *filename*

(**Output**) After parsing the CIF files, store an equivalent but easy to parse CIF description in the specified file. This option removes the include and array commands (see next section) and replaces them with equivalent standard CIF statements. The resulting file is suitable for transmission to other facilities for fabrication.

In the definition of CIF provisions were made for local extensions. All extension commands begin with a number. Part of the purpose of these extensions is to test what features would be suitable to include as part of the standard language. But it is important to realize that these extensions are not standard CIF and that many programs interpreting CIF do not recognize them. If you use these extensions it is advisable to create another CIF file using the **-O** options described above before submitting your circuit for fabrication. The following is a list of extensions recognized by *cifplot*.

0I *filename*;

(**Include**) Read from the specified file as though it appeared in place of this command. Include files can be nested up to 6 deep.

0A *s m n dx dy* ;

(**Array**) Repeat symbol *s* *m* times with *dx* spacing in the x-direction and *n* times with *dy* spacing in the y-direction. *s*, *m*, and *n* are unsigned integers. *dx* and *dy* are signed integers in CIF units.

1 *message*;

(**Print**) Print out the message on standard output when it is read.

2 *"text" transform* ;

2C *"text" transform* ;

(**Text on Plot**) *Text* is placed on the plot at the position specified by the transformation. The allowed transformations are the same as the those allowed for the Call command. The transformation affects only the point at which the

beginning of the text is to appear. The text is always plotted horizontally, thus the mirror and rotate transformations are not really of much use. Normally text is placed above and to the right of the reference point. The **2C** command centers the text about the reference point.

9 *name*;
(**Name symbol**) *name* is associated with the current symbol.

94 *name x y*;

94 *name x y layer*;
(**Name point**) *name* is associated with the point (*x*, *y*). Any mask geometry crossing this point is also associated with *name*. If *layer* is present then just geometry crossing the point on that layer is associated with *name*. For plotting this command is similar to text on plot. When doing circuit extraction this command is used to give an explicit name to a node. *Name* must not have any spaces in it, and it should not be a number.

FILES

~cad/.cadrc
~/.cadrc
~cad/lib/fix.6
~cad/lib/pat.*
/usr/tmp/#cif*

SEE ALSO

cadrc(5)
A Guide to LSI Implementation by Hon and Sequin, Second Edition (Xerox PARC, 1980) for a description of CIF.

AUTHOR

Dan Fitzpatrick

BUGS

The **-r** is somewhat kludgy and does not work well with the other options. Space before semi-colons in local extensions can cause syntax errors.

The **-O** option produces simple cif with no scale factors in the DS commands. Because of this you must supply a scale factor to some programs, such as the **-I** option to *cif2ca*.

The **-X** option is no longer supported.

NAME

crystal – VLSI timing analyzer

SYNOPSIS

crystal [file]

DESCRIPTION

Crystal is a semi-interactive program for analyzing the timing characteristics of large integrated circuits. It estimates the speed of a circuit and prints out information about the critical paths. If **file** is specified, it is read in as though the **build** command (see below) had been invoked. Crystal processes commands from the standard input, one per line. Lines beginning with exclamation points are ignored. Unique abbreviations for commands are acceptable. The order of commands in the input is important. Commands are divided into seven groups, which should appear in the following order:

Model commands
: These commands modify the circuit and timing models used by Crystal, and should appear before the circuit is read in. The model commands are **model**, **parameter**, and **transistor**.

Circuit commands
: Used to input and describe the circuit being analyzed. The circuit commands are **build**, **bus**, **capacitance**, **inputs**, **outputs**, and **resistance**.

Dynamic node command
: This group includes the single command **markdynamic**, used to find and mark the dynamic memory nodes in the circuit.

Check commands
: Includes two commands, **check**, and **ratio**. These commands examine the circuit's structure for suspicious-looking electrical features.

Setup commands
: There are three commands in this group, **flow**, **precharged**, **predischarged**, and **set**. These commands are used to restrict the analysis performed for a given clock phase.

Delay command
: The **delay** command invokes the actual delay analysis.

Miscellaneous commands
: These commands may be invoked at any time: **alias**, **critical**, **dump**, **fillin**, **help**, **options**, **quit**, **prcapacitance**, **prfets**, **prresistance**, **source**, **statistics**, **undump**, and **watch**.

The only command outside these groups is the **clear** command, which resets information that was set by setup and delay commands. After **clear**, input may resume with anything except model commands.

NODE NAMES

Where node names are called for in commands, they can appear in any of several forms:

[1] A simple node name.

[2] A name of the form "a<x:y>b". Crystal tries all names of the form "acb" where c ranges from x to y. To get a "<" character in the name, precede it with a backslash.

[3] A name of the form "*a". Crystal searches the entire node table for names containing the string "a". Note: this kind of name specification is slow on large

chips, since the entire table has to be searched. For example, on a sample 45000 transistor chip, 20 seconds of CPU time were used for each search.

GRAPHICAL COMMAND FILES

Several of the commands can be used with the **-g** argument to generate output for graphical display of information. At present, Crystal will generate command files for either Caesar, Magic, or Squid. To use Caesar command files, run Caesar on the chip and pick a view large enough to hold the whole chip (e.g. with the "v" short command). Then use the ":source" long command to read in the command file. The command files place labels and paint on the error layer to mark places, and also push boxes onto the stack so that you can step from one label to the next using the ":popbox" long command. To use Magic command files, run Magic on the layout, place the cursor in the window containing the layout, and use the ":source" long command to read in the command file. A collection of feedback areas will be generated. These can be examined using Magic's ":feedback" command.

COMMANDS

alias file

Read in aliases from the information in **file**. Each line of the alias file is of the form "= name name name ..." where the first name is a node name that appears in the .sim file and each additional name is another name for the same node. After the **alias** command, any of the names may be used to refer to a node. An alias file shouldn't be read in until after the .sim file has been read.

build file

Build a circuit description from the information in **file**, which must be in .sim format. This command is unnecessary if a file is specified on the shell command line, but is necessary if non-standard models are used.

bus node node ...

This command should only rarely be needed. Each of the nodes gets marked as a bus. When a node is a bus, Crystal assumes that delays through the node can be treated as separate stages to the node and from the node. Nodes with large capacitances are automatically considered to be busses: see the **bus** option below.

capacitance pfs node node ...

The parasitic capacitance value for each **node** is set to the given value. This overrides the capacitance estimate made from the mask layout.

check Make a series of static electrical checks on the circuit. This command prints out information about nodes with no transistors connected to them, nodes that are not driven, nodes that don't drive anything, transistors that are permanently forced off, transistors connecting Vdd and GND, and transistors that are bidirectional but haven't been marked with a flow attribute.

clear All information set by setup and delay commands is cleared, in preparation for an additional timing analysis. Information set by circuit commands isn't affected. After the **clear** command, input may contine with any commands except those in the model group. Information from the **watch** command is also cleared.

critical [file] [-g graphicsfile] [-s spicefile] [pathnumber][m][w]

Print out information about the critical paths. If **pathnumber** isn't specified, then the slowest path is printed. If it is specified, the **pathnumber**'th slowest path is printed. If **pathnumber** is followed by an **m**, then the **pathnumber**'th slowest path leading to a memory node is printed. If **pathnumber** is followed by a **w** then the **pathnumber**'th slowest path leading to a watched node is printed (see the

watch command). Only the very slowest paths are recorded by Crystal, controlled by the **paths**, **mempaths**, and **watchpaths** options (see the **options** command below). Furthermore, Crystal does not record a path if its total delay is within .1% of another path already recorded on the list (this is to alleviate the problem of the lists getting flooded by essentially equivalent paths). If the **file** argument is given, then output goes to that file instead of standard output. If the **-g** argument is given, a graphics command file is generated in **graphicsfile**. If the **-s** argument is given, a SPICE deck is created in **spicefile** describing the transistors and parasitics along the path (no model parameters or body bias voltages are output).

delay node risetime falltime

Propagate delay information through the circuit. Assume that the worst-case time for **node** to become 1 is **risetime**, and the worst-case time for it to become 0 is **falltime**. Propagate timing information through nodes that **node** can impact, until the the worst-case settling times for the entire network have been found. A -1 value for **risetime** or **falltime** means that there is no transition to that level.

dump file

This is a special wizards-only command for saving critical path information in a way that Crystal can read it back later using the **undump** command, without having to reprocess the whole .sim file. Don't use this command unless you really know what you are doing.

fillin time/edgeSpeed inFile outFile keyword path path ...

This command is useful in order to interface Crystal to other programs that process Crystal's output. **InFile** is read by the command, and its contents are copied to **outFile**. Along the way, each occurrence of **keyword** is replaced by a number from one of the critical paths (each **path** is specified as for the **critical** command). If **time** is specified, then the time at the end of each stage along the critical path is used to replace occurrences of **keyword**, with smaller times replacing earlier occurrences. If *edgeSpeed* is specified, then the edge speeds from the stages of the path are used to replace **keywords**. If more than one **path** is specified, the paths are processed in order of their occurrence on the command line. If there are more stages in the **paths** than occurrences of **keyword**, then the last stages are ignored. If there are more occurrences of **keyword** than stages, only the first few **keywords** will be replaced. If no **path** is specified, it defaults to "1".

flow direction attribute attribute ...

For each source/drain **attribute** given, mark the attribute so that information will only be permitted to flow in the given **direction**. **Direction** may be either **in**, **out**, **off**, **ignore**, or **normal**. **In** and **out** require information to flow only in the specified direction. **Off** does not permit any flow through the tagged transistors. If **ignore** is specified then no restrictions are enforced whatsoever. **Normal** returns the flow to its normal operation.

help Print a short listing of the valid commands.

inputs node node ...

Mark each of the nodes as an input. This has two effects. First, it indicates that the node can take on values of either 0 or 1 (otherwise, Crystal may conclude that the node can't ever reach one or both values). Second, if the node isn't also an output node, then Crystal assumes that the timing of the node is fixed by the outside world and is not affected by anything in the circuit: if no **delay** command is given for the node, Crystal assumes the value never changes.

markdynamic node value node value ...

This statement causes Crystal to examine all nodes and mark dynamic memory

nodes. A node is considered to be a dynamic memory node if it is electrically isolated when each **node** takes its corresponding **value**. Normally the command is invoked with all of the clock phases turned off, e.g. "markdynamic Phi1 0 Phi2 0 Phi3 0 Phi4 0".

model name
Use **name** as the model for delay calculations. Currently, two models, **rc** and **slope**, are available. The **rc** model approximates each transistor with a fixed resistance value. The **slope** model uses the gate rise and fall speeds to modify the effective resistance.

options [name [value]] [name [value]] ...
This command is used to see and set a variety of internal options used by Crystal. **Options** with no arguments prints out the current setting of all options. Each option consists of an option name and perhaps a value for the option. Some options do not have values. See the section "OPTIONS" below for the available options.

outputs node node ...
Marks each of the given nodes as an output. Crystal assumes that information (0's and 1's) flows from sources (supply rails and inputs) to targets (gates and outputs). If a piece of circuit doesn't drive any gates, Crystal won't compute delays through it unless the result nodes are labelled as outputs.

parameter [name] [value]
This statement is used to see or change several overall model parameters. **Name** is the name of a parameter, and **value** is a new value for that parameter. If both **name** and **value** are specified, then the value of the parameter is changed. If only **name** is specified, then the current value is printed. If neither **name** or **value** is specified, then the values of all parameters are printed. The valid parameter names are listed in the section "MODEL PARAMETERS" below.

prcapacitance [-g file] [-t threshold] node node ...
Print out information for each of the indicated nodes whose total capacitance is at least **threshold** pf (the default is 0 if the argument isn't present). If no node names are given, then all nodes in the the circuit are checked. The **-g** argument can be used to generate a graphical command file.

precharged node node ...
Mark each of the given nodes as precharged. This means that only falling transitions are considered during timing analysis. Each of the nodes is also treated as a bus.

predischarged node node ...
Mark each of the given nodes as precharged to 0. This means that only rising transitions are considered during timing analysis. Each of the nodes is also treated as a bus.

prfets node node ...
For each **node** that is given, information is printed about all transistors whose gates attach to the node. If no node is specified, then information is printed about all transistors in the circuit.

prresistance [-g file] [-t threshold] node node ...
Print out information for each of the indicated nodes whose internal resistance exceeds **threshold** ohms. If no threshold is given, 0 is used by default. If no node names are given, then check all nodes in the circuit. The **-g** argument is used to generate a graphics command file.

quit Exit Crystal and return to the shell. End-of-file on the input stream will also cause

Crystal to exit.

ratio [limit value] [limit value] ...

Examine the circuit for nMOS ratio violations. Normal circuits are expected to have pullup/pulldown ratios between 3.8 and 4.2. Pass transistor driven circuits must have ratios between 7.8 and 8.2. Ratios outside this range are printed out. If the same illegal pullup/pulldown ratio is duplicated more than 20 times, only the first 20 are printed. The limits of acceptability may be changed by providing arguments to the ratio command. **Limit** must be one of **normallow**, **normalhi**, **passlow**, or **passhi** (unique abbreviations are acceptable).

resistance ohms node node ...

The internal node resistance associated with each **node** is set to **ohms**. This overrides the value computed from the mask layout.

set value node node ...

Force each **node** always to have the given **value** (0 or 1). Furthermore, do a static logic simulation to propagate this information as far as possible throughout the network. Thus, if the input to an inverter or NAND gate is forced to 0, the output is forced to 1, and so on.

source file

Read commands from **file**. On end-of-file, go back to reading commands from the previous source. Source files may be nested.

statistics

Prints a variety of statistics gathered internally by Crystal. Probably not useful except to a system maintainer.

transistor [name [field value] [field value] ...]

The **transistor** command is used to define new transistor types, or see or modify existing types. **Name** is the name of a transistor type, **field** is the name of a field associated with the transistor, and **value** is a new value for that field. The valid field names are listed in the section "TRANSISTOR FIELDS" below. If **name** matches the name of an existing transistor type (see below for the predefined types), then the **field** and **value** arguments are used to change some of its fields. If **name** is not an existing transistor type, a new type is created. If there are no arguments to the **transistor** command, then all fields for all defined types are printed out. If **name** is supplied with no field values, then all the fields for that transistor are printed out. To use a user-defined type for a transistor, place an attribute on the gate of the transistor. The the attribute contains the name of the transistor type to use for it.

undump file

This is another wizards-only command. Don't use it unless you really know what you are doing. The **undump** command is provided to read back the output of the **dump** command, so that Crystal can get critical paths without having to re-extract them.

watch node node ...

Mark each of the given nodes so that delays to them will be recorded on the list of slowest watched nodes (these nodes will still be recorded on the lists of arbitrary and memory nodes too, if they are among the slowest in those categories). The watch flags are cleared by the **clear** command.

OPTIONS

The options defined below are used in various and sundry places inside Crystal to control calculations and printout. They can be changed with the **options** command.

bus value
Gives the amount of capacitance a node must have to automatically be considered a bus by Crystal (default is 2 pfs).

graphics style
Sets the style for graphical output. Currently, three styles are understood: **caesar**, **magic**, and **squid** (default is **caesar**).

limit value
Gives the maximum of stage delays Crystal will calculate before giving up in despair (default is 200000).

mempaths value
Gives the number of worst-case paths Crystal will record for delays to memory nodes (default is 5, maximum is 100).

noprintedgespeeds
When printing critical paths, print only the delay to each node, without the edge rise or fall speeds (default).

noseedelays
Tells Crystal not to print out information about delays as they are calculated in **delay** commands (default).

noseedynamic
Tells Crystal not to print out the dynamic memory nodes as they are found in the **markdynamic** command (default).

noseesettings
Tells Crystal not to print out nodes when they are set to values during the set command (default).

paths value
Gives the number of worst-case paths Crystal will record on the list of slowest nodes overall (memory nodes and watched nodes will also be recorded on lists for each of those categories; **mempaths** and **watchpaths** options are used to control the lengths of those lists). The default is 5 and the maximum is 100.

printedgespeeds
When printing critical paths, in addition to printing the delay to each node, also print the speed at which the edge rises or falls at that node. This only makes sense when using the slope model.

ratiodups value
When printing out ratio errors in the **ratio** command, if a number of errors occur with exactly the same erroneous ratio, only the first **ratiodups** of these duplicate errors will be printed. The default is 20.

ratiolimit value
Controls the maximum number of ratio errors that will be printed in any one **ratio** command. The default is 1000.

seealldelays
Causes Crystal to print out each new delay as it is calculated during the **delay** command.

seeallsettings

Causes Crystal to print out each node setting as it is found during the **set** command.

seedelays
Causes Crystal to print out new delays as they are found during the the **delay** command, but only for nodes whose names have alphabetic first characters.

seedynamic
Causes Crystal to print out the name of every dynamic node as it is found in **markdynamic**.

seesettings
Causes Crystal to print out new node settings during the **set** command, but only for nodes whose names have alphabetic first characters.

units value
Tells Crystal what units to use when printing out information. If **units** is 2.0 (default) then a printed value of 1 corresponds to 2 microns.

watchpaths value
Gives the number of paths to record on the list of slowest watched nodes (default is 5, maximum is 100).

TRANSISTOR FIELDS

Each transistor type is parameterized by the following fields. They can be changed using the **transistor** command.

cperarea
Gate-channel capacitance of the transistor, in pfs per square micron.

cperwidth
Gate-source and gate-drain overlap capacitance, in pfs per micron of transistor width.

histrength
An integer value giving the logical strength of the transistor when it is pulling to Vdd. This is used in simulation to determine which transistor wins when different transistors drive a node in different directions (e.g. **histrength** for pullup loads is less than **lostrength** for enhancement pulldowns).

lostrength
An integer value giving the logical strength of the transistor when it is pulling to ground.

on This field has one of three values: **gate0**, **gate1**, or **always**. **Gate0** means that the transistor is turned on only when the gate is zero (in other words, it is a p-channel enhancement device). **Gate1** means that the transistor is turned on only when the gate is one (it is an n-channel enhancement device). **Always** means the device is always turned on (it is a depletion device).

rdown
The resistance per square of the transistor when it is pulling down. Used to calculate delays in the rc model.

rup The resistance per square of the transistor when it is pulling up. Used to calculate delays in the rc model.

slopeparmsdown
Gives table values used for interpolation in the slope delay model. The value consists of any number of triplets. Each triplet contains an edge speed ratio, an effective resistance, and an output edge speed. The table is used when the

transistor is driving to ground. The *mkcp* program is useful for generating these parameters from SPICE model parameters.

slopeparmsup
Gives table values used for interpolation in the slope delay model. The value consists of any number of triplets. Each triplet contains an edge speed ratio, an effective resistance, and an output edge speed. The table is used when the transistor is driving to Vdd. The *mkcp* program is useful for generating these parameters from SPICE model parameters.

spicebody
Spicebody is the node number to use for the body when outputting this type of transistor in SPICE decks. The body node number must be 0-3. 0 is GND, 1 is Vdd, and 2 and 3 are user-controlled body bias voltages.

spicetype
A single letter identifier used as the type of this transistor in SPICE decks.

PREDEFINED TRANSISTOR TYPES

The following types of transistors are predefined by Crystal. When Crystal reads in files, it selects one of the following transistor types for each transistor, unless overriden by an attribute giving a type not listed below. Their fields can be changed using the **transistor** command.

nenh Enhancement transistors in nMOS.

nenhp Enhancement transistors in nMOS whose gates are driven by pass transistors (i.e. any transistor whose gate is not a circuit input and does not attach to an **nload** or **nsuper** transistor). Transistor types are switched between **nenh** and **nenhp** during flow marking.

ndep Depletion devices in nMOS (most depletion devices are turned into either type **nload** or **nsuper** by Crystal).

nload nMOS depletion devices where the gate connects to either source or drain and the other terminal connects to Vdd.

nsuper
nMOS depletion devices where either the source or drain connects to Vdd but the other terminal doesn't connect to the gate.

nchan N-channel enhancement devices in CMOS. This is provided separately from type **nenh** as a convenience to accomodate different delay characteristics in nMOS and CMOS.

pchan P-channel enhancement devices in CMOS.

MODEL PARAMETERS

The following are the model parameters that aren't associated with particular transistor types. They are used in the **parameter** command.

diffcperarea
Capacitance between diffusion and substrate, in pfs per square micron.

diffcperperim
Sidewall capacitance of diffusion, in pfs per micron of perimeter.

diffresistance
Resistance of diffusion, in ohms per square.

metalcperarea
Capacitance between metal and substrate, in pfs per square micron.

metalresistance
Resistance of metal, in ohms per square.

polycperarea
Capacitance between polysilicon and substrate, in pfs per square micron.

polyresistance
Resistance of polysilicon, in ohms per square.

vdd The supply (logic 1) voltage. Used in the slope model, and also in outputting SPICE decks.

vinv The logic threshold voltage (usually Vdd/2). Used in the slope model to compute edge speeds for resistors.

SEE ALSO

mkcp(1)

AUTHOR

John Ousterhout

NAME

eqntott − generate truth table from Boolean equations

SYNOPSIS

eqntott [**-l**] [**-f**] [**-s**] [**-r**] [**-R**]] [-.key] [cc options] [files]

DESCRIPTION

Eqntott generates a truth table suitable for PLA programming from a set of Boolean equations which define the PLA outputs in terms of its inputs. When neither -f nor -s is specified, input and output variables must be mutually exclusive. If the -s option is given, an output variable may be used in an expression defining another output variable: the expression for the first output is substituted for the the name of that output when it is encountered. The -f option allows outputs to be defined in terms of their previous values in a synchronous system (e.g. an FSM): the same name appearing as both an input and an output may be thought of as referring to two distinct variables, or the same variable at two distinct times. (The -f and -s options are mutually exclusive.)

If the -r option is specified, *eqntott* will attempt to reduce the size of the truth table by merging minterms. The -R option (implies -r) forces *eqntott* to produce a truth table with no redundant minterms. The truth table generated does not represent a minimal covering of the truth functions, but does preserve some "don't care" information for some other program to use.

If the -l option is specified, eqntott will output a truth table which includes the name of the pla and its inputs and outputs as specified in PLA(5).

The form that the output takes is controlled by the string *key*, described below. Input is taken from *files* (standard input default) and run through the C macro preprocessor of *cc*(1), to permit comments, file inclusion, macros, and conditional processing. The *cc options* -D, -I, and -U are recognized and passed on to the preprocessor.

Equation Syntax:

name = expression;

Associates a truth function defined by *expression* with the output *name*, both of which are defined below. If an output name is assigned more than one expression, the effect is identical to a single assignment to the output of the logical disjunction of all the original expressions.

NAME = name ;

Defines the name of the pla to be "name". If not specified, the name of the pla is the name of the input file with any postfixes removed.

INORDER = name [name]... ;

Defines the order in which inputs appear in the truth table. If not specified, the order is that in which the inputs appear in the source.

OUTORDER = name [name]... ;

Defines the order in which outputs appear in the truth table. If not specified, the order is that in which the outputs appear in the source.

Expression Syntax:

name

A name is used to specify an input or output. The name must begin with a letter or underscore; subsequent characters may be letters, digits, underscores, asterisks, periods, square brackets, or angle brackets.

ZERO (or 0)
: Builtin input that always has the value zero (false).

ONE (or 1)
: Builtin input that always has the value one (true).

?
: Builtin input that always has the value "don't care".

(expression)
: Parenthesis may be used to change the order of evaluation.

! expression
: Gives the complement of *expression*.

expression & expression
: Gives the logical conjunction of the two expressions. The & operator associates left to right, and has the same precedence as !.

expression | expression
: Gives the logical disjunction of the two expressions. The | operator also associates left to right, and has a lower precedence than &.

Output Format

The output format may be controlled to a small extent using the character string *key*. The string is scanned left to right, and at each character code, a piece of output is generated corresponding to the character encountered. If -.key is not specified, the string "iopte" is used, or "iopfte" with the -f option.

code	*output generated*
e	**.e**
f	**.f** *output-number input-number* (one line for each feedback path, numbers refer to Or- and And-plane truth table column numbers)
h	a human readable version of the truth table (q.v.)
i	**.i** *number-of-inputs*
I	**.I** *input-name* (one line for each input, in order)
l	a truth table with the name of the pla, its inputs and its outputs
p	**.p** *number-of-product-terms*
n	**.n** *number-of-product-terms*
o	**.o** *number-of-outputs*
O	**.O** *output-name* (one line for each output, in order)
S	PLA connectivity summary
t	PLA personality matrix (q.v.)
v	eqntott version information

The truth table (personality matrix) consists of a line for each minterm, beginning with that minterm and followed by the values of the various outputs. The minterm is composed of a single character (0, 1, or -) for each input in the conventional fashion. The output values are represented by one of the three characters (0, 1, or x). Some white space is added for readability's sake.

In the human readable format, each line of output represents one term in the sum-of-products expression for an output. The line begins with the name of the output, which is enclosed in parentheses for the value "don't care". Then follow the names of the inputs in the product; complemented inputs are preceded by a !.

SEE ALSO

cc(1).

DIAGNOSTICS

Syntax errors are written to the standard error output and should be self-explanatory.

BUGS

-l should be the default, but some pla tools can't handle the full format. Eqntott likes its options separately; i.e. -f -l works but -fl doesn't.

AUTHOR

Bob Cmelik.
-l option added by Jeff Deutsch.

NAME

esim – event driven switch level simulator

SYNOPSIS

esim [file1 [file2 ...]]

DESCRIPTION

Esim is an event-driven switch level simulator for nMOS or CMOS transistor circuits. *Esim* accepts commands from the user, executing each command before reading the next. Commands come in two flavors: those which manipulate the electrical network, and those to direct the simulation. Commands have the following simple syntax:

c arg1 arg2 ... argn

where *c* is a single letter specifying the command to be performed and *arg1* through *argn* are arguments to that command. The arguments are separated by spaces or tabs, and the command is terminated by a newline.

To run *esim* type

esim *file1 file2 ...*

Esim will read and execute commands, first from *file1*, then *file2*, etc. If one of the file names is preceded by a '–', then that file becomes the new output file (the default output is stdout). For example,

esim f.sim -f.out g.sim

This would cause *esim* to read commands from *f.sim*, sending output to the default output. When *f.sim* was exhausted, **f.out** would become the new output file, and the commands in **g.sim** executed.

After all the files have been processed, and if the **q** command has not terminated the simulation run, *esim* will accept further commands from the user, prompting for each one like so:

sim>

The user can type individual commands or direct *esim* to another file using the @ command:

sim> @ *patchfile*.**sim**

This command would cause *esim* to read commands from *patchfile*.**sim**, returning to interactive input when the file was exhausted.

It is common to have an initial network file prepared by a node extractor with perhaps a patch file or two prepared by hand. After reading these files into the simulator, the user would then interactively direct *esim*. This could be accomplished as follows:

esim file.sim patch.1 patch.2

After reading the files, *esim* would prompt for the first command. Or we could have typed:

% esim file.sim
sim> @ patch.1
sim> @ patch.2

Network Manipulation Commands

The electrical network to be simulated is made up of enhancement and depletion mode transistors interconnected by nodes. Components can be added to the network with the following commands:

e gate source drain
e gate source drain length width key xpos ypos area
: Adds enhancement mode transistor to network with the specified gate, source, and drain nodes. The longer form includes size and location information as provided by the node extractor -- when making patches the short form is usually used.

d gate source drain

d gate source drain length width key xpos ypos area
Like **e** except for depletion mode devices.
p gate source drain
p gate source drain length width key xpos ypos area
Like **e** except for pMOS devices in CMOS.
n gate source drain
n gate source drain length width key xpos ypos area
Like **e** except for nMOS devices in CMOS.
C node1 node2 cap
Increase the capictance between *node1* and *node2* by *cap*. *Esim* ignores this unless either *node1* or *node2* is GND.
= node name1 name2 name3
Allows the user to specify synonyms for a given node. Used by the node extractor to relate user-provided node names to the node's internal name (usually just a number).
| comment...
Lines beginning with vertical bar are treated as comments and ignored -- useful for deleting pieces of network in node extractor output files.
i node
Input record -- output by node extractor and not used by *esim*.

Currently, there is no way to remove components from the network once they have been added. You must go back the input files and modify them (using the comment character) to exclude those components you wished removed. **N** records need not be included for new nodes the user wishes to patch into the network.

Simulator Commands

The user can specify which nodes are to have there values displayed after each simulation step:

w node1 -node2 node3 ...
Watch node1 and node3, stop watching node2. At the end of a simulation step, each watched node will displayed like so:
node1=0 node3=X ...
To remove a node from the watched list, preface its name with a '−' in a **w** command.
W label node1 node2 ... noden
Watch bit vector. The values of nodes node1, ..., noden will displayed as a bit vector:
label=010100 20
where the first 0 is the value of node1, the first 1 the value of node2, etc. The number displayed to right is the value of the bit vector interpreted as a binary number; this is omitted if the vector contains an X value. There is no way to unwatch a bit vector.

Before each simulation step the user can force nodes to be either high (1) or low (0) inputs (an input's value cannot be changed by the simulator!):

h node1 node2 ..
Force each node on the argument list to be a high input. overrides previous input commands if necessary.
l node1 node2 ...
Like **h** except forces nodes to be a low input.
x node1 node2 ...
Removes nodes from whatever input list they happen to be on. The next simulation step will determine their correct value in the circuit. This is the default state of most nodes. Note that this does not force nodes to have an X

value -- it simply removes them from the input lists.

The current value of a node can be determined in several ways:

v

View. prints the values of all watched nodes and nodes on the high and low input lists.

? node1 node2 ...

Prints a synopsis of the named nodes including their current values and the state of all transistors that affect the value of these nodes. This is the most common way of wondering through the network in search of what went wrong...

! node1 node2 ...

For each node in the argument list, prints a list of transistors controlled by that node.

? and ! allow the user to go both backwards and forwards through the network in search of that piece causing all the problems.

The simulator is invoked with the following commands:

s

Simulation step. Propogates new values for the inputs through the network, returns when the network has settled. If things don't settle, command will never terminate -- try the **w** and **D** commands to narrow down the problem.

c

Cycle once through the clock, as define by the **K** command.

I

Initialize. Circuits with state are often hard to initialize because the initial value of each node is X. To cure this problem, the I command finds each node whose value is charged-X and changes it to charged-0, then runs a simulation step. If one iterates the I command a couple times, this often leads to a stable initialized condition (indicated when an I command takes 0 events, i.e., the circuit is stable).

Try it -- if circuit does not become stable in 3 or 4 tries, this command is probably of no use.

Miscellaneous Commands

D

toggle debug switch. useful for debugging simulator and/or circuit. If debug switch is on, then during simulation step each time a watched node is encounted in some event, that fact is indicated to the user along with some event info. If a node keeps appearing in this prinout, chances are that its value is oscillating. Vice versa, if your circuit never settles (ie., it oscillates) , you can use the **D** and **w** commands to find the node(s) that are causing the problem.

> filename

write current state of each node into specified file. useful for make a break point in your simulation run. Only stores values so isn't really useful to **dump** a run for later use -- see **<** command.

< filename

read from specified file, reinitializing the value of each node as directed. Note that network must already exist and be identical to the network used to create the dump file with the **>** command. These state saving commands are really provided so that complicated initializing sequences need only be simulated once.

L

invokes network processor that finds all subnets corresponding to simple logic gates and converts them into form that allows faster simulation. Often it does the right thing, leading to a 25% to 50% reduction in the time for a single step. [We know of one case where the transformation was not transparent, so caveat simulee...]

X ...

call extension command -- provides for user extensions to simulator.

q

exit to system.

Local Extensions

V node vector

Define a vector of inputs for the node. The first element is initially set as the input for *node*. Set the next element of the vector as the input after a cycle.

G n

Run the simulator through n cycles (a group). If n is not present make the run as long as the longest vector. All watch nodes are reported back as vectors.

N

Clear all previously defined input vectors.

K node1 vector1 node2 vector2 ... nodeN vectorN

Define the clock. Each cycle, nodes 1 through N must run through their respective vectors.

SEE ALSO

ext2sim(1), sim(5)

AUTHOR

Chris Terman

CMOS enhancements by Mike Klein and Joan Pendelton Changed the old "**R**" (run) command name to "**G**" (group) to avoid naming conflict with Resistor declations from the extracted files (by Fred W. Obermeier).

NAME

espresso – Boolean Minimization

SYNOPSIS

espresso [*type*] [*file*] [*options*]

DESCRIPTION

Espresso takes as input a two-level representation of a two-valued (or a multiple-valued) Boolean function, and produces a minimal equivalent representation. The algorithms used are new and represent an advance in both speed and optimality of solution in heuristic Boolean minimization.

Espresso reads the *file* provided (or standard input if no files are specified), performs the minimization, and writes the minimized result to standard output. *Espresso* automatically verifies that the minimized function is equivalent to the original function.

The default input and output file formats are compatible with the Berkeley standard format for the physical description of a PLA. The input format is described in detail in espresso(5). Note that the input file is a *logical* representation of a set of Boolean equations, and hence the input format differs slightly from that described in pla(5) (which provides for the *physical* representation of a PLA). The input and output formats have been expanded to allow for multiple-valued logic functions, and to allow for the specification of the don't care set which will be used in the minimization.

Type specifies the logical format for the function. The allowed types are -f, -r, -fr, -fd, -dr, and -fdr which have the same meanings assigned in espresso(5).

The command line options described below can be specified anywhere on the command line and must be separated by spaces:

-d Verbose detail describing the progress of the minimization is written to standard output. Useful only for those familiar with the algorithms used.

-do [s] This option executes subprogram [s]. Some of the more useful ones are:
check – checks that the function is a partition of the entire space (i.e., that the ON-set, OFF-set and DC-set are pairwise disjoint, and that their union is the Universe)
echo – implies "-out fdr" and echoes the function to standard output. This can be used to compute the complement of a function.
map – draw the Karnaugh maps for a function.
opo – choose a good assignment of output function phases, and then minimize the function. The chosen phase is reported in the output file.
qm – generate all prime implicants of a function, compute the *reduced prime implicant table*, and perform a heuristic covering of this table. Provides a bound on the size of the minimum solution. Will perform an exact minimization if option -exact is used.
single_output – Minimize each function one at a time as a single-output function. Terms will not be shared among the functions.
stats – provide simple statistics on the size of the function
The remaining subprograms (absmin, allphase, badopo, compact, contain, d1merge, dcset, essen, expand, intersect, irred, lexsort, make_sparse, mincov, minterms, npc, opotol, pair, pairsasao, pop, primes, reduce, sharp, taut, union, unravel, verify) are intended for those heavily into manipulating Boolean functions.

-exact Use an exact algorithm to solve the covering problems encountered during the execution of the algorithms. Not recommended for casual users.

-fast Stop after the first EXPAND and IRREDUNDANT operations (i.e., do not iterate over the solution).

-kiss Sets up a *kiss*-style minimization problem.

-ness Essential primes will not be detected and removed from the minimization.

-nirr The final result will not necessarily be forced irredundant.

-help Provides a quick summary of the available command line options.

-onset Recompute the ON-set before the minimization. Useful when the PLA has a large number of product terms (e.g., an exhaustive list of minterms).

-out [s] Selects the output format. By default, only the ON-set (i.e., type f) is output after the minimization. [s] can be one of **f**, **d**, **r**, **fd**, **dr**, **fr**, or **fdr** to select any combination of the ON-set (f), the OFF-set (r) or the DC-set (d). [s] may also be **eqntott** to output algebraic equations acceptable to eqntott(1), or **pleasure** to output an unmerged PLA (with the *.label* and *.group* keywords) acceptable to pleasure(1).

-pos Swaps the ON-set and OFF-set of the function after reading the function. This can be used to minimize the OFF-set of a function. *.phase* in the input file can also specify an arbitrary choice of output phases.

-s Will provide a short summary of the execution of the program including the initial cost of the function, the final cost, and the computer resources used.

-t Will produce a trace showing the execution of the program. After each main step of the algorithm, a single line is printed which reports the processor time used, and the current cost of the function.

-x Suppress printing of the solution.

DIAGNOSTICS

espresso will issue a warning message if a product term spans more than one line. Usually this is an indication that the number of inputs or outputs of the function is specified incorrectly.

SEE ALSO

kiss(1), pleasure(1), pla(5), espresso(5)

R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.

R. Rudell, "Espresso-MV: Algorithms for Multiple-Valued Logic Minimization," *Proc. Cust. Int. Circ. Conf.*, May 1985.

AUTHOR

Richard Rudell

BUGS

Always passes comments from the input file, and passes unrecognized options straight from the input file to standard output (sometimes this isn't what you want).

There are a lot of options, but the most typical use is the following:

```
eqntott -r file.eqn | espresso >file.pla
```

The -R option of eqntott should not be used (it is much too expensive).

NAME

ext2sim − convert hierarchical **.ext** extracted-circuit files to flat **.sim** files

SYNOPSIS

ext2sim [**−a** *aliasfile*] [**−c** *cthresh*] [**−l** *labelsfile*] [**−o** *simfile*] [**−p** *path*] [**−r** *rthresh*] [**−A**] [**−C**] [**−L**] [**−R**] [**−T** *tech*] *root*

DESCRIPTION

Ext2sim will convert an extracted circuit from the hierarchical representation (**.ext**) produced by Magic to the flat representation (**.sim**) currently required for simulation. The root of the tree to be extracted is the file *root*.**ext**; it and all the files it references are recursively flattened. The result is a single, flat representation of the circuit that is written to the file *root*.**sim**, a list of node aliases written to the file *root*.**al**, and a list of the locations of all nodenames in CIF format, suitable for plotting, to the file *root*.**nodes**. The file *root*.**sim** is suitable for use with programs such as *crystal*(1), *esim*(1), or *sim2spice*(1).

The following options are recognized:

−a *aliasfile*
: Instead of leaving node aliases in the file *root*.**al**, leave it in *aliasfile*.

−c *cap*
: Set the capacitance threshold to *cap* femtofarads. Only capacitances greater than or equal to *cap* will appear in the **.sim** file as **C** lines. This includes both capacitance to substrate (GND) and internodal capacitances. The default value for *cap* is 100 femtofarads. Note that transistor gate capacitance is typically not included in node capacitances, as most analysis tools compute the gate capacitance directly from the gate area. Therefore the -c flag provides a limit only on non-gate capacitance.

−l *labelfile*
: Instead of leaving a CIF file with the locations of all node names in the file *root*.**nodes**, leave it in *labelfile*.

−o *outfile*
: Instead of leaving output in the file *root*.**sim**, leave it in *outfile*.

−p *path*
: Normally, the path to search for **.ext** files is determined by looking for **path** commands in first ~cad/lib/magic/sys/.magic, then ~/.magic, then .magic in the current directory. If **−p** is specified, the colon-separated list of directories specified by *path* is used instead. Each of these directories is searched in turn for the **.ext** files in a design.

−r *res* Set the resistance threshold to *res* ohms. Only nodes with resistances greater than or equal to *res* will appear in the **.sim** file as **R** lines. The default value for *res* is 10 ohms.

−A Don't produce the aliases file.

−C Don't output capacitances (no **C** lines will appear in the **.sim** file). Because this avoids any internodal capacitance processing, *ext2sim* will run faster when this flag is given.

−L Don't produce the label file.

−R Don't output resistances (no **R** lines will appear in the **.sim** file). This is required if the **.sim** file is to be read by programs that don't understand about explicit resistances in **.sim** files.

−T *tech*

Set the technology in the output **.sim** file to *tech*. This overrides any technology specified in the root **.ext** file.

SCALING AND UNITS

If all of the **.ext** files in the tree read by *ext2sim* have the same geometrical scale (specified in the **scale** line in each **.ext** file), this scale is reflected through to the output, resulting in substantially smaller **.sim** files. Otherwise, the geometrical unit in the output **.sim** file is a centimicron.

Resistance and capacitance are always output in ohms and picofarads, respectively.

SEE ALSO

magic(1), **ext**(5), **sim**(5)

AUTHOR

Walter Scott

NAME

fsleeper − run sleeper remotely

SYNOPSIS

fsleeper [**−t** *tty*] [**−l** *user*] [*remotemachine*]

DESCRIPTION

Fsleeper is used if you wish to run a program such as *magic*(1) on a different machine (*remotemachine*) than the one to which a graphics terminal is attached, and the local graphics terminal has no login process.

Normally, *fsleeper* will start a remote sleeper on the companion graphics terminal for your terminal. This graphics terminal is found by looking in the file ~cad/lib/displays, as described in *displays*(5). If a different graphics terminal is desired, it may be specified by the **-t** flag. Note that this is the terminal on the local machine, not the remote machine. (The remote terminal will be printed by *sleeper*(1) when it starts up on the remote machine).

Also, normally *fsleeper* will attempt to log in as the user **sleeper** on the remote machine. If a different user name is desired, it may be specified with the **-l** flag. This user name must exist on *remotemachine*.

FILES

~cad/lib/displays

SEE ALSO

magic(1), rsleeper(1), sleeper(1), displays(5)

AUTHOR

Walter Scott

BUGS

If no *remotemachine* is specified, it defaults to **ucbkim**. This is fine for Berkeley, but useless elsewhere.

NAME

grSunProg – internal process for Magic's Sun 120 display driver

SYNOPSIS

grSunProg *colorWindowName textWindowName notifyPID requestFD pointFD buttonFD*

DESCRIPTION

GrSunProg is an internal program used by Magic when using the Sun 120 workstation's display. This manual page is intended only for Magic maintainers.

GrSunProg collects button pushes from the color window and sends them over a pipe to Magic. The program also responds to requests from Magic for the mouse position. In addition, this program tells Suntools to forward characters typed in the color window directly to Magic's text window.

ARGUMENTS

All six arguments are required:

colorWindowName

This is the name of the color window that magic is running under (such as **/dev/win3**). Magic normally opens up the color monitor with a single, large, window on it.

textWindowName

This is the name of the text window that contains Magic's command log. Keyboard events are forwarded to this window.

notifyPID

If this processID is not 0, then SIGIO signals are sent to this process when there is data for it.

requestFD pointFD buttonFD

These are the file descriptors that grSunProg should use in its interface (see below). They are small integers printed as strings.

INTERFACE

Button pushes are sent out over file descriptor *buttonFD*. A button push is encoded as two characters followed by two integers giving the location of the button push. The first character is either 'L', 'M', or 'R' depending on the button pushed: Left, Middle, or Right. The next character is either 'D' or 'U' depending on the action: Up or Down. The two numbers are the X and Y coordinates of the button push. This string is followed by a newline. Example: **LD 123 342** means that the left button was pushed down at location (123, 342).

GrSunProg sometimes receives a character from Magic over file descriptor *requestFD*. If this character is an EOF, then the program terminates. If this character is an 'A', then grSunProg responds with a 'P' and the current mouse coordinates over file descriptor *pointFD*. This string is followed by a newline. Example: **P 101 23** means that the mouse is currently at location (101, 23).

SEE ALSO

magic(1) grsunprog2(1)

AUTHOR

Robert N. Mayo

NAME

magic − VLSI layout editor

SYNOPSIS

magic [**−g** *graphics_port*] [**−d** *device_type*] [**−m** *monitor_type*] [**−i** *tablet_port*] [**−T** *technology*] [**−F** *object_file save_file*] [*file*]

DESCRIPTION

Magic is an interactive editor for VLSI layouts that runs under 4.2/4.3 BSD Unix, as well as some look-alikes (such as Sun releases 2.0 and 3.0, and DEC's Ultrix). It also runs under Pyramid's operating system. This man page is a reference manual; if you are a first-time user, you should use the Magic tutorials in "1986 VLSI Tools: Still More Works by the Original Artists" to get acquainted with the system.

Magic runs in several different configurations. For systems like the Sun-2/160 with an integrated color display, one window of the screen is used to display text (commands and Magic's responses) and other windows are used for displaying layouts in color. In systems without integrated color, Magic uses two displays: one for text and a separate color display for displaying layouts. In these systems you should run Magic from the text display.

Normally, Magic gets information about the color display from ~cad/lib/displays. However, the following command line switches can be used if there is insufficient information pre-set in ~cad/lib/displays:

−g The next argument is the name of the device to use for communication with the graphics display. This is usually of the form **/dev/tty***xx* (for displays connected by RS232 lines, such as the AED family) or **/dev/cgone0** (for Sun-2/120 systems with the SunColor option).

−d The next argument is the type of graphics terminal being used. Magic supports these types:

UCB512
An old AED512 with the Berkeley microcode ROMs and an attached bitpad (SummaGraphics Bitpad One). The ROMs are available from AED.

UCB512N
A new "Colorware" AED512 with the Berkeley microcode ROMs and an attached bitpad (SummaGraphics Bitpad One). The ROMs are available from AED. We do not recommend the GTCO bitpad, since we have heard that their Summagraphics emulation mode can't handle up/down button encoding nor double button pushes.

AED767
An AED767 with a SummaGraphics Mouse. Because of missing features in this device, programable cursors and Bit-Blt do not work. Many thanks to Norm Jouppi and DECWRL for porting Magic to this device.

UCB1024
An AED1024 with a SummaGraphics Mouse and AED's Magic microcode ROMs that implement the same operations as the UCB512 ROMs. Thanks to LSI Logic for this port.

AED1024
An AED1024 with a SummaGraphics Mouse and rev. D roms. Because of missing features in this device, programable cursors do not work. Many thanks to Peng Ang and LSI Logic Corp. for porting Magic to this device.

other AEDs

Other AEDs can be handled by modifying Magic's file grAed1.c. There are just too many combinations of options for AEDs for us to be able to supply drivers for all of them.

NULL A null device for running Magic without using a graphics display.

SUN120

A Sun Microsystems workstation, model Sun2/120 with the SunColor option (/dev/cgone0) and the Sun optical mouse. Also works on some old Sun1s with the 'Sun2 brain transplant'. You must be running Suntools on the black and white display.

SUN160

A Sun 160 workstation with a single screen that has 8 bit-planes of color. You must be running Suntools in order for Magic to run with this option. Also, you can not resize or redisplay Magic windows while Magic is collecting a command (since Magic is only one process). Because Sun's window package doesn't do interrupt processing, you can't interrupt Magic unless you point to its text window.

SUNBW

A black & white Sun workstation. Because this system only has one bit-plane, Magic does extra redisplay whenever it erases the box or highlight areas. Also, it is hard to see all the layers since they are drawn as stippled areas instead of colored areas. You must be running Suntools in order for Magic to run with this option, and the caveats for the SUN160 version also apply to this version.

Others

Some people outside of Berkeley have developed drivers for Magic. We have attempted to collect as many of these drivers as possible and put them on our distribution tape, but we haven't attempted to integrate them into Magic. The drivers appear in the directory ~cad/src/magic/graphics.drivers. The list includes at least the following types, but more may have been added since this manual page was updated: **Jupiter**, **Omega440**, **Apollo**, and **Lexidata**. The directory also lists a few other displays and the people who have worked on porting Magic to them. We appreciate the porting that these people have done and their willingness to have their work distributed to other researchers.

On VAXes, NULL is the default. On Sun workstations, the type defaults to SUNBW, SUN120, or SUN160 as appropriate (Magic looks in */dev* and tries to make an educated guess). Types listed in ~cad/lib/displays override the default type.

-m The next argument is used to select the right color map for the monitor's phosphors. "Std" works well for most monitors.

-i The next argument is the name of the port to use for input from the tablet. This defaults to whatever port is being used for the graphics output, and thus only needs to be specified under unusual circumstances.

In addition, Magic accepts the following command line switches:

-T The next argument is the name of a technology. The tile types, display information, and design rules for this technology are read by Magic from a technology file when it starts up. The technology defaults to "scmos".

-F The next two arguments are filenames. The first, *object_file*, is the name of the file that was executed to run this version of Magic. The second, *save_file*, is the name of

a new file. After performing all initializations (reading in the technology file, loading the style information and colormap, etc), an executable image of Magic is stored in *save_file*. This executable image may then be run as a normal Magic, except that it starts up much more quickly. The symbol table from *object_file* is copied to *save_file* so the new version can be debugged. The **-F** feature only works on VAXes.

When Magic starts up it looks for a command file in ~cad/lib/magic/sys/.magic and processes it if it exists. Then Magic looks for a file with the name ".magic" in the home directory and processes it if it exists. Finally, Magic looks for a .magic file in the current directory and reads it as a command file if it exists. The .magic file format is described under the **source** command.

COMMANDS - GENERAL INFORMATION

Magics uses three sorts of commands. Pressing a mouse button is one sort of command. You can also enter commands by typing a : or ; character followed by the text of the command. Multiple commands may be specified on one line by separating them with a semicolon. The third command form consists of single-letter abbreviations called "macros"; macros are invoked by pressing single keys without typing a : first. Certain macros are predefined, but you can override them and add your own macros using the **macro** command (described below under COMMANDS FOR ALL WINDOWS).

Most commands deal with the window underneath the cursor, so if a command doesn't do what you expect make sure that you are pointing to the correct place on the screen. There are several different kinds of windows in Magic (layout, color, and netlist); each window has a different command set, described in a separate section below.

MOUSE BUTTONS FOR LAYOUT

On four-button cursors, the top button is not used by Magic. The remaining three buttons are interpreted in a way that depends on the current tool (see the **tool** command). The various tools are described below. The initial tool is **box**. These interpretations apply only when mouse buttons are pressed in the interior of a layout window.

Box Tool

This is the default tool. It is used to position the box and to paint and erase:

left This button is used to move the box by one of its corners. Normally, this button picks up the box by its lower-left corner. To pick the box up by a different corner, click the right button while the left button is down. This switches the pick-up point to the corner nearest the cursor. When the button is released, the box is moved to position the corner at the cursor location.

right Change the size of the box by moving one corner. Normally this button moves the upper-right corner of the box. To move a different corner, click the left button while the right button is down. This switches the corner to the one nearest the cursor. When you release the button, three corners of the box move in order to place the selected corner at the cursor location (the corner opposite the one you picked up remains fixed).

middle (bottom)
Used to paint or erase. If the crosshair is over paint, then the area of the box is painted with the layer(s) underneath the crosshair. If the crosshair is over white space, then the area of the box is erased.

Wiring Tool

The wiring tool is used to provide an efficient interface to the wiring commands:

left Same as **wire type**.

right Same as **wire leg**.

middle (bottom)
Same as **wire switch**.

Netlist Tool

This tool is used to edit netlists interactively:

left Select the net associated with the terminal nearest the cursor.

right Find the terminal nearest the cursor, and toggle it into the current net (if it wasn't already in the current net) or out of the current net (if it was previously in the net).

middle (bottom)
Find the terminal nearest the cursor, and join its net with the current net.

LONG COMMANDS FOR LAYOUT

These commands work if you are pointing to the interior of a layout window. Commands are invoked by typing a colon (":") or semi-colon (";"), followed by a line containing a command name and zero or more parameters. In addition, macros may be used to invoke commands with single keystrokes. See the macro section below for a list of default macros. Unique abbreviations are acceptable for all keywords in commands. The commands are:

array *xsize ysize*
Make many copies of the selection. There will be *xsize* instances in the x-direction and *ysize* instances in the y-direction. Paint and labels are arrayed by copying them. Subcells are not copied, but instead each instance is turned into an array instance with elements numbered from 0 to *xsize*-1 in the x-direction, and from 0 to *ysize*-1 in the y-direction. The spacing between elements of the array is determined by the box x- and y-dimensions.

array *xlo ylo xhi yhi*
Identical to the form of **array** above, except that the elements of arrayed cells are numbered left-to-right from *xlo* to *xhi* and bottom-to-top from *ylo* to *yhi*. It is legal for *xlo* to be greater than *xhi*, and also for *ylo* to be greater than *yhi*.

box [*args*]
Used to change the size of the box or to find out its size. There are several sorts of arguments that may be given to this command:

(No arguments.)
Show the box size and its location in the edit cell, or root cell of its window if the edit cell isn't in that window.

direction [*distance*]
Move the box *distance* units in *direction*, which may be one of **left**, **right**, **up**, or **down**. *Distance* defaults to the width of the box if *direction* is **right** or **left**, and to the height of the box if *direction* is **up** or **down**.

width [*size*]

height [*size*]
Set the box to the width or height indicated. If *size* is not specified the width or height is reported.

x1 y1 x2 y2

Move the box to the coordinates specified (these are in edit cell coordinates if the edit cell is in the window under the cursor; otherwise these are in the root coordinates of the window). *x1* and *y1* are the coordinates of the lower left corner of the box, while *x2* and *y2* are the upper right corner. The coordinates must all be integers.

calma [*option*] [*args*]

This command is used to read and write files in Calma GDS II Stream format (version 3.0, corresponding to GDS II Release 5.1). This format is like CIF, in that it describes physical mask layers instead of Magic layers. In fact, the technology file specifies a correspondence between CIF and Calma layers. The current CIF output style (see **cif ostyle**) controls how Calma stream layers are generated from Magic layers. If no arguments are given, the **calma** command generates a Calma stream file for the layout in the window beneath the cursor in *file*.**strm**, where *file* is the name of the root cell. This stream file describes the entire cell hierarchy in the window. The name of the library is the same as the name of the root cell. *Option* and *args* may be used to invoke one of several additional operations:

calma flatten

Ordinarily, Magic arrays are output using the Calma ARRAY construct. After a **calma flatten** command, though, arrays will be output instead as a collection of individual cell uses, as occurs when writing CIF.

calma help

Print a short synopsis of all of the **calma** command options.

calma labels

Output labels whenever writing a Calma output file.

calma noflatten

Undoes the effect of a prior :**calma flatten** command, re-enabling the output of Magic arrays using the Calma ARRAY construct.

calma nolabels

Don't output labels when writing a Calma output file.

calma read *file*

The file *file*.**strm** is read in Calma format and converted to a collection of Magic cells. The current CIF input style determines how the Calma layers are converted to Magic layers. The new cells are marked for design-rule checking. Calma format doesn't identify the root of the collection of cells read in, so none of the cells read will appear on the display; use **load** to make them visible. If the Calma file had been produced by Magic, then the name of the root cell is the same as the library name printed by the :**calma read** command.

calma write *fileName*

Writes a stream file just as if no arguments had been entered, except that the output is written into *fileName*.**strm** instead of using the root cell name for the file name.

channels

This command will run just the channel decomposition part of the router, deriving channels for the area under the box. The channels will be displayed as outlined feedback areas over the edit cell.

cif [*option*] [*args*]

Read or write files in Caltech Intermediate Form (CIF). If no arguments are given, this command generates a CIF file for the layout in the window beneath the cursor

in *file*.**cif**, where *file* is the name of the root cell. The CIF file describes the entire cell hierarchy in the window. *Option* and *args* may be used to invoke one of several additional CIF operations:

cif help
Print a short synopsis of all of the cif command options.

cif istyle [*style*]
Select the style to be used for CIF input. If no *style* argument is provided, then Magic prints the names of all CIF input styles defined in the technology file and identifies the current style. If *style* is provided, it is made the current style.

cif ostyle [*style*]
Select the style to be used for CIF output. If no *style* argument is provided, then Magic prints the names of all CIF output styles defined in the technology file and identifies the current style. If *style* is provided, it is made the current style.

cif read *file*
The file *file*.**cif** is read in CIF format and converted to a collection of Magic cells. The current input style determines how the CIF layers are converted to Magic layers. The new cells are marked for design-rule checking. Any information in the top-level CIF cell is copied into the edit cell. Note: this command is not undo-able (it would waste too much space and time to save information for undoing).

cif see *layer*
In this command *layer* must be the CIF name for a layer in the current output style. Magic will display on the screen all the CIF for that layer that falls under the box, using stippled feedback areas. It's a bad idea to look at CIF over a large area, since this command requires the area under the box to be flattened and therefore is slow.

cif statistics
Prints out statistics gathered by the CIF generator as it operates. This is probably not useful to anyone except system maintainers.

cif write *fileName*
Writes out CIF just as if no arguments had been entered, except that the CIF is written into *fileName*.**cif** instead of using the root cell name for the file name. The current output style controls how CIF layers are generated from Magic layers.

clockwise [*degrees*]
Rotate the selection by the largest multiple of 90 degrees less than or equal to *degrees*. After the rotation, the lower-left corner of the selection's bounding box will be in the same place as the lower-left corner of the bounding box before the rotation. *Degrees* defaults to 90. If the box is in the same window as the selection, it is rotated too. If any of the selected information isn't in the edit cell, it is first copied into the edit cell and then rotated with the rest of the selection.

copy [*direction* [*amount*]]
If no arguments are given, the selection is copied by picking up the copy at the point lying underneath the box lower-left corner, and placing it so that this point lies at the cursor position. If *direction* is given, it must be a Manhattan direction (e.g. **north**). The selection is copied in that direction by *amount*. If the box is in the same window as the selection, it is moved too. *Amount* defaults to 1.

corner *direction1 direction2* [layers]
This command is similar to **fill**, except that it generates L-shaped wires that travel across the box first in *direction1* and then in *direction2*. For example, **corner north east** finds all paint under the bottom edge of the box and extends it up to the top of the box and then across to the right side of the box, generating neat corners at the top of the box. The box should be at least as tall as it is wide for this command to work correctly. *Direction1* and *direction2* must be Manhattan directions (see the section DIRECTIONS below) and must be orthogonal to each other. If *layers* is specified then only those layers are used; otherwise all layers are considered.

delete Delete all the information in the current selection that is in the edit cell. When cells are deleted, only the selected use(s) of the cell is (are) deleted: other uses of the cell remain intact, as does the disk file containing the cell. Selected material outside the edit cell is not deleted.

drc option [*args*]
This command is used to interact with the design rule checker. *Option* and *args* (if needed) are used to invoke a **drc** command in one of the following ways:

drc catchup
Let the checker process all the areas that need rechecking. This command will not return until design-rule checking is complete or an interrupt is typed. The checker will run even if the background checker has been disabled with **drc off**.

drc check
Mark the area under the box for rechecking in all cells that intersect the box. The recheck will occur in background after the command completes. This command is not normally necessary, since Magic automatically remembers which areas need to be rechecked. It should only be needed if the design rules are changed.

drc count
Print the number of errors in each cell under the box. Cells with no errors are skipped.

drc find [*nth*]
Place the box over the *nth* error area in the selected cell or edit cell, and print out information about the error just as if **drc why** had been typed. If *nth* isn't given (or is less than 1), the command moves to the next error area. Successive invocations of **drc find** cycle through all the error tiles in the cell. If multiple cells are selected, this command uses the upper-leftmost one. If no cells are selected, this command uses the edit cell.

drc help
Print a short synopsis of all the drc command options.

drc off
Turn off the background checker. From now on, Magic will not recheck design rules immediately after each command (but it will record the areas that need to be rechecked; the commands **drc on** and **drc check** can be used to run the checker).

drc on
Turn on the background checker. From now on, after every command, Magic will reverify design rules in any areas modified by the command.

drc printrules [*file*]
Print out the compiled rule set in *file*, or on the text terminal if *file* isn't

given. For system maintenance only.

drc rulestats
Print out summary statistics about the compiled rule set. This is primarily for use in writing technology files.

drc statistics
Print out statistics kept by the design-rule checker. For each statistic, two values are printed: the count since the last time **drc statistics** was invoked, and the total count in this editing session. This command is intended primarily for system maintenance purposes.

drc why
Recheck the area underneath the box and print out the reason for each violation found. Since this command causes a recheck, the box should normally be placed around a small area (such as an error area).

dump *cellName* [*label*] [*x y*]
Copy the contents of cell *cellName* into the edit cell so that the lower-left corner of label *label* of *cellName* is positioned at point (*x*, *y*) in the edit cell. If *x* and *y* aren't specified, then *label* is positioned at the box lower-left corner (the box must be in a window on the edit cell). If **ORIGIN** is given for *label*, then the cell's origin is used to position it. If *label* isn't specified, then the lower-left corner of *cellName*'s bounding box is used to position it. After this command completes, the new information is selected.

edit Make the selected cell the edit cell, and edit it in context. The edit cell is normally displayed in brighter colors than other cells (see the **see** command to change this). If more than one cell is selected, or if the selected cell is an array, the cursor position is used to selected one of those cells as the new edit cell.

erase [*layers*]
For the area enclosed by the box, erase all paint in *layers*. If *layers* is omitted it defaults to ***,labels**. See your technology manual, or use the **layers** command, to find out about the available layer names.

expand [**toggle**]
If the keyword **toggle** is supplied, all of the selected cells that are unexpanded are expanded, and all of the selected cells that are expanded are unexpanded. If **toggle** isn't specified, then all of the cells underneath the box are expanded recursively until there is nothing but paint under the box.

extract *option* [*args*]
Extract a layout, producing one or more hierarchical **.ext** files that describe the electrical circuit implemented by the layout. The current extraction style (see **extract style** below) determines the parameters for parasitic resistance, capacitance, etc. that will be used. The **extract** command with no parameters checks timestamps and re-extracts as needed to bring all **.ext** files up-to-date for the cell in the window beneath the crosshair, and all cells beneath it. Magic displays any errors encountered during circuit extraction using stippled feedback areas over the area of the error, along with a message describing the type of error. Option and *args* are used in the following ways:

extract all
All cells in the window beneath the cursor are re-extracted regardless of whether they have changed since last being extracted.

extract cell *name*
Extract only the currently selected cell, placing the output in the file *name*.

If more than one cell is selected, this command uses the upper-leftmost one.

extract do [*option*]

extract no *option*

Enable or disable various options governing how the extractor will work. Use **:extract do** with no arguments to print a list of available options and their current settings. When the **adjust** option is enabled, the extractor will compute compensating capacitance and resistance whenever cells overlap or abut; if disabled, the extractor will not compute these adjustments but will run faster. If **capacitance** is enabled, node capacitances to substrate (perimeter and area) are computed; otherwise, all node capacitances are set to zero. Similarly, **resistance** governs whether or not node resistances are computed. The **coupling** option controls whether coupling capacitances are computed or not; if disabled, flat extraction is significantly faster than if coupling capacitance computation is enabled.

extract help

Prints a short synopsis of all the **extract** command options.

extract parents

Extract the currently selected cell and all of its parents. All of its parents must be loaded in order for this to work correctly. If more than one cell is selected, this command uses the upper-leftmost one.

extract showparents

Like **extract parents**, but only print the cells that would be extracted; don't actually extract them.

extract style [*style*]

Select the style to be used for extraction parameters. If no *style* argument is provided, then Magic prints the names of all extraction parameter styles defined in the technology file and identifies the current style. If *style* is provided, it is made the current style.

extract unique

For each cell in the window beneath the cursor, check to insure that no label is attached to more than one node. If such a label does appear in more than one node, and it ends with a "#", append a unique numeric suffix. If the label ends with a "!", do nothing. Otherwise, leave a warning feedback area. This command is provided for converting old designs that were intended for extraction with Mextra, which would automatically append unique suffixes to node names when they appeared more than once.

extract warn [[**no**] *option* | [**no**] **all**]

Normally the extractor only reports fatal errors (**extract warn no all**). To see any warning messages produced by the extractor, warnings may be enabled selectively with **warn** *option*. *Option* may be one of the following: **dup**, to warn about two or more unconnected nodes in the same cell that have the same name, **fets**, to warn about transistors with fewer than the minimum number of terminals, and **labels**, to warn when nodes are not labelled in the area of cell overlap. In addition, **all** may be used to refer to all warnings. If a warning is preceded by **no**, it is disabled. To disable all warnings, use **extract warn no all**.

feedback *option* [*args*]

Examine feedback information that is created by several of the Magic commands to report problems or highlight certain things for users. *Option* and *args* are used in

the following ways:

feedback add *text* [*style*]
Used to create a feedback area manually at the location of the box. This is intended as a way for other programs like Crystal to highlight things on a layout. They can generate a command file consisting of a **feedback clear** command, and a sequence of **box** and **feedback add** commands. *Text* is associated with the feedback (it will be printed by **feedback why** and **feedback find**). *Style* tells how to display the feedback, and is one of **dotted**, **medium**, **outline**, **pale**, and **solid** (if unspecified, *style* defaults to **pale**).

feedback clear
Clears all existing feedback information from the screen.

feedback count
Prints out a count of the current number of feedback areas.

feedback find [*nth*]
Used to locate a particular feedback area. If *nth* is specified, the box is moved to the location of the *nth* feedback area. If *nth* isn't specified, then the box is moved to the next sequential feedback area after the last one located with **feedback find**. In either event, the explanation associated with the feedback area is printed.

feedback help
Prints a short synopsis of all the **feedback** command options.

feedback save *file*
This option will save information about all existing feedback areas in *file*. The information is stored as a collection of Magic commands, so that it can be recovered with the command **source** *file*.

feedback why
Prints out the explanations associated with all feedback areas underneath the box.

fill *direction* [*layers*]
Direction is a Manhattan direction (see the section DIRECTIONS below). The paint visible under one edge of the box is sampled. Everywhere that the edge touches paint, the paint is extended in the given direction to the opposite side of the box. For example, if *direction* is **north**, then paint is sampled under the bottom edge of the box and extended to the top edge. If *layers* is specified, then only the given layers are considered; if *layers* isn't specified; then all layers are considered.

findbox [**zoom**]
Center the view on the box. If the optional **zoom** argument is present, zoom into the area specified by the box. This command will complain if the box is not in the window you are pointing to.

flush [*cellname*]
Cell *cellname* is reloaded from disk. All changes made to the cell since it was last saved are discarded. If *cellname* is not given, the edit cell is flushed.

getcell *cellName* [*label*] [*x y*]
This command adds a child cell instance to the edit cell. The instance refers to the cell *cellName*; it is positioned so that the lower-left corner of label *label* of *cellName* is at point (*x*, *y*) in the edit cell. If *x* and *y* aren't specified, then *label* is positioned at the box lower-left corner (the box must be in a window on the edit cell). If

ORIGIN is used as *label*, then the cell's origin point is used to position it. If *label* isn't specified, then the lower-left corner of *cellName*'s bounding box is used to position it. The new subcell is selected. The difference between this command and **dump** is that **dump** copies the contents of the cell, while **getcell** simply makes a reference to the original cell. *Cellname* must not be the edit cell or one of its ancestors.

grid [*xSpacing* [*ySpacing* [*xOrigin yOrigin*]]]

grid off

If no arguments are given, a one-unit grid is toggled on or off in the window underneath the cursor. **Grid off** always turns the grid off, regardless of whether it was on or off previously. If numerical arguments are given, the arguments determine the grid spacing and origin for the window under the cursor. In its most general form, **grid** takes four integer arguments. **XOrigin** and **yOrigin** specify an origin for the grid: horizontal and vertical grid lines will pass through this point. **XSpacing** and **ySpacing** determine the number of units between adjacent grid lines. If **xOrigin** and **yOrigin** are omitted, they default to 0. If **ySpacing** is also omitted, the xSpacing value is used for both spacings. Grid parameters will be retained for a window until explicitly changed by another *grid* command. When the grid is displayed, a solid box is drawn to show the origin of the edit cell.

identify *instance_id*

Set the instance identifier of the selected cell use to *instance_id*. *Instance_id* must be unique among all instance identifiers in the parent of the selected cell. Initially, Magic guarantees uniqueness of identifiers by giving each cell an initial identifier consisting of the cell definition name followed by an underscore and a small integer.

label *string* [*pos* [*layer*]]

A label with text *string* is positioned at the box location. (If *string* contains spaces, be sure to enclose it in double quotes). Labels may cover points, lines, or areas, and are associated with specific layers. Normally the box is collapsed to either a point or to a line (when labeling terminals on the edges of cells). Normally also, the area under the box is occupied by a single layer. If no *layer* argument is specified, then the label is attached to the layer under the box, or space if no layer covers the entire area of the box. If *layer* is specified but *layer* doesn't cover the entire area of the box, the label will be moved to another layer or space. Labels attached to space will be considered by CIF processing programs to be attached to all layers overlapping the area of the label. *Pos* is optional, and specifies where the label text is to be displayed relative to the box (e.g. "north"). If *pos* isn't given, Magic will pick a position to ensure that the label text doesn't stick out past the edge of the cell.

layers Prints out the names of all the layers defined for the current technology.

load [*file*]

Load the cell hierarchy rooted at *file*.**mag** into the window underneath the cursor. If no *file* is supplied, a new unnamed cell is created. The root cell of the hierarchy is made the edit cell unless there is already an edit cell in a different window.

move [*direction* [*amount*]]

If no arguments are given, the selection is picked up by the point underneath the lower-left corner of the box and moved so that this point lies at the cursor location. If *direction* is given, it must be a Manhattan direction (e.g. **north**). The selection is moved in that direction by *amount*. If the box is in the same window as the selection, it is moved too. *Amount* defaults to 1. If there is selected material that isn't in the edit cell, it is first copied into the edit cell and then moved.

paint *layers*
The area underneath the box is painted in *layers*.

path [*searchpath*]
This command tells Magic where to look for cells. *Searchpath* contains a list of directories separated by colons or spaces (if spaces are used, then *searchpath* must be surrounded by quotes). When looking for a cell, Magic will check each directory in the path in order, until the cell is found. If the cell is not found anywhere in the path, Magic will look in the system library for it. If the *path* command is invoked with no arguments, the current search path is printed.

plot *option* [*args*]
Used to generate hardcopy plots direct from Magic. *Options* and *args* are used in the following ways:

plot gremlin *file* [*layers*]
Generate a Gremlin-format description of everything under the box, and write the description in *file*. If *layers* isn't specified, paint, labels, and unexpanded subcells are all included in the Gremlin file just as they appear on the screen. If *layers* is specified, then just the indicated layers are output in the Gremlin file. *Layers* may include the special layers **labels** and **subcell**. The Gremlin file is scaled to have a total size between 256 and 512 units; you should use the **width** and/or **height** Grn commands to ensure that the printed version is the size you want. Use the **mg** stipples in Grn. No plot parameters are used in Gremlin plotting.

plot help
Print a short synopsis of all the **plot** command options.

plot parameters [*name value*]
If **plot parameters** is invoked with no additional arguments, the values for all of the plot parameters are printed. If *name* and *value* are provided, then *name* is the name of a plot parameter and *value* is a new value for it. Plot parameters are used to control various aspects of plotting; all of them have "reasonable" initial values.. The parameters available now are all used to control Versatec-style plotting. They are:

cellIdFont
The name of the font to use for cell instance ids in Versatec plots. This must be a file in Vfont format.

cellNameFont
The name of the font to use for cell names in Versatec plots. This must be a file in Vfont format.

directory
The name of the directory in which to create raster files for the Versatec. The raster files have names of the form **magicPlotXXXXXX**, where **XXXXXX** is a process-specific identifier.

dotsPerInch
Indicates how many dots per inch there are on the Versatec printer. This parameter is used only for computing the scale factor for plotting. Must be an integer greater than zero.

labelFont
The name of the font to use for labels in Versatec plots. This must be a file in Vfont format.

printer
The name of the printer to which to spool Versatec raster files.

spoolCommand
The command used to spool Versatec raster files. This must be a text string containing to "%s" formatting fields. The first "%s" will be replaced with the printer name, and the second one will be replaced with the name of the raster file.

swathHeight
How many raster lines of Versatec output to generate in memory at one time. The raster file is generated in swaths in order to keep the memory requirements reasonable. This parameter determines the size of the swaths. It must be an integer greater than zero, and should be a multiple of 16 in order to avoid misalignment of stipple patterns.

width The number of pixels across the Versatec printer. Must be an integer greater than 0, and must be an even multiple of 32.

plot versatec [*size* [*layers*]]
Generate a raster file describing all the the information underneath the box in a format suitable for printing on Versatec black-and-white printers, and spool the file for printing. See the plot parameters above for information about the parameters that are used to control Versatec plotting. *Size* is used to scale the plot: a scalefactor is chosen so that the area of the box is *size* inches across on the printed page. *Size* defaults to the width of the printer. *Layers* selects which layers (including labels and subcells) to plot; it defaults to everything visible on the screen.

plow *direction* [*layers*]

plow *option* [*args*]
The first form of this command invokes the plowing operation to stretch and/or compact a cell. *Direction* is a Manhattan direction. *Layers* is an optional collection of mask layers, which defaults to *. One of the edges of the box is treated as a plow and dragged to the opposite edge of the box (e.g. the left edge is used as the plow when **plow right** is invoked). All edges on *layers* that lie in the plow's path are pushed ahead of it, and they push other edges ahead of them to maintain design rules, connectivity, and transistor and contact sizes. Subcells are moved in their entirety without being modified internally. Any mask information overlapping a subcell moved by plowing is also moved by the same amount. *Option* and *args* are used in the following ways:

plow boundary
The box specifies the area that may be modified by plowing. This area is highlighted with a pale stipple outline. Subsequent plows are not allowed to modify any area outside that specified by the box; if they do, the distance the plow moves is reduced by an amount sufficient to insure that no geometry outside the boundary gets affected.

plow help
Prints a short synopsis of all the **plow** command options.

plow horizon *n*

plow horizon
The first form sets the plowing jog horizon to *n* units. The second form simply prints the value of the jog horizon. Every time plowing considers

introducing a jog in a piece of material, it looks up and down that piece of material for a distance equal to the jog horizon. If it finds an existing jog within this distance, it uses it. Only if no jog is found within the jog horizon does plowing introduce one of its own. A jog horizon of zero means that plowing will always introduce new jogs where needed. A jog horizon of infinity (**plow nojogs**) means that plowing will not introduce any new jogs of its own.

plow jogs
Re-enable jog insertion with a horizon of 0. This command is equivalent to **plow horizon 0**.

plow noboundary
Remove any boundary specified with a previous **plow boundary** command.

plow nojogs
Sets the jog horizon to infinity. This means that plowing will not introduce any jogs of its own; it will only use existing ones.

plow nostraighten
Don't straighten jogs automatically after each plow operation.

plow selection [*direction* [*distance*]]
Like the **move** or **stretch** commands, this moves all the material in the selection that belongs to the edit cell. However, any material not in the selection is pushed out of its way, just as though each piece of the selection were plowed individually. If no arguments are given, the selection is picked up by the point underneath the lower-left corner of the box and plowed so that this point lies at the cursor location. The box is moved along with the selection. If *direction* is given, it must be a Manhattan direction (e.g. **north**). The selection is moved in that direction by *amount*. If the box is in the same window as the selection, it is moved too. *Amount* defaults to 1. If there is selected material that isn't in the edit cell, it is ignored (note that this is different from **select** and **move**). If *direction* isn't given and the cursor isn't exactly left, right, up, or down from the box corner, then Magic first rounds the cursor position off to a position that is one of those (whichever is closest).

plow straighten
Straighten jogs automatically after each plow operation. The effect will be as though the **straighten** command were invoked after each plow operation, with the same direction, and over the area changed by plowing.

route *option* [*args*]
This command, with no *option* or *arg*, is used to generate routing in the edit cell to make connections specified in the current netlist. The box is used to indicate the routing area: no routing will be placed outside the area of the box. The new wires are placed in the edit cell. *Options* and *args* have the following effects:

route end [*real*]
Print the value of the channel end constant used by the channel router. If a value is supplied, the channel end constant is set to that value. The channel end constant is a dimensionless multiplier used to compute how far from the end of a channel to begin preparations to make end connections.

route help
Print a short synopsis of all the **route** command options.

route jog [*int*]

Print the value of the minimum jog length used by the channel router. If a value is supplied, the minimum jog length is set to that value. The channel router makes no vertical jogs shorter than the minimum jog length, measured in router grid units. Higher values for this constant may improve the quality of the routing by removing unnecessary jogs; however, prohibiting short jogs may make some channels unroutable.

route metal
Toggle metal maximization on or off. The route command routes the preferred routing layer (termed "metal") horizontally and the alternate routing layer vertically. By default wires on the alternate routing layer are then converted, as much as possible, to the preferred layer before being painted into the layout. Enabling metal maximization improves the quality of the resulting routing, since the preferred routing layer generally has better electrical characteristics; however, designers wishing to do hand routing after automatic routing may find it easier to disable metal maximization and deal with a layer-per-direction layout.

route netlist [*file*]
Print the name of the current netlist. If a file name is specified, it is opened if possible, and the new netlist is loaded. This option is provided primarily as a convenience so you need not open the netlist menu before routing.

route obstacle [*real*]
Print the obstacle constant used by the channel router. If a value is supplied, set the channel router obstacle constant to that value. The obstacle constant is a dimensionless multiplier used in deciding how far in front of an obstacle the channel router should begin jogging nets out of the way. Larger values mean that nets will jog out of the way earlier; however, if nets jog out of the way too early routing area is wasted.

route settings
Print the values of all router parameters.

route steady [*int*]
Print the value of the channel router's steady net constant. If a value is supplied, set the steady net constant to the value. The steady net constant, measured in router grid units, specifies how far beyond the next terminal the channel router should look for a conflicting terminal before deciding that a net is rising or falling. Larger values mean that the net rises and falls less often.

route tech
Print the router technology information. This includes information such as the names of the preferred and alternate routing layers, their wire widths, the router grid spacing, and the contact size.

route vias [*int*]
Print the value of the metal maximization via constant. If a value is supplied, set the via constant to the value. The via constant, measured in router grid units, represents the tradeoff between metal maximization and the via count. In many cases it is possible to convert wiring on the alternate routing layer into routing on the preferred routing layer ("metal") at the expense of introducing one or two vias. The via constant specifies the amount of converted wiring that makes it worthwhile to add vias to the routing.

save [*name*]

Save the edit cell on disk. If the edit cell is currently the "(UNNAMED)" cell, *name* must be specified; in this case the edit cell is renamed to *name* as well as being saved in the file *name*.**mag**. Otherwise, *name* is optional. If specified, the edit cell is saved in the file *name*.**mag**; otherwise, it is saved in the file from which it was originally read.

see *option*

This command is used to control which layers are to be displayed in the window under the cursor. It has several forms:

see no *layers*

Do not display the given layers in the window under the cursor. If **labels** is given as a layer name, don't display labels in that window either. If **errors** is given as a layer, no design-rule violations will be displayed (the checker will continue to run, though).

see *layers*

Reenable display of the given *layers*.

see no Don't display any mask layers or labels. Only subcell bounding boxes will be displayed.

see Reenable display of all mask layers and labels.

see allSame

Display all cells the same way. This disables the facility where the edit cell is displayed in bright colors and non-edit cells are in paler colors. After **see allSame**, all mask information will be displayed in bright colors.

see no allSame

Reenable the facility where non-edit cells are drawn in paler colors.

select *option*

This command is used to select paint, labels, and subcells before operating on them with commands like **move** and **copy** and **delete**. It has several forms:

select If the cursor is over empty space, then this command is identical to **select cell**. Otherwise, paint is selected. The first time the command is invoked, a chunk of paint is selected: the largest rectangular area of material of the same type visible underneath the cursor. If the command is invoked again without moving the cursor, the selection is extended to include all material of the same type, regardless of shape. If the command is invoked a third time, the selection is extended again to include all material that is visible and electrically connected to the point underneath the cursor.

select more

This command is identical to **select** except that the selection is not first cleared. The result is to add the newly-selected material to what is already in the selection.

select [**more**] **area** *layers*

Select material by area. If *layers* are not specified, then all paint, labels, and unexpanded subcells visible underneath the box are selected. If *layers* is specified, then only those layers are selected. If **more** is specified, the new material is added to the current selection rather than replacing it.

select [**more**] **cell** *name*

Select a subcell. If *name* isn't given, this command finds a subcell that is visible underneath the cursor and selects it. If the command is repeated

without moving the cursor then it will step through all the subcells under the cursor. If *name* is given, it is treated as a hierarchical instance identifier starting from the root of the window underneath the cursor. The named cell is selected. If **more** is specified, the new subcell is added to the current selection instead of replacing it.

select clear
Clear out the selection. This does not affect the layout; it merely deselects everything.

select help
Print a short synopsis of the selection commands.

select save *cell*
Save all the information in the selection as a Magic cell on disk. The selection will be saved in file *cell*.**mag**.

sideways
Flip the selection left-to-right about a vertical axis running through the center of the selection's area. If the box is in the same window as the selection, it is flipped too. If some of the selected material isn't in the edit cell, it is first copied into the edit cell and then flipped.

straighten *direction*
Straighten jogs in wires underneath the box by pulling them in *direction*. Jogs are only straightened if doing so will cause no additional geometry to move.

stretch [*direction* [*amount*]]
This command is identical to **move** except that simple stretching occurs as the selection is moved. Each piece of paint in the selection causes the area through which it's moved to be erased in that layer. Also, each piece of paint in the selection that touches unselected material along its back side causes extra material to be painted to fill in the gap left by the move. If *direction* isn't given and the cursor isn't exactly left, right, up, or down from the box corner, then Magic first rounds the cursor position off to a position that is one of those (whichever is closest).

tool [*name* | **info**]
Change the current tool. The result is that the cursor shape is different and the mouse buttons mean different things. The command **tool info** prints out the meanings of the buttons for the current tool. **Tool** *name* changes the current tool to *name*, where *name* is one of **box**, **wiring**, or **netlist**. If **tool** is invoked with no arguments, it picks a new tool in circular sequence: multiple invocations will cycle through all of the available tools.

unexpand
Unexpand all cells that touch the box but don't completely contain it.

upsidedown
Flip the selection upside down about a horizontal axis running through the center of the selection's area. If the box is in the same window as the selection then it is flipped too. If some of the selected material isn't in the edit cell, it is first copied into the edit cell and then flipped.

wire *option* [*args*]
This command provides a centerline-wiring style user interface. *Option* and *args* specify a particular wiring option, as described below. Some of the options can be invoked via mouse buttons when the **wiring** tool is active.

wire help

Print out a synopsis of the various wiring commands.

wire horizontal
Just like **wire leg** except that the new segment is forced to be horizontal.

wire leg
Paint a horizontal or vertical segment of wire from one side of the box over to the cursor's x- or y-location (respectively). The direction (horizontal or vertical) is chosen so as to produce the longest possible segment. The segment is painted in the current wiring material and thickness. The new segment is selected, and the box is placed at its tip.

wire switch [*layer width*]
Switch routing layers and place a contact at the box location. The contact type is chosen to connect the old and new routing materials. The box is placed at the position of the contact, and the contact is selected. If *layer* and *width* are specified, they are used as the new routing material and width, respectively. If they are not specified, the new material and width are chosen to correspond to the material underneath the cursor.

wire type [*layer width*]
Pick a material and width for wiring. If *layer* and *width* are not given, then they are chosen from the material underneath the cursor, a square chunk of material is selected to indicate the layer and width that were chosen, and the box is placed over this chunk. If *layer* and *width* are given, then this command does not modify the box position.

wire vertical
Just like **wire leg** except that the new segment is forced to be vertical.

what Print out information about all the things that are selected.

writeall [**force**]
This command steps through all the cells that have been modified in this edit session and gives you a chance to write them out. If the **force** option is specified, then "autowrite" mode is used: all modified cells are automatically written without asking for permission.

MOUSE BUTTONS FOR WINDOW CONTROL

For systems with pre-existing window packages, such as Suns, Magic generally uses the standard conventions for moving windows in those systems. For systems without pre-existing window packages, such as the AED line of displays, windows can be re-arranged by clicking mouse buttons in window borders. When pressed in the border area of a window, the left and right mouse buttons resize the window, instead of resizing the box as they would when the box tool is active. The buttons behave in the same way that they do for the box tool. For example, the left button moves the whole window by the lower left corner while the right button moves just the upper right corner.

The use of scroll bars and the middle button are explained in "Magic Tutorial #5: Multiple Windows".

COMMANDS FOR ALL WINDOWS

These commands are not used for layout, but are instead used for overall, housekeeping functions. They are valid in all windows.

center Adjust the view in the window under the cursor so the point underneath the cursor is at the center of the window.

closewindow
The window under the cursor is closed. That area of the screen will now show other windows or the background.

echo [**-n**] *str1 str2 ... strN*
Prints *str1 str2 ... strN* on the text terminal, separated by spaces and followed by a newline. If the **-n** switch is given, no newline is output after the command.

grow Grows a window up to full-screen size. Typing the command again causes the window to shrink down to its former size and position.

help [*pattern*]
Displays a synopsis of commands that apply to the window you are pointing to. If *pattern* is given then only command descriptions containing the pattern are printed. *Pattern* may contain '*' and '?' characters, which match a string of non-blank characters or a single non-blank character (respectively).

logcommands [*file* [**update**]]]
If *file* is given, all further commands are logged to that file. If no arguments are given, command logging is terminated. If the keyword **update** is present, commands are output to the file to cause the screen to be updated after each command when the command file is read back in.

macro [*char* [*command*]]
Command is associated with *char* such that typing *char* on the keyboard is equivalent to typing ":" followed by *command*. If *command* is omitted, the current macro for *char* is printed. If *char* is also omitted, then all current macros are printed. If *command* contains spaces, tabs, or semicolons then it must be placed in quotes. The semicolon acts as a command separator allowing multiple commands to be combined in a single macro.

openwindow [*cell*]
Open a new, empty window at the cursor position. It will be of a standard size, but may be resized with the cursor buttons. If *cell* is specified, then that cell is displayed in the new window. Otherwise the area of the box will be displayed in the new window.

over Move the window under the cursor so that it appears above all other windows.

pushbutton *button action*
Simulates a button push. Button should be **left**, **middle**, or **right**. Action is one of **up**, or **down**. This command is normally invoked only from command scripts produced by the **logcommands** command.

redo [*n*]
Redo the last *n* commands that were undone using **undo** (see below). The number of commands to redo defaults to 1 if *n* is not specified.

redraw
Redraw the graphics screen.

reset Reset the graphics controller and redraw the graphics screen. You should usually reset the graphics hardware manually before invoking this command.

scroll *direction* [*amount*]
The window under the cursor is moved by *amount* screenfulls in *direction* relative to the circuit. If *amount* is omitted, it defaults to 0.5.

send *type command*
Send a *command* to the window client named by *type*. The result is just as if

command had been typed in a window of type *type*. See **specialopen**, below, for the allowable types of windows.

setpoint [*x y* [*windowID*]]
Fakes the location of the cursor up until after the next interactive command. Without arguments, just prints out the current point location. This command is normally invoked only from command scripts produced by the **logcommands** command or by wizards that are using Magic without a color display.

If *windowID* is given, then the point is assumed to be in that window's screen coordinate system rather than absolute screen coordinates. This feature is needed for devices like the Sun 160 that have separate coordinate systems for each window. To find out a window's ID on such a device, turn on command logging and look at the file produced.

sleep *n*
Causes Magic to go to sleep for *n* seconds.

source *filename*
Each line of *filename* is read and processed as one command. No colons are necessary. Any line whose last character is backslash is joined to the following line. The commands **setpoint**, **pushbutton**, **echo**, **sleep**, and **updatedisplay** are useful in command files, and seldom used elsewhere.

specialopen [*x1 y1 x2 y2*] *type* [*args*]
Open a window of type *type*. If the optional *x1 y1 x2 y2* coordinates are given, then the new window will have its lower left corner at screen coordinates (*x1*, *y1*) and its upper right corner at screen coordinates (*x2*, *y2*). The *args* arguments are interpreted differently depending upon the type of the window. These types are known:

layout This type of window is used to edit a VLSI cell. The command takes a single argument which is used as the name of a cell to be loaded. The command

open *filename*

is a shorthand for the command

specialopen layout *filename*.

color This type of window allows the color map to be edited. See the section COMMANDS FOR COLORMAP EDITING below.

netlist This type of window presents a menu that can be used to place labels, and to generate and edit net-lists. See the section COMMANDS FOR NETLIST EDITING below.

quit Exit Magic and return to the shell. If any cells, colormaps, or netlists have changed since they were last saved on disk, you are given a chance to abort the command and continue in Magic.

underneath
Move the window pointed at so that it lies underneath the rest of the windows.

undo [*count*]
Undoes the last *count* commands. Almost all commands in Magic are now undoable. The only holdouts left are cell expansion/unexpansion, and window modifications (change of size, zooming, etc.). If *count* is unspecified, it defaults to 1. Only the last twenty modifications are recorded for undoing.

updatedisplay
Update the display. This command is normally invoked only from command scripts produced by the **logcommands** command. Command scripts that do not contain

this command update the screen only at the end of the script.

view Choose a view for the window underneath the cursor so that everything in the window is visible.

windscrollbars [*on*|*off*]
Set the flag that determines if new windows will have scroll bars.

windowpositions [*file*]
Write out the positions of the windows in a format suitable for the **source** command. If *file* is specified, then write it out to that file instead of to the terminal.

zoom [*factor*]
Zoom the view in the window underneath the cursor by *factor*. If *factor* is less than 1, we zoom in; if it is greater than one, we zoom out.

MOUSE BUTTONS FOR NETLIST EDITING

When the netlist menu is opened using the command **special netlist**, a menu appears on the screen. The colored areas on the menu can be clicked with various mouse buttons to perform various actions, such as placing labels and editing netlists. For details on how to use the menu, see "Magic Tutorial #7: Netlists and Routing". The menu buttons all correspond to commands that could be typed in netlist or layout windows.

COMMANDS FOR NETLIST EDITING

The commands described below work if you are pointing to the interior of the netlist menu. They may also be invoked when you are pointing at another window by using the **send netlist** command. Terminal names in all of the commands below are hierarchical names consisting of zero or more cell use ids separated by slashes, followed by the label name, e.g. **toplatch/shiftcell_1/in**. When processing the terminal paths, the search always starts in the edit cell.

add *term1 term2*
Add the terminal named *term1* to the net containing terminal *term2*. If *term2* isn't in a net yet, make a new net containing just *term1* and *term2*.

cleanup
Check the netlist to make sure that for every terminal named in the list there is at least one label in the design. Also check to make sure that every net contains at least two distinct terminals, or one terminal with several labels by the same name. When errors are found, give the user an opportunity to delete offending terminals and nets. This command can also be invoked by clicking the "Cleanup" menu button.

dnet *name name ...*
For each *name* given, delete the net containing that terminal. If no *name* is given, delete the currently-selected net, just as happens when the "No Net" menu button is clicked.

dterm *name name ...*
For each *name* given, delete that terminal from its net.

extract
Pick a piece of paint in the edit cell that lies under the box. Starting from this, trace out all the electrically-connected material in the edit cell. Where this material touches subcells, find any terminals in the subcells and make a new net containing those terminals. Note: this is a different command from the **extract** command in layout windows.

flush [*netlist*]
The netlist named *netlist* is reloaded from the disk file *netlist*.**net**. Any changes made to the netlist since the last time it was written are discarded. If *netlist* isn't given, the current netlist is flushed.

join *term1 term2*
Join together the nets containing terminals *term1* and *term2*. The result is a single net containing all the terminals from both the old nets.

netlist [*name*]
Select a netlist to work on. If *name* is provided, read *name*.**net** (if it hasn't already been read before) and make it the current netlist. If *name* isn't provided, use the name of the edit cell instead.

print [*name*]
Print the names of all the terminals in the net containing *name*. If *name* isn't provided, print the terminals in the current net. This command has the same effect as clicking on the "Print" menu button.

ripup [**netlist**]
This command has two forms. If **netlist** isn't typed as an argument, then find a piece of paint in the edit cell under the box. Trace out all paint in the edit cell that is electrically connected to the starting piece, and delete all of this paint. If **netlist** is typed, find all paint in the edit cell that is electrically connected to any of the terminals in the current netlist, and delete all of this paint.

savenetlist [*file*]
Save the current netlist on disk. If *file* is given, write the netlist in *file*.**net**. Otherwise, write the netlist back to the place from which it was read.

shownet
Find a piece of paint in any cell underneath the box. Starting from this paint, trace out all paint in all cells that is electrically connected to the starting piece and highlight this paint on the screen. To make the highlights go away, invoke the command with the box over empty space. This command has the same effect as clicking on the "Show" menu button.

showterms
Find the labels corresponding to each of the terminals in the current netlist, and generate a feedback area over each. This command has the same effect as clicking on the "Terms" menu button.

trace [*name*]
This command is similar to **shownet** except that instead of starting from a piece of paint under the box, it starts from each of the terminals in the net containing *name* (or the current net if no *name* is given). All connected paint in all cells is highlighted.

verify Compare the current netlist against the wiring in the edit cell to make sure that the nets are implemented exactly as specified in the netlist. If there are discrepancies, feedback areas are created to describe them. This command can also be invoked by clicking the "Verify" menu button.

writeall
Scan through all the netlists that have been read during this editing session. If any have been modified, ask the user whether or not to write them out.

MOUSE BUTTONS FOR COLORMAP EDITING

Color windows display two sets of colored bars and a swatch of the color being edited. The left set of color bars is labelled Red, Green, and Blue; these correspond to the proportion of red, green, and blue in the color being edited. The right set of bars is labelled Hue, Saturation, and Value; these correspond to the same color but in a space whose axes are hue (spectral color), saturation (spectral purity vs. dilution with white), and value (light vs. dark).

The value of a color is changed by pointing inside the region spanned by one of the color bars and clicking any mouse button. The color bar will change so that it extends to the point selected by the crosshair when the button was pressed. The color can also be changed by clicking a button over one of the "pumps" next to a color bar. A left-button click makes a 1% increment or decrement, and a right-button click makes a 5% change.

The color being edited can be changed by pressing the left button over the current color box in the editing window, then moving the mouse and releasing the button over a point on the screen that contains the color to be edited. A color value can be copied from an existing color to the current color by pressing the right mouse button over the current color box, then releasing the button when the cursor is over the color whose value is to be copied into the current color.

COMMANDS FOR COLORMAP EDITING

These commands work if you are pointing to the interior of a colormap window. The commands are:

color [*number*]
: Load *number* as the color being edited in the window. *Number* must be an octal number between 0 and 377; it corresponds to the entry in the color map that is to be edited. If no *number* is given, this command prints out the value of the color currently being edited.

load [*techStyle displayStyle monitorType*]
: Load a new color map. If no arguments are specified, the color map for the current technology style (e.g, **mos**), display style (e.g, **7bit**), and monitor type (e.g, **std**) is re-loaded. Otherwise, the color map is read from the file *techStyle.displayStyle.monitorType*.**cmap** in the current directory or in the system library directory.

save [*techStyle displayStyle monitorType*]
: Save the current color map. If no arguments are specified, save the color map in a file determined by the current technology style, display style, and monitor type as above. Otherwise, save it in the file *techStyle.displayStyle.monitorType*.**cmap** in the current directory or in the system library directory.

DIRECTIONS

Many of the commands take a direction as an argument. The valid direction names are **north**, **south**, **east**, **west**, **top**, **bottom**, **up**, **down**, **left**, **right**, **northeast**, **ne**, **southeast**, **se**, **northwest**, **nw**, **southwest**, **sw**, and **center**. In some cases, only Manhattan directions are permitted, which means only **north**, **south**, **east**, **west**, and their synonyms, are allowed.

LAYERS

The mask layers are different for each technology, and are described in the technology manuals. The layers below are defined in all technologies:

***** All mask layers.

$ All layers underneath the cursor.

errors Design-rule violations (useful primarily in the **see** command).

labels Label layer.

subcell
Subcell layer.

Layer masks may be formed by constructing comma-separated lists of individual layer names. The individual layer names may be abbreviated, as long as the abbreviations are unique. For example, to indicate polysilicon and n-diffusion, use **poly,ndiff** or **ndiff,poly**. The special character **−** causes all subsequent layers to be subtracted from the layer mask. For example, ***−p** means "all layers but polysilicon". The special character **+** reverses the effect of a previous **−**; all subsequent layers are once again added to the layer mask.

SEE ALSO

magicusage(1), ext2sim(1), sleeper(1), fsleeper(1), rsleeper(1), cmap(5), dstyle(5), ext(5), glyphs(5), magic(5), displays(5), net(5)
"Magic Tutorial #1: Getting Started"
"Magic Tutorial #2: Basic Painting and Selection"
etc.
"Magic Technology Manual #1: NMOS"
etc.

FILES

~cad/lib/magic/sys/.magic	startup file to create default macros
~cad/lib/magic/nmos/*	some standard nmos cells
~cad/lib/magic/scmos/*	some standard scmos cells
~cad/lib/magic/sys/*	Magic technology files, colormaps, etc.
~cad/lib/displays	configuration file for Magic workstations

AUTHORS

Gordon Hamachi, Robert Mayo, John Ousterhout, Walter Scott, George Taylor

BUGS

If Magic gets stuck for some reason, try using 'kill -TERM' on it to save your cells in '*cell*.**save.mag**'.

Magic will not run under the Bourne shell.

Report bugs to **magic@ucbkim.berkeley.edu**. Please be specific: tell us exactly what you did to cause the problem, what you expected to happen, and what happened instead. If possible send along small files that we can use to reproduce the bug.

NAME

magicusage − print the names of all cells and files used in a Magic design

SYNOPSIS

magicusage [**−T** *technology*] [**−p** *path*] *rootcell*

DESCRIPTION

Magicusage will print the names of all cells and files used in the design whose root cell is *rootcell*. Each line of the output is of the form

cellname ::: *filename*

where *cellname* is the name of the cell as it is used, and *filename* is the **.mag** file containing the cell. If a cell is not found, a line of the form

cellname ::: **<< not found >>**

is output instead.

If **−p** *path* is specified, the search path used to find **.mag** files will be *path*. Otherwise, the search path is initialized by first reading the system-wide .magic file in ~cad/lib/magic/sys, then the .magic file in the user's home directory, and finally the .magic file in the current directory. The most recent **path** command read from the three files determines the search path used to find cells.

In addition, a library path of ~cad/lib/magic/*techname* is used when searching for cells. By default, *techname* is the technology of the first cell read, but it may be overridden by specifying an explicit technology with the **−T** *techname* flag.

FILES

~cad/lib/magic/*tech*
~cad/lib/magic/sys/.magic
~/.magic

SEE ALSO

magic(1), magic(5)

AUTHOR

Walter Scott

NAME

meg – Mealy finite state machine compiler

SYNOPSIS

meg [*options*] *input_file*

DESCRIPTION

Meg (**Mealy Equation Generator**) is a finite state machine compiler. It translates a high level language desription of a finite state machine into several implementation and simulation formats. **Meg** uses the Mealy model for finite state machines, in which outputs are dependent upon inputs as well as state. Input is read from the named file or from *stdin* if '-' is specified.

Meg can be used to produce boolean equations in the *eqntott(1)* format, truth tables in the *PLA*(5) format, or truth tables in the *espresso*(5) format. **Meg** also generates output for the functional simulators *Slang* and *N.2.mpc*, and input for the *Sungrab* directed graph program.

OPTIONS

-s Print summary information in the file *meg.summary*.

-t Generate a human readable truth table in the file *meg.summary*.

-T Generate a truth table in *PLA(5)* format in the file *meg.tt*.

-s Generate a Slang description of the finite state machine to *stdout*. See the *Slang Slinger's Cyclopedia* for a description of Slang.

-i Generate a truth table in *ESPRESSO(5)* format in the file *meg.imp*. This format can be used as input to *espresso(1)* and *kiss(1)*.

-e Generate equations compatible with *eqntott(1)* to *stdout*.

-n or -N
Generate an N.2 description to stdout. N.2 is a functional simulator that is commercially available from ENDOT, Inc. See the **OUTPUT FORMATS** section below for more details.

-g or -G
Generate a GRAB description to stdout. GRAB is a directed graph language that can be used to print or display the finite state machine on a raster device. GRAB is still in development at this time. See the **OUTPUT FORMATS** section below for more details.

When multiple options are selected that write output to *stdout* they appear in the following order: *N.2*, *GRAB*, *Slang*, and *Equations*.

INPUT FORMAT

A **Meg** program is composed of the following sections:

INPUTS : *signal1 signal2* ... ;
An input signal list consists of the keyword **INPUTS** followed by a colon and a white-space separated list of input signal names, terminated by a semicolon. If the FSM has no inputs this line must be omitted.

OUTPUTS : *signal1 signal2* ... ;

An output signal list consists of the keyword **OUTPUTS** followed by a colon and a white-space separated list of output signal names, terminated by a semicolon. If the FSM has no outputs this line must be omitted.

RESET ON *input_signal* **TO [STATE]** *reset_state*;
This optional statement specifies the reset signal and state. When the signal *input_signal* is asserted, the FSM is forced to *reset_state* while outputting any signals specified in the reset_state specifier. The reset_state specifier is in the *next_state* syntax, described below. The signal *input_signal* must appear in the **INPUTS** list.

These sections are followed by a list of states with the following format:

state_name : *control* ;

Each state is must be given a unique state name. It's control must be specifed in one and only one of the following manners:

IF [NOT] *input* **THEN** *next_state* **ELSE** *next_state* ;
GOTO *next_state* ;
CASE (*input_selector_list*) *cases* **ENDCASE** [*default*] ;

Next_states are specified by a state name optionally followed by a list of output signals to be asserted, enclosed in parentheses.

next_state ::= *state_name* [(*output_signal_name* ...)] ;

Output signals may be set to a specific value using the syntax:

output_signal_name ::= *signal_name* [= { 0 | 1 | ? }]

Signals that do not have the "equals value" syntax are asserted to 1. Signals that do not appear in the list are asserted to 0.

Cases is a list of case selectors of the form:

{ 0 | 1 | ? }+ => *next_state* ;

Where the number of bit entries is the same as the number of input signals in the *input_selector_list*.

When the *input-selector-list* matches the bit pattern of the selector, control is transfered to *next-state*. An error occurs if the same pattern is multiply specified. Input patterns that do not match a case selector result in the default case being taken, or an error if the default is not specified.

The form of *default* is:

=> *next-state* ;

All next-state transitions must be explicitly specified. The special state name **LOOP** may be used to specify that the next state is the same as the current state. The special state name **ANY** may be used to specify that the state transition and its outputs are all *don't cares*. This is usually only used for input conditions that can't occur. This information is used by the logic optimizers to reduce the size of the PLAs.

Comments may appear anywhere in a **Meg** program. They begin with a double dash "--" and terminate with the end of the current line.

The **C** pre-processor can be invoked before the input is parsed. This enables the limited use of symbolic names in the specification of the FSM. This is illustrated in the example below.

OUTPUT FORMATS

N.2 Output Format

The N.2 output option produces a N.2 module containing a *when process* to implement the pla. The module has a port for each input and each output. There is also an output port for the binary encoding of the finite state machine source (assigned from 0 in the order of declaration in the **Meg** source), and two input ports for clocks, *SampleClock* and *AssertionClock*.

The logic samples its inputs on the falling edge of *SampleClock* and asserts its outputs on the rising edge of *AssertionClock*. This is characteristic of static PLA behavior. The initial state is the first state in the **Meg** input file.

GRAB Output Format

The name of the graph is the root of the input file name. Oval nodes are states, named with the **Meg** state name. Rectangular nodes are arcs, named *fromstate_n*, where *fromstate* is the name of the state that the arc eminates from, and *n* is the order of that arc in the state starting at 0.

EXAMPLE

The following example illustrate the use of **Meg**:

```
-- OPUS Multibus controller, Version 2.0, 2/15/85
INPUTS:                 INIT OP1 OP2 SWR MACK;
OUTPUTS:    WAIT MINIT MRD SACK MWR DLI;

-- Define the encoding for the processor operations
-- The C preprocessor does textual substitution on these macros.
#define OP     OP1 OP2
#define NOP    0 0
#define HALT 0 1
#define READ 1 0
#define WRITE          1 1
#define XOP    ? ?

-- describe the reset logic
reset on INIT to swr0(MINIT WAIT=?);

-- wait for slave write from multibus to activate us
-- Note the specification of a don't care output.
swr0:   if SWR then swr1(MINIT WAIT=? DLI SACK)
        else swr0(MINIT WAIT=?);

-- wait for slave write to go away
swr1:   if SWR then swr1(MINIT WAIT=? SACK)
        else op0;

-- wait for request from processor
-- The C preprocessor expands the symbolic name OP into two input
-- signal names, and the symbolic OPs, e.g. NOP, into their binary encoding.
-- Note the use of the special state ANY to indicate a don't care
-- state transition.
op0:    case (SWR MACK OP)
                0 ? NOP => op0;
```

```
                0 ? HALT => swr0(MINIT WAIT=?);
                0 0 READ => op1(WAIT MRD);
                0 0 WRITE => op1(WAIT MWR);
                0 1 READ => op0(WAIT);
                0 1 WRITE => op0(WAIT);
                1 ? XOP => swr1(MINIT WAIT=? DLI SACK);
        endcase => ANY;

-- after starting transaction, wait for acknowledge
op1:    case (SWR MACK OP)
                0 0 READ => op1(MRD);
                0 0 WRITE => op1(MWR);
                0 1 READ => op0(DLI);
                0 1 WRITE => op0;
                1 ? READ => swr1(MINIT WAIT=? DLI SACK);
                1 ? WRITE => swr1(MINIT WAIT=? DLI SACK);
        endcase => ANY;
```

SEE ALSO

espresso(1), kiss(1), mpla(1), eqntott(1), peg(1), espresso(5), pla(5).
R. Rudell, A. Sangiovanni-Vincentelli, G. De Micheli, *A Finite-State Machine Synthesis System.*

AUTHOR

David A. Wood

BUGS

Meg is a direct descendent of **Peg**, by Gordon Hamachi, and inherits many bugs and limitations.

NAME

mkcp – Make Crystal parameters

SYNOPSIS

mkcp [vlow vinv vhigh]

DESCRIPTION

Mkcp is a program that generates the model parameters used by Crystal's slope model. It reads a SPICE deck from its standard input, runs SPICE several times using modified versions of that SPICE deck, extracts Crystal parameter information from the SPICE output, and writes the parameters to standard output. A single run of Mkcp will generate all of the slope parameters for a single transistor type driving its output either high or low (but not both in the same Mkcp run).

The SPICE input deck describes a simple circuit to test the characteristics of a single transistor type. See the files in ~cad/lib/mkcp for examples of input decks. Each deck must contain two particular capacitor cards: one with "c1" in the first columns, and one with "c2" in the first columns. The "c1" card must describe the capacitance on the gate of the transistor being modelled. It has the standard format for a SPICE capacitor card, except that there may be any number of capacitance values, separated by spaces (each of the capacitances *must* be specified in pfs). Mkcp makes one SPICE run with each of the given values and expects that changing the capacitance will change the edge speed of the signal on the transistor gate. The first capacitance value should generate an edge that rises or falls as quickly as possible. The "c2" card describes the capacitance being driven by the transistor, and is used to compute the effective resistance of the device. This card is not modified by Mkcp.

The deck must also have a ".print" card that generates three columns of output. The first column must be time, the second column must be the voltage on the gate of the transistor being modelled, and the third column must be the output being driven by the transistor. There must be a ".tran" card in the deck that allows enough simulation time for the the output to stabilize when using the first value of c1. Mkcp will modify the ".tran" card before each run after the first one so that the simulation time will be long enough for signals to settle in that run.

Mkcp makes one SPICE run for each c1 value that is given. After each run it outputs four values: the edge speed on the gate of the transistor, the ratio of that edge speed to the edge speed on the output of the transistor during the first SPICE run, the effective resistance of the transistor (delay from input to output divided by c2), and the edge speed on the output being driven by the transistor, divided by c2. If the driving transistor is a minimum-size device, then the last three of these values are exactly the slope parameters needed by Crystal. If the transistor isn't minimum-size, then you must divide each of the last two parameters by the length/width ratio of the driving transistor.

Three voltages are used by Mkcp to compute edge speeds and resistances. They can be specified on the command line as *vlow*, *vinv*, and *vhigh*. If any of these voltages is given, then all must be given. The defaults are 2.0 volts for *vlow*, 2.2 volts for *vinv*, and 2.4 volts for *vhigh*; these are about right for the standard MOSIS nMOS process. *Vinv* is the logic threshold voltage; the delay from input to output is the time from when the input reaches *vinv* to when the output reaches *vinv*. *Vlow* and *vhigh* are used to compute edge speeds: the speed of an edge is the time it takes its voltage to pass from *vlow* to *vhigh* (or vice versa) divided by the voltage difference between *vlow* and *vhigh*.

HINTS FOR USING MKCP

You should choose the c2 value and the c1 values so that the range of edge speed ratios is about what you expect to encounter when running Crystal. The fewer data points you use for each transistor, the faster Crystal will run. However, Crystal uses linear interpolation between points, so use enough points to keep the interpolation error low.

Make sure that you use a relatively large value for c2. If you use a small value for c2, then the delay of the circuit will be determined primarily by the internal capacitance of the driving transistor (which Mkcp ignores). To get accurate results, use a c2 value that's larger than the internal capacitance of the circuit. If you're not sure whether you've chosen a good c2 value, try doubling it; if you get a different effective resistance for the same edge speed ratio, then your initial c2 value was probably too small.

It's not at all unusual for a resistance value to come out negative. This happens if the *vinv* value you're using isn't exactly the logic threshold of the circuit; under some conditions the output may reach *vinv* before the input. This is nothing to worry about: negative values can be entered into Crystal and will produce correct results.

SEE ALSO

crystal(1)
J. Ousterhout, *Using Crystal for Timing Analysis*

FILES

~cad/lib/mkcp/*

AUTHOR

John Ousterhout

NAME

mpanda – technology independent PLA generator for multiply-folded PLAs

SYNOPSIS

mpanda [**-acpvV**] [**-s** *style*] [**-G** *num*] [**-S** *numR] [-***l** *num*] [**-t** *template_name*] [**-M** *num*] [**-D** *num1 num2*] [**-o** *output_file*] *input_file*

DESCRIPTION

mpanda is a PLA generator that generates multiply-folded, simply-folded, and non-folded PLAs. **MPanda** is a program written with the **mpack**(3) system.

The input format for **mpanda** is compatible with the *.machine* output of **pleasure**(1). Including the *.machine* control line in the **pleasure** input file results in the proper output format for **mpanda**.

Input files to **mpanda** contain *control lines* and a personality matrix of the PLA to be made. Each control line must begin with a "." (dot or period). The following control lines are understood by **mpanda**:

.[and | or] *num1 num2* [*num3* ...]

This line must be in the input file. It describes the structure of the PLA, with the **and** or **or** specifying that the leftmost plane is an *AND* or *OR* plane. The *AND* plane is the input plane and the *OR* plane is the output plane. The numbers following the first plane designator are the numbers of inputs/outputs (depending on which plane is first) in each successive plane. For instance, the control line: " *.or 9 3 4 5* " means that this PLA has an OR-AND-OR-AND structure with 9 outputs in the first OR plane, 3 inputs in the next AND plane, 4 outputs in the next OR plane, and 5 inputs in the last AND plane. Note that at least two numbers must follow the first plane designator since a PLA must have at least one AND and OR plane.

.row [*number of rows*]

This line describes the height of the input personality matrix.

.top [*l1 l2 l3* ...]
.bottom [*l1 l2 l3* ...]
.left [*l1 l2 l3* ...]
.right [*l1 l2 l3* ...]

These control lines list the labels for inputs and outputs along the top, bottom, left, and right respectively, of the PLA. The label "0" is not allowed. Note that these labels are not designated as being either inputs or outputs. The AND-OR-AND... structure of the PLA is determined by the **.and|or** control line.

.product [*l1 l2 l3* ...]

This control line lists the labels for product rows within the PLA. These labels are used for debugging within the PLA and can be automatically numbered if no labels are put in. The label "0" is not allowed.

.end

This line signals the end of the input file.

The personality matrix format is compatible with **pleasure**, see PLA(5) for details. The table below summarizes the symbols **mpanda** accepts.

Symbols for *AND* Plane		
Contact Signal	No Contact	Explanation
1	-	Normal contact, no splits or folds
!	_	Split below
;	,	Fold to the right
:	.	Split below and fold to the right

Symbols for *OR* Plane		
Contact to Output Signal	No Contact	Explanation
I	~	Normal contact, no splits or folds
i	=	Split below
|	'	Fold to the right
j	"	Split below and fold to the right

Additional Symbols	
Symbol	Explanation
*	Input buffer
+	Output buffer
X	No buffer
c	Contact within AND or OR plane
> <	Routing lines to contacts for multiple folds

That is,

An example of an input file is shown below.

```
.and   2  4  3  1
.row   8
  *  X     ++X+    X  *  X     +
X1---    ~i~I     --1---    ~X
X----    I~~i     ,;1-1-    iX
X_!--    ~|~I     1-_!--    ~X
X1--1    |~~I     -----1    ~X
X--1-    i~~~     ---1--    ~X
*c                           X
X>c                          X
X!_1-    ~|~I     1-:,--    ~X
X----    I~~~     1-----    IX
X---1    ~~~~     --1---    IX
  *  *    ++++    *  *  *     +
.end
```

Note that the AND plane is expanded such that each input is represented as two columns, one for the signal, the other for its complement. Note also, that only one contact row can occur between seuccesive product rows.

STYLES OF PLAs AVAILABLE

As of 2/28/85, there are no templates for mpanda. A Caesar format template for panda is provided as a starting point for template designers. Here is the description of that template:

CS3 CMOS static version with p-channel pull-ups as resistive loads. 3 micron MOSIS rules. Micron-based rules, not lambda-based. The pull-ups are placed in the between the AND and OR planes.

It is easy to create a template for a new style of PLA, and **mpanda**(5) has information on how to do it.

OPTIONS

-a produce **Magic**(1) format (this is the default)

-c produce CIF format

-p (pipe mode) Send the output to **stdout**.

-v Be verbose, and show (in the **magic** output) how the PLA was constructed from its basic components.

-V Be verbose, and print out information about what mpanda is doing. This option implies **-v**.

-s The next argument specifies the style of PLA to generate. (This causes mpanda to use the file **~cad/lib/mpanda/pa-***style***.mag** as its template).

-G Insert an extra ground line every *num* rows in the AND plane and every *num* columns in the OR plane. This defaults to what is appropriate for the static 3 micron **CMOS** PLA, approximately every 10 rows. Note that for styles other than **CS3**, this option should be used for specifying extra ground lines.

-S Stretch power and ground lines by *num* lambda. This defaults to whatever is appropriate for the corresponding **CMOS** PLA. Note that for styles other than **CS3**, this option should be used for specifying stretching power and ground lines.

-l Set cif output style, where *style* is a Magic cif output style. (See mpack(3) for details.)

-t The next argument specifies the template to use, this normally defaults to the standard library. This option is useful for generating styles of PLAs that are not included in the standard library.

-D *num1 num2*

The *Demo* or *Debug* option. This option will cause **mpack** to place only the first *num1* tiles, and the last *num2* of those will be outlined with rectangular labels. In addition, if a tile called **blotch** is defined then a copy of it will be placed in the output tile upon each call to the *align* function during the placing of the last *num2* tiles. The blotch tile will be centered on the first point passed to *align*, and usually consists of a small blotch of brightly colored paint. This has the effect of marking the alignment points of tiles. The last tile painted into is assumed to be the output tile.

-o The next argument is taken to be the base name of the output file. The default is the input file name with any extensions removed. If the input comes from the standard input and the **-o** option is not specified then the output will go to the standard output.

input_file

The file containing the control lines and personality matrix. See PLA(5) for a

description of the personality matrix symbols. If this filename is omitted then the input is taken from the standard input.

FILES

~cad/lib/mpanda/pa-*.mag -- standard templates for PLAs

SEE ALSO

eqntott(1), espresso(1), pleasure(1), pla(5), mpla(5), mpack(3)

AUTHOR

Grace H. Mah

HISTORY

This program is panda converted from tpack to mpack. Conversion by Bob Mayo.

BUGS

The -G and -S options have no way of knowing what the grounding requirements are for the style of PLA actually being generated.

This program inherits any bugs that may exist in **mpack**(3).

This program isn't very useful until someone designs templates for it!

NAME

mpla – technology independent PLA generator

SYNOPSIS

mpla [-acv] [-s *style*] [-o *output_file*] *input_file*

DESCRIPTION

mpla is a PLA generator that generates PLAs in several different styles and technologies. The input format is compatible with **eqntott**, see PLA(5) for details. **Mpla** does not handle split and folded PLAs.

Mpla is a program written with the Mpack system.

STYLES OF PLAs AVAILABLE

The following styles of PLAs are currently supported:

Bcis Buried contacts, nMOS, cis version (inputs and outputs on same side of the PLA). Clocked inputs and outputs are supported. Berkeley design rules.

Btrans

Buried contacts, nMOS, trans version (inputs and outputs on opposite sides of the PLA). Clocked inputs and outputs are supported. Berkeley design rules.

SCD3cis

Scalable CMOS cis version, MOSIS 1.5/2.0/3.0 micron SCMOS process, dynamic PLA with two separate precharge/evaluate lines (switched ground) for the AND (clk1) and OR (clk2) planes, no inverting buffers between the planes. The n-channel transistors in the AND and OR planes are labeled with "Cr:In$" to speed timing analysis using crystal. Frequent ground and metal lines should be used to avoid high diffusion resistance in the AND and OR planes; therefore, "-G 10" or less is recommended. Clocked inputs and outputs are provided. The pass gates should be driven a complementary clock (clka and clkabar for inputs and clkb and clkbbar for outputs). To avoid charge sharing problems, the output clock uses two inverters and a pass gate. Therefore, **your PLA equations should be modified to implement the complement for all the outputs**. This template works well in a multi-clocking environment. However, if you must operate using only a two phase clock, one possible arrangement is to sample the input on phi1 , output the PLA on phi2 by assigning: phi1 = clka, phi1bar = clkabar and clk1, phi2 = clkb and clk2, phi2bar = clkbbar. One limitation to this arrangement is that the non-overlap time between phi1 and phi2 must be sufficiently long to allow the product term lines from the AND plane to set up the inputs to the OR plane before phi2 goes high to cause the OR plane evaluation.

SCD3trans

Same as SCD3cis except trans version.

SCS3cis

Scalable CMOS cis version, MOSIS 1.5/2.0/3.0 micron SCMOS process, pseudo-nmos static PLA with p-channel pullups (pullup/pulldown = 1/2). Recommended ground straps for AND and OR plane diffusion runs is "-G 10" or less. Clocked inputs and outputs are provided. The pass gates should be driven by a complementary clock (clka and clkabar for inputs and clkb and clkbbar for outputs). The clocked outputs require Vdd to be attached to the main PLA Vdd (all Vdd segment are labeled).

SCS3trans

Same as SCS3cis except trans version.

It is easy to create a template for a new style of PLA, and mpla(5) has information on how to do it.

OPTIONS

-I Clock the inputs to the PLA, if this feature is supported for this style.

-O Clock the outputs to the PLA, if this feature is supported for this style.

-G Insert an extra ground line every *num* rows in the AND plane and every *num* columns in the OR plane. This defaults to whatever is appropriate for the corresponding **nMOS** PLA.

-S Stretch power and ground lines by *num* lambda. This defaults to whatever is appropriate for the corresponding **nMOS** PLA.

-v Be verbose, and show (in the Magic output) how the PLA was constructed from its basic components.

-V Be verbose, and print out information about what mpla is doing. This option implies **-v**.

-a produce Magic format (this is the default)

-c produce CIF format (see mpack(3) for details)

-o The next argument is taken to be the base name of the output file. The default is the input file name with any extensions removed. If the input comes from the standard input and the **-o** option is not specified then the output will go to the standard output.

-s The next argument specifies the style of PLA to generate. (This causes mpla to use the file **~cad/lib/mpla/p-*style*.mag** as its template).

-l Set the cif output style to *style*. *Style* is a cif output style as found in the Magic technology file.

-t The next argument specifies the template to use, this normally defaults to the standard library. This option is useful for generating styles of PLAs that are not included in the standard library.

input_file
The file containing the truth_table. If this filename is omitted then the input is taken from the standard input (such as a pipe).

FILES

~cad/lib/mpla/p*.mag – standard templates for PLAs

SEE ALSO

eqntott(1), espresso(1), pla(5), mpla(5), mpack(3)

HISTORY

This is a port of the program 'tpla' to the Mpack system.

AUTHOR

Program by Robert N. Mayo.

Robert Mayo and Fred W. Obermeier: Bcis, Btrans, templates.

SCD3cis, SCD3trans, SCS3cis, and SCS3trans templates by Fred W. Obermeier.

BUGS

The -G and -S options have no way of knowing what the grounding requirements are for the style of PLA actually being generated.

This program inherits any bugs that may exist in mpack(3).

NAME

mquilt − assemble tiles into a rectangular array

SYNOPSIS

mquilt [-acv] [-s standardTemplate] [-t *template*] [-o *output_file*] *text_file*

DESCRIPTION

The user of MQuilt first creates a Magic file, called the *template*, containing a circuit layout over which single-character rectangular labels have been placed. These labels define blocks of the circuit called *tiles*. Using a text editor, the user then creates an array of characters (each line defines one row in the array). MQuilt reads in the array of characters and produces a layout where each character is replaced by the tile of the same name. Spaces and blank lines in the text file are ignored.

For example, we can produce a 3X3 checkerboard with this input file:

```
ABA
BAB
ABA
```

The template file would contain rectangular labels called A and B. The paint and subcells underneath these labels would be placed in the output file in a checkerboard fashion.

Tiles are normally placed so that they abut with each other in the following fashion: the lower edges of all tiles in a row are aligned, tiles are packed together horizontally as closely as possible within a row, and the first tile in a row touches the first tile in the row above it and the first tile in the row below it.

If we wish tiles to be spaced a certain distance apart, instead of what was described previously, we can use *spacing* tiles. Spacing tiles are tiles which indicate, by their size, how far apart two tiles should be spaced. For horizontal spacing, the single-character name of a spacing tile should be placed in parentheses between the names of the two tiles on either side of it. The left edges of the two tiles will be spaced apart by the width of the spacing tile. For example, the form "AB" places tiles A and B next to each other while "A(C)B" places them apart by a distance determined by C. If C is of zero width, A and B will be placed on top of each other. If C is the same width as A, A and B will abut (note that "A(A)B" is the same as "AB"). If the width of C is less than the width of A the tiles will overlap, and if C has a width greater than A they will be separated.

Spacing tiles may also be used to control the vertical spacing. A spacing tile at the beginning of a row (such as "(C)AB") will cause the bottom of the first tile in this row (in this case tile A) to be separated from from the bottom of the first tile in the row above by a distance equal to the height of the spacing tile.

MQuilt is a small program written with the Mpack system.

OPTIONS

-a produce Magic format (this is the default)

-c produce CIF format (see **-l** under mpack(1) for details).

-v be verbose (sequentially label the tiles in the output, for debugging purposes)

-o The next argument is taken to be the base name of the output file. The default is the input file name with any extensions removed.

-t The next argument specifies the template to use.

-s *style* Use the template with the name **q-***style* located in ~cad/lib/mquilt.

text_file

The name of mquilt's text file. If this filename is omitted then the input is taken from the standard input (such as a pipe). If the input comes from the standard input and the **-o** option is not specified then the output will go to the standard output.

other options

Several other options are inherited from mpack(1).

FILES

~cad/lib/mquilt/q-* -- location of standard templates

SEE ALSO

mpack(1), vlsifont(1), quilt(1)

HISTORY

This program is a port of quilt set up to generate Magic files instead of Caesar files.

AUTHOR

Robert N. Mayo

BUGS

This program inherits any bugs that may exist in mpack(3).

NAME

peg − finite state machine compiler

SYNOPSIS

peg [−s] [−t] [file]

DESCRIPTION

Peg (*P*LA *E*quation *G*enerator) is a finite state machine compiler. It translates a high level language description of a finite state machine into the logic equations needed to implement the state machine design. *Peg* uses the Moore model for finite state machines, in which outputs are strictly a function of the current state. Input is read from the named file or from *stdin* if no file is specified.

A set of equations is generated on standard output. The equations are in the *eqn* format used by *eqntott*. Output from *peg* may be piped directly to PLA generators such as *mpla* thus:

peg *infile* | **eqntott** | **mpla −c −s Bcis −I −O −o** *outfile*

This command generates a PLA implementation of the finite state machine in the file *outfile.cif*.

The PLA will have clocked, dynamic latches on all inputs and outputs. From left to right, the PLA inputs and outputs are the fsm inputs, fsm state inputs, fsm state outputs, and fsm outputs. The feedback lines connecting the next-state outputs to the curent-state inputs must be manually added to the resulting circuit.

Peg options have the following meanings.

−t Generate a truth table for the fsm in the file *peg.summary*.

−s Print summary information in the file *peg.summary*.

PROGRAM STRUCTURE

A *peg* program is composed of a list of input signal names, a list of output signal names, and a list of state descriptions, in that order. The input and output lists are optional.

Inputs

An input signal list consists of the keyword **INPUTS** and a list of fsm input signal names, terminated with a semicolon. Every input list must have at least one input. If the fsm has no inputs, this statement is omitted. PLA inputs will have the left-to-right ordering specified in the **INPUTS** list.

Outputs

A list of output signal names begins with the keyword **OUTPUTS** and is terminated with a semicolon. PLA outputs will have the ordering specified in the **OUTPUTS** list.

State List

The remainder of a *peg* program consists of a list of state definitions. A state definition has the form

[state-name] : [**ASSERT** *signal-list* ;] [control ;]

There is at most one **ASSERT** statement per state definition. Asserted output signals are set to 1. Signals that are not asserted have value 0.

There is at most one control statement per state definition. Control may be one of

IF [**NOT**] *input* **THEN** *state-name* [**ELSE** *state-name*]

GOTO *state-name*
CASE (*input-signal-list*) *selectors* **ENDCASE** [*default*]

Each case selector specifies the next-state for a particular set of values of the **CASE** input signals. Case selectors are lines of the form

{ **0** | **1** | **?** }+ => *state-name*

Case selectors are applied one at a time from top to bottom until one is found that applies to the particular set of inputs. Because of this, one must be particularly careful when using don't-cares in case selectors.

If no control is specified – by omitting the **ELSE** clause from an **IF**, by specifying a **CASE** with no default, or by omitting control information entirely – *next state* defaults to the next sequential state on the state list. The default next state is undefined for the last state in the program. Therefore, the *next-state* must be explicitly specified for this state. The special state name **LOOP** may be used to specify that the next state is the same as the current state.

Comments
Comments may appear at any location in a *peg* program. They begin with a double dash, "--", and terminate at the end of the line on which they appear.

Reset Logic
There are two ways of handling fsm initialization. If the keyword **RESET** appears as one of the input signals, then the fsm will jump to the first state on the state list when the signal **RESET** is asserted high. Alternatively, the user may force a jump to the first state on the state list by adding logic to the PLA state outputs to pull all of the state output lines low when a reset is desired.

Example
The following *peg* program illustrates a variety of features:

```
--Decode inputs a, b, and c into
--0, 1, 2, 3, or "other".
INPUTS:  RESET Select a b c;
OUTPUTS:
         Found0 Found1 Found2 Found3 FoundOther;
Start:   --This is the reset state
         IF NOT Select THEN LOOP;
:        CASE (a b c) --Second state
           0 0 0 => Zero;
           0 0 1 => One;
           0 1 0 => Two;
           0 1 1 => Three;
         ENDCASE =>Other;
Zero:    ASSERT Found0; GOTO Start;
One:     ASSERT Found1; GOTO Start;
Two:     ASSERT Found2; GOTO Start;
Three:   ASSERT Found3; GOTO Start;
Other:   ASSERT FoundOther; GOTO Start;
```

SEE ALSO

mpla(1), eqntott(1)
Gordon Hamachi, *Designing Finite State Machines with Peg*

FILES

peg.summary: Summary information file

AUTHOR

Gordon Hamachi

BUGS

The parser quits after the first error is found.

The interpretation of ambiguous case statements has been redefined. Existing *peg* programs may exhibit new behavior.

NAME

pleasure − A PLA Folding Program

SYNOPSIS

pleasure [-abegpsu] [< input] [> output]

DESCRIPTION

Pleasure is a program that performs topological optimization of PLA's using a technique called folding. The primary goal is to optimize the silicon area occupied by the PLA. Two different heuristic algorithms are used in order to achieve the best area minimization.

Pleasure can be run in an interactive mode with the option **-b**. In this case, the input file and output file should be specified on the command line using the file redirection capabilities of the shell. In interactive mode, the following commands are understood by pleasure:

clear Clears the program before restarting with new folding instructions.

help This command provides a list of the available commands.

quit Stops the execution.

read Reads the input file. *pleasure* will ask for the input file name.

reset *options*
Resets the folding instructions. *options* is one of *all, cofold, rofold, window, first, group, side, label, array* or *option*.

run Runs the folding algorithm.

save Saves the output file. *Pleasure* will ask for the format choice and the filename.

select Runs one step of the folding algorithm on user selected row or column folding candidates. Folding instruction '.option' should be set to only one of 'heu1' or 'heu2'.

show Shows the folded/unfolded PLAs on standard output. *pleasure* will ask for the format choice and the heuristic schemes.

status Will show the status of the folding instructions set already.

step Runs one step of the folding algorithm. Folding instruction '.option' should be set to only one of 'heu1' or 'heu2'.

. (any folding instructions)
Sets the folding instructions in the interactive mode. These instructions are identical to the ones to be inserted in the input file under running the program in batch mode. They are explained in detail in *pleasure*(5). Folding instructions are as follows (remember that the brackets indicate optional responses, and are not actually typed):

.area cell-length buffer-length ground-ratio
.cofold [and[=mult]] [or[=mult]]
.rofold [aoa | oao | mult]
.label in1 in2 ... out1 out2 ...
.first [row | column]
.top [c1 c2 ... cn]
.bottom [c1 c2 ... cn]
.left [r1 r2 ... rn]
.right [r1 r2 ... rn]
.order left | right
.window row | column | contact [n1 l1 u1 ... nn ln un]
.array left | right [c1 c2 c3 ... cn]

.group (c1 c2) [(c5 c6 c7 c8) ...]
.side
.option prtall | heu1 | heu2 | expand | unmerged

The input file consists of the folding instructions and the PLA symbolic table. The output file from the logic minimizer *espresso*(1) can be given to *pleasure* by adding the desired folding instructions. Comment lines begin with #.

Pleasure has two different output formats. The default output format is compatible with the *Panda*(1) input format. The option **-g** requests a more human readable format. The optional output format is also required for the program *pain*(1) (see below).

Both output formats report the area estimation based on the minimum rectangular area which encloses the PLA. The parameters used for the area calculation may be specified with the .area folding instruction.

Simply-folded PLAs can be assembled by the program *plaid*(1) provided that the *pleasure* output file (human readable format) is processed by the program *pain*(1) first, to convert the symbolic table to the format described in *pla* (5). Both simply-folded and multiply-folded PLAs can be assembled by the program *panda*.

OPTIONS

- **-a** estimates the area by multiplying the number of columns by the number of rows. It won't consider the buffer area or extra ground line area.
- **-b** runs the program in batch mode (noninteractive)
- **-e** expands the AND plane and imposes group constraints on the signal and its complement.
- **-g** sets the output format as human readable
- **-p** ignores the output (OR) plane phase. It will regard "0" (negative phase) as no transistor in the or-plane.
- **-s** supresses the verbose mode in interactive mode.
- **-u** prevents blank columns (columns with no transistors) from being deleted. This option works only for simple column folding, but is required for a direct path into *panda* if there are blank columns. If there are blank columns which are deleted, then the buffer positions might be incorrect for *panda*.

SEE ALSO

pla(5), eqntott(1), espresso(1), pleasure(5), panda(1)
pain(1), plaid(1)

DIAGNOSTICS

The input routine gives out warning and/or error messages in case of incompatible or wrong folding instructions. The self-checking routine compares the folded PLA with the original unfolded PLA automatically after folding and reports the result if any errors are found.

AUTHOR

Giovanni De Micheli
Modifications by Duksoon Kay

BUGS

As a present limitation for the input PLA, the maximum number of columns or rows is 300, and the maximum number of transistors is 10,000. The maximum number of groups is 50, and each group can have at most 6 elements.

NAME

rsleeper – run sleeper remotely

SYNOPSIS

rsleeper *remotemachine*

DESCRIPTION

Rsleeper is used if you wish to run a program such as *magic*(1) on a different machine (*remotemachine*) than the one to which a graphics terminal is attached, and the local graphics terminal has a login process.

To use it, log in on the graphics terminal and run *rsleeper*. The tty printed will be on the remote machine, and can be used as the graphics display device for programs such as *magic*(1).

For *rsleeper* to work, there must be an account **sleeper** on the remote machine. Its login shell should be the program *sleeper*(1). Users must be able to rlogin to the **sleeper** account without supplying a password.

SEE ALSO

fsleeper(1), magic(1), sleeper(1), displays(5)

NAME

sim2spice − convert from .sim format to spice format

SYNOPSIS

sim2spice [−d defs] file.sim

DESCRIPTION

Sim2spice reads a file in **.sim** format and creates a new file in spice format. The file contains just a list of transistors and capacitors, the user must add the transistor models and simulation information. The new file is appended with the tag **.spice**. One other file is created, which is a list of **.sim** node names and their corresponding spice node numbers. This file is tagged **.names**.

Defs is a file of definitions. A definition can be used to set up equivelences between **.sim** node names and spice node numbers. The form of this type of definition is:

set *sim_name spice_number* [*tech*]

The *tech* field is optional. In NMOS, a special node, 'BULK', is used to represent the substrate node. For CMOS, two special nodes, 'NMOS' and 'PMOS', represent the substrate nodes for the 'n' and 'p' transistors, repectively. For example, for NMOS the **.sim** node 'GND' corresponds to spice node 0, 'Vdd' corresponds to spice node 1, and 'BULK' corresponds to spice node 2. The *defs* file for this set up would look like this:

```
set GND 0 nmos
set Vdd 2 nmos
set BULK 3 nmos
```

A definition also allows you to set a correspondence between **.sim** transistor types and and spice transistor types. The form of this definition is:

def *sim_trans spice_trans* [*tech*]

Again, the *tech* field is optional. For NMOS these definitions would look as follows:

```
def e ENMOS nmos
def d DNMOS nmos
```

Definitions may also be placed in the '.cadrc' file, but the definitions in the *defs* file overrides those in the '.cadrc' file.

SEE ALSO

ext2sim(1), magic(1), spice(1), cadrc(5), ext(5), sim(5)

AUTHOR

Dan Fitzpatrick CMOS fixes by Neil Soiffer

BUGS

The only pre-defined technologies are **nmos**, **cmos-pw**, and **cmos** (the same as **cmos-pw**). Only one definition file is allowed.

NAME

sleeper − acquire a graphics terminal and hang around

SYNOPSIS

sleeper

DESCRIPTION

Certain programs such as *magic*(1) can require the use of a graphics terminal separate from the terminal used to run the program. If the graphics terminal has an ordinary login process running on it, it is necessary to run *sleeper* to acquire ownership of the terminal, set up its modes appropriately, and prevent the login process from eating input destined for the CAD tool.

When *sleeper* is run, it will print a message of the form:

tty is:
/dev/tty*name*

Here, **/dev/tty***name* is the device name of the graphics terminal. This is particularly useful when sleeper is run over the network, or when using *fsleeper*(1) or *rsleeper*(1).

Sleeper may be killed by sending it two **QUIT** signals within ten seconds of each other. This is most easily done by typing two quit characters (usually CTRL-\ or CTRL-SHIFT-L) in a row on the graphics terminal.

For *sleeper* to work best, there should be an account named **sleeper**, whose login shell is ~cad/bin/sleeper and with no password. This enables users to log in as the user **sleeper**, and is also necessary for the programs *fsleeper*(1) and *rsleeper*(1) to work. (Note that you will have to include the full pathname of ~cad/bin/sleeper in /etc/passwd; the initial ~cad does not get expanded).

SEE ALSO

fsleeper(1), magic(1), rsleeper(1)

NAME

spice2summary − summarize numerical spice output

SYNOPSIS

spice2summary *options namesfile* < *spice_output*
or
spice ... | **spice2summary** *options namesfile*

DESCRIPTION

Spice2summary can quickly provide information on a circuit's operating speed and power. **Spice2summary** reads spice output and determines critical signal information from the tabular numeric portion. The spice lines for the numeric format must be set up so that the first column gives the time values, and the second contains the voltage for a reference input node. The rest of the columns can contain any combination of voltages and currents of interest. (See FORMAT for suggested spice format.) The analysis for each column of numbers depends on its type: voltage or current.

For voltage signals, the stable values in response to applied low (Gnd) and high (Vdd) voltages on the reference input are determined (which indicates its logic relationship to the input signal). Also, the time over which this signal is stable is indicated. This is the period that the signal stays within each logic threshold (as set by **−vsl** and **−vsh**).

Signal transitions are measured by the time it takes to pass between the low and high logic thresholds (as set by **−vtl** and **−vth**). Four points are naturally defined by the rising and falling transients for each waveform. Rise time is determined by the time spent moving from the low to the high logic thresholds (2 points). Fall time is determined by the time spent moving from the high logic threshold to the low logic threshold (2 points).

Propagation times from the rising and falling portions of the input reference signal (second column) to the other nodes are also printed. For generality, propagation times are determined by another set of logic thresholds as defined by **−vpl** and **−vph**. Which of these points are used to determine the propagation delay from the input signal and the output signal are controlled by eight flags.

The total delay can be broken down into three distinct time portions which are defined by the logic levels: input change time, intersignal propagation delay, and output change time. First is the rise time (or fall time) of the input reference signal. It is measured from the last time that the reference signal is within the low logic level to the first time that it is within the high logic level (high to low for the input fall time). Next, the intersignal propagation delay is taken from last point where the input has stabilized to the time when the output starts to change. More precisely, the intersignal propagation time is calculated from the point the input reaches the high logic level (low for falling edge) to the time that the output begins to leave the appropriate logic level. The logical relationship of the reference signal to the output signal is determined by checking the output value at the point when the input signal is about to leave each logic threshold. Finally, the transition time for the output signal between the other logic level is defined to be the output change time (i.e. rise time for rising signals and fall time for falling signals).

Each set of four flags is associated with the input rising or input falling portion of the signal. The four possible options for the delay time for the rising portion of the input signal are: **−ripo**, **−rip**, **−rpo** and **−rp**. The delay time from the rising input is calculated by totaling the times associated with each letter given: **i**nput change time, intersignal **p**ropagation time and **o**utput change time. (Similarly **f** stands for the falling input signal for similar options: **−fipo**, **−fip**, **−fpo** and **−fp**). Note that the propagation delay may be a negative number for a slow input and a fast switching output. The default settings are **−rip** and **−fip**. For pessimistic analysis, use the **−ripo** and **−fipo** options since these give the total time the input and output signals are in transition along with the intersignal

propagation delay.

For current signals, the stable values in response to applied low and high input are determined (which indicates the DC current draw under both input conditions). By default, the DC current is calculated by averaging the stable low and high currents. This measure assumes a 50% duty cycle for the current. The duty cycle can be taken into account by using either the **−a** or **−ab** option. The **−a** option causes the DC value of the current to be averaged over the first complete input pulse (reference input's first low and high portions). Care must be taken to avoid averaging current spikes at the beginning part of the analysis due to incorrectly specified initial voltage values. The **−ab** option specifies the current to be averaged over the first high and second low portion of the input reference. The user should avoid terminating the analysis too soon. Both DC averaging methods reject current peaks by using the corresponding stable current values when the current is outside the corresponding limits. (The output text indicates this difference).

The three remaining measures for current examine the transient (AC) effects. The first two provide the average current for each transient (rising and falling). The frequency at which the DC and total current (AC and DC) differ by a percentage (specified by **−it** option) is reported. This figure estimates when DC power (and DC current) will no longer approximate the total power (and total current) for the circuit or circuit portion.

The last lines in the summary list the slowest node (ignoring non-converging nodes) by name, slowest transition direction and time. If I(VDD) appears in the table, the total circuit power dissipation will be printed as well as average power and critical frequency.

Signals that do not reach the user defined limits (such as a node not pulling up to 50% of Vdd before returning to ground) will be flagged with an error message.

If *namesfile* is specified (i.e, the **.names** file output by *sim2spice*(1)), the textual node names are substituted for the node numbers. The other arguments modify the analysis method or redefine signal parameters as defined in the OPTIONS section.

OPTIONS

Valid optional options and their default settings (*v* below is any real number, *pct* is any real percentage from 0.0 to 100.0, and *periods* is an integer. The sum of the percentages for each pair of low and high settings should not exceed 100.0):

Level specifying options:

−vdd *v*
: Set Vdd (power supply - the higher supply value) to *v* volts. Default is 5.0V.

−gnd *v*
: Set Gnd (the lower supply value) to *v* volts. Default is 0.0V.

−vsl *pct*
: Set the highest voltage considered Gnd (low logic threshold). This point is set as a percentage *pct* within Vdd and Gnd from Vdd. This value is used to determine the stable period for signals. Default value is 10%.

−vsh *pct*
: Set the lowest voltage considered Vdd (high logic threshold). This point is set as a percentage *pct* within Vdd and Gnd from Gnd. This value is used to determine the stable period for signals. Default value is 10%.

−vtl *pct*
: Set the highest voltage considered Gnd (low logic threshold). This point is set as a percentage *pct* within Vdd and Gnd from Vdd. This value is used to determine the transition time for signals (i.e. rise time and fall time). Default value is 10%. (Default measures a 10% to 90% rise time or 90% to

10% fall times.)

-vth *pct*
Set the lowest voltage considered Vdd (high logic threshold). This point is set as a percentage *pct* within Vdd and Gnd from Gnd. This value is used to determine the *t*ransition time for signals (i.e. rise time and fall time). Default value is 10%. (Default measures a 10% to 90% rise time or 90% to 10% fall times.)

-vpl *pct*
Set the highest voltage considered Gnd (low switching threshold) for determining the *p*ropagation delay with respect to the reference signal. It is set as a percentage *pct* within Vdd and Gnd (from Gnd). Default value is 50%. (Default measures from 50% point to 50% point. Rise and fall flags options have little effect for this setting. However, 10%/90% times can be found by setting this to 10%.)

-vph *pct*
Set the lowest voltage considered Vdd (high switching threshold) for determining the *p*ropagation delay with respect to the reference signal. It is set as a percentage *pct* within Vdd and Gnd (from Vdd). Default value is 50%. (Default measures from 50% point to 50% point. Rise and fall flags options have little effect for this setting. However, 10%/90% times can be found by setting this to 10%.)

-il *pct*
Set the range of current acceptable as DC for the *l*owest (absolute) stable current. The lowest stable current is lowest of the currents corresponding to the point where the reference input crosses the stable low logic threshold on a rising transient or its dual (stable high logic threshold on falling transient). If both currents are negative, the absolute values (lowest) are used. This point is set as a percentage *pct* of the lowest stable current. Default value is 10%.

-ih *pct*
Set the range of current acceptable as DC for the *h*ighest (absolute) stable current. The highest stable current is highest of the currents corresponding to the point where the reference input crosses the stable low logic threshold on a rising transient or its dual (stable high logic threshold on falling transient). If both currents are negative, the absolute values (highest) are used. This point is set as a percentage *pct* of the highest stable current. Default value is 10%.

-it *pct*
Used to determine the critical frequency. The critical frequency is the frequency at which total power (and total current) (AC and DC) is above or below DC power (and DC current) by *pct* percent. Default value is 5%.

Propagation delay flags:

Calculate the propagation delay from the rising input reference signal to all the changing voltage signals using one of the following options:

-ripo Include the input rise time, intersignal propagation delay time and output change time (rise time or fall time) in the propagation delay.

-rip Include the input rise time and intersignal propagation delay time in the propagation delay. Default.

-rpo Include the intersignal propagation delay time and output change time (rise time or fall time) in the propagation delay.

-rp Report the intersignal propagation delay time as the propagation delay.

Calculate the propagation delay for the falling input reference signal to all the changing voltage signals using one of the following options:

-fipo Include the input fall time, intersignal propagation delay time and output change time (rise time or fall time) in the propagation delay.

-fip Include the input fall time and intersignal propagation delay time in the propagation delay. Default.

-fpo Include the intersignal propagation delay time and output change time (rise time or fall time) in the propagation delay.

-fp Report the intersignal propagation delay time as the propagation delay.

Current flags:

-a Flag to average over the stable high and low portions of the current signal corresponding to the first input low and first input high portions of the input reference signal. One should make sure that the initial conditions on the voltages are provided to avoid unrealistic initial transients. (Rather use the **-ab** option.) This DC averaging method rejects current peaks by using the corresponding stable current values when the current is outside the corresponding limits. The duty cycle is inherently set by the input reference signal. If this option or the **-ab** option is not specified, the two discrete stable high and low values for the current will be averaged. Therefore, the default assumes a 50% duty cycle.

-ab Average over the stable low and high portions of the current signal corresponding to the first input high and second input low portions of the input reference signal. One should make sure that the analysis is inappropriately terminated. This DC averaging method rejects current peaks by using the corresponding stable current values when the current is outside the corresponding limits. The duty cycle is inherently set by the input reference signal. If this option or the **-ab** option is not specified, the two discrete stable high and low values for the current will be averaged.

Miscellaneous:

-s *periods*

Skip past *periods* of the input reference signal to start the analysis in a stablized region. Default is 0 which means take the first gnd, Vdd, gnd pulse. Negative values denote start the search backwards from the end of the spice input. Nonzero values are used to skip past the initial transients due to incorrect initial conditions and to skip possible incomplete periods.

-v Request verbose output. Print the critical points found in the reference signal which indicate which portion of the spice output is used for analysis. (etc.)

FORMAT

Suggested spice input format for meaningful analysis:

... (skipping some lines)

.WIDTH out=133

...

```
* pulse of: init. value=0v, pulsed value=5v, delay=10ns
* rise time=0ns, fall time=0ns, pulse width=120ns
vin 7 0 pulse (0 5 10ns 0ns 0ns 120ns)
.tran ...
.print tran V(7) V(8) ... I(VDD) (0,5)
...
```

where node 7 is the pulse applied input.

NOTES

Spice2summary automatically determines power levels and propagation times with respect to a reference signal. These figures give a designer quick indication of circuit performance by extracting critical information. (Also eliminates much of the tedium of examining volumes of numbers.) If a given node does not reach a particular level, it is reported by a descriptive message.

This program can also be used to verify the slowest nodes thus providing another check for *crystal*(1).

ENHANCEMENTS

Rev. 1 no longer depends on the carriage control characters from SPICE to signal the start and end of .PRINT TRAN output data. Instead a regular expression, specified by a **#define** statement at the top of the program, defines the character pattern that signals the line with the column names. Lines thereafter are ignored until numeric data is encountered. Data lines are read until a line without any numbers is encountered marking the end of the data.

SEE ALSO

crystal(1), sim2spice(1), spice(1)

AUTHOR

Fred W. Obermeier

NAME

vlsifont - create text logos for VLSI chips

SYNOPSIS

vlsifont [**-k** *key*] [**-f** *font*] *word* | **mquilt -s vlsifont**

DESCRIPTION

The *word* on the command line is rasterized into a matrix of characters suitable for input to mquilt or viewing on a text terminal. *word* may be surrounded by quotes to allow embedded spaces. The background characters in the rasterized image will be the same as the first character of *key*, while the foreground characters will be the same as the second character of *key*. *Key* defaults to "**em**".

If the output is piped to mquilt, the user should use the standard template **~cad/lib/mquilt/vlsifont.mag** by specifying the **-s vlsifont** switch, or else supply his own (see mquilt(1) for how to do this using Magic). The standard template recognizes these foreground and background characters:

e -- a small empty square
p -- a small poly square
d -- a small diffusion square
m -- a small metal square
E, P, D, or M -- larger versions of the above

FILES

~cad/lib/mquilt/q-vlsifont.mag -- standard template for quilt
/usr/lib/vfont/* -- standard place for fonts

SEE ALSO

mquilt(1), magic(1), vfont(5), vfontinfo(1)
The Berkeley Font Catalogue

AUTHOR

Robert N. Mayo

BUGS

If the font does not specify the width of a space character then the width of the letter 'e' is used instead.

NOTES

MOSIS will not fabricate chips that contain logos or text over 50 microns high, unless permission is obtained first. (As of January 1983.)

HISTORY

This program is a modified version of the tool 'vfontinfo' from Berkeley.

NAME

MDF – an nMOS frame for the integration of custom VLSI into Multibus-based systems

SYNOPSIS

magic -T nmos ~cad/lib/mdf/MDF
– starts magic with the entire Multibus Design Frame loaded.

magic -T nmos ~cad/lib/mdf/MDFCONNECTIONS
– starts magic with the cell containing the outline of the user circuit cavity.

DESCRIPTION

The **Multibus Design Frame** is an nMOS frame for the integration of custom VLSI into Multibus-based computer systems. The design frame provides a simplified interface to the backplane of the computer system. A circuit designed within the context of a design frame can be quickly integrated into a computer system upon fabrication. In many ways, a design frame is much like a hardware operating system. It can be used to rapidly prototype custom VLSI circuits and for the evaluation of the design in a real system context.

The **Multibus Design Frame** consists of elements at the chip and board levels. The nMOS circuitry within which the user circuit is placed and then sent to fabrication is provided in *magic*(5) format. The top-level cell is **MDF.mag**. It contains all the circuitry of the **Multibus Design Frame**. For the convenience of the designer a cell containing a 3 lambda wide outline of the user circuit cavity is also available (**MDFCONNECTIONS.mag**). All the connections points are labeled with the name of the signal. This cell is called by **MDF.mag**. If a circuit is designed within **MDFCONNECTIONS.mag**, CIF for the entire design can be generated by simply loading **MDF.mag**.

Designers may also find the need to have inputs and outputs other than those to the frame interface. Pads are available in **PADINUSER.mag** and **PADOUTUSER.mag**. The pads used by the design frame can also be used by the user, however, these pads invert their inputs and outputs.

When the fabricated chips are returned they can be placed on the **Multibus Design Frame** board and placed in a Multibus card-cage. The file MDFPCB.cif is the CIF used to fabricate the **Multibus Design Frame** printed circuit board through the PCBIS service of MOSIS. This file is provided as an example of what a printed circuit board description looks like, users are not expected to fabricate their own boards.

FILES

~cad/lib/mdf/MDF.mag
– The top level cell of the Multibus Design Frame

~cad/lib/mdf/MDFCONNECTIONS.mag
– Cell containing the outline of the user circuit cavity and labels on all the connection points.

~cad/lib/mdf/*.mag
– Magic files for all the cells in the Multibus Design Frame

~cad/lib/mdf/MDFPCB.cif
– CIF description of the Multibus Design Frame printed circuit board.

NOTES

Complete documentation, users' guide, and a detailed specification of the **Multibus Design Frame** chip and board level frames is available by writing:

Gaetano Borriello
Computer Science Division
573 Evans Hall
University of California at Berkeley
Berkeley, California 94720

gaetano%ucbkim@Berkeley.ARPA

SEE ALSO

magic(1), magic(5)

AUTHOR

Gaetano Borriello

NAME

mpack – routines for generating semi-regular modules

DESCRIPTION

Mpack is a library of 'C' routines that aid the process of generating semi-regular modules. Decoder planes, barrel shifters, and PLAs are common examples of semi-regular modules.

Using Magic, an mpack user will draw an example of a finished module and then break it into tiles. These tiles represent the building blocks for more complicated instances of the module. The mpack library provides routines to aid in assembling tiles into a finished module.

MAKING AN EXAMPLE MODULE

The first step in using mpack is to create an example instance of the module, called a *template*. The basic building blocks of the structure, or *tiles*, are then chosen. Each tile should be given a name by means of a rectangular label which defines its contents. If the tiles in the module do not abut (e.g. they overlap) it is useful to define another tile whose size indicates how far apart the tiles should be placed.

Templates should be in Magic format and, by convention, end with a **.mag** suffix. With some programs, it is possible to generate the same structure in a different technology or style by changing just the template. If this is the case, each template should have a filename of the
form basename-*style*.**mag**. The *style* part of the filename interacts with the **-s** option (see later part of this manual).

WRITING AN MPACK PROGRAM

An mpack program is the 'C' code which assembles tiles into the desired module. Typically this program reads a file (such as a truth table) and then calls the tile placement routines in the mpack library.

The mpack program must first include the file **~cad/lib/mpack.h** which defines the interface to the mpack system. Next the **TPinitialize** procedure is called. This procedure processes command line arguments, opens an input file as the standard input (**stdin**), and loads in a template.

The program should now read from the standard input and compute where to place the next tile. Tiles may be aligned with previously placed tiles or placed at absolute coordinates. If a tile is to overlap an existing tile the program must space over the distance of the overlap before placing the tile.

When all tiles are placed the program should call the routine **TPwrite_tile** to create the output file that was specified on the command line.

To use the mpack library be sure to include it with your compile or load command (e.g. **cc** *your_file* **~cad/lib/mpack.lib**).

ROUTINES

Initialization and Output Routines

TPinitialize(*argc, argv, base_name*)

The mpack system is initialized, command line arguments are processed, and a template is loaded. The file descriptor **stdin** is attached to the input file specified on the command line. The template's filename is formed by taking the *base_name*, adding any extension indicated by the **-s** option, and then adding the **.mag** suffix. The **-t** option allows the user to override *base_name* from the command line.

Argc and *argv* should contain the command line arguments. *Argc* is a count of the number of arguments, while *argv* is an array of pointers to strings. Strings of length zero are ignored (as is the flag consisting of a single space), in order to make it easy for the calling program to intercept its own arguments. *Argc* and *argv* are of the same structure as the two parameters passed to the main program. A later section of this manual summarizes the command line options.

TPload_tiles(*file_name*)
The given *file_name* is read, and each rectangular label found in the file becomes a tile accessible via TPname_to_tile. No extensions are added to *file_name*.

TILE TPread_tile(*file_name*)
A tile is created and *file_name* is read into it. The tile is returned as the value of the function.

TPwrite_tile(*tile, filename*)
The tile *tile* is written to the file specified by *filename*, with **.ca** or **.cif** extensions added. See the description of the **-o** option for information on what file name is chosen if *filename* is the null string. The choice between Magic or CIF format is chosen with the **-a** or **-c** command line options.

Tile creation, deletion, and access

TPdelete_tile(*tile*)
The tile *tile* is deleted from the database and the space occupied by it is reused.

TILE TPcreate_tile(*name*)
A new, empty tile is created and given the name *name*. This name is used by the routine **TPname_to_tile** and in error messages. The type **TILE** returned is a unique ID for the tile, not the tile itself. Currently this is implemented by defining the type TILE to be a pointer to the internal database representation of the tile.

int TPtile_exists(*name*)
TRUE (1) is returned if a tile with the given *name* exists (such as in the template or from a call to TPcreate_tile).

TILE TPname_to_tile(*name*)
A value of type **TILE** is returned. This value is a unique ID for the tile that has the name *name*. This name comes from a call to TPcreate_tile(), or from the rectangular label that defined it in a template that was read in by TPread_tiles() or TPinitialize(). If the tile does not exist then a value of NULL is returned and an error message is printed.

RECTANGLE TPsize_of_tile(*tile*)
A rectangle is returned that is the same size as the tile *tile*. The rectangle's lower left corner is located at the coordinate (0, 0). All coordinates in mpack are specified in half-lambda.

Painting and Placement Routines

RECTANGLE TPpaint_tile(*from_tile, to_tile, ll_corner*)
The tile *from_tile* is painted into the tile *to_tile* such that its lower left corner is placed at the point *ll_corner* in the tile *to_tile* . The location of the newly painted area in the output tile is returned as a value of type RECTANGLE. The tile *to_tile* is often an empty tile made by **TPcreate_tile**(). The point *ll_corner* is almost never provided directly, it is usually generated by routines such as **align**().

TPdisp_tile(*from_tile, ll_corner*)
A rectangle the size of *from_tile* with the lower left corner located at *ll_corner* is returned. Note that this routine behaves exactly like the routine TPpaint_tile except that no output tile is modified. This routine, in conjunction with the **align** routine, is useful for controlling the overlap of tiles.

RECTANGLE TPpaint_cell(*from_tile, to_tile, ll_corner*)
This routine behaves like **TPpaint_tile**() except that the *from_tile* is placed as a subcell rather than painted into place. The tile *from_tile* must exist in the file system (i.e. it must have been read in from disk or have been written out to disk).

Label Manipulation Routines

TPplace_label(*tile, rect, label_name*)
A label named *label_name* is place in the tile *tile*. The size and location of the label is the given by the RECTANGLE *rect*.

int TPfind_label(*tile, &rect1, str, &rect2*)
The tile *tile* is searched for a label of name *str*. The location of the first such label found is returned in the rectangle *rect2*. The function returns 1 if such a label was found, and 0 otherwise. The rectangle pointer *&rect1*, if non-NULL, restricts the search to an area of the tile.

TPstrip_labels(*tile, ch*)
All labels in the tile *tile* that begin with the character *ch* are deleted.

TPremove_labels(*tile, name, r*)
All labels in the tile *tile* that are completely within the area *r* are deleted. If *name* is non-NULL, then only labels with that name will be affected.

TPstretch_tile(*tile, str, num*)
The string *str* is the name of one or more labels within the tile *tile*. Each of these labels must be of zero width or zero height, i.e. they must be lines. Each of these lines define a line across which the tile will be stretched. The amount of the stretch is specified by *num* in units of **half**-lambda. Stretching such a line turns it into a rectangle. Note that if the tile contains 2 lines that are co-linear, the stretching of one of them will turn both into rectangles.

Point-Valued Routines

POINT tLL(*tile*)
POINT tLR(*tile*)
POINT tUL(*tile*)
POINT tUR(*tile*)

The location of the specified corner of tile *tile*, relative to the tile's lower left corner, is returned as a point. LL stands for lower-left, LR for lower-right, UL for upper-left, and UR for upper-right. Note that **tLL()** returns (0, 0).

POINT rLL(*rect*)
POINT rLR(*rect*)
POINT rUL(*rect*)
POINT rUR(*rect*)

The location of the specified corner of the rectangle *rect* is returned as a point. LL stands for lower-left, LR for lower-right, UL for upper-left, and UR for upper-right.

POINT align(*p1*, *p2*)

A point is computed such that when added to the point *p2* gives the point *p1*. *p1* is normally a corner of a rectangle within a tile and *p2* is normally a corner of a tile. In this case the point computed can be treated as the location for the placement of the tile.

For example, TPpaint_tile(outtile, fromtile, align(rUL(rect), tLL(fromtile))) will paint the tile *fromtile* into *outtile* such that the lower left corner of *fromtile* is aligned with the upper-left corner of *rect*. In this example *rect* would probably be something returned from a previous TPpaint_tile() call.

Point and Rectangle Addition Routines

POINT TPadd_pp(*p1*, *p2*)
POINT TPsub_pp(*p1*, *p2*)

The points *p1* and *p2* are added or subtracted, and the result is returned as a point. In the subtract case *p2* is subtracted from *p1*.

RECTANGLE TPadd_rp(*r1*, *p1*)
RECTANGLE TPsub_rp(*r1*, *p1*)

The rectangle *r1* has the point *p1* added or subtracted from it. This has the effect of displacing the rectangle in the X and/or Y dimensions.

Miscellaneous Functions

int TPget_lambda()

This function returns the current value of lambda in centi-microns.

INTERFACE DATA STRUCTURES

In those cases where tiles must be placed using absolute, (half-lambda) coordinates, it is useful to know that **RECTANGLE**s and **POINT**s are defined as:

```
typedef struct {
  int x_left, x_right, y_top, y_bot;
```

```
} RECTANGLE;

typedef struct {
   int x, y;
} POINT;
```

The variable **origin_point** is predefined to be (0, 0). **origin_rect** is defined to be a zero-sized rectangle located at the origin.

OPTIONS ACCEPTED BY TPinitialize()

Typical command line: *program_name* [-t *template*] [-s *style*] [-o *output_file*] *input_file*

-a produce Magic format (this is the default)

-c produce CIF format

-v be verbose (sequentially label the tiles in the output for debugging purposes; also print out information about the number of rectangles processed by mpack)

-s *style* generate output using the template for this style (see TPinitialize for details)

-o The next argument is taken to be the base name of the output file. The default is the input file name with any extensions removed. If there is not input file specified and no -o option specified, the output will go to **stdout**.

-p (pipe mode) Send the output to **stdout**.

-t The next argument specifies the template base name to use. This overrides the default supplied by the program. A **.mag** extension is automatically added. (see TPinitialize)

-l *name*

Set the cif output style to *name*. *name* is the name of a cif output style as defined in Magic's technology file. If this option is not specified then the first output style in the technology file is used. (Note: In the old tpack system this option set the size of lambda.)

input_file

The name of the file that the program should read from (such as a truth table file). If this filename is omitted then the input is taken from the standard input (such as a pipe).

-M *num*

This option is accepted by mpack, but ignored. It is a leftover from the tpack system.

-D *num1 num2*

The *Demo* or *Debug* option. This option will cause **mpack** to place only the first *num1* tiles, and the last *num2* of those will be outlined with rectangular labels. In addition, if a tile called "blotch" is defined then a copy of it will be placed in the output tile upon each call to the *align* function during the placing of the last *num2* tiles. The blotch tile will be centered on the first point passed to *align*, and usually consists of a small blotch of brightly colored paint. This has the effect of marking the alignment points of tiles. The last tile painted into is assumed to be the output tile.

EXAMPLE

It is highly recommended that the example in **~cad/src/mquilt** be examined. Look at both the template and the 'C' code. A more complex example is in **~cad/src/mpla**.

FILES

~cad/lib/mpack.h	(definition of the mpack interface)
~cad/lib/mpack.lib	(linkable mpack library)
~cad/lib/mpack.ln	(lint-library for lint)
~cad/src/mquilt/*	(an example of an mpack program)
~cad/lib/magic/sys/*.tech*	(technology description files)

ALSO SEE

magic(CAD), mquilt(CAD), mpla(CAD)
Robert N. Mayo *Pictures with Parentheses: Combining Graphics and Procedures in a VLSI Layout Tool,* ®Proceedings of the 20th Design Automation Conference, June, 1983.
'C' Manual

HISTORY

This is a port of the tpack(1) system which generated Caesar files.

AUTHOR

Robert N. Mayo

BUGS

When a tile contains part of a subcell, or touches a subcell, then the whole subcell is considered to be part of the tile. The same goes for arrays of subcells.

NAME

.cadrc – Initialization file

DESCRIPTION

The .cadrc file is an ASCII text file which is used to initialize several CAD programs. Each user may place a .cadrc file in his home directory. Several CAD programs read this file as part of their initialization routine to set up various default settings. In addtion to the .cadrc file in the user's home directory there is a .cadrc file in ~cad. This file is read before the one in the user's directory and is used to tell the program where it can find various files and library programs. This allows program binaries to be transported between systems without recompiling.

The .cadrc file contains several lines, each line is a seperate command. The first word on the line is called the *keyword*. The keyword tells the program how to interpret the line. When a program reads a keyword it doesn't understand it ignores the line. The case of the keyword is ignored. This allows several program to share .cadrc files. What follows is a list of .cadrc command lines.

AreaToCap *layer value*

This command is read by the cifplot circuit extractor and mextra. It is used to set up the default capacitance per unit area. *layer* can be 'metal', 'poly', 'diff', or 'poly/diff'. *value* is in atto-farads (10**-18 farads) per square micron. Also see the command 'perimetertocap'.

CapThreshold *value*

This command is read by mextra. Mextra will not report any node capacitance below *value*. *value* is in femto-farads.

Cifplot *options*

This line allows you to select default command line options for cifplot. Call this command just as you would call cifplot from the shell but without any CIF file.

Device *DevCh xmax ymax resolution DumpProg*

This command sets up information about a particular plotting device. This command is used by cifplot. *DevCh* is a single character which indicates which output device. The characters 'U', 'V', and 'W' are black and white raster scan type devices. Lower case letters are for output in trapezoid format and is generally used for driving random access displays. The letter 'P' is for pen plotters. *DumpProg* is the program to actually display the plot on the device. For raster scan output the program is called with the the name of the dump file. For other type of devices the program is called so that information is piped into standard input. *xmax* and *ymax* indicated the range in device co-ordinates in the x and y direction. *resolution* is the resolution of the device in dots per inch.

FontDir *dirname*

dirname is the name of the directory in which to find font files. This keyword is recognized by cifplot.

MachineName *name*

name is the net address of the machine. (E.g. on Ernie the the command would be "machinename csvax".)

MaxLength *length*

length specifies the maximum length in feet that can be plotted. This command is recognized by cifplot.

PatFile *file*

file is a file of stipple patterns to be used as the default stipple. This command is recognized by cifplot.

PerimeterToCap *layer value*
This command is read by the cifplot circuit extractor and mextra. It is used to set up the default capacitance per unit length. *layer* can be 'metal', 'poly', 'diff', or 'poly/diff'. *value* is in atto-farads (10**-18 farads) per micron. Also see the command 'areatocap'.

Sim2Spl *TransType parameter value*
This command is read by sim2spl. It is used to set default parameters to give to splice. *TransType* is a '.sim' transistor type, either 'e' or 'd'. *parameter* is one of the splice parameters: 'vt', 'kp', 'gam', 'phi', or 'lam'. *value* is the value given to that parameter.

TmpDir *dirname*
dirname is the name of a directory with a lot of free space. This directory is used to set up dump files by cifplot.

SEE ALSO

cifplot(CAD)
mextra(CAD)
sim2spl(CAD)

BUGS

Not yet completely implemented or documented.

NAME

cmap – format of .cmap files (color maps)

DESCRIPTION

Color-map files define the mapping between eight-bit color numbers and red, green and blue intensities used for those numbers. They are read by Magic as part of system startup, and also by the **:load** and **:save** commands in color-map windows. Color-map file names usually have the form *x.y.z*.**cmap***n*, where *x* is a class of technology files, *y* is a class of displays, *z* is a class of monitors, and *n* is a version number (currently 1). The version number will change in the future if the formap of color-map files ever changes. Normally, *x* and *y* correspond to the corresponding parts of a display styles file. For example, the color map file **mos.7bit.std.cmap1** is used today for most nMOS and CMOS technology files using displays that support at least seven bits of color per pixel and standard-phosphor monitors. It corresponds to the display styles file **mos.7bit.dstyle5**.

Color-map files are stored in ASCII form, with each line containing four decimal integers separated by white space. The first three integers are red, green, and blue intensities, and the fourth field is a color number. For current displays the intensities must be integers between 0 and 255. The color numbers must increase from line to line, and the last line must have a color number of 255. The red, green, and blue intensities on the first line are used for all colors from 0 up to and including the color number on that line. For other lines, the intensities on that line are used for all colors starting one color above the color number on the previous line and continuing up and through the color number on the current line. For example, consider the color map below:

```
255     0     0     2
0       0     255   3
255     255   255   256
```

This color map indicates that colors 0, 1, and 2 are to be red, color 3 is to be blue, and all other colors are to be white.

SEE ALSO

magic(1), dstyle(5)

NAME

displays – Display Configuration File

DESCRIPTION

The interactive graphics programs Caesar, Magic, and Gremlin use two separate terminals: a text terminal from which commands are issued, and a color graphics terminal on which graphical output is displayed. These programs use a **displays** file to associate their text terminal with its corresponding display device.

The **displays** file is an ASCII text file with one line for each text terminal/graphics terminal pair. Each line contains 4 items separated by spaces: the name of the port attached to a text terminal, the name of the port attached to the associated graphics terminal, the phosphor type of the graphics terminal's monitor, and the type of graphics terminal.

An applications program may use the phosphor type to select a color map tuned to the monitor's characteristics. Only the **std** phosphor type is supported at UC Berkeley.

The graphics terminal type specifies the device driver a program should use when communicating with its graphics terminal. Magic supports types **UCB512**, **AED1024**, and **SUN120**. Other programs may recognize different display types. See the manual entry for your specific application for more information.

A sample displays file is:

```
/dev/ttyi1 /dev/ttyi0 std UCB512
/dev/ttyj0 /dev/ttyj1 std UCB512
/dev/ttyjf /dev/ttyhf std UCB512
/dev/ttyhb /dev/ttyhc std UCB512
/dev/ttyhc /dev/ttyhb std UCB512 *.in -0.5i
```

In this example, **/dev/ttyi1** connects to a text terminal. An associated **UCB512** graphics terminal with standard phosphor is connected to **/dev/ttyi0**.

FILES

Magic uses the displays file ~cad/lib/displays. Gremlin looks in /usr/local/displays.

SEE ALSO

magic(1)

NAME

dstyle – format of .dstyle files (display styles)

DESCRIPTION

Display styles indicate how to render information on a screen. Each style describes one way of rendering information, for example as a solid area in red or as a dotted outline in purple. Different styles correspond to mask layers, highlights, labels, menus, window borders, and so on. See "Magic Maintainer's Manual #3: Display Styles, Color Maps, and Glyphs" for more information on how the styles are used.

Dstyle files usually have names of the form *x.y.***dstyle***n*, where *x* is a class of technologies, *y* is a class of displays, and *n* is a version number (currently 5). The version number may increase in the future if the format of dstyle files changes. For example, the display style file **mos.7bit.dstyle5** provides all the rendering information for our nMOS and CMOS technologies for color displays with at least 7 bits of color.

Dstyle files are stored in ASCII as a series of lines. Lines beginning with "#" are considered to be comments and are ignored. The rest of the lines of the file are divided up into two sections separated by blank lines. There should not be any blank lines within a section.

DISPLAY_STYLES SECTION

The first section begins with a line **display_styles** *planes* where *planes* is the number of bits of color information per pixel on the screen (between 1 and 8). Each line after that describes one display style and contains eight fields separated by white space: *style writeMask color outline fill stipple shortName longName* The meanings of the fields are:

style The number of this style, in decimal. Styles 1 through 64 are used to display mask layers in the edit cell. The style number(s) to use for each mask layer is (are) specified in the technology file. Styles 65-128 are used for displaying mask layers in non-edit cells. If style *x* is used for a mask layer in the edit cell, style *x*+64 is used for the same mask layer in non-edit cells. Styles above 128 are used by the Magic code for various things like menus and highlights. See the file *styles.h* in Magic for how styles above 128 are used. When redisplaying, the styles are drawn in order starting at 1, so the order of styles may affect what appears on the screen.

writeMask
This is an octal number specifying which bit-planes are to be modified when this style is rendered. For example, 1 means only information in bit-plane 0 will be affected, and 377 means all eight bit-planes are affected.

color An octal number specifying the new values to be written into the bit-planes that are modified. This is used along with *writeMask* to determine the new value of each pixel that's being modified: newPixel = (oldPixel & ~writeMask) | (color & writeMask) The red, green, and blue intensities displayed for each pixel are not deterimined directly by the value of the pixel; they come from a color map that maps the eight-bit pixel values into red, green, and blue intensities. Color maps are stored in separate files.

outline If this field is zero, then no outline is drawn. If the field is non-zero, it specifies that outlines are to be drawn around the rectangular areas rendered in this style, and the octal value gives an eight-bit pattern telling how to draw the outline. For example, 377 means to draw a solid line, 252 means to draw a dotted line, 360 specifies long dashes, etc. This field only indicates *which* pixels will be modified: the *writeMask* and *outline* fields indicate how the pixels are modified.

fill This is a text string specifying how the areas drawn in this style should be filled. It

must have one of the values **solid**, **stipple**, **cross**, **outline**, **grid**. **Solid** means that every pixel in the area is to modified according to *writeMask* and *color*. **Stipple** means that the area should be stippled: the stipple pattern given by *stipple* is used to determine which pixels in the area are to be modified. **Cross** means that an X is drawn in a solid line between the diagonally-opposite corners of the area being rendered. **Outline** means that the area should not be filled at all; only an outline is drawn (if specified by *outline*). **Grid** is a special style used to draw a grid in the line style given by *outline*. The styles **cross** and **stipple** may be supplemented with an outline by giving a non-zero *outline* field. The **outline** and **grid** styles don't make sense without an an outline, and **solid** doesn't make sense with an outline (since all the pixels are modified anyway).

stipple Used when *fill* is **stipple** to specify (in decimal) the stipple number to use.

shortName

This is a one-character name for this style. These names are used in the specification of glyphs and also in a few places in the Magic source code. Most styles have no short name; use a "-" in this field for them.

longName

A more human-readable name for the style. It's not used at all by Magic.

STIPPLES SECTION

The second section of a dstyle file is separated from the first by a blank line. The first line of the second section must be **stipples** and each additional line specifies one stipple pattern with the syntax *number pattern name* *Number* is a decimal number used to name the stipple in the *stipple* fields of style lines. *Number* must be no less than 1 and must be no greater than a device-dependent upper limit. Most devices support at least 15 stipple patterns. *Pattern* consists of eight octal numbers, each from 0-377 and separated by white space. The numbers form an 8-by-8 array of bits indicating which pixels are to be modified when the stipple is used. The *name* field is just a human-readable description of the stipple; it isn't used by Magic.

FILES

~cad/lib/magic/sys/mos.7bit.dstyle5

SEE ALSO

magic(1), cmap(5), glyphs(5)

NAME

espresso – input file format for espresso(1)

DESCRIPTION

Espresso accepts as input a two-level description of a Boolean switching function. This is described as a character matrix with keywords imbedded in the input to specify the size of the matrix and the logical format of the input function. Comments are allowed within the input by placing a pound sign (#) as the first character on a line. Comments and unrecognized keywords are passed directly from the input file to standard output. Any white-space (blanks, tabs, etc.), except when used as a delimiter in an imbedded command, is ignored. It is generally assumed that the PLA is specified such that each row of the PLA fits on a single line in the input file.

KEYWORDS

The following keywords are recognized by *espresso*. The list shows the probable order of the keywords in a PLA description. [d] denotes a decimal number and [s] denotes a text string.

.i [d] Specifies the number of input variables.

.o [d] Specifies the number of output functions.

.type [s] Sets the logical interpretation of the character matrix as described below under "Logical Description of a PLA". This keyword must come before any product terms. [s] is one of f, r, fd, fr, dr, or fdr.

.phase [s] [s] is a string of as many 0's or 1's as there are output functions. It specifies which polarity of each output function should be used for the minimization (a 1 specifies that the ON-set of the corresponding output function should be used, and a 0 specifies that the OFF-set of the corresponding output function should be minimized).

.pair [d] Specifies the number of pairs of variables which will be paired together using two-bit decoders. The rest of the line contains pairs of numbers which specify the binary variables of the PLA which will be paired together. The binary variables are numbered starting with 1. The PLA will be reshaped so that any unpaired binary variables occupy the leftmost part of the array, then the paired multiple-valued columns, and finally any multiple-valued variables.

.kiss Sets up for a *kiss*-style minimization.

.p [d] Specifies the number of product terms. The product terms (one per line) follow immediately after this keyword. Actually, this line is ignored, and the ".e", ".end", or the end of the file indicate the end of the input description.

.e (.end) Marks the end of the PLA description.

LOGICAL DESCRIPTION OF A PLA

When we speak of the ON-set of a Boolean function, we mean those minterms which imply the function value is a 1. Likewise, the OFF-set are those terms which imply the function is a 0, and the DC-set (don't care set) are those terms for which the function is unspecified. A function is completely described by providing its ON-set, OFF-set and DC-set. Note that all minterms lie in the union of the ON-set, OFF-set and DC-set, and that the ON-set, OFF-set and DC-set share no minterms.

The purpose of the *espresso* minimization program is to find a logically equivalent set of product-terms to represent the ON-set and optionally minterms which lie in the DC-set, without containing any minterms of the OFF-set.

A Boolean function can be described in one of the following ways:

1) By providing the ON-set. In this case, *espresso* computes the OFF-set as the complement of the ON-set and the DC-set is empty. This is indicated with the keyword **.type f** in the input file, or **-f** on the command line.

2) By providing the ON-set and DC-set. In this case, *espresso* computes the OFF-set as the complement of the union of the ON-set and the DC-set. If any minterm belongs to both the ON-set and DC-set, then it is considered a don't care and may be removed from the ON-set during the minimization process. This is indicated with the keyword **.type fd** in the input file, or **-fd** on the command line.

3) By providing the ON-set and OFF-set. In this case, *espresso* computes the DC-set as the complement of the union of the ON-set and the OFF-set. It is an error for any minterm to belong to both the ON-set and OFF-set. This error may not be detected during the minimization, but it can be checked with the subprogram "-do check" which will check the consistency of a function. This is indicated with the keyword **on the command line.**

4) By providing the ON-set, OFF-set and DC-set. This is indicated with the keyword **.type fdr** in the input file, or **-fdr** on the command line.

If at all possible, *espresso* should be given the DC-set (either implicitly or explicitly) in order to improve the results of the minimization.

A term is represented by a "cube" which can be considered either a compact representation of an algebraic product term which implies the function value is a 1, or as a representation of a row in a PLA which implements the term. A cube has an input part which corresponds to the input plane of a PLA, and an output part which corresponds to the output plane of a PLA (for the multiple-valued case, see below).

SYMBOLS IN THE PLA MATRIX AND THEIR INTERPRETATION

Each position in the input plane corresponds to an input variable where a 0 implies the corresponding input literal appears complemented in the product term, a 1 implies the input literal appears uncomplemented in the product term, and - implies the input literal does not appear in the product term.

With logical type *f*, for each output, a 1 means this product term belongs to the ON-set, and a 0 or - means this product term has no meaning for the value of this function. This logical type corresponds to an actual PLA where only the ON-set is actually implemented.

With logical type *fd* (the default), for each output, a 1 means this product term belongs to the ON-set, a 0 means this product term has no meaning for the value of this function, and a - implies this product term belongs to the DC-set.

With logical type *fr*, for each output, a 1 means this product term belongs to the ON-set, a 0 means this product term belongs to the OFF-set, and a - means this product term has no meaning for the value of this function.

With logical type *fdr*, for each output, a 1 means this product term belongs to the ON-set, a 0 means this product term belongs to the OFF-set, a - means this product term belongs to the DC-set, and a ~ implies this product term has no meaning for the value of this function.

Note that regardless of the logical type of PLA, a ~ implies the product term has no meaning for the value of this function. 2 is allowed as a synonym for -, 4 is allowed for 1, and 3 is allowed for ~. Also, the logical PLA type can also be specified on the command line.

MULTIPLE-VALUED FUNCTIONS

Espresso will also minimize multiple-valued Boolean functions. There can be an arbitrary number of multiple-valued variables, and each can be of a different size. If there are also binary-valued variables, they should be given as the first variables on the line (for ease of description). Of course, it is always possible to place them anywhere on the line as a two-valued multiple-valued variable. The function size is described by the imbedded option

.mv [num_var] [num_binary_var] [s1] . . . [sn]

Specifies the number of variables (num_var), the number of binary variables (num_binary_var), and the size of each of the multiple-valued variables (s1 through sn).

A multiple-output binary function with *ni* inputs and *no* outputs would be specified as ".mv *ni+1 ni no*." ".mv" cannot be used with either ".i" or ".o" – use one or the other to specify the function size.

The binary variables are given as described above. Each of the multiple-valued variables are given as a bit-vector of 0 and 1 which have their usual meaning for multiple-valued functions. The last multiple-valued variable (also called the output) is interpreted as described above for the output (to split the function into an ON-set, OFF-set and DC-set). A vertical bar | may be used to separate the multiple-valued fields in the input file.

If the size of the multiple-valued field is less than zero, than a symbolic field is interpreted from the input file. The absolute value of the size specifies the maximum number of unique symbolic labels which are expected in this column. The symbolic labels are white-space delimited strings of characters.

To perform a *kiss*-style encoding problem, either the keyword .kiss must be in the file, or the **-kiss** option must be used on the command line. Further, the third to last variable on the input file must be the symbolic "present state", and the second to last variable must be the "next state". As always, the last variable is the output. The symbolic "next state" will be hacked to be actually part of the output.

EXAMPLE #1

A two-bit adder which takes in two 2-bit operands and produces a 3-bit result can be described completely in minterms as:

```
# 2-bit by 2-bit binary adder (with no carry input)
.i 4
.o 3
.type fr
.pair 2 (1 3) (2 4)
.phase 011
00 00      000
00 01      001
00 10      010
00 11      011
01 00      001
01 01      010
01 10      011
01 11      100
10 00      010
10 01      011
10 10      100
10 11      101
11 00      011
11 01      100
11 10      101
11 11      110
.end
```

The logical format for this input file (i.e., type fr) is given to indicate that the file contains both the ON-set and the OFF-set. Note that in this case, the zeros in the output plane are really specifying "value must be zero" rather than "no information".

The imbedded option *.pair* indicates that the first binary-valued variable should be paired with the third binary-valued variable, and that the second variable should be paired with the fourth variable. The function will then be mapped into an equivalent multiple-valued minimization problem.

The imbedded option *.phase* indicates that the positive-phase should be used for the second and third outputs, and that the negative phase should be used for the first output.

EXAMPLE #2

This example shows a description of a multiple-valued function with 5 binary variables and 3 multiple-valued variables (8 variables total) where the multiple-valued variables have sizes of 4 27 and 10 (note that the last multiple-valued variable is the "output" and also encodes the ON-set, DC-set and OFF-set information).

```
.mv 8 5 4 27 10
0-010|1000|100000000000000000000000000|0010000000
10-10|1000|010000000000000000000000000|1000000000
0-111|1000|001000000000000000000000000|0001000000
0-10-|1000|000100000000000000000000000|0001000000
00000|1000|000010000000000000000000000|1000000000
00010|1000|000001000000000000000000000|0010000000
01001|1000|000000100000000000000000000|0000000010
0101-|1000|000000010000000000000000000|0000000000
0-0-0|1000|000000001000000000000000000|1000000000
10000|1000|000000000100000000000000000|0000000000
11100|1000|000000000010000000000000000|0010000000
10-10|1000|000000000001000000000000000|0000000000
11111|1000|000000000000100000000000000|0010000000
                    .
                    .
                    .
11111|0001|000000000000000000000000001|0000000000
```

EXAMPLE #3

This example shows a description of a multiple-valued function setup for *kiss*-style minimization. There are 5 binary variables, 2 symbolic variables (the present-state and the next-state of the FSM) and the output (8 variables total).

```
.mv 8 5 -10 -10 6
.type fr
.kiss
# This is a translation of IOFSM from OPUS
# inputs are      IO1 IO0 INIT SWR MACK
# outputs are    WAIT MINIT MRD SACK MWR DLI
# reset logic
--1--      -          init0      110000
# wait for INIT to go away
--1--      init0      init0      110000
--0--      init0      init1      110000
# wait for SWR
--00-      init1      init1      110000
--01-      init1      init2      110001
# Latch address
--0--      init2      init4      110100
# wait for SWR to go away
--01-      init4      init4      110100
--00-      init4      iowait     000000
# wait for command from MFSM
0000-      iowait     iowait     000000
1000-      iowait     init1      110000
01000      iowait     read0      101000
11000      iowait     write0     100010
01001      iowait     rmack      100000
11001      iowait     wmack      100000
--01-      iowait     init2      110001
# wait for MACK to fall (read operation)
--0-0      rmack      rmack      100000
--0-1      rmack      read0      101000
# wait for MACK to fall (write operation)
--0-0      wmack      wmack      100000
--0-1      wmack      write0     100010
# perform read operation
--0--      read0      read1      101001
--0--      read1      iowait     000000
# perform write operation
--0--      write0     iowait     000000
.end
```

NAME

ext – format of .ext files produced by Magic's hierarchical extractor

DESCRIPTION

Magic's extractor produces a **.ext** file for each cell in a hierarchical design. The **.ext** file for cell *name* is *name*.**ext**. This file contains three kinds of information: environmental information (scaling, timestamps, etc), the extracted circuit corresponding to the mask geometry of cell *name*, and the connections between this mask geometry and the subcells of *name*.

A **.ext** file consists of a series of lines, each of which begins with a keyword. The keyword beginning a line determines how the remainder of the line is interpreted. The following set of keywords define the environmental information:

tech *techname*
Identifies the technology of cell *name* as *techname*, e.g, **nmos**, **cmos**.

timestamp *time*
Identifies the time when cell *name* was last modified. The value *time* is the time stored by Unix, i.e, seconds since 00:00 GMT January 1, 1970. Note that this is *not* the time cell was extracted, but rather the timestamp value stored in the **.mag** file. The incremental extractor compares the timestamp in each **.ext** file with the timestamp in each **.mag** file in a design; if they differ, that cell is re-extracted.

version *version*
Identifies the version of **.ext** format used to write *name*.**ext**. The current version is **4.0**.

scale *rscale cscale lscale*
Sets the scale to be used in interpreting resistance, capacitance, and linear dimension values in the remainder of the **.ext** file. Each resistance value must be multiplied by *rscale* to give the real resistance in milliohms. Each capacitance value must be multiplied by *cscale* to give the real capacitance in attofarads. Each linear dimension (e.g, width, height, transform coordinates) must be multiplied by *lscale* to give the real linear dimension in centimicrons. Also, each area dimension (e.g, transistor channel area) must be multiplied by *scale*scale* to give the real area in square centimicrons. At most one **scale** line may appear in a **.ext** file. If none appears, all of *rscale*, *cscale*, and *lscale* default to 1.

resistclasses *r1 r2 ...*
Sets the resistance per square for the various resistance classes appearing in the technology file. The values *r1*, *r2*, etc. are in milliohms; they are not scaled by the value of *rscale* specified in the **scale** line above. Each node in a **.ext** file has a perimeter and area for each resistance class; the values *r1*, *r2*, etc. are used to convert these perimeters and areas into actual node resistances. See "Magic Tutorial #8: Circuit Extraction" for a description of how resistances are computed from perimeters and areas by the program **ext2sim**.

The following keywords define the circuit formed by the mask information in cell *name*. This circuit is extracted independently of any subcells; its connections to subcells are handled by the keywords in the section after this one.

node *name R C x y a1 p1 a2 p2 ... aN pN [attrs]*
Defines an electrical node in *name*. This node is referred to by the name *name* in subsequent **equiv** lines, connections to the terminals of transistors in **fet** lines, and hierarchical connections or adjustments using **merge** or **adjust**. The node has a total capacitance to ground of *C* attofarads, and a lumped resistance of *R* milliohms.

For purposes of going back from the node name to the geometry defining the node, *(x,y)* is the coordinate of a point inside the node. The values *a1*, *p1*, ... *aN*, *pN* are the area and perimeter for the material in each of the resistance classes described by the **resistclasses** line at the beginning of the **.ext** file; these values are used to compute adjusted hierarchical resistances more accurately. If there were any labels ending in the character "@" attached to geometry in this node, they are considered to be node attributes and appear in the comma-separated list *attrs*, minus their trailing "@" characters. **NOTE:** since many analysis tools compute transistor gate capacitance themselves from the transistor's area and perimeter, the capacitance between a node and substrate (GND!) normally does not include the capacitance from transistor gates connected to that node. If the **.sim** file was produced by *ext2sim*(1), check the technology file that was used to produce the original **.ext** files to see whether transistor gate capacitance is included or excluded; see "Magic Maintainer's Manual #2: The Technology File" for details.

equiv *node1 node2*
Defines two node names in cell *name* as being equivalent: *node1* and *node2*. In a collection of node names related by equiv lines, exactly one must be defined by a **node** line described above.

fet *type xl yl xh yh area perim sub GATE T1 T2 ...*
Defines a transistor in *name*. The kind of transistor is *type*, a string that comes from the technology file and is intended to have meaning to simulation programs. The coordinates of a square entirely contained in the gate region of the transistor are *(xl, yl)* for its lower-left and *(xh, yh)* for its upper-right. All four coordinates are in the *name*'s coordinate space, and are subject to scaling as described in **scale** above. The gate region of the transistor has area *area* square centimicrons and perimeter *perim* centimicrons. The substrate of the transistor is connected to node *sub*, which is defined in the technology file for this type of transistor.

The remainder of a **fet** line consists of a series of triples: *GATE*, *T1*, ... Each describes one of the terminals of the transistor; the first describes the gate, and the remainder describe the transistor's non-gate terminals (e.g, source and drain). Each triple consists of the name of a node connecting to that terminal, a terminal length, and an attribute list. The terminal length is in centimicrons; it is the length of that segment of the channel perimeter connecting to adjacent material, such as polysilicon for the gate or diffusion for a source or drain.

The attribute list is either the single token "0", meaning no attributes, or a comma-separated list of strings. The strings in the attribute list come from labels attached to the transistor. Any label ending in the character "^" is considered a gate attribute and appears on the gate's attribute list, minus the trailing "^". Gate attributes may lie either along the border of a channel or in its interior. Any label ending in the character "$" is considered a non-gate attribute. It appears on the list of the terminal along which it lies, also minus the trailing "$". Non-gate attributes may only lie on the border of the channel.

The keywords in this last section describe the subcells used by *name*, and how it makes connections to and between them.

use *def use-id TRANSFORM*
Specifies that cell *def* with instance identifier *use-id* is a subcell of cell *name*. If cell *def* is arrayed, then *use-id* will be followed by two bracketed subscript ranges of the form: [*lo,hi,sep*]. The first range is for x, and the second for y. The subscripts for a given dimension are *lo* through *hi* inclusive, and the separation between adjacent

array elements is *sep* centimicrons.

TRANSFORM is a set of six integers that describe how coordinates in *def* are to be transformed to coordinates in the parent *name*. It is used by *ext2sim*(1) in transforming transistor locations to coordinates in the root of a design. The six integers of *TRANSFORM* (*ta, tb, tc, td, te, tf*) are interpreted as components in the following transformation matrix, by which all coordinates in *def* are post-multiplied to get coordinates in *name*:

ta	*td*	0
tb	*te*	0
tc	*tf*	1

merge *path1 path2 C a1 p1 a2 p2 ... aN pN*

Used to specify a connection between two subcells, or between a subcell and mask information of *name*. Both *path1* and *path2* are hierarchical node names. To refer to a node in cell *name* itself, its pathname is just its node name. To refer to a node in a subcell of *name*, its pathname consists of the *use-id* of the subcell (as it appeared in a use line above), followed by a slash (*/*), followed by the node name in the subcell. For example, if *name* contains subcell *sub* with use identifier *sub-id*, and *sub* contains node *n*, the full pathname of node *n* relative to *name* will be *sub-id/n*.

Connections between adjacent elements of an array are represented using a special syntax that takes advantage of the regularity of arrays. A use-id in a path may optionally be followed by a range of the form [*lo:hi*] (before the following slash). Such a use-id is interpreted as the elements *lo* through *hi* inclusive of a one-dimensional array. An element of a two-dimensional array may be subscripted with two such ranges: first the y range, then the x range.

Whenever one *path* in a **merge** line contains such a subscript range, the other must contain one of comparable size. For example,

merge sub-id[1:4,2:8]/a sub-id[2:5,1:7]/b

is acceptable because the range 1:4 is the same size as 2:5, and the range 2:8 is the same size as 1:7.

When a connection occurs between nodes in different cells, it may be that some resistance and capacitance has been recorded redundantly. For example, polysilicon in one cell may overlap polysilicon in another, so the capacitance to substrate will have been recorded twice. The values *C*, *a1*, *p1*, etc. in a **merge** line provide a way of compensating for such overlap. Each of *a1*, *p1*, etc. (usually negative) are added to the area and perimeter for material of each resistance class to give an adjusted area and perimeter for the aggregate node. The value *C* attofarads (also usually negative) is added to the sum of the capacitances (to substrate) of nodes *path1* and *path2* to give the capacitance of the aggregate node.

cap *node1 node2 C*

Defines a capacitor between the nodes *node1* and *node2*, with capacitance *C*. This construct is used to specify both internodal capacitance within a single cell and between cells.

AUTHOR

Walter Scott

SEE ALSO

ext2sim(1), magic(1)

NAME

glyphs – format of .glyphs files

DESCRIPTION

Glyph files (".glyph" extension) are used to store commonly-used bit patterns (glyphs) for Magic. Right now, the bit patterns are used for two purposes in Magic. First, they specify patterns for programmable cursors: each cursor shape (e.g. the arrow used for the wiring tool) is read in as a glyph from a glyph file. Second, glyphs are used by the window manager to represent the icons displayed at the ends of scroll bars. Glyph file names normally have the extension **.glyph**.

Glyph files are stored in ASCII format. Lines beginning with "#" are considered to be comments and are ignored. Blank lines are also ignored. The first non-comment line in a glyph file must have the syntax **size** *nGlyphs width height* The *nGlyphs* field must be a number giving the total number of glyphs stored in the file. The *width* and *height* fields give the dimensions of each glyph in pixels. All glyphs in the same file must have the same size.

The **size** line is followed by a description for each of the glyphs. Each glyph consists of *height* lines each containing $2 \times width$ characters. Each pair of characters corresponds to a bit position in the glyph, with the leftmost pair on the topmost line corresponding to the upper-left pixel in the glyph.

The first character of each pair specifies the color to appear in that pixel. The color is represented as as a single character, which must be the short name of a display style in the current display style file. Some commonly-used characters are **K** for black, **W** for white, and **.** for the background color (when **.** is used in a cursor, it means that that pixel position is transparent: the underlying picture appears through the cursor). See "Magic Maintainer's Manual #3: Display Styles, Color Maps, and Glyphs" for more information.

The second character of each pair is normally blank, except for one pixel per glyph which may contain a "*" in the second character. The "*" is used for programmable cursors to indicate the hot-spot: the pixel corresponding to the "*" is the one that the cursor is considered to point to.

For an example of a glyph file, see ~cad/lib/magic/sys/color.glyphs.

SEE ALSO

magic(1), dstyle(5)

NAME

magic − format of **.mag** files read/written by Magic

DESCRIPTION

Magic uses its own internal ASCII format for storing cells in disk files. Each cell *name* is stored in its own file, named *name*.**mag**.

The first line in a **.mag** file is the string

magic

to identify this as a Magic file.

The next line is optional and is used to identify the technology in which a cell was designed. If present, it should be of the form

tech *techname*

If absent, the technology defaults to a system-wide standard, currently **nmos**.

The next line is also optional and gives a timestamp for the cell. The line is of the format

timestamp *stamp*

where *stamp* is a number of seconds since 00:00 GMT January 1, 1970 (i.e, the Unix time returned by the library function *time()*). It should be the last time this cell or any of its children changed. The timestamp is used to detect when a child is edited outside the context of its parent (the parent stores the last timestamp it saw for each of its children; see below). When this occurs, the design-rule checker must recheck the entire area of the child for subcell interaction errors. If this field is omitted in a cell, Magic supplies a default value that forces the rechecks.

Next come lines describing the contents of the cell. There are three kinds of groups of lines, describing mask rectangles, subcell uses, and labels. Each group must appear contiguously in the file, but the order between groups is arbitrary.

Each group of mask rectangles is headed with a line of the format

<< *layer* >>

where *layer* is a layername known in the current technology (see the **tech** line above). Each line after this header has the format

rect *xbot ybot xtop ytop*

where *(xbot, ybot)* is the lower-left corner of the rectangle in Magic (lambda) coordinates, and *(xtop, ytop)* is the upper-right corner. Degenerate rectangles are not allowed; *xbot* must be strictly less than *xtop*, and *ybot* strictly less than *ytop*. The smallest legal value of *xbot* or *ybot* is −67108858, and the largest legal value for *xtop* or *ytop* is 67108858. Values that approach these limits (within a factor of 100 or 1000) may cause numerical overflows in Magic even though they are not strictly illegal. We recommend using coordinates around zero as much as possible.

Rectangles should be non-overlapping, although this is not essential. They should also already have been merged into maximal horizontal strips (the neighbor to the right or left of a rectangle should not be of the same type), but this is also not essential.

The second kind of line group describes a single cell use. Each cell use group is of the following form:

use *filename use-id*
array *xlo xhi xsep ylo yhi ysep*
timestamp *stamp*
transform *a b c d e f*
box *xbot ybot xtop ytop*

A group specifies a single instance of the cell named *filename*, with instance-identifier *use-id*. The instance-identifier *use-id* must be unique among all cells used by this **.mag** file. If *use-id* is not specified, a unique one is generated automatically.

The **array** line need only be present if the cell is an array. If so, the X indices run from *xlo* to *xhi* inclusive, with elements being separated from each other in the X dimension by *xsep* lambda. The Y indices run from *ylo* to *yhi* inclusive, with elements being separated from each other in the Y dimension by *ysep* lambda. If *xlo* and *xhi* are equal, *xsep* is ignored; similarly if *ylo* and *yhi* are equal, *ysep* is ignored.

The **timestamp** line is optional; if present, it gives the last time this cell was aware that the child *filename* changed. If there is no **timestamp** line, a timestamp of 0 is assumed. When the subcell is read in, this value is compared to the actual value at the beginning of the child cell. If there is a difference, the "timestamp mismatch" message is printed, and Magic rechecks design-rules around the child.

The **transform** line gives the geometric transform from coordinates of the child *filename* into coordinates of the cell being read. The six integers *a*, *b*, *c*, *d*, *e*, and *f* are part of the following transformation matrix, which is used to postmultiply all coordinates in the child *filename* whenever their coordinates in the parent are required:

a	*d*	0
b	*e*	0
c	*f*	1

Finally, **box** gives an estimate of the bounding box of cell *filename* (covering all the elements of the array if an **array** line was present), in coordinates of the cell being read.

The third kind of line group in a **.mag** file is a list of labels. It begins with the line

<< labels >>

and is followed by zero or more lines of the following form:

rlabel *layer xbot ybot xtop ytop position text*

Here *layer* is the name of one of the layers specified in the technology file for this cell. The label is attached to material of this type. *Layer* may be **space**, in which case the label is not considered to be attached to any layer.

Labels are rectangular. The lower-left corner of the label (the part attached to any geometry if *layer* is non-space) is at *(xbot, ybot)*, and the upper-right corner at *(xtop, ytop)*. The width of the rectangle or its height may be zero. In fact, most labels in Magic have a lower-left equal to their upper right.

The text of the label, *text*, may be any sequence of characters not including a newline. This text is located at one of nine possible orientations relative to the center of the label's rectangle. *Position* is an integer between 0 and 8, each of which corresponds to a different

orientation:

0	**center**
1	**north**
2	**northeast**
3	**east**
4	**southeast**
5	**south**
6	**southwest**
7	**west**
8	**northwest**

A **.mag** file is terminated by the line

<< end >>

Everything following this line is ignored.

Any line beginning with a pound sigh ("#") is considered to be a comment and ignored. Beware, however, that these comments are discarded by Magic when it reads a cell, so if that cell is written again by Magic, the comments will be lost.

NOTE FOR PROGRAMS THAT GENERATE MAGIC FILES

Magic's incremental design rule checker expects that every cell is either completely checked, or contains information to tell the checker which areas of the cell have yet to be examined for design-rule violations. To make sure that the design-rule checker verifies all the material that has been generated for a cell, programs that generate **.mag** files should place the following rectangle in each file:

<< checkpaint >>
rect *xbot ybot xtop ytop*

This rectangle may appear anywhere a list of rectangles is allowed; immediately following the **timestamp** line at the beginning of a **.mag** file is a good place. The coordinates *xbot* etc. should be large enough to completely cover anything in the cell, and must surround all this material by at least one lambda. Be careful, however, not to make this area too ridiculously large. For example, if you use the maximum and minimum legal tile coordinates, it will take the design-rule checker an extremely long time to recheck the area.

SEE ALSO

magic(1)

NAME

mpanda – template format for mpanda(1)

DESCRIPTION

Making a template for **mpanda** consists of first designing a sample multiply-folded PLA in the desired style, and then labeling *tiles* using the **Magic**(1) graphics editor. A *tile* is a rectangular area of paint, and is defined by a named label outlining the area. **MPanda** assembles these tiles, row by row, to form a multiply-folded PLA.

There are 11 groups of tiles in a **mpanda** template:

1) the core of the AND plane
2) the core of the OR plane
3) the sides of the AND plane
4) the sides of the OR plane
5) the top and bottom of the AND plane
6) the top and bottom of the OR plane
7) the tiles between planes
8) the horizontal spacing tiles in the AND plane
9) the vertical spacing tiles in both planes
10) the horizontal ground tiles
11) the vertical ground tiles

Any of the tiles not in the core areas may contain linear labels with the name **[GND]** or **[Vdd]**. Linear labels are lines with a name attached. Labels with the names **[GND]** and **[Vdd]** will be stretched to allow increased current through the PLA. That is, the tile would be figuratively "cut" along the line label and then the two "pieces" would be stretched apart by a designated amount. A given tile may contain many occurrences of these linear labels, but none of them can be colinear. If 2 labels within a tile are colinear, the stretching of one of them will turn the other one into a rectangle, and it is not possible to stretch along a rectangle.

Point labels may occur anywhere in a tile. Global point labels **GND!** and **Vdd!** may be put in corner tiles, to be placed in all **mpanda**-generated PLAs. The point label **$input$** may occur in tiles on the top or bottom of the AND plane, and the **$output$** label may occur in tiles on the top or bottom of the OR plane. These labels will be replaced with the name of the corresponding input or output. There should be no more than one **$input$** label on each input, and no more than one **$output$** label on each output. Tiles between the AND and OR cores may contain point labels, **$product$**, which are placed when a product term bridges between AND and OR planes. These **$product$** labels are useful for debugging purposes within a PLA.

MPanda builds PLAs by rows of tiles. All of the tiles in a row (except the bottom row) have their bottom edges aligned. The top row of tiles would be made up of edge tiles arranged from the left to the right of the PLA, all stacked together along a line. A product row of the PLA would have a left edge tile, core tiles for the AND and OR plane(s), and a right edge tile, all aligned. Each row of tiles is placed one beneath the next according to the alignment of the first tile on the left side of each row. The alignment of the first tile in each row will be discussed below. In the last row of tiles (bottom tiles), all of the tiles are aligned with each other by their top edges. The whole row is aligned to the previous string above it by aligning to the bottom edge of that previous row (the last product row).

THE CORE OF THE *AND* PLANE:

Required: **sp-and, l0-and, l1-and, l!-and, l;-and, l:-and, r0-and, r1-and, r!-and, r;-and, r:-and, lc-and, rc-and**

Optional: **lu-and, lh-and, lb-and, ru-and, rh-and, rb-and**

These tiles contain transistors that implement the PLA function. The vertical and horizontal pitch of the core of the AND plane is set by the tile **sp-and**. The first character of the tile name indicates whether it is a *left* tile or a *right* tile. Left tiles are placed in every other column in the AND plane core, starting with the first column. Right tiles are placed in every other column starting with the second column. The tile **sp-and** determines the amount of overlap between columns.

The second character of the name represents the function of the tile according to the format for folded PLAs (see PLA(5)). For instance, a *1* stands for for tiles that contain a transistor and a *0* for tiles that pass the input line up to the next tile but have no transistor. For folded PLAs, additional tiles have, for the second character of the name, an for tiles that contain a transistor and are split below, a for tiles that contain a transistor and are folded to the right, and an for tiles that contain a transistor and are folded below and to the right. For multiply-folded PLAs, a represents a contact tile that will make a contact between a vertical and horizontal signal line, as is the case when an input or output is brought into the core from the sides of the PLA.

Optional tiles that minimize excess interconnect in the length and width of the PLA begin with for tiles that do not pass the vertical signal up to the next row, for tiles that do not pass the horizontal signal across to the next column, and for tiles that do not pass both the vertical and the horizontal signals. These tiles are not necessary for the functionality of PLAs, but they help reduce capacitances and therefore delay times throughout the PLA.

Columns in the PLA AND plane are grouped into (left, right) pairs, according to (signal, complement) pairs. The selection of the core tiles in these column pairs is determined by the symbols occuring in a personality matrix as is described in PLA(5). If the symbol is a "0", then the tiles **l0-and** and **r1-and** are placed in the column as a pair. If the symbol is a "1", then the tiles **l1-and** and **r0-and** are placed. If the symbol is a "!", then the tiles **l!-and** and **r0-and** are placed. If the symbol is a "o", then the tiles **l0-and** and **r!-and** are placed. Similar pairings of tiles are done for the symbols "@", ";", "#", and ":".

The optimizing tiles are used to replace **l0-and** and **r0-and** tiles when signal lines do not need to extend the full length and width of the PLA. For example, all **l0-and** tiles above the topmost **l1-and** (**l!-and**, **l;-and**, and **l:-and**) tile are replaced with **lu-and** tiles in order to allow shortened vertical poly lines. A similar substitution is done with **ru-and** tiles in the alternating right columns.

In a similar fashion, all **l0-and** and **r0-and** tiles that extend horizontally to the edge of the PLA are replaced with **lh-and** and **rh-and**, respectively. If a horizontal *and* vertical line can be minimized, the **lb-and** and **rb-and** tiles automatically replace the **l0-and** and **r0-and** tiles.

All of the tiles in this group are assumed to be of the same height. (However, creative designers may design otherwise.) When **mpanda** aligns a row of core tiles in the AND plane, the **sp-and** tile is used to control the overlap between column pair tiles. When the core of the AND plane is made, the lower left corner of a left (or right) tile is aligned to the lower left corner of the **sp-and** tile. The next right (or left) tile is then aligned such that its lower left corner is aligned to the **sp-and** tile's lower right corner. This pattern of placing a core tile (left or right), spacing a distance of **sp-and**, and then placing the next tile (left or right), is done throughout the core of the AND plane. It is highly recommended that the user look at the **pa-CS3.mag** template before trying to define a new one.

THE CORE OF THE *OR* PLANE:

Required: **sp-or, u0-or, u1-or, ui-or, u|-or, uj-or, r0-or, d1-or, di-or, d|-or, dj-or**

Optional: **uu-or, uh-or, ub-or, du-or, dh-or, db-or**

These tiles are similar to the ones in the AND plane. A *u* as the first character indicates that the tile occurs in every other row, starting with the first (the *up* rows). A *d* indicates that the tile will be placed in the other *down* rows. All of the tiles in this group are assumed to be of the same width. The tile **sp-or** sets the horizontal spacing for the OR plane. Since **mpanda** builds PLAs row by row, all tiles defined in the OR plane for one row are aligned in a line by their bottom edges.

THE SIDES OF THE *AND* PLANE:

Required: **ul-and, ll-and, ur-and, lr-and, right-and, left-and, left-in, right-in**

Optional: **hul-and, hur-and, nul-and, nur-and, vul-and, vll-and, vur-and, vlr-and, nright-and, nleft-and**

These tiles align to the left and right sides of the AND plane, when the AND plane is an exterior plane, as in an AND-OR-AND structure. The tile **ul-and** is placed in the upper left corner of the AND plane, while the tile **ll-and** goes in the lower left corner. Along the left side of the AND plane, each product row has a **left-and** tile. For AND planes on the outer right edge of the PLA, the tile **ur-and** is placed in the upper right corner of the AND plane, while the tile **lr-and** goes in the lower right corner. The rows in between the top and bottom contain **right-and** tiles along the right side of the AND plane.

AND planes which are interior to a PLA, such as in an OR-AND-OR PLA, do not have side tiles. When a multiply-folded PLA is made, the tiles **left-in** and **right-in** replace the **left-and** and **right-and** tiles, respectively. These side input buffers are twice as tall as the **left-and** and **right-and** tiles because connections must be made to the input signal and its complement in the multiply-folded column.

Optional corner tiles provide for PLAs that do not have folding. The extra overhead is in either the height (those tiles that begin with) or the width (those tiles that begin with) of the tile. Tiles that begin with are used in PLAs that are not folded and thus, do not have the extra overhead in the horizontal and vertical direction.

The tiles along the top of the PLA are placed in a row with their bottom edges aligned. The tiles along the bottom of the PLA are placed in a row with their top edges aligned. The side tiles of the AND plane are assumed to match the height of the core tiles of the AND plane and to be aligned along the same edge as the core tiles for each row.

Since the first tile in a row determines the alignment for the whole row, the **left-and** or **left-in** tiles (for PLA structures which begin with an AND plane) are assumed to be stacked exactly below succeeding rows. That is, there *is no* overlapping of left side tiles for AND-first PLAs.

THE SIDES OF THE *OR* PLANE:

Required: **ul-or, ll-or, ur-or, lr-or, rightu-or, rightd-or, leftu-or, leftd-or, leftu-out, leftd-out, rightu-out, rightd-out**

Optional: **hul-or, hur-or, nul-or, nur-or, vul-or, vll-or, vur-or, vlr-or nrightu-or, nrightd-or, nleftu-or, nleftd-or**

These tiles function in a manner analogous to the tiles on the sides of the AND plane. Note that the **rightu-or** tile is placed in the *up* rows, while the **rightd-or** tile is placed in the *down* rows. The output buffer tiles, **leftu-out, leftd-out, rightu-out,** and **rightd-out,** are the same height as tiles in the core of the OR plane.

When creating rows of tiles (for PLAs that begin with an OR plane), there is some overlapping of these first "left" tiles. The **leftu-or** (or **leftu-out** or **leftu-out** or **leftd-out**) tiles, for PLA structures which begin with an OR plane, are assumed to be stacked below succeeding rows such that they overlap an amount controled by **sp-or**. That is, when

aligning a new "left-or" tile for a new row, the sequence of alignments is as follows: align the lower left corner of the previous row with the upper left corner of the **sp-or** tile, align the lower left corner of the **sp-or** tile with the lower left corner of the "left-or" tile. Thus, there *is* overlapping of left edge tiles for OR-first PLAs.

THE TOPS AND BOTTOMS OF *AND* PLANES:

Required: **top-in, bot-in, top-and, bots-and, tops-and**

Optional: **ntop-and**

These tiles function in a manner analogous to the tiles on the left and right sides of the AND and OR planes. The **top-in** and **bot-in** tiles contain input buffers coming into the PLA from the top and bottom, respectively. All of these tiles are only placed in every other column, starting with the first because connections must be made with signal and complement lines. The tiles **bots-and** and **tops-and** control the amount of horizontal spacing and overlap between adjacent tiles in the same way that the **sp-and** tile controls horizontal spacing in the AND plane. The **ntop-and** tile is used for PLAs that do not have column folds.

The top tiles are aligned by their bottom edges in a row. The bottom tiles are aligned by their top edges.

THE TOPS AND BOTTOMS OF *OR* PLANES:

Required: **topl-out, topr-out, botl-out, botr-out, topl-or, topl-or**

Optional: **ntopl-or, ntopr-or, nbotl-or, nbotr-or**

These tiles function in a manner analogous to the tiles on the top and bottom sides of the AND plane, except that the **topl-or** tile is placed in every other column starting with the first (the *left* columns) while **topr-or** is placed in the alternating columns. These tiles are also aligned along the top edges (unlike the top tiles in the AND plane which are aligned along the bottom edges).

THE TILES BETWEEN PLANES:

Required: **topmid-ao, botmid-ao, topmid-oa, botmid-oa, midu-ao, midd-ao, midu-oa, midd-oa, nmidu-ao, nmidd-ao, nmidu-oa, nmidd-oa, cmidd-ao, cmidu-ao, cmidd-oa, cmodu-oa**

Optional: **ntopmid-ao, ntopmid-oa**

These tiles are between the AND and OR planes. When an AND plane is followed by an OR plane, the *ao* tiles are used, and when an OR plane is followed by an AND plane, the *oa* tiles are used. The tiles that connect product rows between the AND and OR cores, **midd_ao**, **midu_ao**, **midd_oa**, and **midu_oa**, usually contain some type of circuit element that pulls up each product row. Tiles that begin with a *n* do not contain these circuit elements and are placed when the product row does not need to be connected between two planes. The tiles that begin with a *c* are for middle tiles in contact rows.

In the top row of tiles, the **topmid-ao** tile matches the AND tiles in the top rows by its lower left corner. The **topmid-oa** tile matches the OR tiles in the top rows by its upper left corner. In the bottom row of tiles, the **botmid-ao** and **botmid-oa** tiles match the other tiles in the bottom rows by their top edges. The "midd" and "midu" tiles are assumed to be the same height as the core tiles in the AND plane. Their bottom edges are aligned along the same edge as the other tiles in their row.

THE HORIZONTAL SPACING TILES IN THE *AND* PLANE

Required: **sh-and, both-and, toph-and**

Optional: **shx-and**

These tiles handle the extra horizontal spacing that is needed in the AND plane for folding rows in the AND plane. When two logical rows must share one physical row, the structure of the AND plane (pairings of left and right core tiles) is such that they may need extra spacing in the horizontal direction. This occurs when a right tile (containing a transistor) of one logic row is adjacent to the left tile (containing a transistor) of another logic row. Usually, one contact is used between these transistors to connect them to the product term. However, when a logical fold must be made between these transistors, the contact can not be split, so the duplication of the contact and the extra space needed for physical correctness is contained in the **shx-and** (for no connection between right and left tiles) and **sh-and** tiles (for connecting between right and left tiles, when the logical rows are not folded but horizontal spacing was required on another row). The **both-and** and **toph-and** tiles provide the same amount of horizontal spacing in the top and bottom tiles in the AND plane.

THE VERTICAL SPACING TILES IN BOTH PLANES

Required: **sv-or, lv-and, rv-and, midv-ao, midv-oa, shv-and, leftv-and, leftv-or, rightv-and, rightv-or**

Optional: **svx-or, lvx-and, rvx-and**

These tiles function in a manner analogous to the horizontal space tiles in the AND plane. In the OR plane, the up and down tiles require the **sv-or** (for connecting) and **svx-or** (for non-connecting) tiles for vertical splits along columns in the OR plane. The consequences of spacing out the OR plane vertically, require the tiles, **leftv-and, rightv-and, leftv-or**, and **rightv-or** along the sides of the PLA. In the AND plane, the tiles **lv-and** and **rv-and**, allow for vertical spacing and connect the vertical signals. The tile **lvx-and** and **rvx-and** provide vertical spacing and donot connecting (the in the name) vertical signals. The **shv-and** tile is placed when vertical and horizontal spacing intersect in the core of the AND plane. The **midv-ao** and **midv-oa** tiles contain the vertical spacing in the middle tiles between planes.

These tiles are assumed to be the same height as the core tiles in the AND plane. Their bottom edges are aligned along the same edge as the other tiles in their row.

THE HORIZONTAL GROUND TILES

Required: **HGleft-and, HGright-and, HGl-and, HGr-and, HGright-or, HGleft-or, HGmid-ao, HGmid-oa, HG-or, HGshv-and**

Optional: **nHGleft-and, nHGright-and, nHGright-or, nHGleft-or, HGlx-and, HGrx-and**

For the extra ground lines that are needed for large PLAs, these horizontal ground tiles contain metal lines that run the width of the PLA from one side to the other. These tiles are aligned in the PLA in a manner similar to the horizontal spacing tiles.

THE VERTICAL GROUND TILES

Required: **VGtop-or, VG-or, VGd-or, HVG-or, cHVG-or, VGbot-or**

Optional: **nVGtop-or, VGux-or, VGdx-or**

These tiles are aligned in the PLA in a manner analogous to the horizontal ground tiles and vertical spacing tiles.

SEE ALSO

mpanda(1), mpack(3), PLA(5), mpla(1), mpla(5)

AUTHOR

Grace H. Mah

NAME

mpla – template format for mpla(1)

DESCRIPTION

Making a template for *mpla(1)* consists of first drawing a sample PLA in the desired style, and then labeling *tiles* using the Magic(1) graphics editor. A *tile* is a rectangular area of paint, and is defined by a named label outlining the area. *Mpla* assembles these tiles to form a finished PLA.

GETTING STARTED

Before reading this manual page, it would be helpful to get a plot of one of the MPLA templates located in **~cad/lib/mpla** and a copy of the two gremlin figures located in **~cad/src/mpla**. If this man page is hardcopy, then the two gremlin figures may already be attached.

OVERVIEW OF THE TILES

There are 11 groups of tiles in a *mpla* template:

1) the core of the AND plane
2) the core of the OR plane
3) the left side of the AND plane
4) the top of the AND plane
5) the bottom of the AND plane
6) the top of the OR plane
7) the bottom of the OR plane
8) the right of the OR plane
9) the tiles between the two planes
10) horizontal ground grid tiles
11) vertical ground grid tiles

There are also 2 optional groups of tiles which are used for clocked inputs and outputs:

12) clocking (if any) for the AND plane
13) clocking (if any) for the OR plane

Any of the tiles not in the core areas may contain linear labels with the name **<GND>**, **<Vdd>**. A linear label is just a rectangle label in which has either zero width or zero height. Labels with the names **<GND>** and **<Vdd>** will be stretched to allow increased current through the PLA. A given tile may contain many occurances of these 2 labels, but none of them can be colinear.

The **<input>** label may occur in tiles on the top or bottom of the AND plane, and the **<output>** label may occur in tiles on the top or bottom of the OR plane. The labels will be replaced with the name of the corresponding input or output. There should no more than one **<input>** label on each input, and no more than one **<output>** label on each output.

THE CORE OF THE AND PLANE: sp-and, l0-and, l1-and, l.-and, r0-and, r1-and, r.-and

These tiles contain transistors that implement the PLA function. The vertical pitch of the whole PLA is set by the tile **sp-and**, as is the horizontal pitch of the AND plane.

The first character of the tile name indicates whether it is a *left* tile or a *right* tile. Left tiles are placed in every other column in the AND plane core, starting with the first column. Right tiles, on the other hand, are placed every other column starting with the second column. The second character of the name is a 1 for tiles that contain a transistor, a 0 for tiles that pass the input line up to the next tile but have no transistor, and . for tiles that do not pass the input line up to the next row and have no transistor.

Columns in the PLA AND plane are grouped into (left, right) pairs, and the selection of the core tiles in these column pairs is determined by the input bit in the truth table row for the current minterm. If that bit is a 0, then the tiles **l0-and** and **r1-and** are placed in the column pair. If the bit is a 1, then the tiles **l1-and** and **r0-and** are placed. All **l0-and** tiles above the topmost **l1-and** tile are replaced with **l.-and** tiles in order to allow shortened poly lines, as in the standard nMOS PLA. A similar substitution is done with the **r.-and** tile in the right columns.

The AND plane core will be surrounded by tiles on its perimeter. It is possible for these tiles to overlap the core, this is decribed in the section that deals with these surrounding tiles.

THE CORE OF THE OR PLANE: sp-or, u0-or, u1-or, u.-or, d0-or, d1-or, d.-or

These tiles are similar to the ones in the AND plane. A 'u' as the first character indicates that the tile occurs in every other row, starting with the first (the *up* rows). A 'd' indicates that the tile will be placed in the other *down* rows. The tile **sp-or** sets the horizontal spacing for the OR plane.

THE LEFT SIDE OF THE AND PLANE: Oleft-and, ul-and, leftu-and, leftd-and, ll-and

The tile **ul-and** is placed in the Upper Left corner of the AND plane, while the tile **ll-and** goes in the Lower Left corner. The rows in between contain **leftu-and** and **leftd-and** tiles, with the former going in odd numbered rows (up rows) and the latter in the even numbered rows (down rows). The tile **Oleft-and** controls the amount of overlap between the other 3 tiles and the core of the AND plane. In particular, the right sides of the **ul-and, left-and,** and **ll-and** tiles are lined up such that they overlap the AND plane core by the width of the tile **Oleft-and**.

THE TOP OF THE AND PLANE: Otop-and, ul-and, top-and

These tiles function in a manner analogous to the tiles on the left side of the AND plane, except that the tile **top-and** is only placed in every other column, starting with the first. Note that the tile **ul-and** occurs in two groups of tiles. It is included in this group since its overlap with the *top* of the AND plane is controlled by the tile **Otop-and**, even though its overlap with the *left* side of the plane is controlled by the tile *Oleft-and* in the left-and-plane group.

THE BOTTOM OF THE AND PLANE: Obot-and, ll-and, bot-and

These tiles function in a manner analogous to the tiles on the top side of the AND plane.

THE TOP OR THE OR PLANE: Otop-or, topl-or, topr-or, ur-or

These tiles function in a manner analogous to the tiles on the top side of the AND plane, except that the **topl-or** tile is placed in every other column starting with the first (the *left* columns) while **topr-or** is placed in the other columns.

THE BOTTOM OF THE OR PLANE: Obot-or, botl-or, botr-or, lr-or

These tiles function in a manner analogous to the tiles on the top side of the OR plane.

THE RIGHT SIDE OF THE OR PLANE: Oright-or, rightu-or, rightd-or, ur-or, lr-or

These tiles function in a manner analogous to the tiles on the left side of the and plane. Note that the **rightu-or** tile is placed in the *up* rows, while the **rightd-or** is placed in the *down* rows.

THE AREA BETWEEN PLANES: top-mid, Otop-mid, bot-mid, Obot-mid, midu, midd

Similar to the right side of the OR plane.

HORIZONTAL GROUND LINES: HGleft-and, HGl-and, HGl.-and, HGr-and, HGr.-and, HG-mid, HG-or, HGright-or

Horizontal ground lines may be inserted in the PLA. They always are placed such that there is an even (and nonzero) number of rows above and a nonzero (but odd or even) number of rows below. **HGl.-and** has a vertical input poly line in it, while **HGl-and** does

not.

VERTICAL GROUND LINES: VGtop-or, VGd-or, VGd.-or, VGu-or, VGu.-or, HVG-or

These lines are placed in the OR plane. They always have an even number of OR columns to the left, and either an even or an odd number to the right. Central tiles without a dot in the name contain an extension of the minterm to the right, while tiles with the dot don't. The tile HVG-or is placed at the intersection of a horizontal and vertical ground line.

CLOCKED INPUTS: Cul-and, Ctop-and, Cll-and, Cbot-and

If any of these tiles exist and the user has asked for clocked inputs, they will be used in preference to the tiles which do not start with the letter 'C'.

CLOCKED OUTPUTS: Cur-or, Ctopl-or, Ctopr-or, Clr-or, Cbotl-or, Cbotr-or

Similar to clocked inputs.

SEE ALSO

mpla(1), mpack(1)

HISTORY

Mpla is a port of the program 'tpla'.

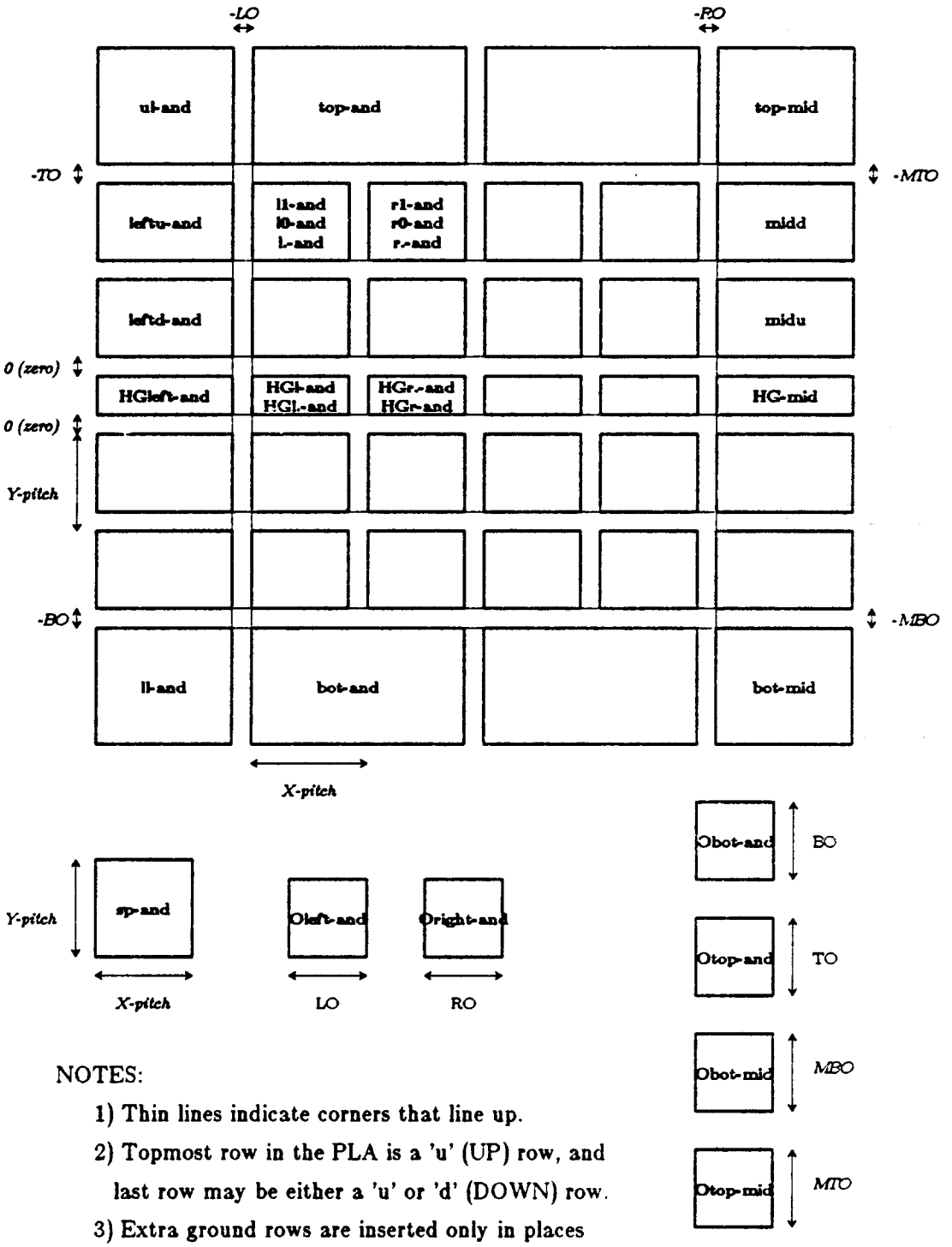
AUTHOR

Robert N. Mayo

BUGS

There really should be a way for template designers to specify what they want aligned with what. Currently this is fixed in the *mpla* code, but you can think of ways to specify this graphically in the templates instead.

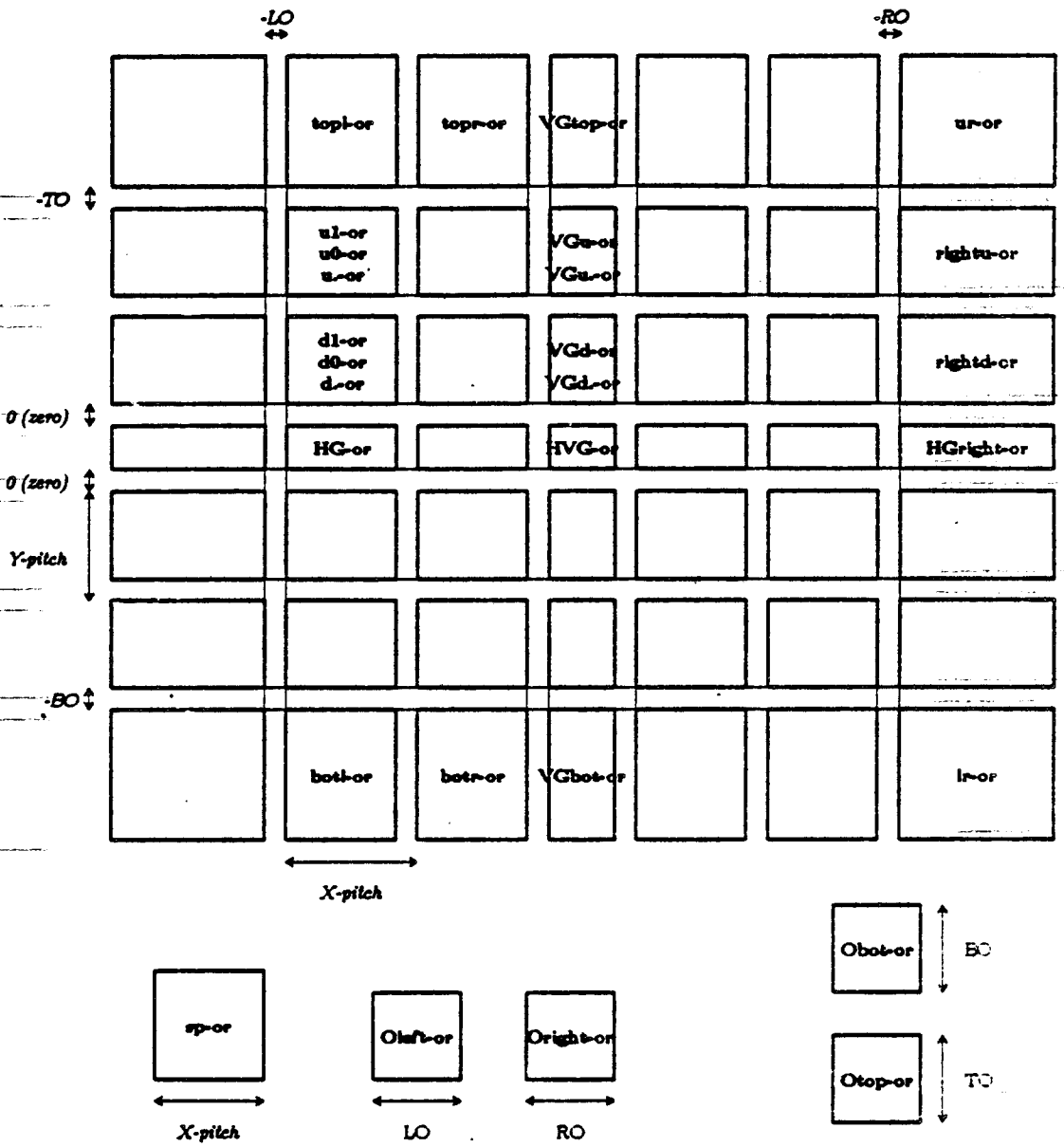
Exploded View of AND Plane



NOTES:

1) Thin lines indicate corners that line up.
2) Topmost row in the PLA is a 'u' (UP) row, and last row may be either a 'u' or 'd' (DOWN) row.
3) Extra ground rows are inserted only in places where there is a 'd' row above and a 'u' row below.
4) Overlap amounts *MTO* and *MBO* apply only to the tiles in the 'mid' section.

Exploded View of OR Plane



NOTES:

1) Thin lines indicate corners that line up.
2) Left most column in the plane is a 'l' (LEFT) row, and
   last column may be either a 'l' or 'r' (RIGHT) column.
3) Extra ground columns are inserted only in places
   where there is a'r' column to the left and a 'l' row to the right.
4) Tiles on the left side of the figure are the tiles
   from the right side of the AND plane
5) Y-pitch of the OR plane is set by the Y-pitch of the AND plane.

NAME

net – format of .net files read/written by Magic's netlist editor

DESCRIPTION

Netlist files are read and written by Magic's netlist editor in a very simple ASCII format. The first line contains the characters " Netlist File" (the leading blank is important). After that comes a blank line and then the descriptions of one or more nets. Each net contains one or more lines, where each line contains a single terminal name. The nets are separated by blank lines. Any line that is blank or whose first character is blank is considered to be a separator line and the rest of its contents are ignored.

Each terminal name is a path, much like a file path name in Unix. It consists of one or more fields separated by slashes. The last field in the path is the name of a label in a cell. The other fields (if any), are cell instance identifiers that form a path from the edit cell down to the label. The first instance identifier must name a subcell of the edit cell, the second must be a subcell of the first, and so on.

Instance identifiers are unique within their parent cells, so a terminal path selects a unique cell to contain the label. However, the same label may appear multiple times within its cell. When this occurs, Magic assumes that the identical labels identify electrically equivalent terminals; it will choose the closest of them when routing to that terminal. Further, after connecting to one of these terminals Magic may take advantage of the internal wiring connecting them together and route through a cell to complete the net's wiring.

An example netlist file follows below. It contains three distinct nets.

```
 Netlist File

alu/bit_1/cout
alu/bit_2/cin


regcell[21,2]/output
latch[2]/input
 This line starts with a blank, so it's a separator.
opcode_pla/out6
shifter/drivers/shift2
```

SEE ALSO

magic(1)

NAME

pla – Format for physical description of Programmable Logic Arrays.

SYNOPSIS

pla

DESCRIPTION

This format is used by programs which manipulate plas to describe the physical implementation. Lines beginning with a '#' are comments and are ignored. Lines beginning with a '.' contain control information about the pla. Currently, the control information is given in the following order:

.i <number of inputs>
.o <number of outputs>
.p <number of product terms (pterms)>
and optionally,
.na<name> (the name to be used for the pla)

What follows then is a description of the AND and OR planes of the pla with one line per product term. Connections in the AND plane are represented with a '1' for connection to the non-inverted input line and a '0' for connection to the inverted input line. No connection to an input line is indicated with 'x', 'X', or '-' with '-' being preferred. Connections in the OR plane are indicated by a '1' with no connection being indicated with 'x', 'X', '0', or '-' with '-' being preferred. Spaces or tabs may be used freely and are ignored.

The end of the pla description is indicated with:

.e

Programs capable of handling split and folded arrays employ the following format:

AND PLANE

Column (1) Contact to input (2) No contact to input

(1)	(2)	
1	–	Normal contacts, no splits or folds
!	_	Split below
;	,	Fold to right
:	.	Split below and fold to right

OR PLANE

Column (1) Contact to output (2) No contact to output

(1)	(2)	
I	~	Normal contacts, no splits or folds
i	=	Split below
|	'	Fold to right
j	"	Split below and fold to right

ADDITIONAL ELEMENTS

*	Input buffer
+	Output buffer
D	Depletion load associated with product term
N	No depletion load associated with product term

Note that the decoding function of the AND plane is separated from the specification of its connectivity. This makes the AND and OR plane specifications identical.

These programs handle the following more general set of .parameters:

.il <number of left-AND plane inputs>
.ir <number of right-AND plane inputs>
.ol <number of left-OR plane inputs>
.or <number of right-OR plane inputs>
.p <number of product terms>

.ilt <labels left-top-AND plane>
.ilb <labels left-bottom-AND plane>
.irt <labels right-top-AND plane>
.irb <labels right-bottom-AND plane>
.olb <labels left-bottom-OR plane>
.olt <labels left-top-OR plane>
.orb <labels right-bottom-Or plane>
.ort <labels right-top-Or plane>
.pl <labels left product terms>
.pr <labels right product terms>

The first group of parameters must precede the second group. If there is only one AND or OR plane it is assumed to be the left one and the companion .parameters may be shortened by dropping their (left,right) designation character.

In order to better deal with folded and split PLAs, the following .parameters are proposed:

.ig <input group>
.og <output group>
.ins <inputs excluded from splitting>
.inf <inputs excluded from folding>
.ons <outputs excluded from splitting>
.onf <outputs excluded from folding>

In order to build finite state machines, the following .parameters are proposed:

.iltf <left-top-AND feedback terms>
.ilbf <left-bottom-AND feedback terms>
.irtf <right-top-AND feedback terms>
.irbf <right-bottom-AND feedback terms>
.oltf <left-top-OR feedback terms>
.olbf <left-bottom-OR feedback terms>
.ortf <right-top-OR feedback terms>
.orbf <right-bottom-OR feedback terms>

.ilr <left re-ordered inputs>
.irr <right re-ordered inputs>
.olrf <left re-ordered outputs>
.orrf <right re-ordered outputs>

The .XXXf parameters must occur in pairs, with the .oXXf line first. Input and output terms must occur on the same side (top, bottom) of the PLA. Feedback terms must be given

in ascending order. The re-order .parameters simplify feedback routing.

SEE ALSO

eqntott(1), espresso(1), espresso(5), panda(1), mpanda(1), tpla(1), mpla(1), pop(1), blam(1), pleasure(1), plaid(1)

NAME

pleasure − input file format for pleasure(1)

1. Input File Format

Input file consists of two parts: the folding instruction part and the PLA symbolic description part. Comment lines must begin with a "#".

(a) Folding instructions.

Folding instructions begin with a period. The period should be placed in the first column of the input file. The instructions can be placed anywhere between ".list" and the PLA table. If symbolic names are assigned to inputs and outputs of the PLA, the instruction "label" can be used to let this assignment be known to **pleasure**. Once a label is assigned to any one column, then all the columns have to have labels as well. In other folding instructions, the same symbolic name has to be used.

The following instructions are understood by **pleasure**:

.list
Input file records following ".list" are displayed on the standard output during the read phase.

.cofold [and [=mult]] [or [=mult]]
Column folding requested in the AND and/or OR plane. Multiple folding is specified by the string "=mult".

.rofold [aoa | oao | mult]
Row folding: the trailing character string specifies AND-OR-AND , OR-AND-OR , or multiple folded architecture.

.label in1 in2 ... out1 out2 ...
Each label should be in one-to-one correspodence to each column. Once every column has been labelled, the assigned name should be used for other folding instructions which refer to the columns such as "top", "bottom", "group", "order" and "window". If this label instruction is not given, the user can use an integer number to match the columns.

.first [row | column]
Specifies if rows or columns are to be folded first. If omitted, the fold sequence will be chosen by the program.

.top [c1 , c2 , ... , cn]
The columns in the list are folded on the top only.

.bottom [c1 , c2 , ... , cn]
The columns in the list are folded on the bottom only.

.left [r1 , r2 , ... , rn]
The rows in the list are folded on the left only.

.right [r1 , r2 , ... , rn]
The rows in the list are folded on the right only.

.order left | right
Column folding in the leftmost (rightmost) array is constrained so that each column can be contacted to a connection row and connection rows are in the same sequence as columns in the original PLA. (See references)

.window row | column | contact [n1,l1,u1, ... ,nn,ln,un]
Folding is constrained so that rows , columns , and contacts to connection rows are kept inside a window. The lower and upper bounds for row (column or contact) nj is specified by lj and uj respectively. Note that row and contact (column) windows are compatible only with column (row) folding.

.array left [c1 , c2 , ... , cn]
.array right [c1 , c2 , ... , cn]
The PLA is segmented and specified columns are placed in the left (right) array.

.group [(c1 c2) (c4 c7 c9 c10) (c34 c35)...]
Folding is constrained so that the signals grouped together can be placed contiguously in the output PLA. The constraints are meant to handle the physical positions of the signal carrying the outputs of 1-input and 2-input decoder. This instruction is compatible with simple/multiple row/column folding and it gives better results for expanded PLA input. At present, the output format for *Panda* cannot be processed directly in case of group constraints because of newly created strange buffers.

.side
Folding is constrained so that the signals are connected by the connection rows from the left or right side.

.option prtall | heu1 | heu2 | expand | unmerged
prtall : prints row, column and contact positions on output file.
heu1 : a fast heuristic selection is used based on the degree of the nodes corresponding to the columns or rows to be folded in the node graph of the problem. (See references) Good for large arrays.
heu2 : another heuristic selection based on the number of ascendant and descendant.
If you don't specify anything for heu1 or heu2, or if you specify both of them, the program will be executed twice and the best result will be selected.
expand : expandas the AND plane and impose the group constraint on the signal and its complement. Same as the option key -e.
unmerged : assumes that the AND plane of the PLA matrix already has been expanded.

.area cell-length buffer-length ground-ratio
Above parameters will be set in micron and be used in computation of the physical area estimatoion. By default, cell length is 8u, buffer length is 40u and ground ratio is 1/6.

.end
End of file.

(b) The PLA symbolic description.

PLA's are described as two-level sum-of-products logical implicants. The PLA input format is compatible with the output format of *Espresso*. In the AND-PLANE, a 1 means a connection to the input variable, a 0 means a connection to the complemented input variable, and any other character (except 0, 1, / or) means no connection. Likewise, in the OR-plane, a 1 or 0 means a connection to the output function, and any other character (except 0, 1, / or) means no connection. A 1 means the positive phase and a 0 means the negative phase. A 0 is also regarded as no connection when the option key is set to -p.

Each input file record is up to 500 characters long. A "/" or " means that the implicant continues on the next record. Blanks separate the AND PLANE from the OR PLANE. No other character is allowed between the two planes. No blank is allowed inside a plane.

2. Output File Format

(a) Format for General Usage

This output format is provided for the user so that he can look at the folding results. It consists of the folding lists, the report of the percentage for the optimal area over the original area and the folded PLA personality matrix. The character set used to represent the folded array is reported in the following table:

AND PLANE

(1) Contact to input
(2) Contact to complement
(3) No contact

(1)	(2)	(3)	
1	0	-	Normal contacts, no splits or folds
!	o	_	Split below
;	@	,	Fold to right
:	#	.	Split below and fold to right

OR PLANE

(1) Contact to output
(2) No contact to output

(1)	(2)	
I	~	Normal contacts, no splits or folds
i	=	Split below
|	'	Fold to right
j	"	Split below and fold to right

(b) Format for the Panda Input

This output format is meant to be processed by a "silicon assembler" program, which generates the folded PLA mask layout in CIF 2.0 standard format. *Panda* is the program which is connected to *Pleasure*. *Pleasure* provides a special output format for *Panda*. For more information on input format, please refer to the manual *Panda*.

The output file consists of the contol lines and a personality matrix of the folded PLA. The symblic name 'B' or 'BLANK' means no signal which corresponds to 'X'(No buffer). However, they are not necessarily in one-to-one correspondence. The personality matrix of the folded PLA has the same character set as the general-usage output format except for the contact symbol for the complement signal in the AND plane. There is no difference between contact symbol of the signal and that of its complement any more since the AND plane has been expanded. The column which is connected to '*'(input buffer) is the signal and the next column which is located just right side of the signal is the complemented signal. Therefore the characters for the contact of the complemented signal are not used here. The additional symbol characters are as follows:

Symbols

*	Input buffer
+	Output buffer
X	No buffer
c	Contact within AND or OR plane
> <	Routing lines to contact for multiple folds

---

3. Interactive Mode

Example of a typical **pleasure** session:

```
pleasure
pleasure => .(dot)xxx          (set the folding instruction)
pleasure => read               (read input file)
        filename
pleasure => status             (look at the status of the instructions set)
pleasure => reset xxx          (reset unnecessary folding instructions)
pleasure => run                (run the folding algorithm)
pleasure => show               (see folded PLA)
pleasure => run                (run the folding algorithm)
pleasure => clear              (clear program before restart)
pleasure => read               (read input file)
        filename
pleasure => .option heu1       (switch to heuristic sheme 1)
pleasure => step               (run one step)
pleasure => show
pleasure => step
pleasure => show
pleasure => run
pleasure => show
pleasure => save               (save folded PLA)
        filename
```

pleasure => *quit*

4. References

(a) G. De Micheli and A. Sangiovanni-Vincentelli, "Multiple folding of programmable logic arrays." Pro. Int. Symp. on Circ. and Syst., Newport Beach (CA), pp. 1026-1029, May 1983.

(b) G. De Micheli and A. Sangiovanni-Vincentelli, "PLEASURE: A computer program for simple/multiple constrained/unconstrained folding of programmable logic arrays." Proc. 20th Design Automation Conference, Miami Beach (FL), pp. 530-537, June 1983.

(c) G. De Micheli and A. Sangiovanni-Vincentelli, "Multiple constrained folding of programmable logic arrays: theory and applications." IEEE Trans. on CAD of Int. Circ. and Syst., Vol. CAD-2, No. 3, pp. 167-180, July 1983.

Example #1: Input file ex1

```
.list
.label in1 in2 in3 in4 in5 in6 out1 out2 out3 out4
.cofold and=mult or
.bottom in3
.option prtall
xx1xx0 1000
x1x0xx 0100
1xxxx0 0001
1xxx1x 0100
0xxxxx 0010
xxxxx1 0001
.end
```

Example #2: Input file ex2

```
# TEST PLA #2
.list
.cofold and=mult or=mult
.label in1 in2 in3 in4 in5 in6 out1 out2 out3 out4
.window contact in1 1 1  in2 1 3   in6 4 6   out1 1 1  out3 4 6
.side
.option prtall
xx1xx0 1000
x1x0xx 0100
1xxxx0 0001
1xxx1x 0100
0xxxxx 0010
xxxxx1 0001
.end
```

Remarks : The PLA examples used in here is rather small compared to its buffer size. So the area may be increased after folding because of the additional buffer area. Area reduction will be more effective for the large PLA's after fold.

Example #3: General use output from ex1 (pleasure -bpg <ex01)

```
# pleasure version  4 on   1/15/85 9:00
# FOLDING REQUESTED:
#         SIMPLE   COLUMN FOLDING IN THE OR PLANE
#         MULTIPLE COLUMN FOLDING IN THE AND PLANE
#         COLUMN FOLDING WITH CONSTRAINED CONTACT POSITIONS
#         GENERAL OUTPUT FILE INCLUDES THE DETAILS
# Unfolded PLA: length=  96u; width= 144u; area=   13824u
# sparsity = 84%, original size = ( 12 +   4) *  6
 Column folding:

Ordered column folding list #   1
out1 out3

Ordered column folding list #   2
in6 in5 in2

Ordered column folding list #   3
in1 in4

Ordered column folding list #   4
out4 out2

 Columns from the top
in1 in3 in6 out1 out4

 Rows from the left
   1   3   6   4   5   2

 Contacts on the left plane :
in6 in1 in3 in5 in2 in4

 Contacts on the right plane :
out4 out1 out3 out2 B B
# After folding: length= 136u; width= 144u; area=   19584u (100.00%)
# sparsity = 67%, new size = (  6 +   2) *  6
# PLA cell= 8u x 8u; buffer length=40u; ground ratio= 1 / 6
# Area computation based on the minimum enclosing rectangle of
# the folded PLA.
# This output based on the heuristic scheme2
# In the other scheme, length= 136 width= 160 area=   21760(157.41%)

 PERSONALITY MATRIX
-10  I~
1-0  =I
--!  ~i
1-!  ~I
o--  I~
0-1  ~I
 #elapsed run time:    7.2 seconds
```

Example #4: General use output from ex2 (pleasure -bpg <ex02)

```
# pleasure version  4 on   1/15/85 9:00
# FOLDING REQUESTED:
#         MULTIPLE COLUMN FOLDING IN THE AND PLANE
#         MULTIPLE COLUMN FOLDING IN THE OR PLANE
#         COLUMN FOLDING WITH CONSTRAINED CONTACT POSITIONS
#         GENERAL OUTPUT FILE INCLUDES THE DETAILS
#         I/O BUFFERS ARE ALL SET FROM SIDES
# Unfolded PLA: length=  96u; width= 144u; area=   13824u
# sparsity = 84%, original size = ( 12 +   4) *  6
 Column folding:

Ordered column folding list #   1
out1 out4 out2 out3

Ordered column folding list #   2
in3 in2

Ordered column folding list #   3
in6 in4

 Columns from the top
in1 in3 in5 in6 out1

 Rows from the left
   1   3   6   4   2   5

 Contacts on the left plane :
in1 in3 in2 in6 in4 in5

 Contacts on the right plane :
out1 out4 B out2 B out3
# After folding: length=  56u; width= 152u; area=    8512u ( 61.57%)
# sparsity = 71%, new size = (  8 +   1) *  6
# PLA cell= 8u x 8u; buffer length=40u; ground ratio= 1 / 6
# Area computation based on the minimum enclosing rectangle of
# the folded PLA.
# This output based on the heuristic scheme 1
# In the other scheme, length=  56 width= 152 area=    8512( 61.57%)

 PERSONALITY MATRIX
-1-0  i
1_-0  I
---1  i
1-1_  I
-1-0  i
0---  I
 #elapsed run time:    7.7 seconds
```

Example #5: Panda input generated from ex1(pleasure -bp < ex01)

```
# pleasure version  4 on   1/15/85 9:00
# FOLDING REQUESTED:
#         SIMPLE   COLUMN FOLDING IN THE OR PLANE
#         MULTIPLE COLUMN FOLDING IN THE AND PLANE
#         COLUMN FOLDING WITH CONSTRAINED CONTACT POSITIONS
#         GENERAL OUTPUT FILE INCLUDES THE DETAILS
#         OUTPUT FORMAT IS SET FOR PANDA
# Unfolded PLA: length=  96u; width= 144u; area=   13824u
# sparsity = 84%, original size = ( 12 +   4) *  6
# After folding: length= 136u; width= 144u; area=   19584u (141.67%)
# sparsity = 67%, new size = (  6 +   2) *  6
# PLA cell=  8u x 8u; buffer length=40u; ground ratio= 1 / 6
# Area computation based on the minimum enclosing rectangle of
# the folded/unfolded PLA.
# This output based on the heuristic scheme2
# In the other scheme,  length= 136 width= 160 area=   21760(157.41%)
.top in1 in6 out1 out4
.bottom in4 in3 in2 out3 out2
.left in5
.and    3   2
.row      6
 * X *   ++
X--1--1  I~X
X1----1  =IX
X----!_  ~iX
*>>>>c     X
X>>>>>c    X
X1---!_  ~IX
X_!----  I~X
X-1--1-  ~IX
 * * *   ++
.end
 #elapsed run time:    6.6 seconds
```

Example #6: Panda input generated from ex2(pleasure -bp <ex02)

```
# pleasure version  4 on   1/15/85 9:00
# FOLDING REQUESTED:
#         MULTIPLE COLUMN FOLDING IN THE AND PLANE
#         MULTIPLE COLUMN FOLDING IN THE OR PLANE
#         COLUMN FOLDING WITH CONSTRAINED CONTACT POSITIONS
#         GENERAL OUTPUT FILE INCLUDES THE DETAILS
#         OUTPUT FORMAT IS SET FOR PANDA
#         I/O BUFFERS ARE ALL SET FROM SIDES
# Unfolded PLA: length=  96u; width= 144u; area=   13824u
# sparsity = 84%, original size = ( 12 +   4) *  6
# After folding: length=  56u; width= 152u; area=    8512u ( 61.57%)
# sparsity = 71%, new size = (  8 +   1) *  6
# PLA cell=  8u x 8u; buffer length=40u; ground ratio= 1 / 6
# Area computation based on the minimum enclosing rectangle of
# the folded/unfolded PLA.
# This output based on the heuristic scheme 1
# In the other scheme,  length=  56 width= 152 area=    8512( 61.57%)
.left in1 in3 in2 in6 in4 in5
.right out1 out4 out2 out3
.and    4   1
.row      6
 X X X X  X
*c         c+
X>c         X
X--1----1  iX
*>>c       c+
X>>>c       X
X1-__---1  IX
*>>c        X
X>>>c       X
X------1-  iX
*>>>>>>c   c+
X>>>>>>>c   X
X1---1-__  IX
*>>>>>>c    X
X>>>>>>>c   X
X--1----1  iX
*>>>>c     c+
X>>>>>c     X
X-1-----  IX
 X X X X   X
.end
 #elapsed run time:    7.5 seconds

 Remarks : For multiple folding, the implementation is assumed
           in double metal layer.
```

NAME

sim – format of .sim files read by esim, crystal, etc.

DESCRIPTION

The simulation tools *crystal*(1) and *esim*(1) accept a circuit description in **.sim** format. There is a single **.sim** file for the entire circuit, unlike Magic's *ext*(5) format in which there is a **.ext** file for every cell in a hierarchical design.

A **.sim** file consists of a series of lines, each of which begins with a key letter. The key letter beginning a line determines how the remainder of the line is interpreted. The following are the list of key letters understood.

| units: *s* **tech:** *tech*

If present, this must be the first line in the **.sim** file. It identifies the technology of this circuit as *tech* and gives a scale factor for units of linear dimension as *s*. All linear dimensions appearing in the **.sim** file are multiplied by *s* to give centimicrons.

type g s d l w x y **g=***gattrs* **s=***sattrs* **d=***dattrs*

Defines a transistor of type *type*. Currently, *type* may be **e** or **d** for NMOS, or **p** or **n** for CMOS. The name of the node to which the gate, source, and drain of the transistor are connected are given by *g*, *s*, and *d* respectively. The length and width of the transistor are *l* and *w*. The next two tokens, *x* and *y*, are optional. If present, they give the location of a point inside the gate region of the transistor. The last three tokens are the attribute lists for the transistor gate, source, and drain. If no attributes are present for a particular terminal, the corresponding attribute list may be absent (i.e, there may be no **g=** field at all). The attribute lists *gattrs*, etc. are comma-separated lists of labels. The label names should not include any spaces, although some tools can accept label names with spaces if they are enclosed in double quotes.

C *n1 n2 cap*

Defines a capacitor between nodes *n1* and *n2*. The value of the capacitor is *cap* femtofarads. **NOTE:** since many analysis tools compute transistor gate capacitance themselves from the transistor's area and perimeter, the capacitance between a node and substrate (GND!) normally does not include the capacitance from transistor gates connected to that node. If the **.sim** file was produced by *ext2sim*(1), check the technology file that was used to produce the original **.ext** files to see whether transistor gate capacitance is included or excluded; see "Magic Maintainer's Manual #2: The Technology File" for details.

R *node res*

Defines the lumped resistance of node *node* to be *res* ohms. This construct is only interpreted by a few programs.

N *node darea dperim parea pperim marea mperim*

As an alternative to computed capacitances, some tools expect the total perimeter and area of the polysilicon, diffusion, and metal in each node to be reported in the **.sim** file. The **N** construct associates diffusion area *darea* (in square centimicrons) and diffusion perimeter *dperim* (in centimicrons) with node *node*, polysilicon area *parea* and perimeter *pperim*, and metal area *marea* and perimeter *mperim*. *This construct is technology dependent and obsolete.* ®.TP **A** *node attr* Associates attribute *attr* for node *node*. The string *attr* should contain no blanks.

= *node1 node2*

Each node in a **.sim** file is named implicitly by having it appear in a transistor

definition. All node names appearing in a **.sim** file are assumed to be distinct. Some tools, such as *esim*(1), recognize aliases for node names. The = construct allows the name *node2* to be defined as an alias for the name *node1*. Aliases defined by means of this construct may not appear anywhere else in the **.sim** file.

SEE ALSO

crystal(1), esim(1), ext2sim(1), sim2spice(1), ext(5)

NAME

prleak − aid for debugging programs using malloc/free

SYNOPSIS

prleak [**−a**] [**−d**] [**−l**] [*objfile* [*tracefile*]]

DESCRIPTION

Prleak is a tool for use in debugging programs that make use of Magic's versions of *malloc* and *free*. It examines the trace file produced by special versions of *malloc* and *free* produced when they are compiled with the **−DMALLOCTRACE** flag. The output of prleak is the average allocation size, a list of 'leaky' allocations (blocks still allocated at program exit) if **−l** is specified, a list of duplicate frees (blocks that the program attempted to free after they had already been deallocated) if **−d** is specified, and a list of all calls to malloc and free if **−a** is specified. If no switches are given, the default action is as though **−l** and **−d** were in effect.

For each entry output, both the address of the allocated block and a stack backtrace at the time of the call to *malloc* or *free* are printed. *Prleak* attempts to use the namelist from *objfile* (**a.out** if no file is given) to produce a symbolic backtrace. If no namelist can be found, the backtrace is printed in hex. If *tracefile* is specified, the malloc trace is read from it; otherwise, it is read from the file **malloc.out** in the current directory.

An example output might be as follows:

```
Average allocation size = 12 bytes

------
Leaks:
------

0x11540     [11 bytes]
      at _foo+0x14
      called from ~main+026

0x11556     [14 bytes]
      at _bar+0x50
      called from _foo+0x36
      called from ~main+0x26

--------- ------
Duplicate frees:
--------- ------

0x11556
      at _bar+0x40
      called from _foo+0x36
      called from ~main+0x26
```

FILES

malloc.out

SEE ALSO

ACM SIGPLAN Notices, Vol 17, No 5 (May 1982), the article by Barach and Taenzer.

AUTHOR

Walter Scott

BUGS

Local symbols (beginning with "~") in the backtrace output should be tagged with the source file to which they refer.

Changes to Magic in Version 4

Walter Scott
Gordon Hamachi
Robert Mayo
John Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

This document corresponds to Magic version 4.

The 1986 VLSI Tools tape contains Version 4 of the Magic graphical layout editor. This new version has changed in a number of ways from Version 3, the previous release that was distributed with the 1985 VLSI Tools tape. This document lists the most important changes that have been made to Magic since that release. Only some of these are user visible; others are internal changes that should only affect you if you have made any local modifications to Magic.

New features

1. The user interface has undergone two massive changes. The first change is the addition of a selection-style interface that replaces yank and stuff. Read the new Tutorial #2 for a description of how it works. Secondly, a new wiring-style interface has been implemented. Read the new Tutorial #3 for a description of this.
2. There is now a **:plow** command for stretching and compacting cells, along with a **:straighten** command to eliminate unnecessary jogs. See Tutorial #3 for details.
3. Calma GDS-II stream format can now be both written and read, in much the same way as CIF.
4. A new command has been added to do plotting directly from Magic without having to generate CIF and use CIFplot. See the description of **:plot** in *magic*(1).
5. We now distribute the official MOSIS SCMOS technology for Magic. This was provided by Shih-Lien Lu of USC/ISI.

General

6. Label layer selection has been generalized. For a label to be attached to a layer, it used to be necessary for the layer to cover the entire area of the label. Now it's only necessary for a layer to be a "component" of material that covers the entire area of the label. This means, for example, that a label attached to metal can span an area containing both metal and poly-metal-contact and diffusion-metal contact.
7. The **:grid** command is more general now. It permits different spacings in the x- and y-directions, and also allows you to specify an explicit origin.
8. The tech file now supports more general layer names, such as "$\sim x$" to get all layers but x. See Maintainer's Manual #2 for details.

CIF and Calma

9. In the CIF geometrical commands, there is now an operator **and-not** for doing inversions. Also, several bugs in CIF output (having to do with arrays and bloating tiles) have been fixed.
10. CIF and Calma output have been improved to generate 30% smaller files by merging rectangles wherever possible.
11. There have been lots of changes to the CIF reading and writing code. The CIF reader didn't used to handle wires correctly; it should work now. Also, if there are non-Manhattan wires or polygons in a CIF input file, Magic approximates them with a crude staircase. In CIF writing, there used to be several bugs, particularly having to do with cell interactions. There have been many fixes made.

Windows, redisplay, and graphics

12. Sun-2/160's and the Sun 2.0 operating system are now supported.
13. Minimal support for external window packages is now available. For example, on the Sun we allow SunTools to manage our windows. See the code in the windows module for details.
14. The graphics display drivers have been changed so that they can run on more sorts of systems. As a consequence, any old drivers that you may have written need to be modified in order to work with the new scheme. To port an old driver, the procedures related to the "TrackingRect" stuff need to be deleted, and the command input scheme changed. In the new scheme, the driver supplies a procedure for each input device (normally one for stdin, and one for the graphics device). These procedures collect input (such as button pushes and keystrokes) from the device and put them into a queue for later processing in the textio module. See the AED driver and grLock.c for details.

    In addition, all of the Magic modules that do redisplay have been changed so that they lock the window before drawing, and then unlock it when they are done.

15. Transformations between root-cell and screen coordinates have been reimplemented to avoid numerical problems with the old scheme. Instead of calling GeoTransToScreen, call WindSurfaceToScreen. Also, the calling sequences for WindScreenToSurface and WindPointToSurface have been changed slightly. There should be no user-visible changes from this, but if you've been writing software that uses the window package then you'll have to make changes.
16. The names of display styles files, color maps, and glyph files have changed to allow a few files to be shared widely instead of having many files that are near-duplicates. The numbering of styles in the **.dstyle** files has changed. See the tutorials and man pages for details.

Circuit extraction

17. A number of bugs have been fixed in the circuit extractor. The mechanism for reporting warnings and fatal errors has been cleaned up, so you can now select the kinds of errors you want to see. Multiple but unconnected nets with the same name in a single cell are now a warning error instead of a fatal one.
18. The circuit extractor has been changed to output perimeter and area information for each node, along with capacitance and approximate resistance. See *ext*(5) for details of the new **.ext** format. *Ext2sim* has also been changed to accept the new information and compute better adjustments to resistance due to overlap. You will have to re-extract all your old cells to use the new ext2sim, however.
19. The extractor also now gives you more flexibility in what you choose to extract. You can enable or disable the extraction of resistance, substrate capacitance, or coupling capacitance. You can also enable or disable hierarchical adjustments to capacitance, perimeter, and area that result from overlap.
20. The extractor's accuracy has been improved. It now knows about shielding when extracting coupling capacitance (e.g, metal1 shields metal2 from poly if it lies between the two). It compensates for coupling capacitance hierarchically (e.g, a poly bus from one cell overlapping a metal bus from another cell).
21. There are now several extraction "styles", allowing you to specify different combinations of extraction parameters for different processes. See Tutorial #8 for details of this and the previous changes to the extractor.

Netlists and routing

22. The router channel decomposition has been modified to treat array elements at the top level as single cells. This allows routing to all 4 sides of each of the arrayed cells.

23. The router stem generator has been improved to make fewer jogs. It also allows terminals to be inside cells: when this occurs, the stem generator routes straight across the cell to the closest channel.
24. The netlist editor now has a **:flush** command, and prints out the names of terminals as they are selected and deselected.
25. The router has been generalized slightly, so that it can generate contacts with additional surrounds in the wiring layers. For example, you can ask that each contact placed by the router be surrounded by an additional 1 unit of metal around the contact.
26. Previously the router assumed that labels with exactly the same path name were to be wired together. Magic now treats identically-labeled terminals as electrically-equivalent, and only connects to the closest of these terminals. Further, after making the initial connection Magic may take advantage of the connectivity between equivalent terminals and feed the net through a cell and out any of the other electrically-equivalent terminals to connect to other terminals in the net.
27. The **:route** command now takes a number of options. See Tutorial #7 and *magic*(1) for details.
28. There's a new command called **:corner** that makes L-shaped bends around corners. It is useful for routing large busses.

Design-rule checking

29. DRC redisplay has been improved in two ways. First, it has been made much smarter. Whereas before it used to do way too much redisplay when there were several error areas on the screen, now it is much smarter, and only redisplays where error information has changed. But, if even this is too much redisplay, you can type **:see no errors**, in which case DRC continues to run but you aren't bothered with any redisplay at all. When you get to the point that you'd like to see errors, type **:see errors**. Also, **:drc why** has been speeded up quite a bit.

Magic Tutorial #1: Getting Started

John Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

This tutorial corresponds to Magic version 4.

1. What is Magic?

Magic is an interactive system for creating and modifying VLSI circuit layouts. With Magic, you use a color graphics display and a mouse or graphics tablet to design basic cells and to combine them hierarchically into large structures. Magic is different from other layout editors you may have used. The most important difference is that Magic is more than just a color painting tool: it understands quite a bit about the nature of circuits and uses this information to provide you with additional operations. For example, Magic has built-in knowledge of layout rules; as you are editing, it continuously checks for rule violations. Magic also knows about connectivity and transistors, and contains a built-in hierarchical circuit extractor. Magic also has a *plow* operation that you can use to stretch or compact cells. Lastly, Magic has routing tools that you can use to make the global interconnections in your circuits.

Magic is based on the Mead-Conway style of design. This means that it uses simplified design rules and circuit structures. The simplifications make it easier for you to design circuits and permit Magic to provide powerful assistance that would not be possible otherwise. However, they result in slightly less dense circuits than you could get with more complex rules and structures. For example, Magic permits only *Manhattan* designs (those whose edges are vertical or horizontal). Circuit designers tell us that our conservative design rules cost 5-10% in density. We think that the density sacrifice is compensated for by reduced design time.

Magic Tutorial #1: Getting Started
Magic Tutorial #2: Basic Painting and Selection
Magic Tutorial #3: Advanced Painting (Wiring and Plowing)
Magic Tutorial #4: Cell Hierarchies
Magic Tutorial #5: Multiple Windows
Magic Tutorial #6: Design-Rule Checking
Magic Tutorial #7: Netlists and Routing
Magic Tutorial #8: Circuit Extraction
Magic Tutorial #9: Format Conversion for CIF and Calma
Magic Maintainer's Manual #1: Hints for System Maintainers
Magic Maintainer's Manual #2: The Technology File
Magic Maintainer's Manual #3: Display Styles, Color Maps, and Glyphs
Magic Technology Manual #1: NMOS
Magic Technology Manual #2: SCMOS

Table I. The Magic tutorials, maintenance manuals, and technology manuals.

2. How to Get Help and Report Problems

There are several ways you can get help about Magic. If you are trying to learn about the system, you should start off with the Magic tutorials, of which this is the first. Each tutorial introduces a particular set of facilities in Magic. There is also a set of manuals intended for system maintainers. These describe things like how to create new technologies. Finally, there is a set of technology manuals. Each one of the technology manuals describes the features peculiar to a particular technology, such as layer names and design rules. Table I lists all of the Magic manuals. The tutorials are designed to be read while you are running Magic, so that you can try out the new commands as they are explained. You needn't read all the tutorials at once; each tutorial lists the other tutorials that you should read first.

The tutorials are not necessarily complete. Each one is designed to introduce a set of facilities, but it doesn't necessarily cover every possibility. The ultimate authority on how Magic works is the reference manual, which is a standard Unix *man* page. The *man* page gives concise and complete descriptions of all the Magic commands. Once you have a general idea how a command works, the *man* page is probably easier to consult than the tutorial. However, the *man* page may not make much sense until after you've read the tutorial.

A third way of getting help is available on-line through Magic itself. The **:help** command will print out one line for each Magic command, giving the command's syntax and an extremely brief description of the command. This facility is useful if you've forgotten the name or exact syntax of a command. After each screenful of help information, **:help** stops and prints "--More--". If you type a space, the next screenful of data will be output, and if you type **q** the rest of the output will be skipped. If you're interested in information about a particular subject, you can type

:help *subject*

This command will print out each command description that contains the *subject* string.

If you have a question or problem that can't be answered with any of the above approaches, don't hesitate to contact a member of the Magic team. We are: Gordon Hamachi, Bob Mayo, John Ousterhout, and Walter Scott. Magic is a relatively young program, so you will probably run across bugs and unpleasant features. When this happens, please let us know by sending mail to magic@ucbkim (or magic@kim.berkeley.edu in the new domain-based internet naming system). This will reach everyone in the group and will also log the message in a system file so we can't forget about the problem. Please tell us about problems with the manuals too. When you report a bug, *please* be specific. We get quite a few messages of the form "Magic dumped core while I was working today. Can you please fix it?" Obviously, these messages aren't much help to us. When you report a bug, describe the exact sequence of events that led to the problem, what you expected to happen, and what actually happened. The best thing of all is to find a small example that reproduces the problem and send us the relevant (small!) files so we can make it happen here.

3. Hardware Configuration

Magic runs in several configurations. One configuration uses VAX processors: each workstation consists of a standard video terminal, called the *text display*, and a color display. You use the keyboard on the text display to type in commands, and Magic uses its screen to log the commands and their results. The color display is used to display one or more portions of the circuit you are designing. You will use a graphics tablet or mouse to point to things on the color display and to invoke some commands. If there is a keyboard attached to the color display (as, for example, with AED512 displays) it is not used except to reset the display. The current version of Magic supports the AED family of displays. Most of the displays are now available with special ROMs in them that provide extra Magic support (talk to your local AED sales rep to make sure you get the UCB ROMs). More displays are being added, so check the Unix man page for the most up-to-date information.

The second configuration supported by Magic is a Sun model 120 with the SunColor option. Magic uses the black-and-white display to log commands (we'll refer to it as the "text display"), and the color screen to display layouts. The optical mouse is used to point on the color screen. If you run with the Sun configuration, we recommend that you get as much memory as you possibly can (e.g. 4 or 8 Mbytes).

A third configuration supported by Magic is the Sun model 160, which has an integrated color display. In this configuration, both the command log and the layout windows appear on the same screen. In the manuals, we'll still refer to the command log as the "text display" even though it's really just a window. As with the Sun 120s, we recommend getting a lot of physical memory.

4. Running Magic

From this point on, you should be sitting at a Magic workstation so you can experiment with the program as you read the manuals. Starting up Magic is usually pretty simple. Just log in at the text display. If you're using a Sun workstation you should also run Suntools and open up a shell window. Then type the shell command

magic tut1

Tut1 is the name of a library cell that you will play with in this tutorial. At this point, several colored rectangles should appear on the color display along with a white box and a cursor (if you're running on a Sun 120 workstation, the cursor will still be over a window on the black-and-white screen; slide it off the right edge of the black-and-white screen to get it onto the color screen). A message will be printed on the text display to tell you that **tut1** isn't writable (it's in a read-only library), and a "**>**" prompt should appear. If this has happened, then you can skip the rest of this section (except for the note below) and go directly to Section 5.

Note: in the tutorials, when you see things printed in boldface, for example, **magic tut1** from above, they refer to things you type exactly, such as command names and file names. These are usually case sensitive (**A** is different from **a**). When you see things printed in italics, they refer to classes of things you might type. Arguments in square brackets are optional. For example, a more complete description of the shell command for Magic is

magic [*file*]

You could type any file name for *file*, and Magic would start editing that file. It turns out that **tut1** is just a file in Magic's cell library. If you didn't type a file name, Magic would load a new blank cell.

If things didn't happen as they should have when you tried to run Magic, any of several things could be wrong. If a message of the form "magic: Command not found" appears on your screen it is because the shell couldn't find the Magic program. The most stable version of Magic is the directory ~cad/bin, and the newest public version is in ~cad/new. You should make sure that both these directories are in your shell path. Normally, ~cad/new should appear before ~cad/bin. If this sounds like gibberish, find a Unix hacker and have him or her explain to you about paths. If worst comes to worst, you can invoke Magic by typing its full name:

~cad/bin/magic tut1

Another possible problem is that Magic might not know which color display to use. This happens when you run Magic from a video terminal that isn't hardwired to a VAX. On VAXes, Magic reads the file **~cad/lib/displays** to find out which color display to use for each hardwired terminal, but if you connect via a patchboard, port selector, or Ethernet then Magic might not know what to use. In this case, you must tell Magic which port to use for the color display using the **-g** (graphics port) switch:

magic -g *port* **tut1**

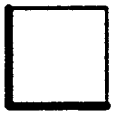
Port is the device name for a port, e.g. **/dev/ttyj1**.

If you were running with an AED display and got a message of the form "Unable to open mouse..." when you tried to run Magic, it is because there was a login process on the color display, and it prevented Magic from opening the port for reading. In this case, you must login a special job called "sleeper" on the color display. Sleeper will reset the port so that Magic can use it. Type breaks on the color display if necessary to get a login prompt, then login a user named "sleeper". No password should be necessary. After this, Magic should work correctly. Most of the displays at Berkeley are configured without login processes, so sleeper is not usually needed. By the way, there are special versions of sleeper called rsleeper and fsleeper, which can be used to run Magic remotely over the Ethernet. Consult the man pages or your local wizard for details.

5. The Box and the Cursor

Two things, called the *box* and the *cursor*, are used to select things on the color display. As you move the mouse, the cursor moves on the screen. The cursor starts out with a crosshair shape, but you'll see later that its shape changes as you work to provide feedback about what you're doing. The left and right mouse buttons are used to position the box. If you press the left mouse button and then release it, the box will move so that its lower left corner is at the cursor position. If you press and release the right mouse button, the upper right corner of the box will move to the cursor position, but the lower left corner will not change. These two buttons are enough to position the box anywhere on the screen. Try using the buttons to place the box around each of the colored rectangles on the screen.

Sometimes it is convenient to move the box by a corner other than the lower left. To do this, press the left mouse button and *hold it down*. The cursor shape changes to show you that you are moving the box by its lower left corner:



(some displays don't support programmable cursors, so the cursor will never change shape). While holding the button down, move the cursor near the lower right corner of the box, and now click the right mouse button (i.e. press and release it, while still holding down the left button). The cursor's shape will change to indicate that you are now moving the box by its lower right corner. Move the cursor to a different place on the screen and release the left button. The box should move so that its lower right corner is at the cursor position. Try using this feature to move the box so that it is almost entirely off-screen to the left. Try moving the box by each of its corners.

You can also reshape the box by corners other than the upper right. To do this, press the right mouse button and hold it down. The cursor shape shows you

that you are reshaping the box by its upper right corner:

Now move the cursor near some other corner of the box and click the left button, all the while holding the right button down. The cursor shape will change to show you that now you are reshaping the box by a different corner. When you release the right button, the box will reshape so that the selected corner is at the cursor position but the diagonally opposite corner is unchanged. Try reshaping the box by each of its corners.

6. Invoking Commands

Commands can be invoked in Magic in three ways: by pressing buttons on the puck or mouse; by typing single keystrokes on the text keyboard (these are called *macros*); or by typing longer commands on the text keyboard (these are called *long commands*). Many of the commands use the box and cursor to help guide the command.

To see how commands can be invoked from the buttons, first position the box over a small blank area in the middle of the screen. Then move the cursor over the red rectangle and press the middle mouse button (if you're using a four-button puck, use the bottom button wherever the Magic documentation says "middle"). At this point, the area of the box should get painted red. Now move the cursor over empty space and press the middle button again. The red paint should go away. Note how this command uses both the cursor and box locations to control what happens.

As an example of a macro, type the **g** key on the text keyboard. A grid will appear on the color display, along with a small black box marking the origin of the cell. If you type **g** again, the grid will go away. You may have noticed earlier that the box corners didn't move to the exact cursor position: you can see now that the box is forced to fall on grid points.

Long commands are invoked by typing a colon (":") or semi-colon (";"). After you type the colon or semi-colon, the ">" prompt on the text screen will be replaced by a ":" prompt. This indicates that Magic is waiting for a long command. At this point you should type a line of text, followed by a return. When the long command has been processed, the ">" prompt reappears on the text display. Try typing semi-colon followed by return to see how this works. Occasionally a "]" (right bracket) prompt will appear. This means that the design-rule checker is reverifying part of your design. For now you can just ignore this and treat "]" like ">".

Each long command consists of the name of the command followed by arguments, if any are needed by that command. The command name can be abbreviated, just as long as you type enough characters to distinguish it from all other long commands. For example, **:h** and **:he** may be used as abbreviations for **:help**. On the other hand, **:u** may not be used as an abbreviation for **:undo**

because there is another command, **:upsidedown**, that has the same abbreviation. Try typing **:u**.

As an example of a long command, put the box over empty space on the color display, then invoke the long command

:paint red

The box should fill with the red color, just as if you had used the middle mouse button to paint it. Everything you can do in Magic can be invoked with a long command. It turns out that the macros are just conveniences that are expanded into long commands and executed. For example, the long command equivalent to the **g** macro is

:grid

Magic permits you to define new macros if you wish. Once you've become familiar with Magic you'll almost certainly want to add your own macros so that you can invoke quickly the commands you use most frequently. See the *magic(1)* man page under the command **:macro**.

One more long command is of immediate use to you. It is

:quit

Invoke this command. Note that before exiting, Magic will give you one last chance to save the information that you've modified. Type **y** to exit without saving anything.

Magic Tutorial #2: Basic Painting and Selection

John Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

This tutorial corresponds to Magic version 4.

Tutorials to read first:

Magic Tutorial #1: Getting Started

Commands introduced in this tutorial:

:box, :clockwise, :copy, :erase, :findbox :grid, :label, :layers, :macro, :move, :paint, :redo, :save, :select, :sideways, :undo, :upsidedown, :view, :what, :writeall, :zoom

Macros introduced in this tutorial:

a, A, c, d, ^D, e, E, g, G, q, Q, r, R, s, S, t, T, u, U, v, w, W, z, Z, 4

1. Cells and Paint

In Magic, a circuit layout is a hierarchical collection of *cells*. Each cell contains three things: colored shapes, called *paint*, that define the circuit's structure; textual *labels* attached to the paint; and *subcells*, which are instances of other cells. The paint is what determines the eventual function of the VLSI circuit. Labels and subcells are a convenience for you in managing the layout and provide a way of communicating information between various synthesis and analysis tools. This tutorial explains how to create and edit paint and labels in simple single-cell designs, using the basic painting commands. "Magic Tutorial #3: Advanced Painting (Wiring and Plowing)" describes some more advanced features for manipulating paint. For information on how to build up cell hierarchies, see "Magic Tutorial #4: Cell Hierarchies".

2. Painting and Erasing

Enter Magic to edit the cell **tut2a** (type **magic tut2a** to the Unix shell; follow the directions in "Tutorial #1: Getting Started" if you have any problems with this). The **tut2a** cell is a sort of palette: it shows a splotch of each of several paint layers and gives the names that Magic uses for the layers.

The two basic layout operations are painting and erasing. They can be invoked using the **:paint** and **:erase** long commands, or using the buttons. The easiest way to paint and erase is with the mouse buttons. To paint, position the box over the area you'd like to paint, then move the cursor over a color and click the middle mouse button. To erase everything in an area, place the box over the area, move the cursor over a blank spot, and click the middle mouse button. Try painting and erasing various colors. If the screen gets totally messed up, you can always exit Magic and restart it. While you're painting, white dots may occasionally appear and disappear. These are design rule violations detected by Magic, and will be explained in "Magic Tutorial #6: Design Rule Checking". You can ignore them for now.

It's completely legal to paint one layer on top of another. When this happens, one of three things may occur. In some cases, the layers are independent, so what you'll see is a combination of the two, as if each were a transparent colored foil. This happens, for example, if you paint metal1 (blue) on top of polysilicon (red). In other cases, when you paint one layer on top of another you'll get something different from either of the two original layers. For example, painting poly on top of ndiff produces ntransistor (try this). In still other cases the new layer replaces the old one: this happens, for example, if you paint a pcontact on top of ntransistor. Try painting different layers on top of each other to see what happens. The meaning of the various layers is discussed in more detail in Section 11 below.

There is a second way of erasing paint that allows you to erase some layers without affecting others. This is the macro **^D** (control-D, for "**D**elete paint"). To use it, position the box over the area to be erased, then move the crosshair over a splotch of paint containing the layer(s) you'd like to erase. Type **^D** key on the text keyboard: the colors underneath the cursor will be erased from the area underneath the box, but no other layers will be affected. Experiment around with the **^D** macro to try different combinations of paints and erases. If the cursor is over empty space then the **^D** macro is equivalent to the middle mouse button: it erases everything.

You can also paint and erase using the long commands

:paint *layers*
:erase *layers*

In each of these commands *layers* is one or more layer names separated by commas (you can also use spaces for separators, but only if you enclose the entire list in double-quotes). Any layer can be abbreviated as long as the abbreviation is unambiguous. For example, **:paint poly,metal1** will paint the polysilicon and metal1 layers. The macro **^D** is predefined by Magic to be **:erase $** (**$** is a pseudo-layer that means "all layers underneath the cursor").

3. Undo

There are probably going to be times when you'll do things that you'll later wish you hadn't. Fortunately, Magic has an undo facility that you can use to restore things after you've made mistakes. The command

:undo

(or, alternatively, the macro **u**) will undo the effects of the last command you invoked. If you made a mistake several commands back, you can type **:undo** several times to undo successive commands. However, there is a limit to all this: Magic only remembers how to undo the last ten or so commands. If you undo something and then decide you wanted it after all, you can undo the undo with the command

:redo

(**U** is a macro for this command). Try making a few paints and erases, then use **:undo** and **:redo** to work backwards and forwards through the changes you made.

4. The Selection

Once you have painted a piece of layout, there are several commands you can invoke to modify the layout. Many of them are based on the *selection*: you select one or more pieces of the design, and then perform operations such as copying, deletion, and rotation on the selected things. To see how the selection works, load cell **tut2b**. You can do this by typing **:load tut2b** if you're still in Magic, or by starting up Magic with the shell command **magic tut2b**.

The first thing to do is to learn how to select. Move the cursor over the upper portion of the L-shaped blue area in **tut2b**, and type **s**, which is a macro for **:select**. The box will jump over to cover the vertical part of the "L". This operation selected a chunk of material. Move the box away from the chunk, and you'll see that a thin white outline is left around the chunk to show that it's selected. Now move the cursor over the vertical red bar on the right of the cell and type **s**. The box will move over that bar, and the selection highlighting will disappear from the blue area.

If you type **s** several times without moving the cursor, each command selects a slightly larger piece of material. Move the cursor back over the top of the blue "L", and type **s** three times without moving the cursor. The first **s** selects a chunk (a rectangular region all of the same type of material). The second **s** selects a *region* (all of the blue material in the region underneath the cursor, rectangular or not). The third **s** selects a *net* (all of the material that is electrically connected to the original chunk; this includes the blue metal, the red polysilicon, and the contact that connects them).

The macro **S** (short for **:select more**) is just like **s** except that it adds on to the selection, rather than replacing it. Move the cursor over the vertical red bar on the right and type **S** to see how this works. You can also type **S** multiple times to add regions and nets to the selection.

If you accidentally type **s** or **S** when the cursor is over space, you'll select a cell (**tut2b** in this case). You can just undo this for now. Cell selection will be discussed in "Magic Tutorial #4: Cell Hierarchies".

You can also select material by area: place the box around the material you'd like to select and type **a** (short for **:select area**). This will select all of the material underneath the box. You can use the macro **A** to add material to the selection by area, and you can use the long command

:select [**more**] **area** *layers*

to select only material on certain layers. Place the box around everything in **tut2b** and type **:select area metal1** followed by **:select more area poly**.

If you'd like to clear out the selection without modifying any of the selected material, you can use the command

:select clear

or type the macro **C**. For a synopsis of all the options to the **:select** command, type

:select help

5. Operations on the Selection

Once you've made a selection, there are a number of operations you can perform on it:

:delete
:move [*direction* [*distance*]]
:stretch [*direction* [*distance*]]
:copy
:upsidedown
:sideways
:clockwise [*degrees*]

The **:delete** command deletes everything that's selected. Watch out: **:delete** is different from **:erase**, which erases paint from the area underneath the box. Select the red bar on the right in **tut2b** and type **d**, which is a macro for **:delete**. Undo the deletion with the **u** macro.

The **:move** command picks up both the box and the selection and moves them so that the lower-left corner of the box is at the cursor location. Select the red bar on the right and move it so that it falls on top of the vertical part of the blue "L". You can use **t** ("translate") as a macro for **:move**. Practice moving various things around the screen. The command **:copy** and its macro **c** are just like **:move** except that a copy of the selection is left behind at the original position.

There is also a longer form of the **:move** command that you can use to move the selection a precise amount. For example, **:move up 10** will move the selection (and the box) up 10 units. The *direction* argument can be any direction

like **left**, **south**, **down**, etc. See the Magic manual page for a complete list of the legal directions. The macros **q**, **w**, **e**, and **r** are defined to move the selection left, down, up, and right (respectively) by one unit.

The **:stretch** command is similar to **:move** except that it stretches and erases as it moves. **:stretch** does not operate diagonally, so if you use the cursor to indicate where to stretch to, Magic will either stretch up, down, left, or right, whichever is closest. The **:stretch** command moves the selection and also does two additional things. First, for each piece of paint that moves, **:stretch** will erase that layer from the region that the paint passes through as it moves, in order to clear material out of its way. Second, if the back edge of a piece of selected paint touches non-selected material, one of the two pieces of paint is stretched to maintain the connection. The macros **Q**, **W**, **E**, and **R** just like the macros **q**, etc. described above for **:move**. The macro **T** is predefined to **:stretch**. To see how stretching works, select the horizontal piece of the green wire in **tut2b** and type **W**, then **E**. Stretching only worries about material in front of and behind the selection; it ignores material to the sides (try the **Q** and **R** macros to see). You can use plowing (described in Tutorial #3) if this is a problem.

The command **:upsidedown** will flip the selection upside down, and **:sideways** flips the selection sideways. Both commands leave the selection so it occupies the same total area as before, but with the contents flipped. The command **:clockwise** will rotate the selection clockwise, leaving the lower-left corner of the new selection at the same place as the lower-left corner of the old selection. *Degrees* must be a multiple of 90, and defaults to 90.

At this point you know enough to do quite a bit of damage to the **tut2b** cell. Experiment with the selection commands. Remember that you can use **:undo** to back out of trouble.

6. Labels

Labels are pieces of text attached to the paint of a cell. They are used to provide information to other tools that will process the circuit. Most labels are node names: they provide an easy way of referring to nodes in tools such as routers, simulators, and timing analyzers. Labels may also be used for other purposes: for example, some labels are treated as *attributes* that give Crystal, the timing analyzer, information about the direction of signal flow through transistors.

Load the cell **tut2c** and place a cross in the middle of the red chunk (to make a cross, position the lower-left corner of the box with the left button and then click the right button to place the upper-right corner on top of the lower-left corner). Then type type the command **:label test**. A new label will appear at the position of the box. The complete syntax of the **:label** command is

:label [*text* [*position* [*layer*]]]

Text must be supplied, but the other arguments can be defaulted. If *text* has any spaces in it, then it must be enclosed in double quotes. *Position* tells where the text should be displayed, relative to the point of the label. It may be any of

north, **south**, **east**, **west**, **top**, **bottom**, **left**, **right**, **up**, **down**, **center**, **northeast**, **ne**, **southeast**, **se**, **southwest**, **sw**, **northwest**, **nw**. For example, if **ne** is given, the text will be displayed above and to the right of the label point. If no *position* is given, Magic will pick a position for you. *Layer* tells which paint layer to attach the label to. If *layer* covers the entire area of the label, then the label will be associated with the particular layer. If *layer* is omitted, or if it doesn't cover the label's area, Magic initially associates the label with the "space" layer, then checks to see if there's a layer that covers the whole area. If there is, Magic moves the label to that layer. It is generally a bad idea to place labels at points where there are several paint layers, since it will be hard to tell which layer the label is attached to. As you edit, Magic will ensure that labels are only attached to layers that exist everywhere under the label. To see how this works, paint the layer pdiff (brown) over the label you just created: the label will switch layers. Finally, erase poly over the area, and the label will move again.

Although many labels are point labels, this need not be the case. You can label any rectangular area by setting the box to that area before invoking the label command. This feature is used for labelling terminals for the router (see below), and for labelling tiles used by Mpack, the tile packing program. **Tut2e** has examples of point, line, and rectangular labels.

All of the selection commands apply to labels as well as paint. Whenever you select paint, the labels attached to that paint will also be selected. Selected labels are highlighted in white. Select some of the chunks of paint in **tut2c** to see how the labels are selected too. When you use area selection, labels will only be selected if they are completely contained in the area being selected. If you'd like to select *just* a label without any paint, make the box into a cross and put the cross on the label: **s** and **S** will select just the label.

There are several ways to erase a label. One way is to select and then delete it. Another way is to erase the paint that the label is attached to. If the paint is erased all around the label, then Magic will delete the label too. Try attaching a label to a red area, then paint blue over the red. If you erase blue the label stays (since it's attached to red), but if you erase the red then the label is deleted.

You can also erase labels using the **:erase** command and the pseudo-layer **labels**. The command

:erase labels

will erase all labels that lie completely within the area of the box. Finally, you can erase a label by making the box into a cross on top of the label, then clicking the middle button with the cursor over empty space. Technically, this will erase all paint layers and labels too. However, since the box has zero area, erasing paint has no effect: only the labels are erased.

7. Labelling Conventions

When creating labels, Magic will permit you to use absolutely any text whatsoever. However, many other tools, and even parts of Magic, expect label names to observe certain conventions. Except for the special cases described

below, labels shouldn't contain any of the characters "/$@!^". Spaces, control characters, or parentheses within labels are probably a bad idea too. Many of the programs that process Magic output have their own restrictions on label names, so you should find out about the restrictions that apply at your site. Most labels are node names: each one gives a unique identification to a set of things that are electrically connected. There are two kinds of node names, local and global. Any label that ends in "!" is treated as a global node name; it will be assumed that all nodes by this name, anywere in any cell in a layout, are electrically connected. The most common global names are **Vdd!** and **GND!**, the power rails. You should always use these names exactly, since many other tools require them. Nobody knows why "GND!" is all in capital letters and "Vdd!" isn't.

Any label that does not end in "!" or any of the other special characters discussed below is a local node name. It refers to a node within that particular cell. Local node names should be unique within the cell: there shouldn't be two electrically distinct nodes with the same name. On the other hand, it is perfectly legal, and sometimes advantageous, to give more than one name to the same node. It is also legal to use the same local node name in different cells: the tools will be able to distinguish between them and will not assume that they are electrically connected.

The only other labels currently understood by the tools are *attributes*. Attributes are pieces of text associated with a particular piece of the circuit: they are not node names, and need not be unique. For example, an attribute might identify a node as a chip input, or it might identify a transistor terminal as the source of information for that transistor. Any label whose last character is "@", "$", or "^" is an attribute. There are three different kinds of attributes. Node attributes are those ending with "@"; they are associated with particular nodes. Transistor source/drain attributes are those ending in "$"; they are associated with particular terminals of a transistor. A source or drain attribute must be attached to the channel region of the transistor and must fall exactly on the source or drain edge of the transistor. The third kind of attribute is a transistor gate attribute. It ends in "^" and is attached to the channel region of the transistor. To see examples of attributes and node names, edit the cell **tut2d** in Magic.

There is one last convention about label usage. The routing tools ignore all labels except for those on the edges of cells. If you expect to use the router to connect to a particular node, you should place the label for that node on its outermost edge. The label should not be a point label, but should instead be a horizontal or vertical line covering the entire edge of the wire. The router will choose a connection point somewhere along the label. A good rule of thumb is to label all nodes that enter or leave the cell in this way. For more details on how labels are used in routing, see "Magic Tutorial #7: Netlists and Routing".

8. Files and Formats

Magic provides a variety of ways to save your cells on disk. Normally, things are saved in a special Magic format. Each cell is a separate file, and the name of the file is just the name of the cell with **.mag** appended. For example, the cell

tut2a is saved in file **tut2a.mag**. To save cells on disk, invoke the command

:writeall

This command will run through each of the cells that you have modified in this editing session, and ask you what to do with the cell. Normally, you'll type **write**, or just hit the return key, in which case the cell will be written back to the disk file from which it was read (if this is a new cell, then you'll be asked for a name for the cell). If you type **autowrite**, then Magic will write out all the cells that have changed without asking you what to do on a cell-by-cell basis. **Flush** will cause Magic to delete its internal copy of the cell and reload the cell from the disk copy, thereby expunging all edits that you've made. **Skip** will pass on to the next cell without writing this cell (but Magic still remembers that it has changed, so the next time you invoke **:writeall** Magic will ask about this cell again). **Abort** will stop the command immediately without writing or checking any more cells.

IMPORTANT NOTE: Unlike vi and other text editors, Magic doesn't keep checkpoint files. This means that if the system should crash in the middle of a session, you'll lose all changes since the last time you wrote out cells. It's a good idea to save your cells frequently during long editing sessions.

You can also save the cell you're currently editing with the command

:save *name*

This command will append ".mag" to *name* and save the cell you are editing in that location. If you don't provide a name, Magic will use the cell's name (plus the ".mag" extension) as the file name, and it will prompt you for a name if the cell hasn't yet been named.

Once a cell has been saved on disk you can edit it by invoking Magic with the command

magic *name*

where *name* is the same name you used to save the cell (no ".mag" extension).

Magic can also read and write files in CIF and Calma Stream formats. See "Magic Tutorial #9: Format Conversion for CIF and Calma" for details.

9. Plotting

Magic can generate hardcopy plots of layouts in two ways: versatec and gremlin. The first style is for printers like the black-and-white Versatec family: for these, Magic will output a raster file and spool the file for printing. To plot part of your design, place the box around the part you'd like to plot and type

:plot versatec [*width* [*layers*]]

This will generate a plot of the area of the box. Everything visible underneath the box will appear in more-or-less the same way in the plot. *Width* specifies how wide the plot will be, in inches. Magic will scale the plot so that the area of the box comes out this wide. The default for *width* is the width of the plotter (if *width* is larger than the plotter width, it's reduced to the plotter width). If *layers*

is given, it specifies exactly what information is to be plotted. Only those layers will appear in the plot. The special "layer" **labels** will enable label plotting.

The second form of plotting is for generating Gremlin-format files, which can then be edited with the Gremlin drawing system or included in documents processed by Grn and Ditroff. The command to get Gremlin files is

:plot gremlin *file* [*layers*]

It will generate a Gremlin-format file in *file* that describes everything underneath the box. If *layers* is specified, it indicates which layers are to appear in the file; otherwise everything visible on the screen is output. The Gremlin file is output without any particular scale; use the **width** or **height** commands in Grn to scale the plot when it's printed. You should use the **mg** stipples when printing Magic Gremlin plots; these will produce the same stipple patterns as **:plot versatec**.

There are several plotting parameters used internally to Magic, such as the width of the Versatec printer and the number of dots per inch on the Versatec printer. You can modify most of these to work with different printers. For details, read about the various **:plot** command options in the *man* page.

10. Utility Commands

There are several additional commands that you will probably find useful once you start working on real cells. The command

:grid [*spacing*]
:grid *xSpacing ySpacing*
:grid *xSpacing ySpacing xOrigin yOrigin*
:grid off

will display a grid over your layout. Initially, the grid has a one-unit spacing. Typing **:grid** with no arguments will toggle the grid on and off. If a single numerical argument is given, the grid will be turned on, and the grid lines will be *spacing* units apart. The macro **g** provides a short form for **:grid** and **G** is short for **:grid 2**. If you provide two arguments to **:grid**, they are the x- and y-spacings, which may be different. If you provide four arguments, the last two specify a reference point through which horizontal and vertical grid lines pass; the default is to use (0,0) as the grid origin. The command **:grid off** always turns the grid off, regardless of whether or not is was previously on. When the grid is on, a small black box is displayed to mark the (0,0) coordinate of the cell you're editing.

If you want to create a cell that doesn't fit on the screen, you'll need to know how to change the screen view. This can be done with three commands:

:zoom *factor*
:findbox [**zoom**]
:view

If *factor* is given to the zoom command, it is a zoom-out factor. For example, the command **:zoom 2** will change the view so that there are twice as many units across the screen as there used to be (**Z** is a macro for this). The new view will

have the same center as the old one. The command **:zoom .5** will increase the magnification so that only half as much of the circuit is visible.

The **:findbox** command is used to change the view according to the box. The command alone just moves the view (without changing the scale factor) so that the box is in the center of the screen. If the **zoom** argument is given then the magnification is changed too, so that the area of the box nearly fills the screen. **z** is a macro for **:findbox zoom** and **B** is a macro for **:findbox**.

The command **:view** resets the view so that the entire cell is visible in the window. It comes in handy if you get lost in a big layout. The macro **v** is equivalent to **:view**.

The command **:box** prints out the size and location of the box in case you'd like to measure something in your layout. The macro **b** is predefined to **:box**. The **:box** command can also be used to set the box to a particular location, height, or width. See the man page for details.

The command

:what

will print out information about what's selected. This may be helpful if you're not sure what layer a particular piece of material is, or what layer a particular label is attached to.

If you forget what a macro means, you can invoke the command

:macro [*char*]

This command will print out the long command that's associated with the macro *char*. If you omit *char*, Magic will print out all of the macro associations. The command

:macro *char command*

We set up *char* to be a macro for *command*, replacing the old *char* macro if there was one. If *command* contains any spaces then it must be enclosed in double-quotes. To see how this works, type the command **:macro 1 "echo You just typed the 1 key."**, then type the 1 key.

One of the macros, ".", has special meaning in Magic. This macro is always defined by the system to be the last long command you typed. Whenever you'd like to repeat a long command, all you have to do is use the dot macro.

11. What the Layers Mean

The paint layers available in Magic are different from those that you may be used to in Caesar and other systems because they don't correspond exactly to the masks used in fabrication. We call them *abstract layers* because they correspond to constructs such as wires and contacts, rather than mask layers. We also call them *logs* because they look like sticks except that the geometry is drawn fully fleshed instead of as lines. In Magic there is one paint layer for each kind of conducting material (polysilicon, ndiffusion, metal1, etc.), plus one additional paint layer for each kind of transistor (ntransistor, ptransistor, etc.), and, finally,

one further paint layer for each kind of contact (pcontact, ndcontact, m2contact, etc.) Each layer has one or more names that are used to refer to that layer in commands. To find out the layers available in the current technology, type the command

:layers

In addition to the mask layers, there are a few pseudo-layers that are valid in all technologies; these are listed in Table I. Each Magic technology also has a technology manual describing the features of that technology, such as design rules, routing layers, CIF styles, etc. If you haven't seen any of the technology manuals yet, this is a good time to take a look at the one for your process.

errors (design-rule violations)
labels
subcells
* (all mask layers)
$ (all mask layers visible under cursor)

Table I. Pseudo-layers available in all technologies.

If you're used to designing with mask layers (e.g. you've been reading the Mead-Conway book), Magic's log style will take some getting used to. One of the reasons for logs is to save you work. In Magic you don't draw implants, wells, buried windows, or contact via holes. Instead, you draw the primary conducting layers and paint some of their overlaps with special types such as n-transistor or polysilicon contact. For transistors, you draw only the actual area of the transistor channel. Magic will generate the polysilicon and diffusion, plus any necessary implants, when it creates a CIF file. For contacts, you paint the contact layer in the area of overlap between the conducting layers. Magic will generate each of the constituent mask layers plus vias and buried windows when it writes the CIF file. Figure 1 shows a simple cell drawn with both mask layers (as in Caesar) and with logs (as in Magic). If you're curious about what the masks will look like for a particular layout, you can use the **:cif see** command to view the mask information.

An advantage of the logs used in Magic is that they simplify the design rules. Most of the formation rules (e.g. contact structure) go away, since Magic automatically generates correctly-formed structures when it writes CIF. All that are left are minimum size and spacing rules, and Magic's abstract layers result in fewer of these than there would be otherwise. This helps to make Magic's built-in design rule checker very fast (see "Magic Tutorial #6: Design Rule Checking"), and is one of the reasons plowing is possible.

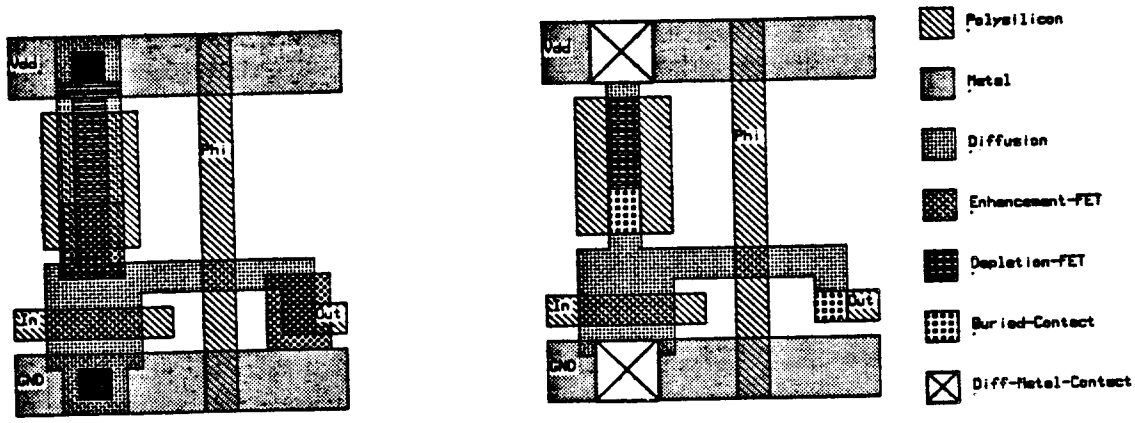


Figure 1. An example of how the logs are used. The figure on the left shows actual mask layers for an nMOS shift register cell, and the figure on the right shows the layers used to represent the cell in Magic.

Magic Tutorial #3: Advanced Painting (Wiring and Plowing)

John Ousterhout
Walter Scott

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

This tutorial corresponds to Magic version 4.

Tutorials to read first:

Magic Tutorial #1: Getting Started
Magic Tutorial #2: Basic Painting and Selection

Commands introduced in this tutorial:

:array, :corner, :fill, :flush, :plow, :straighten, :tool, :wire

Macros introduced in this tutorial:

<space>

1. Introduction

Tutorial #2 showed you the basic facilities for placing paint and labels, selecting, and manipulating the things that are selected. This tutorial describes two additional facilities for manipulating paint: wiring and plowing. These commands aren't absolutely necessary, since you can achieve the same effect with the simpler commands of Tutorial #2; however, wiring and plowing allow you to perform certain kinds of manipulations much more quickly than you could otherwise. Wiring is described in Section 2; it allows you to place wires by pointing at the ends of legs rather than by positioning the box, and also provides for convenient contact placement. Plowing is the subject of Section 3. It allows you to re-arrange pieces of your circuit without having to worry about design-rule violations being created: plowing automatically moves things out of the way to avoid trouble.

2. Wiring

The box-and-painting paradigm described in Tutorial #2 is sufficient to create any possible layout, but it's relatively inefficient since three keystrokes are required to paint each new area: two button clicks to position the box and one more to paint the material. This section describes a different painting mechanism based on *wires*. At any given time, there is a current wiring material and wire thickness. With the wiring interface you can create a new area of material with a single button click: this paints a straight-line segment of the current material and width between the end of the previous wire segment and the cursor location. Each additional button click adds an additional segment. The wiring interface also makes it easy for you to place contacts.

2.1. Tools

Before learning about wiring, you'll need to learn about tools. Until now, when you've pressed mouse buttons in layout windows the buttons have caused the box to change or material to be painted. The truth is that buttons can mean different things at different times. The meaning of the mouse buttons depends on the *current tool*. Each tool is identified by a particular cursor shape and a particular interpretation of the mouse buttons. Initially, the current tool is the box tool; when the box tool is active the cursor has the shape of a crosshair. To get information about the current tool, you can type the long command

:tool info

This command prints out the name of the current tool and the meaning of the buttons. Run Magic on the cell **tut3a** and type **:tool info**.

The **:tool** command can also be used to switch tools. Try this out by typing the command

:tool

Magic will print out a message telling you that you're using the wiring tool, and the cursor will change to an arrow shape. Use the **:tool info** command to see what the buttons mean now. You'll be using the wiring tool for most of the rest of this section. The macro " " (space) corresponds to **:tool**. Try typing the space key a few times: Magic will cycle circularly through all of the available tools. There are three tools in Magic right now: the box tool, which you already know about, the wiring tool, which you'll learn about in this tutorial, and the netlist tool, which has a square cursor shape and is used for netlist editing. "Tutorial #7: Netlists and Routing" will show you how to use the netlist tool.

The current tool affects only the meanings of the mouse buttons. It does not change the meanings of the long commands or macros. This means, for example, that you can still use all the selection commands while the wiring tool is active. Switch tools to the wiring tool, point at some paint in **tut3a**, and type the **s** macro. A chunk gets selected just as it does with the box tool.

2.2. Basic Wiring

There are three basic wiring commands: selecting the wiring material, adding a leg, and adding a contact. This section describes the first two commands. At this point you should be editing the cell **tut3a** with the wiring tool active. The first step in wiring is to pick the material and width to use for wires. This can be done in two ways. The easiest way is to find a piece of material of the right type and width, point to it with the cursor, and click the left mouse button. Try this in **tut3a** by pointing to the label **1** and left-clicking. Magic prints out the material and width that it chose, selects a square of that material and width around the cursor, and places the box around the square. Try pointing to various places in **tut3a** and left-clicking.

Once you've selected the wiring material, the right button paints legs of a wire. Left-click on label **1** to select the red material, then move the cursor over label **2** and right-click. This will paint a red wire between **1** and **2**. The new wire leg is selected so that you can modify it with selection commands, and the box is placed over the tip of the leg to show you the starting point for the next wire leg. Add more legs to the wire by right-clicking at **3** and then **4**. Use the mouse buttons to paint another wire in blue from **5** to **6** to **7**.

Each leg of a wire must be either horizontal or vertical. If you move the cursor diagonally, Magic will still paint a horizontal or vertical line (whichever results in the longest new wire leg). To see how this works, left-click on **8** in **tut3a**, then right-click on **9**. You'll get a horizontal leg. Now undo the new leg and right-click on **10**. This time you'll get a vertical leg. You can force Magic to paint the next leg in a particular direction with the commands

:wire horizontal
:wire vertical

Try out this feature by left-clicking on **8** in **tut3a**, moving the cursor over **10**, and typing **:wire ho** (abbreviations work for **:wire** command options just as they do elsewhere in Magic). This command will generate a short horizontal leg instead of a longer vertical one.

2.3. Contacts

When the wiring tool is active, the middle mouse button places contacts. Undo all of your changes to **tut3a** by typing the command **:flush** and answering **yes** to the question Magic asks. This throws away all of the changes made to the cell and re-loads it from disk. Draw a red wire leg from **1** to **2**. Now move the cursor over the blue area and click the middle mouse button. This has several effects. It places a contact at the end of the current wire leg, selects the contact, and moves the box over the selection. In addition, it changes the wiring material and thickness to match the material you middle-clicked. Move the cursor over **3** and right-click to paint a blue leg, then make a contact to purple by middle-clicking over the purple material. Continue by drawing a purple leg to **4**.

Once you've drawn the purple leg to **4**, move the cursor over red material and middle-click. This time, Magic prints an error message and treats the click

just like a left-click. Magic only knows how to make contacts between certain combinations of layers, which are specified in the technology file (see "Magic Maintainer's Manual #2: The Technology File"). For this technology, Magic doesn't know how to make contacts directly between purple and red.

2.4. Wiring and the Box

In the examples so far, each new wire leg appeared to be drawn from the end of the previous leg to the cursor position. In fact, however, the new material was drawn from the *box* to the cursor position. Magic automatically repositions the box on each button click to help set things up for the next leg. Using the box as the starting point for wire legs makes it easy to start wires in places that don't already have material of the right type and width. Suppose that you want to start a new wire in the middle of an empty area. You can't left-click to get the wire started there. Instead, you can left-click some other place where there's the right material for the wire, type the space bar twice to get back the box tool, move the box where you'd like the wire to start, hit the space bar once more to get back the wiring tool, and then right-click to paint the wire. Try this out on **tut3a**.

When you first start wiring, you may not be able to find the right kind of material anywhere on the screen. When this happens, you can select the wiring material and width with the command

:wire type *layer width*

Then move the box where you'd like the wire to start, switch to the wiring tool, and right-click to add legs.

2.5. Wiring and the Selection

Each time you paint a new wire leg or contact using the wiring commands, Magic selects the new material just as if you had placed the cursor over it and typed **s**. This makes it easy for you to adjust its position if you didn't get it right initially. The **:stretch** command is particularly useful for this. In **tut3a**, paint a wire leg in blue from **5** to **6** (use **:flush** to reset the cell if you've made a lot of changes). Now type **R** two or three times to stretch the leg over to the right. Middle-click over purple material, then use **W** to stretch the contact downward.

It's often hard to position the cursor so that a wire leg comes out right the first time, but it's usually easy to tell whether the leg is right once it's painted. If it's wrong, then you can use the stretching commands to shift it over one unit at a time until it's correct.

2.6. Bundles of Wires

Magic provides two additional commands that are useful for running *bundles* of parallel wires. The commands are:

fill *direction* [*layers*]
corner *direction1 direction2* [*layers*]

To see how they work, load the cell **tut3b**. The **:fill** comand extends a whole bunch of paint in a given direction. It finds all paint touching one side of the box and extends that paint to the opposite side of the box. For example, **:fill left** will look underneath the right edge of the box for paint, and will extend that paint to the left side of the box. The effect is just as if all the colors visible underneath that edge of the box constituted a paint brush; Magic sweeps the brush across the box in the given direction. Place the box over the label "Fill here" in **tut3b** and type **:fill left**.

The **:corner** command is similar to **:fill** except that it generates L-shaped wires that follow two sides of the box, travelling first in *direction1* and then in *direction2*. Place the box over the label "Corner here" in **tut3b** and type **:corner right up**.

In both **:fill** and **:corner**, if *layers* isn't specified then all layers are filled. If *layers* is given then only those layers are painted. Experiment on **tut3b** with the **:fill** and **:corner** commands.

When you're painting bundles of wires, it would be nice if there were a convenient way to place contacts across the whole bundle in order to switch to a different layer. There's no single command to do this, but you can place one contact by hand and then use the **:array** command to replicate a single contact across the whole bundle. Load the cell **tut3c**. This contains a bundle of wires with a single contact already painted by hand on the bottom wire. Type **s** with the cursor over the contact, and type **S** with the cursor over the stub of purple wiring material next to it. Now place the box over the label "Array" and type the command **:array 1 10**. This will copy the selected contact across the whole bundle.

The syntax of the **:array** command is

:array *xsize ysize*

This command makes the selection into an array of identical elements. *Xsize* specifies how many total instances there should be in the x-direction when the command is finished and *ysize* specifies how many total instances there should be in the y-direction. In the **tut3c** example, **xsize** was one, so no additional copies were created in that direction; **ysize** was 10, so 9 additional copies were created. The box is used to determine how far apart the elements should be: the width of the box determines the x-spacing and the height determines the y-spacing. The new material always appears above and to the right of the original copy.

In **tut3c**, use **:corner** to extend the purple wires and turn them up. Then paint a contact back to blue on the leftmost wire, add a stub of blue paint above it, and use **:array** to copy them across the top of the bundle. Finally, use **:fill** again to extend the blue bundle farther up.

3. Plowing

Magic contains a facility called *plowing* that you can use to stretch and compact cells. The basic plowing command has the syntax

:plow *direction* [*layers*]

where *direction* is a Manhattan direction like **left** and *layers* is an optional, comma-separated list of mask layers. The plow command treats one side of the box as if it were a plow, and shoves the plow over to the other side of the box. For example, **:plow up** treats the bottom side of the box as a plow, and moves the plow to the top of the box.

As the plow moves, every edge in its path is pushed ahead of it (if *layers* is specified, then only edges on those layers are moved). Each edge that is pushed by the plow pushes other edges ahead of it in a way that preserves design rules, connectivity, and transistor and contact sizes. This means that material ahead of the plow gets compacted down to the minimum spacing permitted by the design rules, and material that crossed the plow's original position gets stretched behind the plow.

You can compact a cell by placing a large plow off to one side of the cell and plowing across the whole cell. You can open up space in the middle of a cell by dragging a small plow across the area where you want more space.

To try out plowing, edit the cell **tut3d**, place the box over the rectangle that's labelled "Plow here", and try plowing in various directions. Also, try plowing only certain layers. For example, with the box over the "Plow here" label, try

:plow right metal2

Nothing happens. This is because there are no metal2 *edges* in the path of the plow. If instead you had typed

:plow right metal1

only the metal would have been plowed to the right.

In addition to plowing with the box, you can plow the selection. The command to do this has the following syntax:

:plow selection [*direction* [*distance*]]

This is very similar to the **:stretch** command: it picks up the selection and the box and moves both so that the lower-left corner of the box is at the cursor location. Unlike the **:stretch** command, though, **:plow selection** insures that design rule correctness and connectivity are preserved.

Load the cell **tut3e** and use **a** to select the area underneath the label that says "select me". Then point with the cursor to the point labelled "point here" and type **:plow selection**. Practice selecting things and plowing them. Like the **:stretch** command, there is also a longer form of **:plow selection**. For example, **:plow selection down 5** will plow the selection and the box down 10 units.

Selecting a cell and plowing it is a good way to move the cell. Load **tut3f** and select the cell **tut3e**. Point to the label "point here" and plow the selection

with **:plow selection**. Notice that all connections to the cell have remained attached. The cell you select must be in the edit cell, however.

The plowing operation is implemented in a way that tries to keep your design as compact as possible. To do this, it inserts jogs in wires around the plow. In many cases, though, the additional jogs are more trouble than they're worth. To reduce the number of jogs inserted by plowing, type the command

:plow nojogs

From now on, Magic will insert as few jogs as possible when plowing, even if this means moving more material. You can re-enable jog insertion with the command

:plow jogs

Load the cell **tut3d** again and try plowing it both with and without jog insertion.

There is another way to reduce the number of jogs introduced by plowing. Instead of avoiding jogs in the first place, plowing can introduce them freely but clean them up as much as possible afterward. This results in more dense layouts, but possibly more jogs than if you had enabled **:plow nojogs**. To take advantage of this second method for jog reduction, re-enable jog insertion (**:plow jogs**) and enable jog cleanup with the command

:plow straighten

From now on, Magic will attempt to straighten out jogs after each plow operation. To disable straightening, use the command

:plow nostraighten

It might seem pointless to disable jog introduction with **:plow nojogs** at the same time straightening is enabled with **:plow straighten**. While it is true that **:plow nojogs** won't introduce any new jogs for **:plow straighten** to clean up, plowing will straighten out any existing jogs after each operation.

In fact, there is a separate command that is sometimes useful for cleaning up layouts with many jogs, namely the command

:straighten *direction*

where *direction* is a Manhattan direction, e.g., **up**, **down**, **right**, or **left**. This command will start from one side of the box and pull jogs toward that side to straighten them. Load the cell **tut3g**, place the box over the label "put box here", and type **:straighten left**. Undo the last command and type **:straighten right** instead. Play around with the **:straighten** command.

There is one more feature of plowing that is sometimes useful. If you are working on a large cell and want to make sure that plowing never affects any geometry outside of a certain area, you can place a *boundary* around the area you want to affect with the command

:plow boundary

The box is used to specify the area you want to affect. After this command, subsequent plows will only affect the area inside this boundary.

Load the cell **tut3h** place the box over the label "put boundary here", and type **:plow boundary**. Now move the box away. You will see the boundary highlighted with dotted lines. Now place the box over the area labelled "put plow here" and plow up. This plow would cause geometry outside of the boundary to be affected, so Magic reduces the plow distance enough to prevent this and warns you of this fact. Now undo the last plow and remove the boundary with

:plow noboundary

Put the box over the "put plow here" label and plow up again. This time there was no boundary to stop the plow, so everything was moved as far as the height of the box. Experiment with placing the boundary around an area of this cell and plowing.

Magic Tutorial #4: Cell Hierarchies

John Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

This tutorial corresponds to Magic version 4.

Tutorials to read first:

Magic Tutorial #1: Getting Started
Magic Tutorial #2: Basic Painting and Selection

Commands introduced in this tutorial:

:array, :edit, :expand, :flush, :getcell, :identify, :load, :path, :see, :unexpand

Macros introduced in this tutorial:

x, X, ^X

1. Introduction

In Magic, a layout is a hierarchical collection of cells. Each cell contains three things: paint, labels, and subcells. Tutorial #2 showed you how to create and edit paint and labels. This tutorial describes Magic's facilities for building up cell hierarchies. Strictly speaking, hierarchical structure isn't necessary: any design that can be represented hierarchically can also be represented "flat" (with all the paint and labels in a single cell). However, many things are greatly improved if you use a hierarchical structure, including the efficiency of the design tools, the speed with which you can enter the design, and the ease with which you can modify it later.

2. Selecting and Viewing Hierarchical Designs

"Hierarchical structure" means that each cell can contain other cells as components. To look at an example of a hierarchical layout, enter Magic with the shell command **magic tut4a**. The cell **tut4a** contains four subcells plus some blue paint. Two of the subcells are instances of cell **tut4x** and two are instances of **tut4y**. Initially, each subcell is displayed in *unexpanded* form. This means that no details of the subcell are displayed; all you see is the cell's bounding box, plus two names inside the bounding box. The top name is the name of the subcell (the name you would type when invoking Magic to edit the cell). The cell's contents are stored in a file with this name plus a **.mag** extension. The bottom name inside each bounding box is called an *instance identifier*, and is used to distinguish different instances of the same subcell. Instance id's are used for routing and circuit extraction, and are discussed in Section 6.

Subcells can be manipulated using the same selection mechanism that you learned in Tutorial #2. To select a subcell, place the cursor over the subcell and type **f** ("find cell"), which is a macro for **:select cell**. You can also select a cell by typing **s** when the cursor is over a location where there's no paint; **f** is probably more convenient, particularly for cells that are completely covered with paint. When you select a cell the box will be set to the cell's bounding box, the cell's name will be highlighted, and a message will be printed on the text display. All the selection operations (**:move**, **:copy**, **:delete**, etc.) apply to subcells. Try selecting and moving the top subcell in **tut4a**. You can also select subcells using area selection (the **a** and **A** macros): any unexpanded subcells that intersect the area of the box will be selected.

To see what's inside a cell instance, you must *expand* it. Select one of the instances of **tut4y**, then type the command

:expand toggle

or invoke the macro **^X** which is equivalent. This causes the internals of that instance of **tut4y** to be displayed. If you type **^X** again, the instance is unexpanded so you only see a bounding box again. The **:expand toggle** command expands all of the selected cells that are unexpanded, and unexpands all those that are expanded. Type **^X** a third time so that **tut4y** is expanded.

As you can see now, **tut4y** contains an array of **tut4x** cells plus some additional paint. In Magic, an array is a special kind of instance containing multiple copies of the same subcell spaced at fixed intervals. Arrays can be one-dimensional or two-dimensional. The whole array is always treated as a single instance: any command that operates on one element of the array also operates on all the other elements simultaneously. The instance identifiers for the elements of the array are the same except for an index. Now select one of the elements of the array and expand it. Notice that the entire array is expanded at the same time.

When you have expanded the array, you'll see that the paint in the top-level cell **tut4a** is displayed more brightly than the paint in the **tut4x** instances. **Tut3a** is called the *edit cell*, because its contents are currently editable. The paint in the edit cell is normally displayed more brightly than other paint to make

it clear that you can change it. As long as **tut4a** is the edit cell, you cannot modify the paint in **tut4x**. Try erasing paint from the area of one of the **tut4x** instances: nothing will be changed. Section 4 tells how to switch the edit cell.

Place the cursor over one of the **tut4x** array elements again. At this point, the cursor is actually over three different cells: **tut4x** (an element of an array instance within **tut4y**), **tut4y** (an instance within **tut4a**), and **tut4**. Even the topmost cell in the hierarchy is treated as an instance by Magic. When you press the **s** key to select a cell, Magic initially chooses the smallest instance visible underneath the cursor, **tut4x** in this case. However, if you invoke the **s** macro again (or type **:select**) without moving the cursor, Magic will step through all of the instances under the cursor in order. Try this out. The same is true of the **f** macro and **:select cell**.

When there are many different expanded cells on the screen, you can use the selection commands to select paint from any of them. You can select anything that's visible, regardless of which cell it's in. However, as mentioned above, you can only modify paint in the edit cell. If you use **:move** or **:upsidedown** or similar commands when you've selected information outside the edit cell, the information outside the edit cell is removed from the selection before performing the operation.

There are two additional commands you can use for expanding and unexpanding cells:

:expand
:unexpand

Both of these commands operate on the area underneath the box. The **:expand** command will recursively expand every cell that intersects the box until there are no unexpanded cells left under the box. The **:unexpand** command will unexpand every cell whose area intersects the box but doesn't completely contain it. The macro **x** is equivalent to **:expand**, and **X** is equivalent to **:unexpand**. Try out the various expansion and unexpansion facilities on **tut4a**.

3. Manipulating Subcells

There are a few other commands, in addition to the selection commands already described, that you'll need in order to manipulate subcells. The command

:getcell *name*

will find the file *name*.**mag** on disk, read the cell it contains, and create an instance of that cell with its lower-left corner aligned with the lower-left corner of the box. Use the **getcell** command to get an instance of the cell **tut4z**. After the **getcell** command, the new instance is selected so you can move it or copy it or delete it. There are several other forms of the **getcell** command that allow you to position the cell using a label contained within the cell. See the *man* page for details.

To turn a normal instance into an array, select the instance and then invoke the **:array** command. It has two forms:

:array *xsize ysize*
:array *xlo xhi ylo yhi*

In the first form, *xsize* indicates how many elements the array should have in the x-direction, and *ysize* indicates how many elements it should have in the y-direction. The spacing between elements is controlled by the box's width (for the x-direction) and height (for the y-direction). By changing the box size, you can space elements so that they overlap, abut, or have gaps between them. The elements are given indices from 0 to *xsize*-1 in the x-direction and from 0 to *ysize*-1 in the y-direction. The second form of the command is identical to the first except that the elements are given indices from *xlo* to *xhi* in the x-direction and from *ylo* to *yhi* in the y-direction. Try making a 4x4 array out of the **tut4z** cell with gaps between the cells.

You can also invoke the **:array** command on an existing array to change the number of elements or spacing. Use a size of 1 for *xsize* or *ysize* in order to get a one-dimensional array. If there are several cells selected, the **:array** command will make each of them into an array of the same size and spacing. It also works on paint and labels: if paint and labels are selected when you invoke **:array**, they will be copied many times over to create the array. Try using the array command to replicate a small strip of paint.

4. Switching the Edit Cell

At any given time, you are editing the definition of a single cell. This definition is called the *edit cell*. You can modify paint and labels in the edit cell, and you can re-arrange its subcells. You may not re-arrange or delete the subcells of any cells other than the edit cell, nor may you modify the paint or labels of any cells except the edit cell. You may, however, copy information from other cells into the edit cell, using the selection commands. To help clarify what is and isn't modifiable, Magic displays the paint of the edit cell in brighter colors than other paint.

When you rearrange subcells of the edit cell, you aren't changing the subcells themselves. All you can do is change the way they are used in the edit cell (location, orientation, etc.). When you delete a subcell, nothing happens to the file containing the subcell; the command merely deletes the instance from the edit cell.

Besides the edit cell, there is one other special cell in Magic. It's called the *root cell* and is the topmost cell in the hierarchy, the one you named when you ran Magic (**tut4a** in this case). As you will see in Tutorial #5, there can actually be several root cells at any given time, one in each window. For now, there is only a single window on the screen, and thus only a single root cell. The window caption at the top of the color display contains the name of the window's root cell and also the name of the edit cell.

Up until now, the root cell and the edit cell have been the same. However, this need not always be the case. You can switch the edit cell to any cell in the hierarchy by selecting an instance of the definition you'd like to edit, and then

typing the command

:edit

Use this command to switch the edit cell to one of the **tut4x** instances in **tut4a**. Its paint brightens, while the paint in **tut4a** becomes dim. If you want to edit an element of an array, select the array, place the cursor over the element you'd like to edit, then type **:edit**. The particular element underneath the cursor becomes the edit cell.

When you edit a cell, you are editing the master definition of that cell. This means that if the cell is used in several places in your design, the edits will be reflected in all those places. Try painting and erasing in the **tut4x** cell that you just made the edit cell: the modifications will appear in all of its instances.

There is a second way to change the edit cell. This is the command

:load *name*

The **:load** command loads a new hierarchy into the window underneath the cursor. *Name* is the name of the root cell in the hierarchy. If no *name* is given, a new unnamed cell is loaded and you start editing from scratch. The **:load** command only changes the edit cell if there is not already an edit cell in another window.

5. Subcell Usage Conventions

Overlaps between cells are occasionally useful to share busses and control lines running along the edges. However, overlaps cause the analysis tools to work much harder than they would if there were no overlaps: wherever cells overlap, the tools have to combine the information from the two separate cells. Thus, you shouldn't use overlaps any more than absolutely necessary. For example, suppose you want to create a one-dimensional array of cells that alternates between two cell types, A and B: "ABABABABABAB". One way to do this is first to make an array of A instances with large gaps between them ("A A A A A A"), then make an array of B instances with large gaps between them ("B B B B B B"), and finally place one array on top of the other so that the B's nestle in between the A's. The problem with this approach is that the two arrays overlap almost completely, so Magic will have to go to a lot of extra work to handle the overlaps (in this case, there isn't much overlap of actual paint, but Magic won't know this and will spend a lot of time worrying about it). A better solution is to create a new cell that contains one instance of A and one instance of B, side by side. Then make an array of the new cell. This approach makes it clear to Magic that there isn't any real overlap between the A's and B's.

If you do create overlaps, you should use the overlaps only to connect the two cells together, and not to change their structure. This means that the overlap should not cause transistors to appear, disappear, or change size. The result of overlapping the two subcells should be the same electrically as if you placed the two cells apart and then ran wires to hook parts of one cell to parts of the other. The convention is necessary in order to be able to do hierarchical circuit extraction easily (it makes it possible for each subcell to be circuit-extracted

independently).

Three kinds of overlaps are flagged as errors by the design-rule checker. First, you may not overlap polysilicon in one subcell with diffusion in another cell in order to create transistors. Second, you may not overlap transistors or contacts in one cell with different kinds of transistors or contacts in another cell (there are a few exceptions to this rule in some technologies). Third, if contacts from different cells overlap, they must be the same type of contact and must coincide exactly: you may not have partial overlaps. This rule is necessary in order to guarantee that Magic can generate CIF for fabrication.

You will make life a lot easier on yourself (and on Magic) if you spend a bit of time to choose a clean hierarchical structure. In general, the less cell overlap the better. If you use extensive overlaps you'll find that the tools run very slowly and that it's hard to make modifications to the circuit.

6. Instance Identifiers

Instance identifiers are used to distinguish the different subcells within a single parent. The cell definition names cannot be used for this purpose because there could be many instances of a single definition. Magic will create default instance id's for you when you create new instances with the **:get** or **:copy** commands. The default id for an instance will be the name of the definition with a unique integer added on. You can change an id by selecting an instance (which must be a child of the edit cell) and invoking the command

:identify *newid*

where *newid* is the identifier you would like the instance to have. *Newid* must not already be used as an instance identifier of any subcell within the edit cell.

Any node or instance can be described uniquely by listing a path of instance identifiers, starting from the root cell. The standard form of such names is similar to Unix file names. For example, if **id1** is the name of an instance within the root cell, **id2** is an instance within **id1**, and **node** is a node name within **id2**, then **id1/id2/node** can be used unambiguously to refer to the node. When you select a cell, Magic prints out the complete path name of the instance.

Arrays are treated specially. When you use **:identify** to give an array an instance identifier, each element of the array is given the instance identifier you specified, followed by one or two array subscripts enclosed in square brackets, e.g, **id3[2]** or **id4[3][7]**. When the array is one-dimensional, there is a single subscript; when it is two-dimensional, the first subscript is for the y-dimension and the second for the x-dimension.

7. Writing and Flushing Cells

When you make changes to your circuit in Magic, there is no immediate effect on the disk files that hold the cells. You must explicitly save each cell that has changed, using either the **:save** command or the **:writeall** command. Magic keeps track of the cells that have changed since the last time they were saved on

disk. If you try to leave Magic without saving all the cells that have changed, the system will warn you and give you a chance to return to Magic to save them. Magic never flushes cells behind your back, and never throws away definitions that it has read in. Thus, if you edit a cell and then use **:load** to edit another cell, the first cell is still saved in Magic even though it doesn't appear anywhere on the screen. If you then invoke **:load** a second time to go back to the first cell, you'll get the edited copy.

If you decide that you'd really like to discard the edits you've made to a cell and recover the old version, there are two ways you can do it. The first way is using the **flush** option in **:writeall**. The second way is to use the command

:flush [*cellname*]

If no *cellname* is given, then the edit cell is flushed. Otherwise, the cell named *cellname* is flushed. The **:flush** command will expunge Magic's internal copy of the cell and replace it with the disk copy.

When you are editing large chips, Magic may claim that cells have changed even though you haven't modified them. Whenever you modify a cell, Magic makes changes in the parents of the cell, and their parents, and so on up to the root of the hierarchy. These changes record new design-rule violations, as well as timestamp and bounding box information used by Magic to keep track of design changes and enable fast cell read-in. Thus, whenever you change one cell you'll generally need to write out new copies of its parents and grandparents. If you don't write out the parents, or if you edit a child "out of context" (by itself, without the parents loaded), then you'll incur extra overhead the next time you try to edit the parents. "Timestamp mismatch" warnings are printed when you've edited cells out of context and then later go back and read in the cell as part of its parent. These aren't serious problems; they just mean that Magic is doing extra work to update information in the parent to reflect the child's new state.

8. Search Paths

When many people are working on a large design, the design will probably be more manageable if different pieces of it can be located in different directories of the file system. Magic provides a simple mechanism for managing designs spread over several directories. The system maintains a *search path* that tells which directories to search when trying to read in cells. By default, the search path is ".", which means that Magic looks only in the working directory. You can change the path using the command

:path [*searchpath*]

where *searchpath* is the new path that Magic should use. *Searchpath* consists of a list of directories separated by colons. For example, the path ".:~ouster/x:a/b" means that if Magic is trying to read in a cell named "foo", it will first look for a file named "foo.mag" in the current directory. If it doesn't find the file there, it will look for a file named "~ouster/x/foo.mag", and if that doesn't exist, then it will try "a/b/foo.mag" last. To find out what the current path is, type **:path**

with no arguments. In addition to your path, this command will print out the system cell library path (where Magic looks for cells if it can't find them anywhere in your path), and the system search path (where Magic looks for files like colormaps and technology files if it can't find them in your current directory).

If you're working on a large design, you should use the search path mechanism to spread your layout over several directories. A typical large chip will contain a few hundred cells; if you try to place all of them in the same directory there will just be too many things to manage. For example, place the datapath in one directory, the control unit in another, the instruction buffer in a third, and so on. Try to keep the size of each directory down to a few dozen files. You can place the **:path** command in a **.magic** file in your home directory or the directory you normally run Magic from; this will save you from having to retype it each time you start up (see the Magic man page to find out about **.magic** files).

Because there is only a single search path that is used everywhere in Magic, you must be careful not to re-use the same cell name in different portions of the chip. A common problem with large designs is that different designers use the same name for different cells. This works fine as long as the designers are working separately, but when the two pieces of the design are put together using a search path, a single copy of the cell (the one that is found first in the search path) gets used everywhere.

9. Additional Commands

This section describes a few additional cell-related commands that you may find useful. One of them is the command

:select save *file*

This command takes the selection and writes it to disk as a new Magic cell in the file *file*.**mag**. You can use this command to break up a big file into smaller ones, or to extract pieces from an existing cell.

The command

:dump *cellName* [*labelName*]

does the opposite of **select save**: it copies the contents of cell *cellName* into the edit cell, such that the lower-left corner of label *labelName* is at the lower-left corner of the box. The new material will also be selected. This command is similar in form to the **getcell** command except that it copies the contents of the cell instead of using the cell as a subcell. There are several forms of **dump**; see the *man* page for details.

The main purpose of **dump** is to allow you to create a library of cells representing commonly-used structures such as standard transistor shapes or special contact arrangements. You can then define macros that invoke the **dump** command to place the cells. The result is that a single keystroke is all you need to copy one of them into the edit cell.

As mentioned earlier, Magic normally displays the edit cell in brighter colors than non-edit cells. This helps to distinguish what is editable from what is not,

but may make it hard for you to view non-edit paint since it appears paler. If you type the command

:see allSame

you'll turn off this feature: all paint everywhere will be displayed in the bright colors. The word **allSame** must be typed just that way, with one capital letter. If you'd like to restore the different display styles, type the command

:see no allSame

You can also use the **:see** command to selectively disable display of various mask layers in order to make the other ones easier to see. For details, read about **:see** in the Magic man page.

Magic Tutorial #5: Multiple Windows

Robert N. Mayo

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

This tutorial corresponds to Magic version 4.

Tutorials to read first:

Magic Tutorial #1: Getting Started
Magic Tutorial #2: Basic Painting and Selection

Commands introduced in this tutorial:

:center :closewindow, :openwindow, :over, :specialopen, :under, :windowpositions

Macros introduced in this tutorial:

o, O, ","

1. Introduction

A window is a rectangular viewport. You can think of it as a magnifying glass that may be moved around on your chip. Magic initially displays a single window on the screen. This tutorial will show you how to create new windows and how to move old ones around. Multiple windows allow you to view several portions of a circuit at the same time, or even portions of different circuits.

Some operations are easier with multiple windows. For example, let's say that you want to paint a very long line, say 3 units by 800 units. With a single window it is hard to align the box accurately since the magnification is not great enough. With multiple windows, one window can show the big picture while other windows show magnified views of the areas where the box needs to be aligned. The box can then be positioned accurately in these magnified windows.

2. Manipulating Windows

2.1. Opening and Closing Windows

Initially Magic displays one large window. The

:openwindow [*cellname*]

command opens another window and loads the given cell. To give this a try, start up Magic with the command **magic tut5a**. Then point anywhere in a Magic window and type the command **:openwindow tut5b** (make sure you're pointing to a Magic window). A new window will appear and it will contain the cell **tut5b**. If you don't give a *cellname* argument to **:openwindow**, it will open a new window on the cell containing the box, and will zoom in on the box. The macro **o** is predefined to **:openwindow**. Try this out by placing the box around an area of **tut5b** and then typing **o**. Another window will appear. You now have three windows, all of which display pieces of layout. There are other kinds of windows in Magic besides layout windows: you'll learn about them later. Magic doesn't care how many windows you have (within reason) nor how they overlap.

To get rid of a window, point to it and type

:closewindow

or use the macro **O**. Point to a portion of the original window and close it.

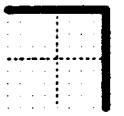
2.2. Resizing and Moving Windows on a Sun-160

If you have been experimenting with Magic while reading this you will have noticed that windows opened by **:openwindow** are all the same size. If you'd prefer a different arrangement you can resize your windows or move them around on the screen. The techniques used for this are different, however, depending on what kind of display you're using. If you're using a Sun-2/160, read this section. If you're using an AED display, Sun-2/120 (with SunColor option), or some other kind of separate color display, skip to the next section.

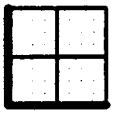
On the Sun-2/160, Magic is a standard window tool, so you can use the standard window operations to re-arrange windows (click the right button inside the window caption and select a menu entry to resize or move). There's also a separate "Magic" menu available from the caption. It has an entry **View**, which is equivalent to the **:view** command, and an entry **Grow**. The **Grow** menu entry will cause the window to fill the entire screen. Once you've grown a window, the **Grow** entry changes to **Shrink**, which will restore the previous screen configuration.

2.3. Resizing and Moving Windows on Standalone Displays

For displays like the AED family, which don't have a built-in window package, Magic implements its own window manager. To re-arrange windows on the screen you can use techniques similar to those you learned for moving the box for painting operations. Point somewhere in the border area of a window, except for the lower left corner, and press and hold the right button. The cursor will change to a shape like this:



This indicates that you have hold of the upper right corner of the window. Point to a new location for this corner and release the button. The window will change shape so that the corner moves. Now point to the border area and press and hold the left button. The cursor will now look like:



This indicates that you have hold of the entire window by its lower left window. Move the cursor and release the button. The window will move so that its lower left corner is where you pointed.

The other button commands for positioning the box by any of its corners also work for windows. Just remember to point to the border of a window before pushing the buttons.

The middle button can be used to grow a window up to full-screen size. To try this, click the middle button over the caption of the window. The window will now fill the entire screen. Click in the caption again and the window will shrink back to its former size.

2.4. Shuffling Windows

By now you know how to open, close, and resize windows. This is sufficient for most purposes, but sometimes you want to look at a window that is covered up by another window. The **:underneath** and **:over** commands help with this.

The **:underneath** command moves the window that you are pointing at underneath all of the other windows. The **:over** command moves the window on top of the rest. Create a few windows that overlap and then use these commands to move them around. You'll see that overlapping windows behave just like sheets of paper: the ones on top obscure portions of the ones underneath.

2.5. Scrolling Windows

Some of the windows have thick bars on the left and bottom borders. These are called *scroll bars*, and the slugs within them are called *elevators*. The size and position of an elevator indicates how much of the layout (or whatever is in the window) is currently visible. If an elevator fills its scroll bar, then all of the layout is visible in that window. If an elevator fills only a portion of the scroll bar, then only that portion of the layout is visible. The position of the elevator indicates which part is visible – if it is near the bottom, you are viewing the bottom part of the layout; if it is near the top, you are viewing the top part of the

layout. There are scroll bars for both the vertical direction (the left scroll bar) and the horizontal direction (the bottom scroll bar).

Besides indicating how much is visible, the scroll bars can be used to change the view of the window. Clicking the middle mouse button in a scroll bar moves the elevator to that position. For example, if you are viewing the lower half of a chip (elevator near the bottom) and you click the middle button near the top of the scroll bar, the elevator will move up to that position and you will be viewing the top part of your chip. The little squares with arrows in them at the ends of the scroll bars will scroll the view by one screenful when the middle button is clicked on them. They are useful when you want to move exactly one screenful. The **:scroll** command can also be used to scroll the view (though we don't think it's as easy to use as the scroll bars). See the man page for information on it.

If you only want to make a small adjustment in a window's view, you can use the command

:center

It will move the view in the window so that the point that used to be underneath the cursor is now in the middle of the window. The macro , is predefined to **:center**.

The bull's-eye in the lower left corner of a window is used to zoom the view in and out. Clicking the left mouse button zooms the view out by a factor of 2, and clicking the right mouse button zooms in by a factor of 2. Clicking the middle button here makes everything in the window visible and is equivalent to the **:view** command.

2.6. Saving Window Configurations

After setting up a bunch of windows you may want to save the configuration (for example, you may be partial to a set of 3 non-overlapping windows). To do this, type:

:windowpositions *filename*

A set of commands will be written to the file. This file can be used with the **:source** command to recreate the window configuration later. (However, this only works well if you stay on the same kind of display; if you create a file on a Sun and then **:source** it on a VAX, you won't get very satisfactory results).

3. How Commands Work Inside of Windows

Each window has a caption at the top. Here is an example:

mychip EDITING shiftcell

This indicates that the window contains the root cell **mychip**, and that a subcell of it called **shiftcell** is being edited. You may remember from the Tutorial #4 that at any given time Magic is editing exactly one cell. If the edit cell is in another window then the caption on this window will read:

mychip [NOT BEING EDITED]

Let's do an example to see how commands are executed within windows. Close any layout windows that you may have on the screen and open two new

windows, each containing the cell **tut5a**. (Use the **:closewindow** and **:openwindow tut5a** commands to do this.) Try moving the box around in one of the windows. Notice that the box also moves in the other window. Windows containing the same root cell are equivalent as far as the box is concerned: if it appears in one it will appear in all, and it can be manipulated from them interchangeably. If you change **tut5a** by painting or erasing portions of it you will see the changes in both windows. This is because both windows are looking at the same thing: the cell **tut5a**. Go ahead and try some painting and erasing until you feel comfortable with it. Try positioning one corner of the box in one window and another corner in another window. You'll find it doesn't matter which window you point to, all Magic knows is that you are pointing to **tut5a**.

These windows are independent in some respects, however. For example, you may scroll one window around without affecting the other window. Use the scrollbars to give this a try. You can also expand and unexpand cells independently in different windows.

We have seen how Magic behaves when both windows view a single cell. What happens when windows view different cells? To try this out load **tut5b** into one of the windows (point to a window and type **:load tut5b**). You will see the captions on the windows change — only one window contains the cell currently being edited. The box cannot be positioned by placing one corner in one window and another corner in the other window because that doesn't really make sense (try it). However, the selection commands work between windows: you can select information in one window and then copy it into another (this only works if the window you're copying into contains the edit cell; if not, you'll have to use the **:edit** command first).

The operation of many Magic commands is dependent upon which window you are pointing at. If you are used to using Magic with only one window you may, at first, forget to point to the window that you want the operation performed upon. For instance, if there are several windows on the screen you will have to point to one before executing a command like **:grid** — otherwise you may not affect the window that you intended!

4. Special Windows

In addition to providing multiple windows on different areas of a layout, Magic provides several special types of windows that display things other than layouts. For example, there are special window types to edit netlists and to adjust the colors displayed on the screen. One of the special window types is described in the section below; others are described in the other tutorials. The

:specialopen *type* [*args*]

command is used to create these sorts of windows. The *type* argument tells what sort of window you want, and *args* describe what you want loaded into that window. The **:openwindow** *cellname* command is really just short for the command **:specialopen layout** *cellname*.

Each different type of window (layout, color, etc.) has its own command set. If you type **:help** in different window types, you'll see that the commands are different. Some of the commands, such as those to manipulate windows, are valid

in all windows, but for other commands you must make sure you're pointing to the right kind of window or the command may be misinterpreted. For example, the **:extract** command means one thing in a layout window and something totally different in a netlist window.

5. Color Editing

Special windows of type **color** are used to edit the red, green, and blue intensities of the colors displayed on the screen. To create a color editing window, invoke the command

:specialopen color [*number*]

Number is optional; if present, it gives the octal value of the color number whose intensities are to be edited. If *number* isn't given, 0 is used. Try opening a color window on color 0.

A color editing window contains 6 "color bars", 12 "color pumps" (one on each side of each bar), plus a large rectangle at the top of the window that displays a swatch of the color being edited (called the "current color" from now on). The color bars display the components of the current color in two different ways. The three bars on the left display the current color in terms of its red, green, and blue intensities (these intensities are the values actually sent to the display). The three bars on the right display the current color in terms of hue, saturation, and value. Hue selects a color of the spectrum. Saturation indicates how diluted the color is (high saturation corresponds to a pure color, low saturation corresponds to a color that is diluted with gray, and a saturation of 0 results in gray regardless of hue). Value indicates the overall brightness (a value of 0 corresponds to black, regardless of hue or saturation).

There are several ways to modify the current color. First, try pressing any mouse button while the cursor is over one of the color bars. The length of the bar, and the current color, will be modified to reflect the mouse position. The color map in the display is also changed, so the colors will change everywhere on the screen that the current color is displayed. Color 0, which you should currently be editing, is the background color. You can also modify the current color by pressing a button while the cursor is over one of the "color pumps" next to the bars. If you button a pump with "+" in it, the value of the bar next to it will be incremented slightly, and if you button the "-" pump, the bar will be decremented slightly. The left button causes a change of about 1% in the value of the bar, and the right button will pump the bar up or down by about 5%. Try adjusting the bars by buttoning the bars and the pumps.

If you press a button while the cursor is over the current color box at the top of the window, one of two things will happen. In either case, nothing happens until you release the button. Before releasing the button, move the cursor so it is over a different color somewhere on the screen. If you pressed the left button, then when the button is released the color underneath the cursor becomes the new current color, and all future editing operations will affect this color. Try using this feature to modify the color used for window borders. If you pressed the right button, then when the button is released the value of the current color is copied from whatever color is present underneath the cursor.

There are only a few commands you can type in color windows, aside from those that are valid in all windows. The command

:color [*number*]

will change the current color to *number*. If no *number* is given, this command will print out the current color and its red, green, and blue intensities. The command

:save [*techStyle displayStyle monitorType*]

will save the current color map in a file named *techStyle.displayStyle.monitorType*.**cmap**, where *techStyle* is the type of technology (e.g., **mos**), *displayStyle* is the kind of display specified by a **styletype** in the **style** section of a technology file (e.g., **7bit**), and *monitorType* is the type of the current monitor (e.g., **std**). If no arguments are given, the current technology style, display style, and monitor type are used. The command

:load [*techStyle displayStyle monitorType*]

will load the color map from the file named *techStyle.displayStyle.monitorType*.**cmap** as above. If no arguments are given, the current technology style, display style, and monitor type are used. When loading color maps, Magic looks first in the current directory, then in the system library.

Magic Tutorial #6: Design-Rule Checking

John Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

This tutorial corresponds to Magic version 4.

Tutorials to read first:

Magic Tutorial #1: Getting Started
Magic Tutorial #2: Basic Painting and Selection
Magic Tutorial #4: Cell Hierarchies

Commands introduced in this tutorial:

:drc

Macro introduced in this tutorial:

y

1. Continuous Design-Rule Checking

When you are editing a layout with Magic, the system automatically checks design rules on your behalf. Every time you paint or erase, and every time you move a cell or change an array structure, Magic rechecks the area you changed to be sure you haven't violated any of the layout rules. If you do violate rules, Magic will display little white dots in the vicinity of the violation. This error paint will stay around until you fix the problem; when the violation is corrected, the error paint will go away automatically. Error paint is written to disk with your cells and will re-appear the next time the cell is read in. There is no way to get rid of it except to fix the violation.

Continuous design-rule checking means that you always have an up-to-date picture of design-rule errors in your layout. There is never any need to run a massive check over the whole design unless you change your design rules. When you make small changes to an existing layout, you will find out immediately if

you've introduced errors, without having to completely recheck the entire layout.

To see how the checker works, run Magic on the cell **tut6a**. This cell contains several areas of metal (blue), some of which are too close to each other or too narrow. Try painting and erasing metal to make the error paint go away and re-appear again.

2. Getting Information about Errors

In many cases, the reason for a design-rule violation will be obvious to you as soon as you see the error paint. However, Magic provides several commands for you to use to find violations and figure what's wrong in case it isn't obvious. All of the design-rule checking commands have the form

:drc *option*

where *option* selects one of several commands understood by the design-rule checker. If you're not sure why error paint has suddenly appeared, place the box around the error paint and invoke the command

:drc why

This command will recheck the area underneath the box, and print out the reasons for any violations that were found. You can also use the macro **y** to do the same thing. Try this on some of the errors in **tut6a**. It's a good idea to place the box right around the area of the error paint: **:drc why** rechecks the entire area under the box, so it may take a long time if the box is very large.

If you're working in a large cell, it may be hard to see the error paint. To help locate the errors, select a cell and then use the command

:drc find [*nth*]

If you don't provide the *nth* argument, the command will place the box around one of the errors in the selected cell, and print out the reason for the error, just as if you had typed **:drc why**. If you invoke the command repeatedly, it will step through all of the errors in the selected cell. (remember, the "." macro can be used to repeat the last long command; this will save you from having to retype **:drc find** over and over again). Try this out on the errors in **tut6a**. If you type a number for *nth*, the command will go to the *nth* error in the selected cell, instead of the next one. If you invoke this command with no cell selected, it searches the edit cell.

A third drc command is provided to give you summary information about errors in hierarchical designs. The command is

:drc count

This command will search every cell (visible or not) that lies underneath the box to see if any have errors in them. For each cell with errors, **:drc count** will print out a count of the number of error areas.

3. Errors in Hierarchical Layouts

The design-rule checker works on hierarchical layouts as well as single cells. There are three overall rules that describe the way that Magic checks hierarchical designs:

[1] The paint in each cell must obey all the design rules by itself, without considering the paint in any other cells, including its children.

[2] The combined paint of each cell and all of its descendants (subcells, sub-subcells, etc.) must be consistent. If a subcell interacts with paint or with other subcells in a way that introduces a design-rule violation, an error will appear in the parent. In designs with many levels of hierarchy, this rule is applied separately to each cell and its descendants.

[3] Each array must be consistent by itself, without considering any other subcells or paint in its parent. If the neighboring elements of an array interact to produce a design-rule violation, the violation will appear in the parent.

To see some examples of interaction errors, edit the cell **tut6b**. This cell doesn't make sense electrically, but illustrates the features of the hierarchical checker. On the left are two subcells that are too close together. In addition, the subcells are too close to the red paint in the top-level cell. Move the subcells and/or modify the paint to make the errors go away and reappear. On the right side of **tut6b** is an array whose elements interact to produce a design-rule violation. Edit an element of the array to make the violation go away. When there are interaction errors between the elements of an array, the errors always appear near one of the four corner elements of the array (since the array spacing is uniform, Magic only checks interactions near the corners; if these elements are correct, all the ones in the middle must be correct too).

It's important to remember that each of the three overall rules must be satisfied independently. This may sometimes result in errors where it doesn't seem like there should be any. Edit the cell **tut6c** for some examples of this. On the left side of the cell there is a sliver of paint in the parent that extends paint in a subcell. Although the overall design is correct, the sliver of paint in the parent is not correct by itself, as required by the first overall rule above. On the right side of **tut6c** is an array with spacing violations between the array elements. Even though the paint in the parent masks some of the problems, the array is not consistent by itself so errors are flagged. The three overall rules are more conservative than strictly necessary, but they reduce the amount of rechecking Magic must do. For example, the array rule allows Magic to deduce the correctness of an array by looking only at the corner elements; if paint from the parent had to be considered in checking arrays, it would be necessary to check the entire array since there might be parent paint masking some errors but not all (as, for example, in **tut6c**).

Error paint appears in different cells in the hierarchy, depending on what kind of error was found. Errors resulting from paint in a single cell cause error paint to appear in that cell. Errors resulting from interactions and arrays appear in the parent of the interacting cells or array. Because of the way Magic makes interaction checks, errors can sometimes "bubble up" through the hierarchy and

appear in multiple cells. When two cells overlap, Magic checks this area by copying all the paint in that area from both cells (and their descendants) into a buffer and then checking the buffer. Magic is unable to tell the difference between an error from one of the subcells and an error that comes about because the two subcells overlap incorrectly. This means that errors in an interaction area of a cell may also appear in the cell's parent. Fixing the error in the subcell will cause the error in the parent to go away also.

4. Turning the Checker Off

We hope that in most cases the checker will run so quickly and quietly that you hardly know it's there. However, there will probably be some situations where the checker is irksome. This section describes several ways to keep the checker out of your hair.

If you're working on a cell with lots of design-rule violations the constant redisplay caused by design-rule checking may get in your way more than it helps. This is particularly true if you're in the middle of a large series of changes and don't care about design-rule violations until the changes are finished. You can stop the redisplay using the command

:see no errors

After this command is typed, design-rule errors will no longer be displayed on the screen. The design-rule checker will continue to run and will store error information internally, but it won't bother you by displaying it on the screen. When you're ready to see errors again, type

:see errors

There can also be times when it's not the redisplay that's bothersome, but the amount of CPU time the checker takes to recheck what you've changed. For example, if a large subcell is moved to overlap another large subcell, the entire overlap area will have to be rechecked, and this could take several minutes. If the prompt on the text screen is a "]" character, it means that the command has completed but the checker hasn't caught up yet. You can invoke new commands while the checker is running, and the checker will suspend itself long enough to process the new commands.

If the checker takes too long to interrupt itself and respond to your commands, you have several options. First, you can hit the break key on the keyboard. This will stop the checker immediately and wait for your next command. As soon as you issue a command or push a mouse button, the checker will start up again. To turn the checker off altogether, type the command

:drc off

From this point on, the checker will not run. Magic will record the areas that need rechecking but won't do the rechecks. If you save your file and quit Magic, the information about areas to recheck will be saved on disk. The next time you read in the cell, Magic will recheck those areas, unless you've still got the checker turned off. There is nothing you can do to make Magic forget about areas to

recheck; **:drc off** merely postpones the recheck operation to a later time.

Once you've turned the checker off, you have two ways to make sure everything has been rechecked. The first is to type the command

:drc catchup

This command will run the checker and wait until everything has been rechecked and errors are completely up to date. When the command completes, the checker will still be enabled or disabled just as it was before the command. If you get tired of waiting for **:drc catchup**, you can always hit the break key to abort the command; the recheck areas will be remembered for later. To turn the checker back on permanently, invoke the command

:drc on

5. Porting Layouts from Other Systems

You should not need to read this section if you've created your layout from scratch using Magic or have read it from CIF using Magic's CIF or Calma reader. However, if you are bringing into Magic a layout that was created using a different editor or an old version of Magic that didn't have continuous checking, read on. You may also need to read this section if you've changed the design rules in the technology file.

In order to find out about errors in a design that wasn't created with Magic, you must force Magic to recheck everything in the design. Once this global recheck has been done, Magic will use its continuous checker to deal with any changes you make to the design; you should only need to do the global recheck once. To make the global recheck, load your design, place the box around the entire design, and type

:drc check

This will cause Magic to act as if the entire area under the box had just been modified: it will recheck that entire area. Furthermore, it will work its way down through the hierarchy; for every subcell found underneath the box, it will recheck that subcell over the area of the box.

If you get nervous that a design-rule violation might somehow have been missed, you can use **:drc check** to force any area to be rechecked at any time, even for cells that were created with Magic. However, this should never be necessary unless you've changed the design rules. If the number of errors in the layout ever changes because of a **:drc check**, it is a bug in Magic and you should notify us immediately.

Magic Tutorial #7: Netlists and Routing

John Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

This tutorial corresponds to Magic version 4.

Tutorials to read first:

Magic Tutorial #1: Getting Started
Magic Tutorial #2: Basic Painting and Selection
Magic Tutorial #3: Advanced Painting (Wiring and Plowing)
Magic Tutorial #4: Cell Hierarchies
Magic Tutorial #5: Multiple Windows

Netlist commands introduced in this tutorial:

:extract, :flush, :ripup, :savenetlist, :trace, :writeall

Layout commands introduced in this tutorial:

:channel, :route

Macros introduced in this tutorial:

(none)

1. Introduction

This tutorial describes how to use Magic's automatic routing tools to make interconnections between subcells in a design. The tools are unusual because they provide an *obstacle-avoidance* capability: if there is mask material in the routing areas, the Magic router can work under, over, or around that material to complete the connections. This means that you can pre-route key signals by hand and have Magic route the less important signals automatically. In addition, you can route power and ground by hand (right now we don't have any power-ground routing tools, so you *have* to route them by hand). Note that Magic's automatic router *only* makes connections between subcells; to make point-to-point

connections between pieces of layout within a single cell you should use the wiring command described in "Magic Tutorial #3: Advanced Painting (Wiring and Plowing) ". Also, the router is designed to make a large number of connections at once; if you only need to make a few connections you are probably better off doing them manually.

The first step in routing is to tell Magic what should be connected to what. This information is contained in a file called a *netlist*. Sections 2, 3, 4, and 5 describe how to create and modify netlists using Magic's interactive netlist editing tools. Once you've created a netlist, the next step is to invoke the router. Section 6 shows how to do this, and gives a brief summary of what goes on inside the routing tools. Unless your design is very simple and has lots of free space, the routing probably won't succeed the first time. Section 7 describes the feedback provided by the routing tools. Sections 8 and 9 discuss how you can modify your design in light of this feedback to improve its routability. You'll probably need to iterate a few times until the routing is successful.

2. Terminals and Netlists

A netlist is a file that describes a set of desired connections. It contains one or more *nets*. Each net names a set of *terminals* that should all be wired together. A terminal is simply a label attached to a piece of mask material within a subcell; it is distinguishable from ordinary labels within a subcell by its presence within a netlist file and by certain characteristics common to terminals, as described below.

The first step in building a netlist is to label the terminals in your design. Figure 1 shows an example. Each label should be a line or rectangle running along the edge of the cell (point terminals are not allowed). The router will make a connection to the cell somewhere along a terminal's length. If the label isn't at the edge of the cell, Magic will route recklessly across the cell to reach the terminal, taking the shortest path between the terminal and a routing channel. It's almost always a good idea to arrange for terminal labels to be at cell edges. The label must be at least as wide as the minimum width of the routing material; the wider you make the label, the more flexibility you give the router to choose a good point to connect to the terminal.

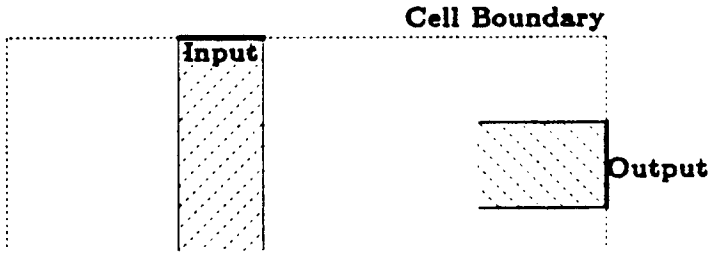


Figure 1. An example of terminal labels. Each terminal should be labelled with a line or rectangle along the edge of the cell.

Terminal labels must be attached to mask material that connects directly to one of Magic's two routing layers (Routing layers are defined in Magic's

technology file). For example, in the SCMOS process where the routing layers are metal1 and metal2, diffusion may not be used as a terminal since neither of the routing layers will connect directly to it. On the other hand, a terminal may be attached to diffusion-metal1 contact, since the metal1 routing layer will connect properly to it. Terminals can have arbitrary names, except that they should not contain slashes ("/") or the substring "feedthrough", and should not end in "@", "$", or "^". See Tutorial #2 for a complete description of labelling conventions.

For an example of good and bad terminals, edit the cell **tut7a**. The cell doesn't make any electrical sense, but contains several good and bad terminals. All the terminals with names like **bad1** are incorrect or undesirable for one of the reasons given above, and those with names like **good4** are acceptable.

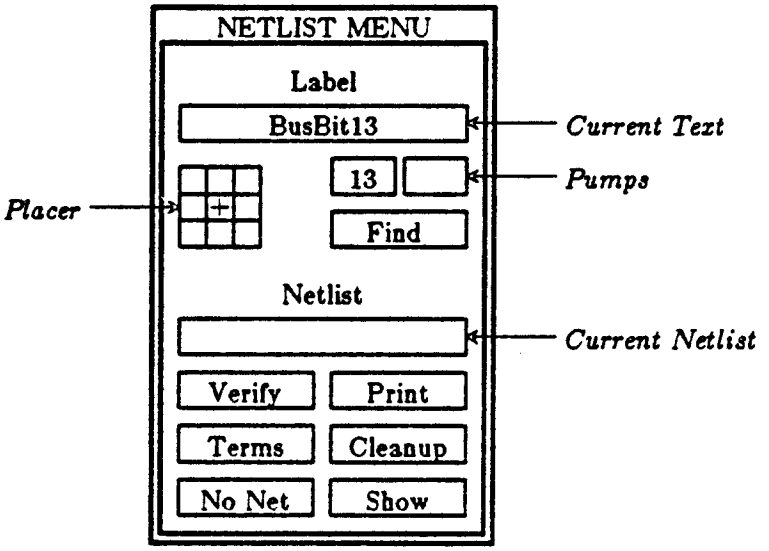


Figure 2. The netlist menu.

If you create two or more terminal labels with the same name in the same cell the router will assume that they are electrically equivalent (connected together within the cell). Because of this, when routing the net it will feel free to connect to whichever one of the terminals is most convenient, and ignore the others. In some cases the router may take advantage of electrically equivalent terminals by using *feed throughs*: entering a cell at one terminal to make one connection, and exiting through an equivalent terminal on the way to make another connection for the same net.

3. Menu for Label Editing

Magic provides a special menu facility to assist you in placing terminal labels and editing netlists. To make the menu appear, invoke the Magic command

Button	Action
Current Text	Left-click: prompt for more labels Right-click: advance to next label
Placer	Left-click: place label Right-click: change label text position
Pumps	Left-click: decrement number Right-click: increment number
Find	Search under box, highlight labels matching current text
Current Netlist	Left-click: prompt for new netlist name Right-click: use edit cell name as netlist name
Verify	Check that wiring matches netlist (same as typing **:verify** command)
Print	Print names of all terminals in selected net (same as typing **:print** command)
Terms	Place feedback areas on screen to identify all terminals in current netlist (same as **:showterms** command)
Cleanup	Check current netlist for missing labels and nets with less than two terminals (same as typing **:cleanup** command)
No Net	Delete selected net (same as **:dnet** command)
Show	Highlight paint connected to material under box (same as typing **:shownet** command)

Table I. A summary of all the netlist menu button actions.

:specialopen netlist

A new window will appear in the lower-left corner of the screen, containing several rectangular areas on a purple background. Each of the rectangular areas is called a *button*. Clicking mouse buttons inside the menu buttons will invoke various commands to edit labels and netlists. Figure 2 shows a diagram of the netlist menu and Table I summarizes the meaning of button clicks in various menu items. The netlist menu can be grown, shrunk, and moved just like any other window; see "Magic Tutorial #5: Multiple Windows" for details. It also has its own private set of commands. To see what commands you can type in the netlist menu, move the cursor over the menu and type

:help

You shouldn't need to type commands in the netlist menu very often, since almost everything you'll need to do can be done using the menu. See Section 9 for a description of a few of the commands you can type; the complete set is described in the manual page *magic(1)*. One of the best uses for the commands is so that you can define macros for them and avoid having to go back and forth to the menu; look up the **:send** command in the man page to see how to do this. The top half of the menu is for placing labels and the bottom half is for editing

netlists. This section describes the label facilities, and Section 4 describes the netlist facilities.

The label menu makes it easy for you to enter lots of labels, particularly when there are many labels that are the same except for a number, e.g. **bus1**, **bus2**, **bus3**, etc. There are four sections to the label menu: the current text, the placer, two pumps, and the **Find** button. To place labels, first click the left mouse button over the current text rectangle. Then type one or more labels on the keyboard, one per line. You can use this mechanism to enter several labels at once. Type return twice to signal the end of the list. At this point, the first of the labels you typed will appear in the current text rectangle.

To place a label, position the box over the area you want to label, then click the left mouse button inside one of the squares of the placer area. A label will be created with the current text. Where you click in the placer determines where the label text will appear relative to the label box: for example, clicking the left-center square causes the text to be centered just to the left of the box. You can place many copies of the same label by moving the box and clicking the placer area again. You can re-orient the text of a label by clicking the right mouse button inside the placer area. For example, if you would like to move a label's text so that it appears centered above the label, place the box over the label and right-click the top-center placer square.

If you entered several labels at once, only the first appears in the current text area. However, you can advance to the next label by right-clicking inside the current text area. In this way you can place a long series of labels entirely with the mouse. Try using this mechanism to add labels to **tut7a**.

The two small buttons underneath the right side of the current text area are called pumps. To see how these work, enter a label name containing a number into the current text area, for example, **bus1**. When you do this, the "1" appears in the left pump. Right-clicking the pump causes the number to increment, and left-clicking the pump causes the number to decrement. This makes it easy for you to enter a series of numbered signal names. If a name has two numbers in it, the second number will appear in the second pump, and it can be incremented or decremented too. Try using the pumps to place a series of numbered names.

The last entry in the label portion of the menu is the **Find** button. This can be used to locate a label by searching for a given pattern. If you click the **Find** button, Magic will use the current text as a pattern and search the area underneath the box for a label whose name contains the pattern. Pattern-matching is done in the same way as in *csh*, using the special characters "*", "?", "\", "[", and "]". Try this on **tut7a**: enter "good*" into the current text area, place the box around the whole cell, then click on the "Find" button. For each of the good labels, a feedback area will be created with white stripes to highlight the area. The **:feedback find** command can be used to step through the areas, and **:feedback clear** will erase the feedback information from the screen. The **:feedback** command has many of the same options as **:drc** for getting information about feedback areas; see the Magic manual page for details, or type **:feedback help** for a synopsis of the options.

4. Netlist Editing

After placing terminal labels, the next step is to specify the connections between them; this is called netlist editing. The bottom half of the netlist menu is used for editing netlists. The first thing you must do is to specify the netlist you want to edit. Do this by clicking in the current netlist box. If you left-click, Magic will prompt you for the netlist name and you can type it at the keyboard. If you right-click, Magic will use the name of the edit cell as the current netlist name. In either case, Magic will read the netlist from disk if it exists and will create a new netlist if there isn't currently a netlist file with the given name. Netlist files are stored on disk with a ".net" extension, which is added by Magic when it reads and writes files. You can change the current netlist by clicking the current netlist button again. Startup Magic on the cell **tut7b**, open the netlist menu, and set the current netlist to **tut7b**. Then expand the subcells in **tut7b** so that you can see their terminals.

Button	Action
Left	Select net, using nearest terminal to cursor.
Right	Toggle nearest terminal into or out of current net.
Middle	Find nearest terminal, join its net with the current net.

Table II. The actions of the mouse buttons when the terminal tool is in use.

Netlist editing is done with the netlist tool. If you haven't already read "Tutorial #3: Advanced Painting (Wiring and Plowing)", you should read it now, up through Section 2.1. Tutorial #3 explained how to change the current tool by using the space macro or by typing **:tool**. Switch tools to the netlist tool (the cursor will appear as a thick square).

When the netlist tool is in use the left, right, and middle buttons invoke select, toggle, and join operations respectively (see Table II). To see how they work, move the cursor over the terminal **right4** in the top subcell of **tut7b** and click the left mouse button (you may have to zoom in a bit to see the labels; terminals are numbered in clockwise order: **right4** is the fourth terminal from the top on the right side). This causes the net containing that terminal to be selected. Three hollow white squares will appear over the layout, marking the terminals that are supposed to be wired together into **right4**'s net. Left-click over the **left3** terminal in the same subcell to select its net, then select the **right4** net again.

The right button is used to toggle terminals into or out of the current net. If you right-click over a terminal that is in the current net, then it is removed from the current net. If you right-click over a terminal that isn't in the current net, it is added to the current net. A single terminal can only be in one net at a time, so if a terminal is already in a net when you toggle it into another net then Magic will remove it from the old net. Toggle the terminal **top4** in the bottom cell out of, then back into, the net containing **right4**. Now toggle **left3** in the bottom cell into this net. Magic warns you because it had to remove **left3** from another

net in order to add it to **right4**'s net. Type **u** to undo this change, then left-click on **left3** to make sure it got restored to its old net by the undo. All of the netlist-editing operations are undo-able.

The middle button is used to merge two nets together. If you middle-click over a terminal, all the terminals in its net are added to the current net. Play around with the three buttons to edit the netlist **tut7b**.

Note: the router does not make connections to terminals in the top level cell. It only works with terminals in subcells, or sub-subcells, etc. Because of this, the netlist editor does not permit you to select terminals in the top level cell. If you click over such a terminal Magic prints an error message and refuses to make the selection.

If you left-click over a terminal that is not currently in a net, Magic creates a new net automatically. If you didn't really want to make a new net, you have several choices. Either you can toggle the terminal out of its own net, you can undo the select operation, or you can click the **No Net** button in the netlist menu (you can do this even while the cursor is in the square shape). The **No Net** button removes all terminals from the current net and destroys the net. It's a bad idea to leave single-net terminals in the netlist: the router will treat them as errors.

There are two ways to save netlists on disk; these are similar to the ways you can save layout cells. If you type

:savenetlist [*name*]

with the cursor over the netlist menu, the current netlist will be saved on disk in the file *name*.**net**. If no *name* is typed, the name of the current netlist is used. If you type the command

:writeall

then Magic will step through all the netlists that have been modified since they were last written, asking you if you'd like them to be written out. If you try to leave Magic without saving all the modified netlists, Magic will warn you and give you a chance to write them out.

If you make changes to a netlist and then decide you don't want them, you can use the **:flush** netlist command to throw away all of the changes and re-read the netlist from its disk file. If you create netlists using a text editor or some other program, you can use **:flush** after you've modified the netlist file in order to make sure that Magic is using the most up-to-date version.

The **Print** button in the netlist menu will print out on the text screen the names of all the terminals in the current net. Try this for some of the nets in **tut7b**. The official name of a terminal looks a lot like a Unix file name, consisting of a bunch of fields separated by slashes. Each field except the last is the id of a subcell, and the last field is the name of the terminal. These hierarchical names provide unique names for each terminal, even if the same terminal name is re-used in different cells or if there are multiple copies of the same cell.

The **Verify** button will check the paint of the edit cell to be sure it implements the connections specified in the current netlist. Feedback areas are created to show nets that are incomplete or nets that are shorted together.

The **Terms** button will cause Magic to generate a feedback area over each of the terminals in the current netlist, so that you can see which terminals are included in the netlist. If you type the command **:feedback clear** in a layout window then the feedback will be erased.

The **Cleanup** button is there as a convenience to help you cleanup your netlists. If you click on it, Magic will scan through the current netlist to make sure it is reasonable. **Cleanup** looks for two error conditions: terminal names that don't correspond to any labels in the design, and nets that don't have at least two terminals. When it finds either of these conditions it prints a message and gives you the chance to either delete the offending terminal (if you type **dterm**), delete the offending net (**dnet**), skip the current problem without modifying the netlist and continue looking for other problems (**skip**), or abort the **Cleanup** command without making any more changes (**abort**).

The **Show** button provides an additional mechanism for displaying the paint in the net. If you place the box over a piece of paint and click on **Show**, Magic will highlight all of the paint in the net under the box. This is similar to pointing at the net and typing **s** three times to select the net, except that **Show** doesn't select the net (it uses a different mechanism to highlight it), and **Show** will trace through all cells, expanded or not (the selection mechanism only considers paint in expanded cells). Once you've used **Show** to highlight a net, the only way to make the highlighting go away is to place the box over empty space and invoke **Show** again. **Show** is an old command that pre-dates the selection interface, but we've left it in Magic because some people find it useful.

5. Netlist Files

Netlists are stored on disk in ordinary text files. You are welcome to edit those files by hand or to write programs that generate the netlists automatically. For example, a netlist might be generated by a schematic editor or by a high-level simulator. See the manual page *net(5)* for a description of netlist file format.

6. Running the Router

Once you've created a netlist, it is relatively easy to invoke the router. First, place the box around the area you'd like Magic to consider for routing. No terminals outside this area will be considered, and Magic will not generate any paint more than a few units outside this area (Magic will may use the next routing grid line outside the area). Load **tut7b**, **:flush** the netlist if you made any changes to it, set the box to the bounding box of the cell, and then invoke the command

:route

When this command completes, the netlist should be routed. Click the **Verify**

netlist button to make sure the connections were made correctly. Try deleting a piece from one of the wires and verify again. Feedback areas should appear to indicate where the routing was incorrect. Use the **:feedback** command to step through the areas and, eventually, to delete the feedback (**:feedback help** gives a synopsis of the command options).

All of the wires placed by the router are of the same width, so the router won't be very useful for power and ground wiring. Instead, you should wire power and ground by hand before running the router. The router will be able to work around your hand-placed connections to make the connections in the netlist. If there are certain key signals that you want to wire carefully by hand, you can do this too; the router will work around them. Signals that you route by hand should not be in the netlist. **Tutorial7b** has an example of "hand routing" in the form of a piece of metal in the middle of the circuit. Undo the routing, and try modifying the metal and/or adding more hand routing of your own to see how it affects the routing.

You've probably noticed by now that the router sometimes generates unnecessary wiring, such as inserting extra jogs and U-shapes in wires (look next to **right3** in the top cell). These jogs are particularly noticeable in small examples. However, the router actually does *better* on larger examples: there will still be a bit of extra wire, but it's negligible in comparison to the total wire length on a large chip. We hope to change Magic to reduce the amount of extra wire it generates, but some of it is necessary and important: it helps the router to avoid several problem situations that would cause it to fail on more difficult examples. For the present, use the **straighten** command described in "Magic Tutorial #3: Advanced Painting (Wiring and Plowing)" to remove unnecessary jogs. Please don't judge the router by its behavior on small examples. On the other hand, if it does awful things on big examples, we'd like to know about it.

If the router is unable to complete the connections, it will report errors to you (you can make this happen in **tut7b** by hand-routing on both routing layers to block some of the terminals). Errors may be reported in several ways. For some errors, such as non-existent terminal names, messages will be printed. For other errors, cross-hatched feedback areas will be created. Most of the feedback areas have messages similar to "Net shifter/bit[0]/phi1: Can't make bottom connection." To see the message associated with a feedback area, place the box over the feedback area and type **:feedback why**. In this case the message means that for some reason the router was unable to connect the specified net (named by one of its terminals) within one of the routing channel. The terms "bottom", "top", etc. may be misnomers because Magic sometimes rotates channels before routing: the names refer to the direction at the time the channel was routed, not the direction in the circuit. However, the location of the feedback area indicates where the connection was supposed to have been made.

The **route** command has a number of options useful for getting information about the routing and setting routing parameters. You need to invoke the **route** command once for each option you want to specify; then type **:route** with no options to start up the router with whatever parameters you've set. Type **:route netlist** *file* to specify a netlist for the routing without having to open up the

netlist menu. The **metal** option lets you toggle metal maximization on and off; if metal mazimization is turned on, the router converts routing from the alternate routing layer ("poly") to the preferred routing layer ("metal") wherever possible. The **via** option controls metal maximization by specifying how many grid units of "metal" conversion make it worthwhile to place vias; setting this to 5 means that metal maximization will add extra vias only if 5 or more grid units of "poly" can be converted to "metal". View the current technology's router parameters with the **tech** option. The **jog**, **obstacle**, and **steady** options let you view and change parameters to control the channel router (this feature is for advanced users). Finally, show all parameter values with the **settings** option. The options and their actions are summarized in Table III.

Option	Action
end	Print the channel router end constant
end *real*	Set the channel router end constant
help	Print a summary of the router options
jog	Print the channel router minimum jog length
jog *int*	Set the minimum jog length, measured in grid units
metal	Toggle metal maximization on or off
netlist	Print the name of the current net list
netlist *file*	Set the current net list
obstacle	Print the channel router obstacle constant
obstacle *real*	Set the obstacle constant
settings	Print a list of all router parameters
steady	Print the channel router steady net constant
steady *int*	Set the steady net constant, measured in grid units
tech	Print router technology information
vias	Print the metal maximization via limit
vias *int*	Set the via limit

Table III. A summary of all of the router options.

7. How the Router Works

In order to make the router produce the best possible results, it helps to know a little bit about how it works. The router runs in three stages, called *channel definition*, *global routing*, and *channel routing*. In the channel definition phase, Magic divides the area of the edit cell into rectangular routing areas called channels. The channels cover all the space under the box except the areas occupied by subcells. All of Magic's routing goes in the channel areas, except that stems (Section 8.2) may extend over subcells.

To see the channel structure that Magic chose, place the box in **tut7b** as if you were going to route, then type the command

:channel

in the layout window. Magic will compute the channel structure and display it on the screen as a collection of feedback areas. In this case, each feedback area is displayed as a white rectangle. The feedback areas make it hard to see where the box is, so type **:feedback clear** when you're through looking at them.

The second phase of routing is global routing. In the global routing phase, Magic considers each net in turn and chooses the sequence of channels the net must pass through in order to connect its terminals. The *crossing points* (places where the net crosses from one channel to another) are chosen at this point, but not the exact path through each channel.

In the third phase, each channel is considered separately. All the nets passing through that channel are examined at once, and the exact path of each net is decided. Once the routing paths have been determined, paint is added to the edit cell to implement the routing.

The Magic router is grid-based: it places all its wires on a uniform grid. For the standard nMOS process the grid spacing is 7 units, and for the standard SCMOS process it is 8 units. If you type **:grid 8** after routing **tut7b**, you'll see that all of the routing lines up with its lower and left sides on grid lines. Fortunately, you don't have to make your cell terminals line up on even grid boundaries. During the routing Magic generates *stems* that connect your terminals up to grid lines at the edges of channels. Notice that there's space left by Magic between the subcells and the channels; this space is used by the stem generator.

8. What to do When the Router Fails

Don't be surprised if the router is unable to make all the connections the first time you try it on a large circuit. Unless you have extra routing space in your chip, you may have to make slight re-arrangements to help the router out. The paragraphs below describe things you can do to make life easier for the router. This section is not very well developed, so we'd like to hear about techniques you use to improve routability. If you discover new techniques, send us mail and we'll add them to this section.

8.1. Channel Structure

One of the first things to check when the router fails is the channel structure. Use the **:channel** command to look at the channels. One common mistake is to have some of the desired routing area covered by subcells; Magic only runs wires where there are no subcells. Check to be sure that there are channels everywhere that you're expecting wires to run. If you place cells too close together, there may not be enough room to have a channel between the cells; when this happens Magic will route willy-nilly across the tops of cells to bring terminals out to channels, and will probably generate shorts or design-rule violations. To solve the problem, move the cells farther apart. If there are many skinny channels, it will be difficult for the router to produce good routing. Try to re-arrange the cell structure to line up edges of nearby cells so that there are as few channels as possible and they are as large as possible (before doing this you'll probably want

to get rid of the existing routing by undo-ing or by flushing the edit cell).

8.2. Stems

Another problem has to do with the stem generator. Stems are the pieces of wiring that connect terminals up to grid points on the edges of channels. The current stem generation code doesn't know about connectivity or design rules. It simply finds the nearest routing grid point and wires out to that point, without considering any other terminals. If a terminal is not on the edge of the cell, the stem runs straight across the cell to the nearest channel, without any consideration for other material in the cell. If two terminals are too close together, Magic may decide to route them both to the same grid point. When this happens, you have two choices. Either you can move the cell so that the terminals have different nearest grid points (for example, you can line its terminals up with the grid lines), or if this doesn't work you'll have to modify the cell to make the terminals farther apart.

The place where stems cause the most trouble is in PLAs, many of which have been optimized to space the outputs as closely together as possible. In some cases the outputs are closer together than the routing grid, which is an impossible situation for the stem generator. In this case, we think the best approach is to change the PLA templates to space the outputs farther apart. Either space them exactly the same as the router grid (in which case you can line the PLAs up before routing so the terminals are already on the grid), or space the outputs at least 1.5 grid units apart so the stem generator won't have troubles. Having tightly-spaced PLA outputs is false economy: it makes it more difficult to design the PLAs and results in awful routing problems. Even if Magic could river-route out from tightly-spaced terminals to grid lines (which it can't), it would require N^2 space to route out N lines; it takes less area to stretch the PLA.

8.3. Obstacles

The router tends to have special difficulties with obstacles running along the edges of channels. When you've placed a power wire or other hand-routing along the edge of a channel, the channel router will often run material under your wiring in the other routing layer, thereby blocking both routing layers and making it impossible to complete the routing. Where this occurs, you can increase the chances of successful routing by moving the hand-routing away from the channel edges. It's especially important to keep hand-routing away from terminals. The stem generator will not pay any attention to hand-routing when it generates stems (it just makes a bee-line for the nearest grid point), so it may accidentally short a terminal to nearby hand-routing.

When placing hand-routing, you can get better routing results by following the advice illustrated in Figure 3. First, display the routing grid. For example, if the router is using a 8-unit grid (which is true for the standard SCMOS technology), type **:grid 8**. Then place all your hand routing with its left and bottom edges along the grid lines. Because of the way the routing tools work, this approach results in the least possible amount of lost routing space.

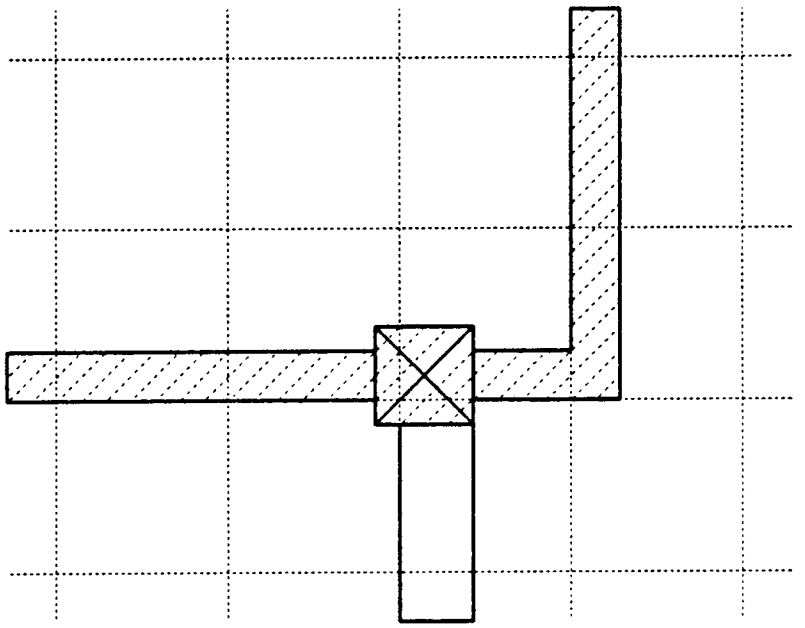


Figure 3. When placing hand routing, it is best to place wires with their left and bottom edges along grid lines, and contacts centered on the wires. In this fashion, the hand routing will block as few routing grid lines as possible.

9. More Netlist Commands

In addition to the netlist menu buttons and commands described in Section 4, there are a number of other netlist commands you can invoke by typing in the netlist window. Many of these commands are textual equivalents of the menu buttons. However, they allow you to deal with terminals by typing the hierarchical name of the terminal rather than by pointing to it. If you don't know where a terminal is, or if you have deleted a label from your design so that there's nothing to point to, you'll have to use the textual commands. Commands that don't just duplicate menu buttons are described below; see the *magic(1)* manual page for details on the others.

The netlist command

:extract

will generate a net from existing wiring. It looks under the box for paint, then traces out all the material in the edit cell that is connected electrically to that paint. Wherever the material touches subcells it looks for terminals in the subcells, and all the terminals it finds are placed into a new net. Warning: there is also an **extract** command for layout windows, and it is totally different from the **extract** command in netlist windows. Make sure you've got the cursor over the netlist window when you invoke this command!

The netlist editor provides two commands for ripping up existing routing (or other material). They are

:ripup
:ripup netlist

The first command starts by finding any paint in the edit cell that lies underneath the box. It then works outward from that paint to find all paint in the edit cell that is electrically connected to the starting paint. All of this paint is erased. (**:ripup** isn't really necessary, since the same effect can be achieved by selecting all the paint in the net and deleting the selection; it's a hangover from olden days when there was no sclection). The second form of the command, **:ripup netlist**, is similar to the first except that it starts from each of the terminals in the current netlist instead of the box. Any paint in the edit cell that is electrically connected to a terminal is erased. The **:ripup netlist** command may be useful to ripup existing routing before rerouting.

The command

:trace [*name*]

provides an additional facility for examining router feedback. It highlights all paint connected to each terminal in the net containing *name*, much as the **Show** menu button does for paint connected to anything under the box. The net to be highlighted may be specified by naming one of its terminals, for example, **:trace shifter/bit[0]/phi1**. Use the trace command in conjunction with the nets specified in router feedback to see the partially completed wiring for a net. Where no net is specified, the **:trace** command highlights the currently selected net.

Magic Tutorial #8: Circuit Extraction

Walter Scott

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

This tutorial corresponds to Magic version 4.

Tutorials to read first:

Magic Tutorial #1: Getting Started
Magic Tutorial #2: Basic Painting and Selection
Magic Tutorial #4: Cell Hierarchies

Commands introduced in this tutorial:

:extract

Macros introduced in this tutorial:

none

Programs introduced in this tutorial:

ext2sim

1. Introduction

This tutorial covers the use of Magic's incremental and hierarchical circuit extractor. The extractor is incremental: only part of the entire layout must be re-extracted after each change. Because it is hierarchical, the structure of the extracted circuit parallels the structure of the layout being extracted. The extractor produces a separate **.ext** file for each **.mag** file in a hierarchical design. This is in contrast to previous extractors, such as Mextra, which produces a single **.sim** file that represents the flattened (fully-instantiated) layout.

Sections 2 and 3 introduce Magic's **:extract** command. Section 4 describes what information actually gets extracted, and discusses limitations and inaccuracies. Section 5 talks about extraction styles. Although the hierarchical **.ext** format fully describes the circuit implemented by a layout, none of our tools

yet accept it. Instead, it is necessary to convert it to the flat **.sim** format using the **ext2sim** program described in Section 6.

2. Basic Extraction

You can use Magic's extractor in one of several ways. Normally it is not necessary to extract all cells in a layout. To extract only those cells that have changed since they were extracted, use:

:load *root*
:extract

It looks for a **.ext** file for every cell in the tree that descends from the cell *root*. The **.ext** file is searched for in the same directory as the one containing the cell's **.mag** file. Any cells that have been modified since they were last extracted, and all of their parents, are re-extracted. If a cell has no **.ext** file, this also causes it to be re-extracted.

To try out the extractor on an example, copy all the **tut8***x* cells to your current directory with the following shell command:

cp cad/lib/magic/tutorials/tut8*.mag .

Start magic on the cell **tut8a** and type **:extract**. Magic will print the name of each cell as it extracts it. Now type **:extract** a second time. This time nothing gets printed, since Magic didn't have to re-extract any cells. Now delete the piece of poly labelled "**delete me**" and type **:extract** again. This time, only the cell **tut8a** is extracted as it is the only one that changed. If you make a change to cell **tut8b** (do it) and then extract again, both **tut8b** and **tut8a** will be re-extracted, since **tut8a** is the parent of **tut8b**.

To force all cells in the subtree rooted at cell *root* to be re-extracted, use **:extract all**:

:load *root*
:extract all

Try this also on **tut8a**.

You can also use the **:extract** command to extract a single cell as follows:

:extract cell *name*

will extract just the selected (current) cell, and place the output in the file *name*. If more than one cell is selected, one is chosen at random. You should be careful about using this form of the **:extract** command, since even though you may only make a change to a child cell, all of its parents may have to be re-extracted. To re-extract all of the parents of the selected cell, you may use

:extract parents

Finally, to see what cells would be extracted by **:extract parents** without actually extracting them, use

:extract showparents

Try these commands.

3. Feedback: Errors and Warnings

When the extractor encounters problems, it leaves feedback in the form of stippled white rectangular areas on the screen. Each area covers the portion of the layout that caused the error. Each area also has an error message associated with it, which you can see by using the **:feedback** command. (Type **:feedback help** while in Magic for assistance in using the **:feedback** command.)

The extractor will always report extraction errors. These are problems in the layout that may cause the output of the extractor to be incorrect. The layout should be fixed to eliminate extraction errors before attempting to simulate the circuit.

Extraction errors can come from violations of transistor rules. There are two rules about the formation of transistors: no transistor can be formed, and none can be destroyed, as a result of cell overlaps. For example, it is illegal to have poly in one cell overlap diffusion in another cell, as that would form a transistor in the parent where none was present in either child. It is also illegal to have a buried contact in one cell overlap a transistor in another, as this would destroy the transistor. Violating these transistor rules will cause design-rule violations as well as extraction errors.

In general, it is an error for material of two types on the same plane to overlap or abut if they don't connect to each other. For example, in CMOS it is illegal for p-diffusion and n-diffusion to overlap or abut.

In addition to errors, the extractor can give warnings. If only warnings are present, the extracted circuit is simulatable. Normally, warnings are not reported or displayed as feedback. To cause them all to be displayed, use **:extract warn all**. The command **:extract warn** *warning* may be used to enable specific warnings selectively; see below. To revert to the default of not displaying warnings, use **:extract warn no all**; **:extract warn no** *warning* disables display of the particular *warning*.

Three different kinds of warnings are generated. The **dup** warning checks to see whether you have two electrically unconnected nodes in the same cell labelled with the same name. If so, you are warned because the two unconnected nodes could appear to be connected in the resulting **.ext** file, which means that the extracted circuit would not represent the actual layout. When two unconnected nodes share the same label name, the extractor leaves feedback squares over each instance of the shared name.

The **fets** warning checks to see whether any transistors have fewer diffusion terminals than the minimum for their types. For example, a **dfet** in nMOS must have two diffusion terminals: a source and a drain. If a capacitor with only one diffusion terminal is desired, **dcap** should be used instead. This warning is a consistency check for shorted transistor terminals, or for unconnected terminals on transistors.

The third warning, **labels**, is generated if you violate the following guideline for placement of labels. Whenever geometry from two subcells abuts or overlaps, you should make sure that there is a label attached to the geometry in each subcell, *in the area of the overlap, or along the line of abutment.* If you do not follow this guideline, the extractor will still work correctly, but will run slower.

Load the cell **tut8e**, expand all its children, and enable all extractor warnings with **:extract warn all**. Now extract **tut8e** and all of its children with **:extract**, and examine the feedback for examples of fatal errors and warnings.

4. What Gets Extracted; Limitations

Magic's extractor computes from the layout the information needed to run simulation tools such as *crystal*(1) and *esim*(1). This information includes the sizes and shapes of transistors, and the connectivity, resistance, and parasitic capacitance of nodes. Both capacitance to substrate and several kinds of internodal coupling capacitances are extracted.

The details of the **.ext** files output by Magic may be found in the manual page *ext*(5). "Magic Maintainer's Manual #2: The Technology File" describes how extraction parameters are specified for the extractor. The remainder of this section is intended as a description of the information that gets extracted, and the limitations of the extractor.

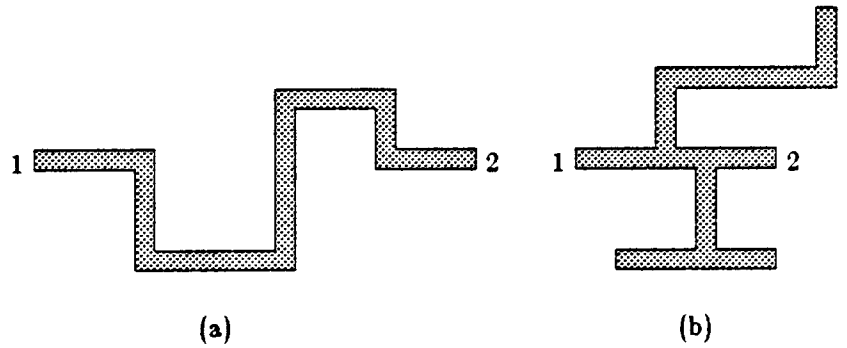


Figure 1. Magic approximates the resistance of a node by assuming that it is a simple rectangular region. The perimeter and area of such a region are used to compute its length and width. (a) For non-branching nodes, this approximation is a good one. (b) The computed resistance for this node is the same as for (a) because the side branches are counted, yet the actual resistance between points 1 and 2 is significantly less than in (a).

4.1. Resistance

Magic extracts a lumped resistance for each node, rather than a point-to-point resistance between each pair of devices connected to that node. The result is that all such point-to-point resistances are approximated by the worst-case resistance between any two points in that node.

Node resistances are approximated rather than computed exactly. For a node comprised entirely of a single type of material, Magic will compute the

node's total perimeter and area. It then solves a quadratic equation to find the width and height of a simple rectangle with this same perimeter and area, and approximates the resistance of the node as the resistance of this "equivalent" rectangle. The resistance is always taken in the longer dimension of the rectangle. When a node contains more than a single type of material, Magic computes an equivalent rectangle for each type, and then sums the resistances as though the rectangles were laid end-to-end.

This approximation for resistance does not take into account any branching, so it can be significantly in error for nodes that have side branches. Figure 1 gives an example. For global signal trees such as clocks or power, Magic's estimate of resistance will likely be several times higher than the actual resistance between two points.

The approximated resistance also does not lend itself well to hierarchical adjustments, as does capacitance. To allow programs like **ext2sim** to incorporate hierarchical adjustments into a resistance approximation, the each node in the **.ext** file also contains a perimeter and area for each "resistance class" that was defined in the technology file (see "Maintainer's Manual #2: The Technology File," and *ext*(5)). When flattening a circuit, **ext2sim** uses this information along with adjustments to perimeter and area to produce the value it actually uses for node resistance.

If you wish to disable the extraction of resistances and node perimeters and areas, use the command **:extract no resistance**. This will cause all node resistances, perimeters, and areas in the **.ext** file to be zero. To re-enable extraction of resistance, use the command **:extract do resistance**.

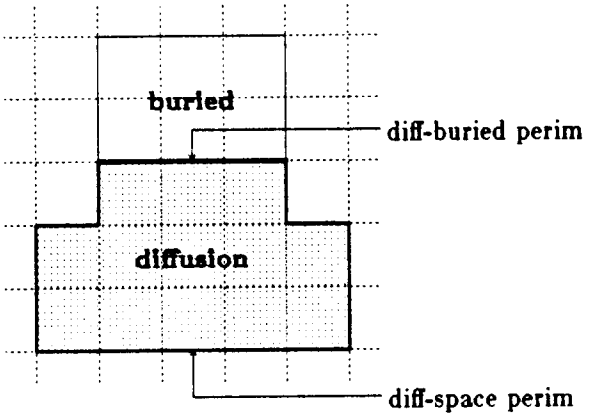


Figure 2. Each type of edge has capacitance to substrate per unit length. Here, the diffusion-space perimeter of 13 units has one value per unit length, and the diffusion-buried perimeter of 3 units another. In addition, each type of material has capacitance per unit area.

4.2. Capacitance

Capcitance to substrate comes from two different sources. Each type of material has a capacitance to substrate per unit area. Each type of edge (i.e, each pair of types) has a capacitance to substrate per unit length. See Figure 2. The computation of capacitance may be disabled with **:extract no capacitance**,

which causes all substrate capacitance values in the **.ext** file to be zero. It may be re-enabled with **:extract do capacitance**.

Internodal capacitance comes from three sources, as shown in Figure 3. When materials of two different types overlap, the capacitance to substrate of the one on top (as determined by the technology) is replaced by an internodal capacitance to the one on the bottom. Its computation may be disabled with **:extract no coupling**, which will also cause the extractor to run 30% to 50% faster. Extraction of coupling capacitances can be re-enabled with **:extract do coupling**.

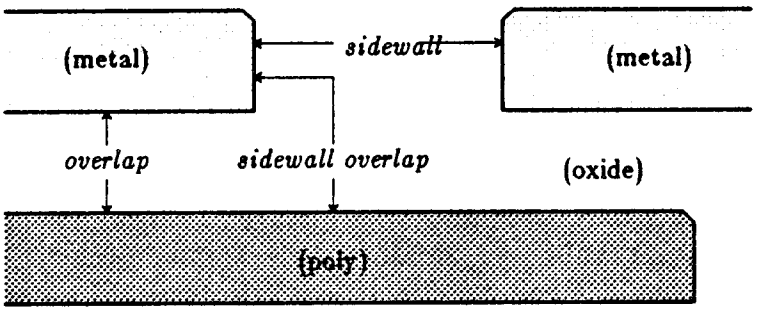


Figure 3. Magic extracts three kinds of internodal coupling capacitance. This figure is a cross-section (side view, not a top view) of a set of masks that shows all three kinds of capacitance. *Overlap* capacitance is parallel-plate capacitance between two different kinds of material when they overlap. *Sidewall* capacitance is parallel-plate capacitance between the vertical edges of two pieces of the same kind of material. *Sidewall overlap* capacitance is orthogonal-plate capacitance between the vertical edge of one piece of material and the horizontal surface of another piece of material that overlaps the first edge.

Whenever material from two subcells overlaps or abuts, the extractor computes adjustments to substrate capacitance, coupling capacitance, and node perimeter and area. Often, these adjustments make little difference to the type of analysis you are performing, as when you wish only to compare netlists. Even when running Crystal for timing analysis, the adjustments can make less than a 5% difference in the timing of critical paths in designs with only a small amount of inter-cell overlap. To disable the computation of these adjustments, use **:extract no adjustment**, which will result in approximately 50% faster extraction. (This speedup is not entirely additive with the speedup resulting from **:extract no coupling**). To re-enable computation of adjustments, use **:extract do adjustment**.

4.3. Transistors

Like the resistances of nodes, the lengths and widths of transistors are approximated. Magic computes the contribution to the total perimeter by each of the terminals of the transistor. See Figure 4. For rectangular transistors, this yields an exact L/W. For non-branching, non-rectangular transistors, it is still possible to approximate L/W fairly well, but substantial inaccuracies can be introduced if the channel of a transistor contains branches. Since most transistors are rectangular, however, Magic's approximation works well in practice.

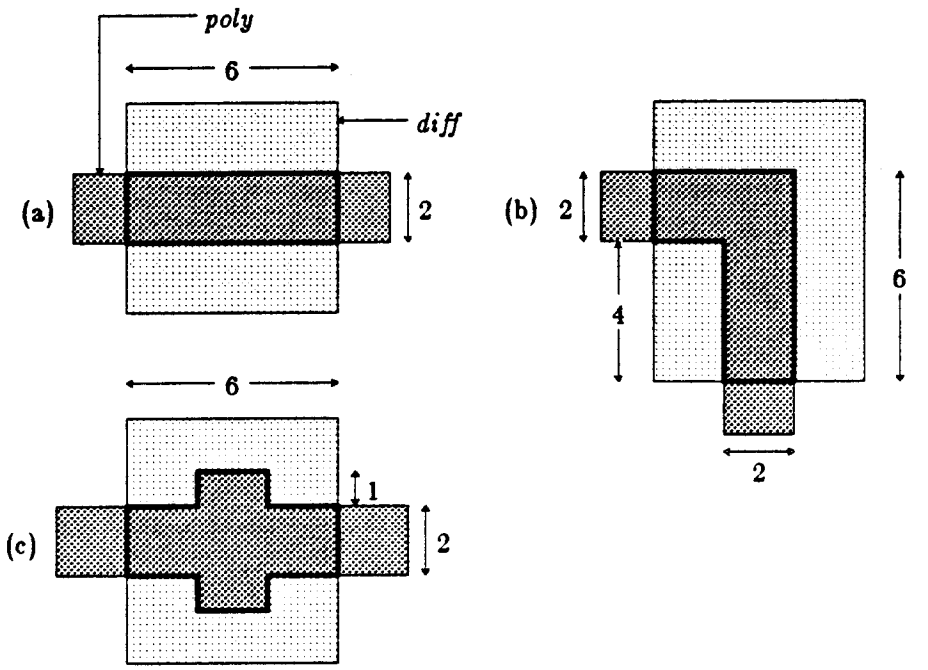


Figure 4.
(a) When transistors are rectangular, it is possible to compute L/W exactly. Here *gateperim* = 4, *sourceperim* = 6, *drainperim* = 6, and $L/W = 2/6$. (b) The L/W of non-branching transistors can be approximated. Here *gateperim* = 4, *sourceperim* = 6, *drainperim* = 10. By averaging *sourceperim* and *drainperim* we get $L/W = 2/8$. (c) The L/W of branching transistors is not well approximated. Here *gateperim* = 16, *sourceperim* = 2, *drainperim* = 2. Magic's estimate of L/W is 8/2, whereas in fact because of current spreading, W is effectively larger than 2 and L effectively smaller than 8, so L/W is overestimated.

5. Extraction styles

Magic usually knows several different ways to extract a circuit from a given layout. Each of these ways is called a *style*. Different styles can be used to handle different fabrication facilities, which may differ in the parameters they have for parasitic capacitance and resistance. Extraction styles are described in the technology file that Magic reads when it starts up; the exact number and nature of the styles is determined by whoever wrote your technology file. At any given time, there is one current extraction style.

To print a list of the extraction styles available, type the command

:extract style.

To change the style to *style*, use the command

:extract style *style*

Each style has a specific scalefactor between Magic units and physical units (e.g, microns); you can't use a particular style with a different scalefactor. To change the scalefactor, you'll have to edit the appropriate style in the **extract** section of the technology file. This process is described in "Magic Maintainer's Manual #2: The Technology File."

6. Ext2sim

Unfortunately, none of our tools yet take advantage of the **.ext** files produced by Magic's extractor. To use these files for simulation or timing analysis, you need to create a **.sim** file. You can do this by running the program *ext2sim*, which is described in a separate manual page, *ext2sim*(1). Note that this is not a Magic command, but a separate program. Writers of tools that read **.ext** format will probably find the code for *ext2sim* a good starting point.

Magic Tutorial #9: Format Conversion for CIF and Calma

John Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

This tutorial corresponds to Magic version 4.

Tutorials to read first:

Magic Tutorial #1: Getting Started
Magic Tutorial #2: Basic Painting and Selection
Magic Tutorial #4: Cell Hierarchies

Commands covered in this tutorial:

:calma, :cif

Macros covered in this tutorial:

None.

1. Basics

CIF (Caltech Intermediate Form) and Calma Stream Format are standard layout description languages used to transfer mask-level layouts between organizations and design tools. This tutorial describes how Magic can be used to read and write files in CIF and Stream formats. The version of CIF that Magic supports is CIF 2.0; it is the most popular layout language in the university design community. The Calma format that Magic supports is GDS II Stream format, version 3.0, corresponding to GDS II Release 5.1. This is probably the most popular layout description language for the industrial design community.

To write out a CIF file, place the cursor over a layout window and type the command

:cif

This will generate a CIF file called *name*.**cif**, where *name* is the name of the root

cell in the window. The CIF file will contain a description of the entire cell hierarchy in that window. If you wish to use a name different from the root cell, type the command

:cif write *file*

This will store the CIF in *file*.**cif**. Start Magic up to edit **tut9a** and generate CIF for that cell. The CIF file will be in ASCII format, so you can use Unix commands like **more** and **vi** to see what it contains.

To read a CIF file into Magic, place the cursor over a layout window and type the command

:cif read *file*

This will read the file *file*.**cif** (which must be in CIF format), generate Magic cells for the hierarchy described in the file, make the entire hierarchy a subcell of the edit cell, and run the design-rule checker to verify everything read from the file. Information in the top-level cell (usually just a call on the "main" cell of the layout) will be placed into the edit cell. Start Magic up afresh and read in **tut9a.cif**, which you created above. It will be easier if you always read CIF when Magic has just been started up: if some of the cells already exist, the CIF reader will not overwrite them, but will instead use numbers for cell names.

To read and write Stream-format files, use the commands **:calma read** and **:calma**, respectively. These commands have the same effect as the CIF commands, except that they operate on files with **.strm** extensions. Stream is a binary format, so you can't examine **.strm** files with a text editor.

Stream files do not identify a top-level cell, so you won't see anything on the screen after you've used the **:calma read** command. You'll have to use the **:load** command to look at the cells you read. However, if Magic was used to write the Calma file being read, the library name reported by the **:calma read** command is the same as the name of the root cell for that library.

Also, Calma format places some limitations on the names of cells: they can only contain alphanumeric characters, "$", and "_", and can be at most 32 characters long. If the name of a cell does not meet these limitations, **:calma write** converts it to a unique name of the form ___*n*, where *n* is a small integer. To avoid any possible conflicts, you should avoid using names like these for your own cells.

You shouldn't need to know much more than what's above in order to read and write CIF and Stream. The sections below describe the different styles of CIF/Calma that Magic can generate and the limitations of the CIF/Calma facilities (you may have noticed that when you wrote and read CIF above you didn't quite get back what you started with; Section 3 describes the differences that can occur). Although the discussion mentions only CIF, the same features and problems apply to Calma.

2. Styles

Magic usually knows several different ways to generate CIF/Calma from a given layout. Each of these ways is called a *style*. Different styles can be used to handle different fabrication facilities, which may differ in the names they use for layers or in the exact mask set required for fabrication. Different styles can be also used to write out CIF/Calma with slightly different feature sizes or design rules. CIF/Calma styles are described in the technology file that Magic reads when it starts up; the exact number and nature of the styles is determined by whoever wrote your technology file. There are separate styles for reading and writing CIF/Calma; at any given time, there is one current input style and one current output style.

The standard SCMOS technology file provides an example of how different styles can be used. Start up Magic with the SCMOS technology (**magic -Tscmos**). Then type the commands

:cif ostyle
:cif istyle

The first command will print out a list of all the styles in which Magic can write CIF/Calma (in this technology) and the second command prints out the styles in which Magic can read CIF/Calma. You use the **:cif** command to change the current styles, but the styles are used for both CIF and Calma format conversion. The SCMOS technology file provides several output styles. The initial (default) style for writing CIF is **lambda=1.0(gen)**. This style generates mask layers for the MOSIS scalable CMOS process, where each Magic unit corresponds to 1 micron and both well polarities are generated. See the technology manual for more information on the various styles that are available. You can change the output style with the command

:cif ostyle *newStyle*

where *newStyle* is the new style you'd like to use for output. After this command, any future CIF or Calma files will be generated with the new style. The **:cif istyle** command can be used in the same way to see the available styles for reading CIF and to change the current style.

Each style has a specific scalefactor; you can't use a particular style with a different scalefactor. To change the scalefactor, you'll have to edit the appropriate style in the **cifinput** or **cifoutput** section of the technology file. This process is described in "Magic Maintainer's Manual #2: The Technology File."

3. Rounding

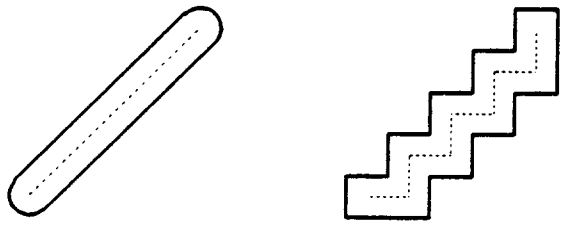
The units used for coordinates in Magic are generally different from those in CIF files. In Magic, most technology files use lambda-based units, where one unit is typically half the minimum feature size. In CIF files, the units are centimicrons (hundredths of a micron). When reading CIF and Calma files, an integer scalefactor is used to convert from centimicrons to Magic units. If the CIF file contains coordinates that don't scale exactly to integer Magic units, Magic rounds

the coordinates up or down to the closest integer Magic units. A CIF coordinate exactly halfway between two Magic units is rounded down. The final authority on rounding is the procedure CIFScaleCoord in the file cif/CIFreadutils.c When rounding occurs, the resulting Magic file will not match the CIF file exactly.

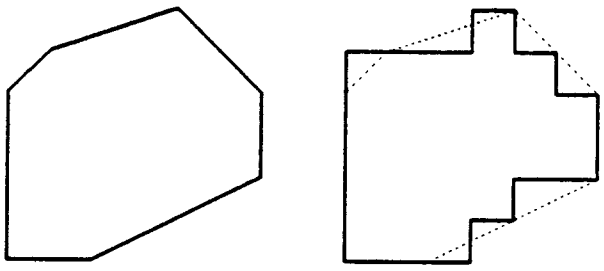
Technology files usually specify geometrical operations such as bloating, shrinking, and-ing, and or-ing to be performed on CIF geometries when they are read into Magic. These geometrical operations are all performed in the CIF coordinate system (centimicrons) so there is no rounding or loss of accuracy in the operations. Rounding occurs only AFTER the geometrical operations, at the last possible instant before entering paint into the Magic database.

4. Non-Manhattan Geometries

Magic only supports Manhattan features. When CIF or Calma files contain non-Manhattan features, they are approximated with Manhattan ones. The approximations occur for wires (if the centerline contains non-Manhattan segments) and polygons (if the outline contains non-Manhattan segments). In these cases, the non-Manhattan segments are replaced with one or more horizontal and vertical segments before the figure is processed. Conversion is done by inserting a one-unit stairstep on a 45-degree angle until a point is reached where a horizontal or vertical line can reach the segment's endpoint. Some examples are illustrated in the figure below: in each case, the figure on the left is the one specified in the CIF file, and the figure on the right is what results in Magic.



CIF Wire Resulting Magic Shape



CIF Polygon Resulting Magic Shape

The shape of the Magic stairstep depends on the order in which vertices appear in the CIF or Calma file. The stairstep is made by first incrementing or decrementing the x-coordinate, then incrementing or decrementing the y-coordinate, then x, then y, and so on. For example, in the figure above, the

polygon was specified in counter-clockwise order; if it had been specified in clockwise order the result would have been slightly different.

An additional approximation occurs for wires. The CIF wire figure assumes that round caps will be generated at each end of the wire. In Magic, square caps are generated instead. The top example of the figure above illustrates this approximation.

5. Other Problems with Reading and Writing CIF

You may have noticed that when you wrote out CIF for **tut9a** and read it back in again, you didn't get back quite what you started with. Although the differences shouldn't cause any serious problems, this section describes what they are so you'll know what to expect. There are three areas where there may be discrepancies: labels, arrays, and contacts. These are illustrated in **tut9b**. Load this cell, then generate CIF, then read the CIF back in again. When the CIF is read in, you'll get a couple of warning messages because Magic won't allow the CIF to overwrite existing cells: it uses new numbered cells instead (this is why you should normally read CIF with a "clean slate"; in this case it's convenient to have both the original and reconstructed infromation present at the same time; just ignore the warnings). The information from the CIF cell appears as a subcell named **1** right on top of the old contents of **tut9b**; select **1**, move it below **tut9b**, and expand it so you can compare its contents to **tut9b**.

The first problem area is that CIF cannot handle labels unless they are points. Where you have line or box labels in Magic, CIF labels are generated at the center of the Magic labels. The label **in** in **tut9y** is an example of a line label that gets smashed in the CIF processing.

The second problem is with arrays. CIF has no standard array construct, so when Magic outputs arrays it does it as a collection of cell instances. When the CIF file is read back in, each array element comes back as a separate subcell. The array of **tut9y** cells is an example of this. Most designs only have a few arrays that are large enough to matter; where this is the case, you should go back after reading the CIF and replace the multiple instances with a single array. Calma format does have an array construct, so it doesn't have this problem.

The third discrepancy is that where there are large contact areas, when CIF is read and written the area of the contact may be reduced slightly. This happened to the large poly contact in **tut9b**. The shrink doesn't reduce the effective area of the contact; it just reduces the area drawn in Magic. To see what's happening here, place the box around **tut9b** and **1**, expand everything, then type

:cif see CCP

This causes feedback to be displayed showing CIF layer "CCP" (contact cut to poly). You may have to zoom in a bit to distinguish the individual via holes. Magic generates lots of small contact vias over the area of the contact, and if contacts aren't exact multiples of the hole size and spacing then extra space is left around the edges. When the CIF is read back in, this extra space isn't turned

back into contact. The circuit that is read in is functionally identical to the original circuit, even though the Magic contact appears slightly smaller.

There is an additional problem with generating CIF having to do with the cell hierarchy. When Magic generates CIF, it performs geometric operations such as "grow" and "shrink"on the mask layers. Some of these operations are not guaranteed to work perfectly on hierarchical designs. Magic detects when there are problems and creates feedback areas to mark the trouble spots. When you write CIF, Magic will warn you that there were troubles. These should almost never happen if you generate CIF from designs that don't have any design-rule errors. If they do occur, you'll have to either get a Magic wizard to help you, or read the document on technology files: it describes the problem and its solutions.

Magic Maintainer's Manual #1: Hints for System Maintainers

John Ousterhout
Walter Scott

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

This tutorial corresponds to Magic version 4.

Tutorials to read first:

All of them.

Commands covered in this tutorial:

:*profile, :*runstats, :*seeflags, :*watch

Macros covered in this tutorial:

None.

1. Introduction

This document provides some information to help would-be Magic maintainers learn about the system. It is not at all complete, and like most infrequently-used documentation, will probably become less and less correct over time as the system evolves but this tutorial doesn't. So, take what you read here with a grain of salt. We believe that everything in this tutorial was up-to-date as of the January 1986 Magic release. Before doing anything to the internals of Magic, you should read at least the first, and perhaps all four, of the papers on Magic that appeared together in the *1984 IDesign Automation Conference.* In addition, the *1985 Chapel Hill Conference on VLSI* contains a paper desribing Magic's global router, and the extractor is described in a paper in the *1985 Design Automation Conference.*

2. Installing Magic

If you've received Magic from Berkeley on the 1986 VLSI Tools tape, then it shouldn't take much work to get it running. The tools tape should be read into ~cad, which means that there will be a binary version of Magic in ~cad/bin and a set of library subdirectories in ~cad/lib/magic. If this isn't so, then at the very least you'll need to get a Magic system library set up in ~cad/lib/magic/sys: this directory contains information like technology files and colormaps and Magic can't run at all without it. **Be sure that there is a user named "cad" on your system; if not, Magic won't be able to find any of the library or technology files.**

If you're running on a Sun you shouldn't need to do anything besides what's mentioned above. Just run Suntools and then run Magic.

If you're running on a VAX with an attached color display, you'll probably need to do some additional setup. If the display is an AED512 or similar display, it will be attached to the VAX via an RS232 port. Magic needs to be able to read from this port, and there are two ways to do this. The first is simply to have no login process for that port and have your system administrator change the protection to allow all processes to read from the port and write to it. The second way is to have users log in on the display and run a process that changes the protection of the display. There is a program called Sleeper that we distribute with Magic; if it's run from an AED port it will set everything up so Magic can use the port. Sleeper is clumsy to use, so we recommend that you use the first solution (no login process).

When you're running on VAXes, Magic will need to know which color display port to use from each terminal port. Users can type this information as command-line switches but it's clumsy. To simplify things, Magic checks the file **~cad/lib/displays** when it starts up. The displays file tells which color display port to use for which text terminal port and also tells what kind of display is attached. Once this file is set up, users can run Magic without worrying about the system configuration. See the manual page for *displays*(5).

One last note: if you're running on an AED display, you'll need to set communication switches 3-4-5 to up-down-up.

3. Source Directory Structure; Making Magic

If you are working on ucbkim at Berkeley, the Magic sources are rooted in the directory **~magic/src**. At most other sites, the root for the Magic sources should be **~cad/src/magic**. All pathnames given in this manual will assume that your current working directory is the root of the Magic sources.

There are approximately 30 source subdirectories in Magic. Most of these consist of modules of source code for the system, for example **database**, **main**, and **utils**. See Section 4 of this document for brief descriptions of what's in each source directory. Besides the source code, the other subdirectories are:

doc — Contains sources for all the documentation, including *man* pages, tutorials, and maintenance manuals. Subdirectories of **doc**, e.g. **doc/scmos**, contain the technology manuals.

	The Makefile in each directory can be used to run off the documentation. The tutorials, maintenance manuals, and technology manuals all use the Berkeley Grn/Ditroff package, which means that you can't run them off without Grn/Ditroff unless you change the sources.
include	Contains installed (i.e. "safe") versions of all the header files (*.h) from all the modules.
lib	Contains installed (i.e. "safe") versions of each of the compiled and linked modules (*.o).
installed	Contains one subdirectory for each of the source code directories. Each subdirectory contains "safe" versions of the source files for that module. These files correspond to the installed **.o** files in **lib**.
magic	This directory is where the modules of Magic are combined together to form an executable version of the system.
cadlib	This is a symbolic link to the directory where Magic stores cell libraries and official installed versions of technology files and color maps. Normally, **cadlib** is a symbolic link to ~**cad/lib/magic**.

Magic is a relatively large system: there are around 250 source files, 120000 lines of C code, and as many as four maintainers working on the system at one time at Berkeley. In order to make all of this manageable, we've organized the sources in a two-level structure. Each module has its own subdirectory, and you can make changes to the module and recompile it by working within that subdirectory. In addition to the information in the subdirectory, there is an "installed" version of each module, which consists of the files in the **lib**, **include**, and **installed** subdirectories. The installed version of each module is supposed to be stable and reliable. At Berkeley, when a module is changed it is tested carefully without re-installing it, and is only re-installed when it is in good condition. Note that "installed" doesn't mean that Magic users see the module; it only means that other Magic maintainers will see it.

By keeping modules separate, it's possible for several maintainers to work at once as long as they are modifying different source subdirectories. Each maintainer works with the uninstalled version of a module, and links that with the installed versions of all other modules. Thus, for example, one maintainer can modify **database/DBcell.c** and another can modify **dbwind/DBWundo.c** at the same time.

Putting together a runnable Magic system proceeds in two steps after a source file has been modified. First, the source file is compiled, and all the files in its module are linked together into a single file *xyz*.**o**, where *xyz* is the name of the module. Then all of the modules are linked together to form an executable version of Magic. The command **make** in each source directory will compile and link the module locally; **make install** will compile and link it, and also install it in the **include**, **lib**, and **installed** directories. The command **make** in the subdirectory **magic** will produce a runnable version of Magic in that directory,

using the installed versions of all modules. To work with the uninstalled version of a module, create another subdirectory identical to **magic**, and modify the Makefile so that it uses uninstalled versions of the relevant modules. For example, at Berkeley, there are subdirectories **hamachitest**, **mayotest**, **oustertest**, and **wsstest** that we use to test new versions of modules before installing them. If you want to remake the entire system, type "make magic" in the top-level directory (**~cad/src/magic**), then switch to the directory **magic** underneath the top-level directory and type "make" there.

4. Summary of Magic Modules

This section contains brief summaries of what is in each of the Magic source subdirectories.

calma — Contains code to read and write Calma Stream-format files. It uses many of the procedures in the **cif** module.

cif — Contains code to process the CIF sections of technology files, and to generate CIF files from Magic.

cmwind — Contains code to implement special windows for editing color maps.

commands — The procedures in this module contain the top-level command routines for layout commands (commands that are valid in all windows are handled in the **windows** module). These routines generally just parse the commands, check for errors, and call other routines to carry out the actions.

database — This is the largest and most important Magic module. It implements the hierarchical corner-stitched database, and reads and writes Magic files.

dbwind — Provides display functions specific to layout windows, including managing the box, redisplaying layout, and displaying highlights and feedback.

debug — There's not much in this module, just a few routines used for debugging purposes.

drc — This module contains the incremental design-rule checker. It contains code to read the **drc** sections of technology files, record areas to be rechecked, and recheck those areas in a hierarchical fashion.

ext2sim — The **ext2sim** directory isn't part of Magic itself. It's a self-contained program that flattens the hierarchical **.ext** files generated by Magic's extractor into a single file in **.sim** format. See the manual page **ext2sim(1)**.

extract — Contains code to read the **extract** sections of technology files, and to generate hierarchical circuit descriptions (**.ext** files) from Magic layouts.

fsleeper Like **ext2sim**, this directory is a self-contained program that allows a graphics terminal attached to one machine to be used with Magic running on a different machine. See the manual page **fsleeper(1)**.

gcr Contains the channel router, which is an extension of Rivest's greedy router that can handle switchboxes and obstacles in the channels.

graphics This is the lowest-level graphics module. It contains driver routines for the AED family of displays and for Sun workstations. The code here does basic clipping and drawing. If you want to make Magic run on a new kind of display, this is the only module that should have to change.

graphics.drivers This isn't really a Magic source module. It contains several sub-directories, each of which is a contributed graphics driver. These drivers were provided by various Magic users around the country. We have not tested them with the rest of Magic, so we make no guarantees about how well they'll work. We also can't help you if you have difficulties installing the driver. To use a contributed driver, look in its directory for a file named **ReadMe**, **README**, or something similar. This file should explain how to integrate that driver with the rest of Magic. If you're lucky, the **ReadMe** file will also contain the name of someone you can contact if you run into trouble.

grouter The files in this module implement the global router, which computes the sequence of channels that each net is to pass through.

macros Implements simple keyboard macros.

magicusage Like **ext2sim**, this is also a self-contained program. It searches through a layout to find all the files that are used in it. See **magicusage**(1).

main This module contains the main program for Magic, which parses command-line parameters, initializes the world, and then transfers control to **textio**.

misc A few small things that didn't belong anyplace else.

mpack Contains routines that implement the Tpack tile-packing interface using the Magic database.

netmenu Implements netlists and the special netlist-editing windows.

parser Contains the code that parses command lines into arguments.

plot The internals of the **:plot** command.

plow This module contains the code to support the **:plow** and **:straighten** commands.

prleak Also not part of Magic itself. Prleak is a self-contained program intended for use in debugging Magic's memory allocator. It analyzes a trace of mallocs/frees to look for memory leaks. See the manual page **prleak(8)** for information on what the program does.

router Contains the top-level routing code, including procedures to read the router sections of technology files, chop free space up into channels, analyze obstacles, and paint back the results produced by the channel router.

select This module contains files that manage the selection. The routines here provide facilities for making a selection, enumerating what's in the selection, and manipulating the selection in several ways, such as moving it or copying it.

signals Handles signals such as the break key and control-Z.

tech This module contains the top-level technology file reading code, and the current technology files. The code does little except to read technology file lines, parse them into arguments, and pass them off to clients in other modules (such as **drc** or **database**).

textio The top-level command interpreter. This module grabs commands from the keyboard or mouse and sends them to the window module for processing. Also provides routines for message and error printout, and to manage the prompt on the screen.

tiles Implements basic corner-stitched tile planes. This module was separated from **database** in order to allow other clients to use tile planes without using the other database facilities too.

undo The **undo** module provides the overall framework for undo and redo operations, in that it stores lists of actions. However, all the specific actions are managed by clients such as **database** or **netmenu**.

utils This module implements a whole bunch of utility procedures, including a geometry package for dealing with rectangles and points and transformations, a heap package, a hash table package, a stack package, a revised memory allocator, and lots of other stuff.

windows This is the overall window manager. It keeps track of windows and calls clients (like **dbwind** and **cmwind**) to process window-specific operations such as redisplaying or processing commands. Commands that are valid in all windows, such as resizing or moving windows, are implemented here.

wiring The files in this directory implement the **:wire** command. There are routines to select wiring material, add wire legs,

and place contacts.

5. Technology and Other Support Files

Besides the source code files, there are a number of other files that must be managed by Magic maintainers, including color maps, technology files, and other stuff. Below is a listing of those files and where they are located.

5.1. Technology Files

See "Magic Maintainer's Manual #2: The Technology File" for information on the contents of technology files. The sources for technology files are contained in the subdirectory **tech**, in files like **scmos.tech** and **nmos.tech**. The technology files that Magic actually uses at runtime are kept in the directory **cadlib/sys**; **make install** in **tech** will copy the sources to **cadlib/sys**. Technology file formats have evolved rapidly during Magic's life, so we use version numbers to allow multiple formats of technology files to exist at once. The installed versions of technology files have names like **nmos.tech20**, where **20** is a version number. The current version is defined in the Makefile for **tech**, and should be incremented if you ever change the format of technology files; if you install a new format without changing the version number, pre-existing versions of Magic won't be able to read the files. After incrementing the version number, you'll also have to re-make the **tech** module since the version number is referenced by the code that reads the files.

5.2. Display Styles

The display style file sources are contained in the source directory **graphics**. See "Magic Maintainer's Manual #3: The Display Style and Glyph Files" and the manual page *dstyle*(5) for a description of their contents. **Make install** in **graphics** will copy the files to **cadlib/sys**, which is where Magic looks for them when it executes.

5.3. Glyph Files

Glyph files are described in Maintainer's Manual #3 and the manual page *glyphs*(5); they define patterns that appear in the cursor. The sources for glyph files appear in two places: some of them are in **graphics**, in files like **UCB512.glyphs**, and some others are defined in **windows/window.glyphs**. When you **make install** in those directories, the glyphs are copied to **cadlib/sys**, which is where Magic looks for them when it executes.

5.4. Color Maps

The color map sources are also contained in the source directory **graphics**. Color maps have names like **mos.7bit.std.cmap**, where **mos** is the name of the technology style to which the color map applies, **7bit** is the display style, and **std** is a type of monitor. If monitors have radically different phosphors, they may require different color maps to achieve the same affects. Right now we only support the **std** kind of monitor. When Magic executes, it looks for color maps in

cadlib/sys; **make install** in **graphics** will copy them there. Although color map files are textual, you shouldn't edit them by hand; use Magic's color map editing window instead.

6. New Display Drivers

The most common kind of change that will be made to Magic is probably to adapt it for new kinds of color displays. Each display driver contains a standard collection of procedures to perform basic functions such as placing text, drawing filled rectangles, or changing the shape of the cursor. A table (defined in **graphics/grMain.c**) holds the addresses of the routines for the current display driver. At initialization time this table is filled in with the addresses of the routines for the particular display being used. All graphics calls pass through the table.

If you have a display other than an AED or Sun, the first thing you should do is check the directory **graphics.drivers**. Each of the sudirectories in **graphics.drivers** contains an additional display driver that was contributed by some site outside of Berkeley. Each subdirectory should contain a file named **ReadMe** (or something similar), which explains how to install and use the driver. If you have any troubles with one of the these drivers, you'll have to contact the people that contributed the driver; we here at Berkeley don't know anything about them. If the contributors wish to be contacted, they will identify themselves in the **ReadMe** file.

If you need to build a new display driver, we recommend starting with the routines for either the AED (all the files in **graphics** with names like **grAed1.c**), or the Sun (names like **grSunW1.c**). For stand-alone displays, the AED routines are probably the easiest to work from; for integrated workstations with pre-existing window packages, the Sun routines may be easiest. Copy the files into a new set for your display, change the names of the routines, and modify them to perform the equivalent functions on your display. Write an initialization routine like **aedSetDisplay**, and add information to the display type tables in **graphics/grMain.c**. At this point you should be all set. There shouldn't be any need to modify anything outside of the graphics module.

7. Debugging and Wizard Commands

At Berkeley, we use *sdb* to debug Magic on VAXes and *dbx* on the Suns (there's no *sdb* for the Suns). The Makefiles are set up to compile all files with the **-gold** switch, which creates debugging information in *sdb*'s format. If you want to use *dbx* you'll have to change this to a **-g** switch and recompile the world.

Because of the size of Magic and the way Unix handles debugging symbols, it's extremely slow to compile a complete version of Magic with debugging information for everything, and the executable file ends up being enormous. To solve this problem the Makefiles are set up to strip off debugging information before installing. Thus, you have to link with uninstalled versions to get debugging information. In most cases, debugging information is only needed for a

few modules at a time, namely the modules you're currently modifying. The database module is set up to install with debugging symbols, since it seems to be involved in almost all debugging.

If you try to use *dbx*, you'll discover that Magic has too many procedures for the default table sizes; *dbx* runs out of space and dies. The solution is either to recompile *dbx* with larger tables or throw away pieces of Magic to reduce the number of procedures (we recommend the first alternative).

There are a number of commands that we implemented in Magic to assist in debugging. These commands are called *wizard commands*, and aren't visible to normal Magic users. They all start with "*". To get terse online help for the wizard commands, type **:help wizard** to Magic. The wizard commands aren't documented very well. Some of the more useful ones are:

***watch** *plane*
This causes Magic to display on the screen the corner-stitched tile structure for one of the planes of the edit cell. For example, ***watch subcell** will display the structure of the subcell tile plane, including the address of the record for each tile and the values of its corner stitches. Without this command it would have been virtually impossible to debug the database module.

***profile on | off**
If you're using the Unix profiling tools to figure out where the cycles are going, this command can be used to turn profiling off for everything except the particular operation you want to measure. This command doesn't work on Pyramid systems, because their operating system doesn't support selective enabling and disabling of profiling.

***runstats**
This command prints out the CPU time usage since the last invocation of this command, and also the total since starting Magic.

***seeflags** *flag*
If you're working on the router, this command allows you to see the various channel router flags by displaying them as feedback areas. The cursor should first be placed over the channel whose flags you want to see.

Magic Maintainer's Manual #2: The Technology File

Walter Scott
John Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

This tutorial corresponds to Magic version 4
(technology suffix ".tech20")

Tutorials to read first:

Magic Tutorial #1: Getting Started
Magic Tutorial #2: Basic Painting and Selection
Magic Tutorial #6: Design-Rule Checking

You should also read at least the first, and probably all four, of the papers on Magic that appeared in the *ACM IEEE 21st Design Automation Conference*, and the paper "Magic's Circuit Extractor", which appeared in the *ACM IEEE 22nd Design Automation Conference*. The overview paper from the DAC was also reprinted in *IEEE Design and Test* magazine in the February 1985 issue. The circuit extractor paper will also appear in the February 1986 issue of *IEEE Design and Test* magazine.

Commands covered in this tutorial:

:*watch

Macros covered in this tutorial:

none

1. Introduction

Magic is a technology independent layout editor. All technology-specific information—mask layers, design rules, etc.—comes from a *technology file*. There is a different technology file for each technology supported by Magic. You can run Magic with a different technology by specifying the **-T***techfile* flag on the

command line you use to start Magic, where *techfile* is the name of a file of the form *techname*.**techn** in either the current directory, or the library directory ~cad/lib/magic/sys. (The *n* is a numeric suffix to identify the version of the technology file, which is currently **20**).

This tutorial describes the contents of a technology file, and gives hints for building a new one. It assumes that your current working directory is the Magic source directory, **~magic/src** on ucbkim, or **~cad/src/magic** on other machines.

A technology file is organized into sections, each of which begins with a line containing a single keyword and ends with a line containing the single word **end**. If you examine one of the Magic technology files in the directory **~cad/src/magic/tech**, e.g, **nmos.tech**, you can see that it contains the following sections: **tech**, **planes**, **types**, **styles**, **contact**, **compose**, **connect**, **cifoutput**, **cifinput**, **drc**, **extract**, **wiring**, **router**, **plowing**, and **plot**. These sections must appear in this order in all technology files. Every technology file must have all of the sections, although the sections need not have any lines between the section header and the **end** line.

A technology file can contain comments, which are blocks of text beginning with the characters "**/***" and ending with the characters "***/**". Comments are ignored when processing a technology file. In **nmos.tech** you can see several lines just before the **drc** section (near the end of the technology file) that are of the form "**#define** ...". These lines are definitions of macros that may be used in subsequent lines in the technology file.

The form of comments and macro definitions should look familiar to "C" programmers, for good reason: the "C" macro preprocessor is used to expand macros and eliminate comments. Technology files cannot be read directly by Magic in their "raw" form; the "C" preprocessor is run to produce a Magic-readable version of the technology file. The last section in this tutorial describes how to install technology files.

Each section in a technology file consists of a series of lines. Each line consists of a series of words, separated by spaces or tabs. If a line ends with the character "\", the "\" and the following newline are ignored. For example,

```
width   allDiff   2 \
"Diffusion width must be at least 2"
```

is treated as though it had all appeared on a single line with no intervening "\". The rest of this part of the tutorial will describe each of the technology file sections in turn.

2. Tech section

Magic stores the technology of a cell in the cell's file on disk. When reading a cell back in to Magic from disk, the cell's technology must match the name of the current technology, which appears as a single word in the **tech** section of the technology file. See Table 1 for an example.

```
tech
nmos
end
```

Table 1. **Tech** section

It may seem that storing the technology name as part of the technology file is redundant, since the name of the file is probably the same as the name of the technology. (e.g., "nmos.tech20" for technology **nmos**, or "scmos.tech20" for technology **scmos**). This feature is leftover from olden days when slight variants of a technology would be created by having a new technology file with a different file name but the same "official" name given in the **tech** section. This has the advantage that cells designed with one variant could be edited with any of the other files implementing the same technology without having to modify the technology names in the **.mag** files. The disadvantage of this approach, however, is that it defeats Magic technology-defaulting mechanism: if no explicit technology is specified when Magic starts up, it reads the technology from the **.mag** file being edited and looks for a technology file by this name. If there are several variants of the same technology, Magic will pick the one with the desired technology file name. Anyhow, we recommend that the internal names of technologies should always match the file names.

3. Planes, types, and contact sections

The **planes**, **types**, and **contact** sections are used to define the layers used in the technology. Magic uses a new data structure, called *corner-stitching*, to represent layouts. Corner-stitching represents mask information as a collection of non-overlapping rectangular *tiles*. Each tile has a type that corresponds to a single Magic layer. An individual corner-stitched data structure is referred to as a *plane*.

Magic allows you to see the corner-stitched planes it uses to store a layout. We'll use this facility to see how several corner-stitched planes are used to store the layers of a layout. Enter Magic to edit the cell **m2a**. Type the command **:*watch poly-diff demo**. You are now looking at the **poly-diff** plane. Each of the boxes outlined in black is a tile. (The arrows are *stitches*, but are unimportant to this discussion.) You can see that some tiles contain layers (polysilicon, diffusion, poly-metal-contact, diff-metal-contact, and enhancement-fet), while others contain empty space. Corner-stitching is unusual in that it represents empty space explicitly. Each tile contains exactly one type of material, or space.

You have probably noticed that metal does not seem to have a tile associated with it, but instead appears right in the middle of a space tile. This is because metal is stored on a different plane, the **metal** plane. Type the command **:*watch metal demo**. Now you can see that there are metal tiles, but the

polysilicon, diffusion, and transistor tiles have disappeared. The two contacts, poly-metal-contact and diff-metal-contact, still appear to be tiles.

The reason Magic uses several planes to store mask information is that corner-stitching can only represent non-overlapping rectangles. If a layout were to consist of only a single layer, such as polysilicon, then only two types of tiles would be necessary: polysilicon, and space. As more layers are added, overlaps can be represented by creating a special tile type for each kind of overlap area. For example, when polysilicon overlaps diffusion, the overlap area is marked with the tile type enhancement-fet.

Although some overlaps correspond to actual electrical constructs (e.g., transistors), other overlaps have little electrical significance. For example, metal can overlap polysilicon without changing the connectivity of the circuit or creating any new devices. To create new tile types for all possible overlapping combinations of metal with polysilicon, diffusion, transistors, etc. would be wasteful, since these new overlapping combinations would have no electrical significance.

Instead, Magic partitions the layers into separate planes. Layers whose overlaps have electrical significance must be stored in a single plane. For example, polysilicon, diffusion, and their overlaps (enhancement-fet, depletion-fet, and buried-contact) are all stored in the **poly-diff** plane. Metal does not interact with any of these tile types, so it is stored in its own plane, the **metal** plane.

Contacts between layers in one plane and layers in another are a special case and are represented on *both* planes. This explains why the poly-metal-contact and diff-metal-contact tiles appeared on both the **poly-diff** plane and on the **metal** plane.

The **planes** section of the technology file specifies how many planes will be used to store tiles in a given technology, and gives each plane a name. Each line in this section defines a plane by giving a comma-separated list of the names by which it is known. Any name may be used in referring to the plane in later sections, or in commands like the **:*watch** command you used earlier. Table 2 gives the **planes** section from the nMOS technology file.

```
planes
poly-diff,poly,diff
metal
end
```

Table 2. **Planes** section

Magic uses three other planes internally. The **subcell** plane is used for storing cell instances rather than storing mask layers. The **designRuleCheck** and **designRuleError** planes are used by the design rule checker to store areas to be reverified, and areas containing design rule violations, respectively.

There is a limit on the maximum number of planes in a technology, including the internal planes. This limit is currently 8. To increase the limit, it is necessary to change **MAXPLANES** in the file **database/database.h** and then recompile all of Magic as described in "Maintainer's Manual #1". Each additional plane involves additional storage space in every cell and additional processing time for searches, so we recommend that you keep the number of planes as small as you can do cleanly.

types	
p	polysilicon,poly,red
p	diffusion,diff,green
p	poly-metal-contact,pmc
p	diff-metal-contact,dmc
p	enhancement-fet,efet
p	depletion-fet,dfet
p	depletion-capacitor,dcap
p	buried-contact,bc
m	metal,blue
m	glass-contact
end	

Table 3. **Types** section

The **types** section identifies the technology-specific tile types used by Magic. Table 3 gives this section for the nMOS technology file. Each line in this section is of the following form:

plane names

Each type defined in this section is allowed to appear on exactly one of the planes defined in the **planes** section, namely that given by the *plane* field above. For contacts types such as **poly-metal-contact**, the plane will be the contact's home plane; there will be other tile types used to represent the contact on the other planes it connects (this is described later in this section).

The *names* field is a comma-separated list of names. The first name in the list is the "long" name for the type; it appears in the **.mag** file and whenever error messages involving that type are printed. Any unique abbreviation of any of a type's names is sufficient to refer to that type, both from within the technology file and in any commands such as **:paint** or **:erase**.

Magic has certain built-in types as shown in Table 4. Empty space (**space**) is special in that it can appear on any plane. The types **error_p**, **error_s**, and **error_ps** record design rule violations. The types **checkpaint** and **checksubcell** record areas still to be design-rule checked.

There is a limit on the maximum number of types in a technology, including all the built-in types. Currently, the limit is 40 tile types. To increase the limit,

Tile type	Plane
space	*all*
error_p, EP	**designRuleError**
error_s, ES	**designRuleError**
error_ps, EPS	**designRuleError**
checkpaint, CP	**designRuleCheck**
checksubcell, CS	**designRuleCheck**

Table 4. Built-in Magic types.

you'll have to change **TT_MAXTYPES** in the file **database/database.h** and then recompile all of Magic as described in "Maintainer's Manual #1". A number of macros in **database.h** also depend on the value of **TT_MAXTYPES**/32. They are currently set up assuming that **TT_MAXTYPES** is between 33 and 64; if **TT_MAXTYPES** is changed to lie outside this region they should be changed. See the comments in **database.h** for more information. Because there are a number of tables whose size is determined by the square of **TT_MAXTYPES**, it is very expensive to increase **TT_MAXTYPES** much beyond 64.

```
contact
poly-metal-contact      poly        metal
diff-metal-contact      diff        metal
end
```

Table 5. **Contact** section

As mentioned before, contacts in Magic are represented on each plane containing material connected by the contact. Also mentioned before, though, is that each tile type defined in the **types** section appears on exactly one plane. This seeming conflict is resolved by having Magic automatically generate new tile types for each of the planes on which a contact appears. The **contact** section lets Magic know which types are contacts, and the planes that they connect.

Each line in the **contact** section begins with a tile type that is to be considered as a contact. This tile type is referred to as the *base* type of the contact. In Table 5, for example, the type **poly-metal-contact** is the base type of a contact. The remainder of each line is a list of tile types that are not contacts, each of which must have a different home plane. These tile types are referred to as the *component* types of the contact, and are the layers that would be present if there were no electrical connection. In the example, the component layers are **polysilicon** and **metal**.

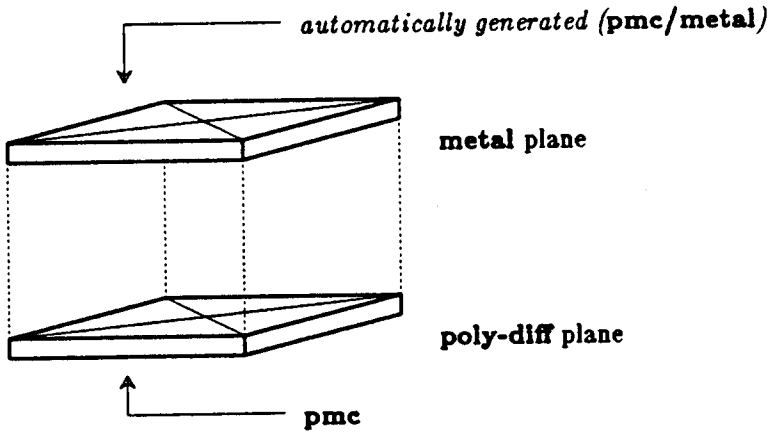


Figure 1. A different tile type is used to represent a contact on each plane that it connects. Here, a contact between poly on the **poly-diff** plane and metal on the **metal** plane is stored as two tile types. One, **pmc**, is specified in the technology file as residing on the **poly-diff** plane; the other is automatically-generated for the **metal** plane.

New types get generated for all planes of a contact except for the home plane of its base type. In the example, this means that a new tile type will be generated to represent the contact on the metal plane. These generated types are called *images* of the contact. The type used to represent the contact on the poly-diff plane is **poly-metal-contact** itself. Figure 1 depicts the situation graphically. In later sections of the technology file, it is sometimes useful to refer separately to the various images of contact. A special notation using a "/" is used for this. If a tile type *aaa/bbb* is specified in the technology file, this refers to the image of contact *aaa* on plane *bbb*. For example, **pmc/metal** refers to the image of the poly-metal contact that lies on the metal plane, and **pmc/poly-diff** refers to the image on the poly plane, which is the same as **pmc**.

4. Specifying Type-lists

In several places in the technology file you'll need to specify groups of tile types. For example, in the **connect** section you'll specify groups of tiles that are mutually connected. These are called *type-lists* and there are several ways to specify them. The simplest form for a type-list is a comma-separated list of tile types, for example

poly,diff,pmc,dmc

There must not be any spaces in the type-list. Type-lists may also use tildes ("~") to select all tiles but a specified set, and parentheses for grouping. For example,

~(pmc,dmc)

selects all tile types but pmc and dmc. When a contact name appears in a type-list, it selects *all* images of the contact unless a "/" is used to indicate a particular one. The example above will not select any of the images of pmc or dmc. Slashes can also be used in conjunction with parentheses and tildes. For example,

~(pmc,dmc)/poly-diff,metal

selects all of the tile types on the poly-diff plane except for pmc and dmc, and also selects metal. Tildes have higher operator precedence than slashes, and commas have lowest precedence of all.

Note: in the CIF sections of the technology file, only simple comma-separated names are permitted; tildes and parentheses are not understood. However, everywhere else in the technology file the full generality can be used. Sorry for this inconsistency...

styles	
styletype	mos
polysilicon	1
diffusion	2
metal	20
enhancement-fet	6
enhancement-fet	7
depletion-fet	6
depletion-fet	10
depletion-capacitor	6
depletion-capacitor	11
buried-contact	6
buried-contact	33
poly-metal-contact	1
poly-metal-contact	20
poly-metal-contact	32
diff-metal-contact	2
diff-metal-contact	20
diff-metal-contact	32
glass-contact	20
glass-contact	34
error_p	42
error_s	42
error_ps	42
end	

Table 6. **Styles** section

5. Styles section

Magic can be run on several different types of graphical displays. Although it would have been possible to incorporate display-specific information into the technology file, a different technology file would have been required for each display type. Instead, the technology file gives one or more display-independent

styles for each type that is to be displayed, and uses a per-display-type styles file to map info the colors and stipplings specific to the display being used. The styles file is described in Magic Maintainer's Manual #3: "Styles and Colors", so we will not describe it further here.

Table 6 shows the **styles** section from the nMOS technology file. The first line specifies the type of style file for use with this technology, which in this example is **mos**. Each subsequent line consists of a tile type and a style number (an integer between 1 and 63). The style number is nothing more than a reference between the technology file and the styles file. Notice that a given tile type can have several styles (e.g., poly-metal-contact uses styles #1, #33, and #3), and that a given style may be used to display several different tiles (e.g., style #4 is used in enhancement-fet, depletion-fet, and buried-contact). If a tile type should not be displayed, it has no entry in the **styles** section.

```
compose
compose       efet    poly diff
decompose     dfet    poly diff
decompose     dcap    poly diff
decompose     bc      poly diff
erase         glass   metal      space
end
```

Table 7. **Compose** section

6. Compose section

The semantics of Magic's paint operation are defined by a collection of rules of the form, "given material *HAVE* on plane *PLANE*, if we paint *PAINT*, then we get *Z*", plus a similar set of rules for the erase operation. The default paint and erase rules are simple. Assume that we are given material *HAVE* on plane *PLANE*, and are painting or erasing material *PAINT*.

(1) *You get what you paint.* If the home plane of *PAINT* is *PLANE*, or *PAINT* is space, you get *PAINT*; otherwise, nothing changes and you get *HAVE*.

(2) *You can erase all or nothing.* Erasing space or *PAINT* from *PAINT* will give space; erasing anything else has no effect.

These rules apply for contacts as well. Painting the base type of a contact paints the base type on its home plane, and each automatically-generated type on its home plane. Erasing the base type of a contact erases both the base type and the automatically-generated types.

It is sometimes desirable for certain tile types to behave as though they were "composed" of other, more fundamental ones. For example, painting poly over diffusion in nMOS produces enhancement-fet, instead of diffusion. Also, painting

either poly or diffusion over enhancement-fet leaves enhancement-fet, erasing poly from enhancement-fet leaves diffusion, and erasing diffusion leaves poly. The semantics for enhancement-fet are a result of the following rule in the **compose** section of the nMOS technology file:

compose efet poly diff

Sometimes, not all of the "component" layers of a type are layers known to magic. For example, although both enhancement-fet and depletion-fet contain poly and diffusion, depletion-fet can be thought of as also containing implant (which is not a tile type). So while we can't construct depletion-fet by painting poly and then diffusion, we'd still like it to behave as though it contained both materials. Painting poly or diffusion over a depletion-fet should not change it, and erasing either poly or diffusion should give the other. These semantics are the result of the following rule:

decompose dfet poly diff

The general syntax of both types of composition rules, **compose** and **decompose**, is:

compose *type a1 b1 a2 b2 ...*
decompose *type a1 b1 a2 b2 ...*

The idea is that each of the pairs *a1 b1*, *a2 b2*, etc comprise *type*. In the case of a **compose** rule, painting any *a* atop its corresponding *b* will give *type*, as well as vice-versa. In both **compose** and **decompose** rules, erasing *a* from *type* gives *b*, erasing *b* from *type* gives *a*, and painting either *a* or *b* over *type* leaves *type* unchanged.

Contacts are implicitly composed of their component types, so the result obtained when painting a type *PAINT* over a contact type *CONTACT* will by default depend only on the component types of *CONTACT*. If painting *PAINT* doesn't affect the component types of the contact, then it is considered not to affect the contact itself either. If painting *PAINT* does affect any of the component types, then the result is as though the contact had been replaced by its component types in the layout before type *PAINT* was painted. Similar rules hold for erasing.

A poly-metal-contact has component types poly and metal. Since painting poly doesn't affect either poly or metal, it doesn't affect a poly-metal-contact either. Painting diffusion does affect poly—it turns it into an enhancement-fet—. Hence, painting diffusion over a poly-metal-contact breaks up the contact, leaving enhancement-fet on the poly-diff plane and metal on the metal plane.

The **compose** and **decompose** rules are normally sufficient to specify the desired semantics of painting or erasing. In unusual cases, however, it may be necessary to provide Magic with explicit **paint** or **erase** rules. For example, to specify that erasing metal from a glass contact causes the contact to disappear, the technology file contains the rule:

erase glass metal space

This rule could not have been written as a **decompose** rule because it is

asymmetric; erasing space from a glass contact does not yield metal. The general syntax for these explicit rules is:

paint *have t result* [*p*]
erase *have t result* [*p*]

Here, *have* is the type already present, on plane *p* if it is specified; otherwise, on the home plane of *have*. Type *t* is being painted or erased, and the result is type *result*. Table 7 gives the **compose** section for nMOS.

It's easiest to think of the paint and erase rules as being built up in four passes. The first pass generates the default rules for all non-contact types, and the second pass replaces these as specified by the **compose**, **decompose**, etc. rules, also for non-contact types. At this point, the behavior of the component types of contacts has been completely determined, so the third pass can generate the default rules for all contact types, and the fourth pass can modify these as per any **compose**, etc. rules for contacts.

connect	
poly	pmc,efet,dfet,dcap,bc
diff	bc,dmc
efet,dfet,dcap	pmc,bc
metal	glass,pmc,dmc
glass	pmc,dmc
end	

Table 8. **Connect** section

7. Connect section

For circuit extraction, routing, and some of the net-list operations, Magic needs to know what types are electrically connected. Magic's model of electrical connectivity used is based on signal propagation. Two types should be marked as connected if a signal will *always* pass between the two types, in either direction. For the most part, this will mean that all non-space types within a plane should be marked as connected. The exceptions to this rule are devices (transistors). A transistor should be considered electrically connected to adjacent polysilicon, but not to adjacent diffusion. This models the fact that polysilicon connects to the gate of the transistor, but that the transistor acts as a switch between the diffusion areas on either side of the channel of the transistor.

The lines in the **connect** section of a technology file, as shown in Table 8, each contain a pair of type-lists in the format described in Section 4. Each type in the first list connects to each type in the second list. This does not imply that the types in the first list are themselves connected to each other, or that the types in the second list are connected to each other.

Because connectivity is a symmetric relationship, only one of the two possible orders of two tile types need be specified. Tiles of the same type are always considered to be connected. Contacts are treated specially; they should be specified as connecting to material in all planes spanned by the contact. For example, poly-metal-contact is shown as connecting to several types in the poly-diff plane, as well as several types in the metal plane. The connectivity of a contact should usually be that of its component types, so poly-metal-contact should connect to everything connected to poly, and to everything connected to metal.

```
cifoutput
style fab4.0
scalefactor 200 200
layer NP poly,pmc,efet,dfet,dcap,bc
    labels poly,efet,dfet,dcap,bc
    calma 1 1
layer ND diff,dmc,efet,dfet,dcap,bc
    labels diff
    calma 2 1
layer NM metal,pmc,dmc,glass
    labels metal,pmc,dmc,glass
    calma 3 1
layer NI
    bloat-or dfet,dcap * 200 diff,bc 400
    grow 100
    shrink 100
    calma 4 1
layer NC dmc
    squares 400
    calma 5 1
layer NC pmc
    squares 400
layer NG glass
    calma 6 1
layer NB
    bloat-or bc * 200 diff,dmc 400 dfet 0
    grow 100
    shrink 100
    calma 7 1
end
```

Table 9. **Cifoutput** section

8. Cifoutput section

The layers stored by Magic do not always correspond to physical mask layers. For example, there is no physical layer corresponding to depletion-fet; instead, the actual circuit must be built up by overlapping poly and diffusion, then covering the entire transistor area with a depletion implant. When writing CIF (Caltech Intermediate Form) or Calma GDS-II files, Magic generates the actual geometries that will appear on the masks used to fabricate the circuit. The **cifoutput** section of the technology file describes how to generate mask layers from Magic's abstract layers.

8.1. CIF styles

The technology file can contain several different specifications of how to generate CIF. Each of these is called a CIF *style*. Different styles may be used for fabrication at different feature sizes, or for totally different purposes. For example, some of the Magic technology files contain a style "plot" that generates CIF pseudo-layers that have exactly the same shapes as the Magic layers. This style is used for generating plots that look just like what appears on the color display; it makes no sense for fabrication. Lines of the form

style *name*

are used to end the description of the previous style and start the description of a new style. The Magic command **:cif ostyle** *name* is typed by users to change the current style used for output. The first style in the technology file is used by default for CIF output if the designer doesn't issue a **:cif style** command. If the first line of the **cifoutput** section isn't a **style** line, then Magic uses an initial style name of **default**.

8.2. Scaling

Each style must contain a line of the form

scalefactor *scale* [*reducer*]

that tells how to scale Magic coordinates into CIF coordinates. The argument *scale* indicates how many hundredths of a micron correspond to one Magic unit. Because of certain numerical problems with the CIF representation, *scale* must always be an even number. The second parameter, *reducer*, is optional. If it is specified, it is used to increase the readability and decrease the size of CIF files. Each CIF coordinate is divided by *reducer* before being written to the CIF file, then a uniform upward scalefactor of *reducer* is specified once for the whole file. This has no effect on the CIF except to make the individual CIF numbers smaller and thereby reduce the sizes of CIF files. *Reducer* must be a positive integer, and must evenly divide into every other dimension specified in any statement for this style. *Reducer* must also divide one-half of *scale*. If this sounds confusing, the easiest thing is to leave *reducer* unspecified, in which case the value 1 is used.

8.3. Layer descriptions

The main body of information for each CIF style is a set of layer descriptions. Each layer description consists of one or more lines describing how to generate the CIF for a single layer. The first line of each description is one of

layer *name* [*layers*]
or
templayer *name* [*layers*]

These statements are identical, except that templayers are not output in the CIF file. They are used only to build up intermediate results used in generating the "real" layers. In each case, *name* is the CIF name to be used for the layer. If *layers* is specified, it consists of a comma-separated list of Magic layers and previously-defined CIF layers in this style; these layers form the initial contents of the new CIF layer (note: the layer lists in this section are less general than what was described in Section 4; tildes and parentheses are not allowed). If *layers* is not specified, then the new CIF layer is initially empty. The following statements are used to modify the contents of a CIF layer before it is output.

After the **layer** or **templayer** statement come several statements specifying geometrical operations to apply in building the CIF layer. Each statement takes the current contents of the layer, applies some operation to it, and produces the new contents of the layer. The last geometrical operation for the layer determines what is actually output in the CIF file. The geometrical operations are:

or *layers*
and *layers*
and-not *layers*
grow *amount*
shrink *amount*
bloat-or *layers layers2 amount layers2 amount ...*
bloat-max *layers layers2 amount layers2 amount ...*
bloat-min *layers layers2 amount layers2 amount ...*
squares *size*
squares *border size separation*

The operation **or** takes all the *layers* (which may be either Magic layers or previously-defined CIF layers), and or's them with the material already in the CIF layer. The operation **and** is similar to **or**, except that it and's the layers with the material in the CIF layer (in other words, any CIF material that doesn't lie under material in *layers* is removed from the CIF layer). **And-not** finds all areas covered by *layers* and erases current CIF material from those areas. **Grow** and **shrink** will uniformly grow or shrink the current CIF layer by *amount* units, where *amount* is specified in CIF units, not Magic units.

The three **bloat** operations provide selective forms of growing. In these statements, all the layers must be Magic layers. Each operation examines all the tiles in *layers*, and grows the tiles by a different distance on each side, depending on the rest of the line. Each pair *layers2 amount* specifies some tile types and a distance (in CIF units). Where a tile of type *layers* abuts a tile of type *layers2*, the first tile is grown on that side by *amount*. The result is or'ed with the current

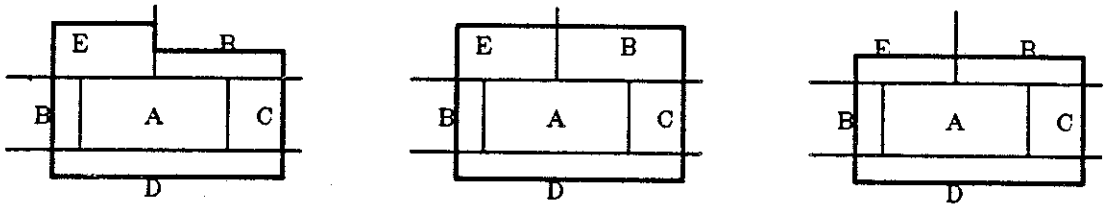


Figure 2. The three different forms of **bloat** behave slightly differently when two different bloat distances apply along the same side of a tile. In each of the above examples, the CIF that would be generated is shown in bold outline. If **bloat-or** is specified, a jagged edge may be generated, as on the left. If **bloat-max** is used, the largest bloat distance for each side is applied uniformly to the side, as in the center. If **bloat-min** is used, the smallest bloat distance for each side is applied uniformly to the side, as on the right.

contents of the CIF plane. The layer "*" may be used as *layers2* to indicate all tile types. Where tiles only have a single type of neighbor on each side, all three forms of **bloat** are identical. Where the neighbors are different, the three forms are slightly different, as illustrated in Figure 2. Note: all the layers specified in any given **bloat** operation must lie on a single Magic plane. For **bloat-or** all distances must be positive. In **bloat-max** and **bloat-min** the distances may be negative to provide a selective form of shrinking.

In retrospect, it's not clear that **bloat-max** and **bloat-min** are very useful operations. The problem is that they operate on tiles, not regions. This can cause unexpected behavior on concave regions. For example, if the region being bloated is in the shape of a "T", a single bloat factor will be applied to the underside of the horizontal bar. If you use **bloat-max** or **bloat-min**, you should probably specify design-rules that require the shapes being bloated to be convex.

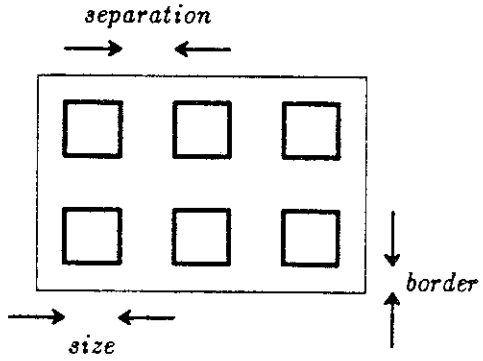


Figure 3. The **squares** operator chops each tile up into squares, as determined by the *border*, *size*, and *separation* parameters. In the example, the bold lines show the CIF that would be generated by a **squares** operation. The squares of material are always centered so that the borders on opposite sides are the same.

The last geometric operation is called **squares**. It examines each tile on the CIF plane, and replaces that tile with one or more squares of material. Each square is *size* CIF units across, and squares are separated by *separation* units. A border of at least *border* units is left around the edge of the original tile, if

possible. This operation is used to generate contact vias, as in Figure 3. If only one argument is given in the **squares** statement, then *separation* defaults to *size* and *border* defaults to *size*/2. If a tile doesn't hold an integral number of squares, extra space is left around the edges of the tile and the squares are centered in the tile. If the tile is so small that not even a single square can fit and still leave enough border, then the border is reduced. If a square won't fit in the tile, even with no border, then no material is generated. The **squares** operation must be used with some care, in conjunction with the design rules. For example, if there are several adjacent skinny tiles, there may not be enough room in any of the tiles for a square, so no material will be generated at all. Whenever you use the **squares** operator, you should use design rules to prohibit adjacent contact tiles, and you should always use the **no_overlap** rule to prevent unpleasant hierarchical interactions. The problems with hierarchy are discussed in Section 8.6 below, and design rules are discussed in Section 10.

8.4. Labels

There is an additional statement permitted in the **cifoutput** section as part of a layer description:

labels *Magiclayers*

This statement tells Magic that labels attached to Magic layers *Magiclayers* are to be associated with the current CIF layer. Each Magic layer should only appear in one such statement for any given CIF style. If a Magic layer doesn't appear in any **labels** statement, then it is not attached to a specific layer when output in CIF.

8.5. Calma (GDS II Stream format) layers

Each layer description in the **cifoutput** section may also contain one of the following statements:

calma *calmaNumber calmaType*

This statement tells Magic which layer number and data type to use when the **calma** command outputs Calma GDS II Stream format for this CIF layer. Both *calmaNumber* and *calmaType* should be positive integers, between 0 and 63. Each CIF layer should have a different *calmaNumber*. If there is no **calma** line for a given CIF layer, then that layer will not be output by the **:calma** command.

8.6. Hierarchy

Hierarchical designs make life especially difficult for the CIF generator. The CIF corresponding to a collection of subcells may not necessarily be the same as the sum of the CIF's of the individual cells. For example, if a layer is generated by growing and then shrinking, nearby features from different cells may merge together so that they don't shrink back to their original shapes (see Figure 4). If Magic generates CIF separately for each cell, the interactions between cells will not be reflected properly. The CIF generator attempts to avoid these problems. Although it generates CIF in a hierarchical representation that matches the Magic cell structure, it tries to ensure that the resulting CIF patterns are exactly the

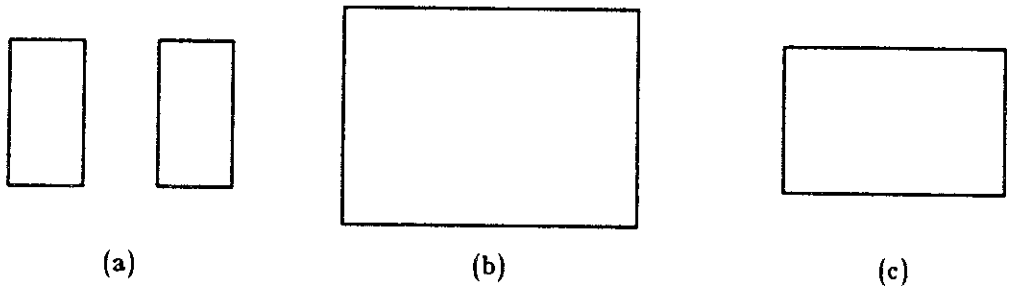


Figure 4. If the operator **grow 100** is applied to the shapes in (a), the merged shape in (b) results. If the operator **shrink 100** is applied to (b), the result is (c). However, if the two original shapes in (a) belong to different cells, and if CIF is generated separately in each cell, the result will be the same as in (a). Magic handles this by outputting additional information in the parent of the subcells to fill in the gap between the shapes.

same as if the entire Magic design had been flattened into a single cell and then CIF were generated from the flattened design. It does this by looking in each cell for places where subcells are close enough to interact with each other or with paint in the parent. Where this happens, Magic flattens the interaction area and generates CIF for it; then Magic flattens each of the subcells separately and generates CIF for them. Finally, it compares the CIF from the subcells with the CIF from the flattened parent. Where there is a difference, Magic outputs extra CIF in the parent to compensate.

Magic's hierarchical approach only works if the overall CIF for the parent ends up covering at least as much area as the CIFs for the individual components, so all compensation can be done by adding extra CIF to the parent. In mathematical terms, this requires each geometric operation to obey the rule

$$Op(A \cup B) \supseteq Op(A) \cup Op(B)$$

The operations **and**, **or**, **grow**, and **shrink** all obey this rule. Unfortunately, the **and-not**, **bloat**, and **squares** operations do not. For example, if there are two partially-overlapping tiles in different cells, the squares generated from one of the cells may fall in the separations between squares in the other cell, resulting in much larger areas of material than expected. There are two ways around this problem. One way is to use the design rules to prohibit problem situations from arising. This applies mainly to the **squares** operator. Tiles from which squares are made should never be allowed to overlap other such tiles in different cells unless the overlap is exact, so each cell will generate squares in the same place. You can use the **exact_overlap** design rule for this.

The second approach is to leave things up to the designer. When generating CIF, Magic issues warnings where there is less material in the children than the parent. The designer can locate these problems and eliminate the interactions that cause the trouble. Warning: Magic does not check the **squares** operations for hierarchical consistency, so you absolutely must use **exact_overlap** design rule checks! Right now, the **cifoutput** section of the technology is one of the trickiest things in the whole file, particularly since errors here may not show up until your chip comes back and doesn't work. Be extremely careful when writing

this part!

```
cifinput
style lambda=2microns
scalefactor 200
layer poly NP
    labels NP
layer diff ND
    labels ND
layer metal NM
    labels NM
layer efet NP
    and ND
layer dfet NI
    and NP
    and ND
layer pmc NC
    grow 200
    and NM
    and NP
layer dmc NC
    grow 200
    and NM
    and ND
layer bc NB
    and NP
    and ND
layer glass NG
ignore NX
calma NP 1 *
calma ND 2 *
calma NM 3 *
calma NI 4 *
calma NC 5 *
calma NG 6 *
calma NB 7,8 *
end
```

Table 10. **Cifinput** section. The order of the layers is important, since each Magic layer overrides the previous ones just as if they were painted by hand.

9. Cifinput section

In addition to writing CIF, Magic can also read in CIF files using the **:cif read** *file* command. The **cifinput** section of the technology file describes how to convert from CIF mask layers to Magic tile types. In addition, it provides information to the Calma reader to allow it to read in Calma GDS II Stream format files. The **cifinput** section is very similar to the **cifoutput** section. It can contain several styles, with a line of the form

style *name*

used to end the description of the previous style (if any), and start a new CIF input style called *name*. If no initial style name is given, the name **default** is assigned. Each style must have a statement of the form

scalefactor *centimicrons*

to indicate how many hundredths of a micron correspond to one unit in Magic.

Like the **cifoutput** section, each style consists of a number of layer descriptions. A layer description contains one or more lines describing a series of geometric operations to be performed on CIF layers. The result of all these operations is painted on a particular Magic layer just as if the user had painted that information by hand. A layer description begins with a statement of the form

layer *magicLayer* [*layers*]

In the **layer** statement, *magicLayer* is the Magic layer that will be painted after performing the geometric operations, and *layers* is an optional list of CIF layers. If *layers* is specified, it is the initial value for the layer being built up. If *layers* isn't specified, the layer starts off empty. As in the **cifoutput** section, each line after the *layer* statement gives a geometric operation that is applied to the previous contents of the layer being built in order to generate new contents for the layer. The result of the last geometric operation is painted into the Magic database.

The geometric operations that are allowed in the **cifinput** section are a subset of those permitted in the **cifoutput** section:

or *layers*
and *layers*
and-not *layers*
grow *amount*
shrink *amount*

In these commands the *layers* must all be CIF layers, and the *amounts* are all CIF distances (centimicrons). As with the **cifoutput** section, layers can only be specified in simple comma-separated lists: tildes and slashes are not permitted.

When CIF files are read, all the mask information is read for a cell before performing any of the geometric processing. After the cell has been completely read in, the Magic layers are produced and painted in the order they appear in the technology file. In general, the order that the layers are processed is important since each layer will usually override the previous ones. For example,

in the nMOS tech file shown in Table 10 the commands for **efet** will result in the **efet** layer being generated not only where there are enhancement transistors but also where there are depletion transistors and buried contacts. The descriptions for **dfet** and **bc** appear later in the section, so those layers will replace the **efet** material that was originally painted.

Labels are handled in the **cifinput** section just like in the **cifoutput** section. A line of the form

labels *layers*

means that the current Magic layer is to receive all CIF labels on *layers*. This is actually just an initial layer assignment for the labels. Once a CIF cell has been read in, Magic scans the label list and re-assigns labels if necessary. In the example of Table 10, if a label is attached to the CIF layer NP then it will be assigned to the Magic layer **poly**. However, the polysilicon may actually be part of a poly-metal contact, which is Magic layer **pmc**. After all the mask information has been processed, Magic checks the material underneath the layer, and adjusts the label's layer to match that material (**pmc** in this case). This is the same as what would happen if a designer painted **poly** over an area, attached a label to the material, then painted **pmc** over the area.

No hierarchical mask processing is done for CIF input. Each cell is read in and its layers are processed independently from all other cells; Magic assumes that there will not be any unpleasant interactions between cells as happens in CIF output (and so far, at least, this seems to be a valid assumption).

If Magic encounters a CIF layer name that doesn't appear in any of the lines for the current CIF input style, it issues a warning message and ignores the information associated with the layer. If you would like Magic to ignore certain layers without issuing any warning messages, insert a line of the form

ignore *cifLayers*

where *cifLayers* is a comma-separated list of one or more CIF layer names.

Calma layers are specified via **calma** lines, which should appear at the end of the **cifinput** section. They are of the form:

calma *cifLayer calmaLayers calmaTypes*

The *cifLayer* is one of the CIF types mentioned in the **cifinput** section. Both *calmaLayers* and *calmaTypes* are one or more comma-separated integers between 0 and 63. The interpretation of a **calma** line is that any Calma geometry whose layer is any of the layers in *calmaLayers*, and whose type is any of the types in *calmaTypes*, should be treated as the CIF layer *cifLayer*. Either or both of *calmaLayers* and *calmaTypes* may be the character * instead of a comma-separated list of integers; this character means *all* layers or types respectively. It is commonly used for *calmaTypes* to indicate that the Calma type of a piece of geometry should be ignored.

Just as for CIF, Magic also issues warnings if it encounters unknown Calma layers while reading Stream files. If there are layers that you'd like Magic to ignore without issuing warnings, assign them to a dummy CIF layer and ignore the CIF layer.

#define	allDiff	diff,dmc,bc,efet,dfet,dcap
#define	allPoly	poly,pmc,bc,efet,dfet,dcap
#define	tran	efet,dfet
#define	contact	pmc,dmc
#define	allMetal	metal,pmc,dmc,glass
#define	allpdTypes	s,diff,poly,dmc,pmc,bc,dfet,dcap,efet

Table 11a. Abbreviations for sets of tile types.

width	allDiff	2	*"Diffusion width must be at least 2"*
width	dmc	4	*"Metal-diff contact width must be at least 4"*
width	allPoly	2	*"Polysilicon width must be at least 2"*
width	pmc	4	*"Metal-poly contact width must be at least 4"*
width	bc	2	*"Buried contact width must be at least 2"*
width	efet	2	*"Enhancement FET width must be at least 2"*
width	dfet	2	*"Depletion FET width must be at least 2"*
width	dcap	2	*"Depletion capacitor width must be at least 2"*
width	allMetal	3	*"Metal width must be at least 3"*

Table 11b. Width rules in the **drc** section.

10. Drc section

The design rules used by Magic's design rule checker come entirely from the technology file. We'll look first at two simple kinds of rules, **width** and and **spacing**. Most of the rules in the **drc** section are one or the other of these kinds of rules.

10.1. Width rules

The minimum width of a collection of types, taken together, is expressed by a **width** rule. Such a rule has the form:

width *type-list width error*

where *type-list* is a set of tile types (see Section 4 for syntax), *width* is an integer, and *error* is a string, enclosed in double quotes, that can be printed by the command **:drc why** if the rule is violated. A width rule requires that all regions containing any types in the set *types* must be wider than w in both dimensions. For example, in Table 11b, the rule

width dfet 2 *"Depletion FET width must be at least 2"*

means that depletion-mode devices must be at least 2 units wide whenever they appear. The *type-list* field may contain more than a single type, as in the following rule:

width allMetal 3 *"Metal width must be at least 3"*

which means that all regions consisting of the types metal, poly-metal-contact, diff-metal-contact, or glass must be at least 3 units wide. Because many of the rules in the **drc** section refer to the same sets of layers, the **#define** facility of the C preprocessor is used to define a number of macros for these sets of layers. Table 11a gives a complete list.

All of the layers named in any one width rule must lie on the same plane. However, if some of the layers are contacts, Magic will substitute a different contact image if the named image isn't on the same plane as the other layers.

spacing	allDiff	allDiff	3	**touching_ok** \
	"Diff-diff separation must be at least 3"			
spacing	allPoly	allPoly	2	**touching_ok** \
	"Poly-poly separation must be at least 2"			
spacing	tran	contact	1	**touching_illegal** \
	"Transistor-contact separation must be at least 1"			
spacing	efet	dfet,dcap	3	**touching_illegal** \
	"Enhancement-depletion transistor separation must be at least 3"			
spacing	allMetal	allMetal	3	**touching_ok** \
	"Metal-metal separation must be at least 3"			
spacing	bc	efet	3	**touching_illegal** \
	"Buried contact-transistor separation must be at least 3"			

Table 11c. Spacing rules in the **drc** section.

10.2. Spacing rules

The second simple kind of design rule is a **spacing** rule. It comes in two flavors: **touching_ok**, and **touching_illegal**, both with the following syntax:

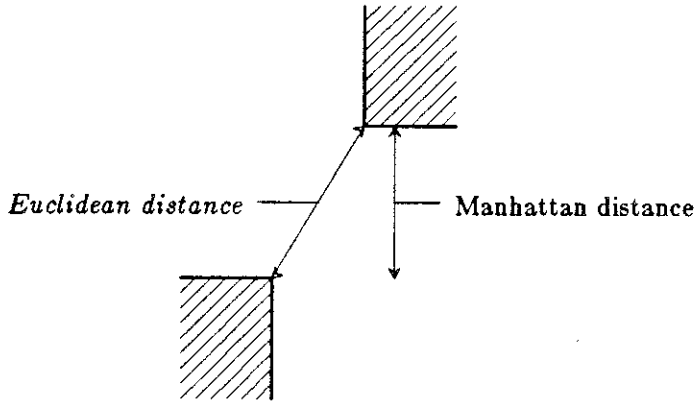


Figure 5. For design rule checking, the Manhattan distance between two horizontally or vertically aligned points is just the normal Euclidean distance. If they are not so aligned, then the Manhattan distance is the length of the longest side of the right triangle forming the diagonal line between the points.

spacing *types1 types2 distance flavor error*

The first flavor, **touching_ok**, does not prohibit *types1* and *types2* from being immediately adjacent. It merely requires that any type in the set *types1* must be separated by a "Manhattan" distance of at least *distance* units from any type in the set *types2* that is not immediately adjacent to the first type. See Figure 5 for an explanation of Manhattan distance for design rules. As an example, consider the metal separation rule:

spacing allMetal allMetal 3 **touching_ok** \
"Metal-metal separation must be at least 3"

This rule is symmetric (*types1* is equal to *types2*), and requires, for example, that a poly-metal-contact be separated by at least 3 units from a piece of metal. However, this rule does not prevent the poly-metal-contact from touching a piece of metal. In **touching_ok** rules, all of the layers in both *types1* and *types2* must be stored on the same plane (Magic will substitute different contact images if necessary).

The second flavor of spacing rule, **touching_illegal**, disallows adjacency. It is used for rules where *types1* and *types2* can never touch, as in the following:

spacing bc efet 3 **touching_illegal** \
"Buried contact-transistor separation must be at least 3"

Buried contacts and enhancement mode transistors must be 3 units apart; they cannot touch. It is only with the **touching_illegal** rules that *types1* and *types2* are not the same. In **touching_illegal** rules, the layers in *types1* and *types2* may be on different planes; Magic will find violations between material on different planes.

10.3. Edge rules

The width and spacing rules just described are actually translated by Magic into an underlying, edge-based rule format. This underlying format can handle rules more general than simple widths and spacings, and is accessible to the writer of a technology file via **edge** rules. These rules are applied at boundaries between material of two different types, in any of four directions as shown in Figure 6. The design rule table contains a separate list of rules for each possible combination of materials on the two sides of an edge.

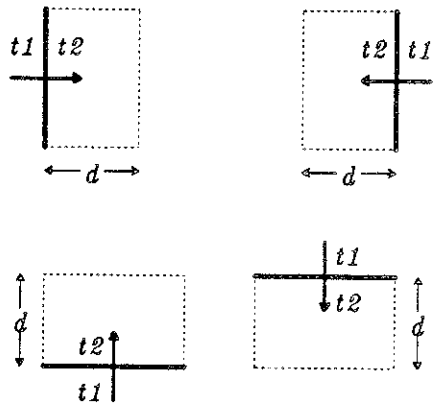


Figure 6. Design rules are applied at the edges between tiles in the same plane. A rule is specified in terms of type *t1* and type *t2*, the materials on either side of the edge. Each rule may be applied in any of four directions, as shown by the arrows. The simplest rules require that only certain mask types can appear within distance *d* on *t2*'s side of the edge.

In its simplest form, a rule specifies a distance and a set of mask types: only the given types are permitted within that distance on *type2*'s side of the edge. This area is referred to as the *constraint region*. Unfortunately, this simple scheme will miss errors in corner regions, such as the case shown in Figure 7. To eliminate these problems, the full rule format allows the constraint region to be extended past the ends of the edge under some circumstances. See Figure 8 for an illustration of the corner rules and how they work. Table 12 gives a complete description of the information in each design rule.

Edge rules are specified in the technology file using the following syntax:

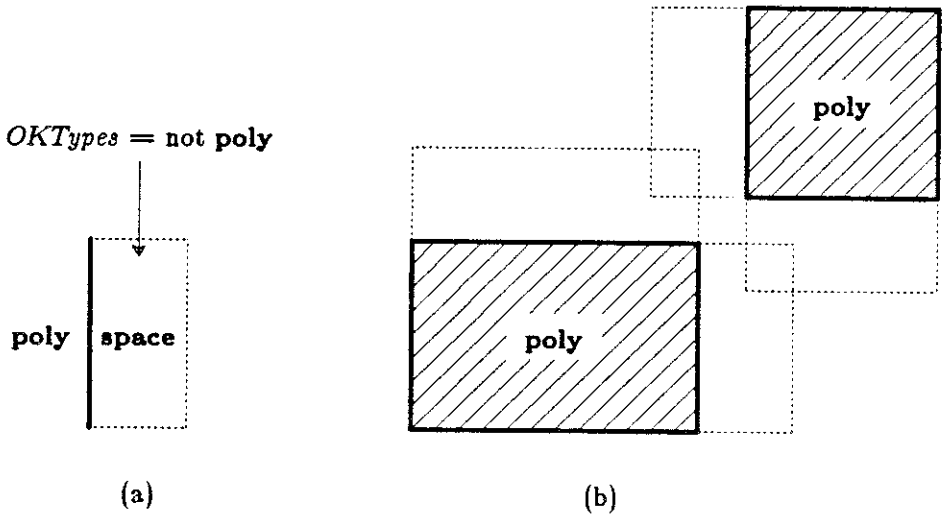


Figure 7. If only the simple rules from Figure 6 are used, errors may go unnoticed in corner regions. For example, the polysilicon spacing rule in (a) will fail to detect the error in (b).

edge *types1 types2 d OKTypes cornerTypes cornerDist error* [*plane*]

Both *types1* and *types2* are type-lists. An edge rule is generated for each pair consisting of a type from *types1* and a type from *types2*. All the types in *types1*, *types2*, and *cornerTypes* must lie on a single plane. See Figure 8 for an example edge rule.

Some of the edge rules in Magic have the property that if a rule is violated between two pieces of geometry, the violation can be discovered looking from either piece of geometry toward the other. To capitalize on this, Magic normally applies an edge rule only in two of the four possible directions: bottom-to-top and left-to-right, reducing the work it has to do by a factor of two. Also, the corner extension is only performed to one side of the edge: to the top for a left-to-right rule, and to the left for a bottom-to-top rule. All of the width and spacing rules translate neatly into edge rules.

However, you'll probably find it easiest when you're writing edge rules to insist that they be checked in all directions. To do this, write the rule the same way except use the keyword **edge4way** instead of **edge**:

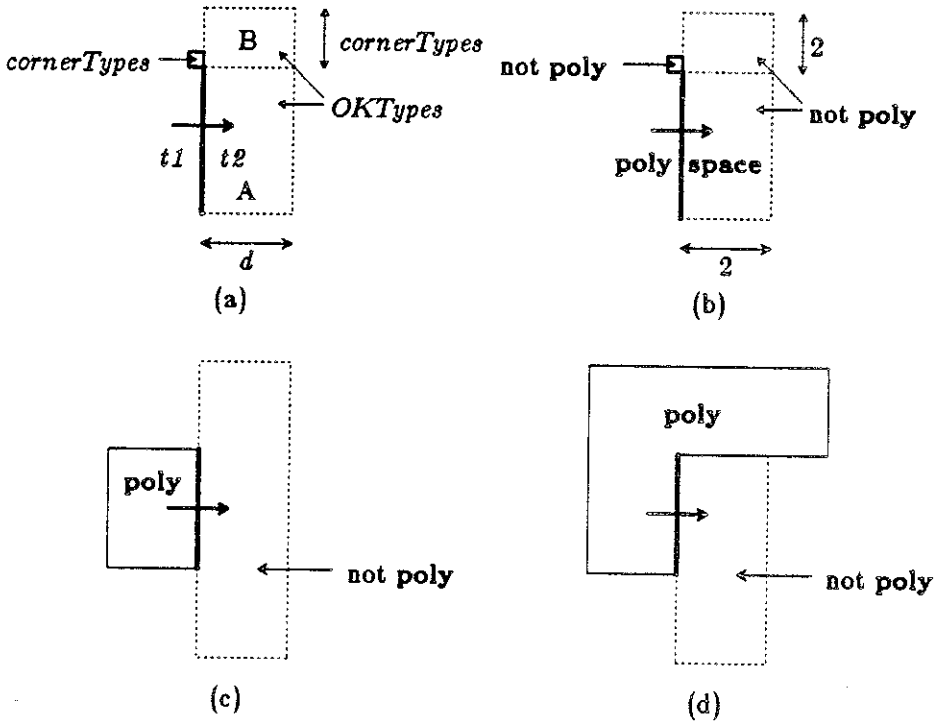


Figure 8. The complete design rule format is illustrated in (a). Whenever an edge has *type1* on its left side and *type2* on its right side, the area A is checked to be sure that only *OKTypes* are present. If the material just above and to the left of the edge is one of *cornerTypes*, then area B is also checked to be sure that it contains only *OKTypes*. A similar corner check is made at the bottom of the edge. Figure (b) shows a polysilicon spacing rule, (c) shows a situation where corner extension is performed on both ends of the edge, and (d) shows a situation where corner extension is made only at the bottom of the edge. If the rule described in (d) were to be written as an **edge** rule, it would look like:

edge poly space 2 ~poly ~poly 2 \
"*Poly-poly separation must be at least 2*"

edge4way tran diff 2 diff,dmc diff 2 \
"Diffusion must overhang transistor by at least 2"

Not only are **edge4way** rules checked in all four directions, but the corner extension is performed on *both* sides of the edge. For example, when checking a rule from left-to-right, the corner extension is performed both to the top and to the bottom. **Edge4way** rules take twice as much time to check as **edge** rules, so it's to your advantage to use **edge** rules wherever you can.

Normally, an edge rule is checked completely within a single plane: both the edge that triggers the rule and the constraint area to check fall in the same plane. However, the *plane* argument can be specified in an edge rule to force Magic to perform the constraint check on a plane different from the one containing the triggering edge. In this case, *OKTypes* must all be tile types in *plane*. This feature is used, for example, in our CMOS process to ensure that polysilicon and diffusion edges don't lie underneath metal2 contacts:

Parameter	Meaning
type1	Material on first side of edge.
type2	Material on second side of edge.
d	Distance to check on second side of edge.
OKTypes	List of layers that are permitted within *d* units on second side of edge.
cornerTypes	List of layers that cause corner extension.
cornerDist	Amount to extend constraint area when *cornerTypes* matches.
plane	Plane on which to check constraint region (defaults to same plane as *type1* and *type2* and *cornerTypes*).

Table 12. The parts of an edge-based rule.

```
edge          diff      s,poly,pmc      1       s           s,poly,pmc      1 \
              "Diff-poly separation must be at least 1"
edge          poly      s,diff,dmc      1       s           s,diff,dmc      1 \
              "Diff-poly separation must be at least 1"
edge          dmc       s,poly          1       s           s,poly          1 \
              "Diff-poly separation must be at least 1"
edge          pmc       s,diff          1       s           s,diff          1 \
              "Diff-poly separation must be at least 1"
edge          tran      s               1       0           0               0 \
              "Transistor overhang is missing"
edge          s         tran            1       0           0               0 \
              "Transistor overhang is missing"
edge4way      bc        diff,dmc        4       ~efet       allpdTypes      3 \
              "Buried contact-transistor separation must be at least 4 on diff side"
edge4way      dfet      bc              3       bc          0               0 \
              "Buried contact next to depletion transistor must be at least 3x2"
edge4way      tran      poly            2       poly,pmc    poly            2 \
              "Polysilicon must overhang transistor by at least 2"
edge4way      tran      diff            2       diff,dmc    diff            2 \
              "Diffusion must overhang transistor by at least 2"
```

Table 11d. Edge rules in the **drc** section.

edge4way allPoly ~allPoly/poly-diff 1 ~m2c/metal2 0 0 \
"Polysilicon edges must not appear under m2contact" metal2

10.4. Overlap Rules

In order for CIF generation and circuit extraction to work properly, certain kinds of overlaps between subcells must be prohibited. The design-rule checker provides two kinds of rules for restricting overlaps. They are

exact_overlap *type-list*
no_overlap *type-list1 type-list2*

In the **exact_overlap** rule, *type-list* indicates one or more tile types. If a cell contains a tile of one of these types and that tile is overlapped by another tile of the same type from a different cell, then the overlap must be exact: the tile in each cell must cover exactly the same area. Abutment between tiles from different cells is considered to be a partial overlap, so it is prohibited too. This rule is used to ensure that the CIF **squares** operator will work correctly, as described in Section 8.6. See Table 11e for the **exact_overlap** rule from the standard nmos technology file.

exact_overlap	dmc,pmc	
no_overlap	efet,dfet	efet,dfet

Table 11e. Exact_overlap rule in the **drc** section.

The **no_overlap** rule makes illegal any overlap between a tile in *type-list1* and a tile in *type-list2*. You should rarely, if ever, need to specify **no_overlap** rules, since Magic automatically prohibits many kinds of overlaps between subcells. After reading the technology file, Magic examines the paint table and applies the following rule: if two tile types A and B are such that the result of painting A over B is neither A nor B, or the result of painting B over A isn't the same as the result of painting A over B, then A and B are not allowed to overlap. Such overlaps are prohibited because they change the structure of the circuit. Overlaps are supposed only to connect things without making structural changes. Thus, for example, poly can overlap poly-metal-contact without violating the above rules, but poly may not overlap diffusion because the result is efet, which is neither poly nor diffusion. The only **no_overlap** rules you should need to specify are rules to keep transistors from overlapping other transistors of the same type.

11. Extract section

The **extract** section of a technology file contains the parameters used by Magic's circuit extractor. Each line in this section begins with a keyword that determines the interpretation of the remainder of the line. Table 13 gives an example **extract** section.

This section is like the **cifinput** and **cifoutput** sections in that it supports multiple extraction styles. Each style is preceded by a line of the form

```
extract
style       default
lambda      200
step        100
sidehalo    4

resist      poly,pmc,efet,dfet,bc     30000
resist      diff,dmc                  10000
resist      metal,glass                  30

areacap     poly,efet,dfet              200
areacap     metal,glass                 120
areacap     diff                        400
areacap     bc                          600
areacap     dmc                         520
areacap     pmc                         320

perimc      diff,dmc,bc   space,dfet,efet        200

overlap     metal         diff                  100
overlap     metal         poly                  120

#define     extPoly       poly,pmc
#define     extMet        metal,pmc/m,dmc/m

sidewall    extPoly   space   space      extPoly    50
sidewall    extMet    space   space      extMet     60

sideoverlap extMet    space   ndiff,pdiff   80
sideoverlap extMet    space   poly          70

fet         efet      diff    2          efet    GND!   0   0
fet         dfet      diff,bc 2          dfet    GND!   0   0
fet         dcap      diff,bc 1          dcap    GND!   0   0
end
```

Table 13. **Extract** section

style *stylename*

All subsequent lines up to the next **style** line or the end of the section are interpreted as belonging to extraction style *stylename*. If there is no initial **style** line, the first style will be named "default".

The keywords **areacap**, **perimcap**, and **resist** define the capacitance to substrate and the sheet resistivity of each of the Magic layers in a layout. All capacitances that appear in the **extract** section are specified as an integral number of attofarads (per unit area or perimeter), and all resistances as an integral number of milliohms per square.

The **areacap** keyword is followed by a list of types and a capacitance to substrate, as follows:

areacap *types C*

Each of the types listed in *types* has a capacitance to substrate of *C* attofarads per square lambda. Each type can appear in at most one **areacap** line. If a type does not appear in any **areacap** line, it is considered to have zero capacitance to substrate per unit area. Since most analysis tools compute transistor gate capacitance directly from the area of the transistor's gate, Magic should produce node capacitances that do not include gate capacitances. To ensure this, all transistors should have zero **areacap** values.

The **perimcap** keyword is followed by two type-lists and a capacitance to substrate, as follows:

perimcap *intypes outtypes C*

Each edge that has one of the types in *intypes* on its inside, and one of the types in *outtypes* on its outside, has a capacitance to substrate of *C* attofarads per lambda. This can also be used as an approximation of the effects due to the sidewalls of diffused areas. As for **areacap**, each unique combination of an *intype* and an *outtype* may appear at most once in a **perimcap** line. Also as for **areacap**, if a combination of *intype* and *outtype* does not appear in any **perimcap** line, its perimeter capacitance per unit length is zero.

The **resist** keyword is followed by a type-list and a resistance as follows:

resist *types R*

The sheet resistivity of each of the types in *types* is *R* milliohms per square.

Each **resist** line in fact defines a "resistance class". When the extractor outputs the area and perimeter of nodes in the **.ext** file, it does so for each resistance class. Normally, each resistance class consists of all types with the same resistance. However, if you wish to obtain the perimeter and area of each type separately in the **.ext** file, you should make each into its own resistance class by using a separate **resist** line for each type.

Magic also extracts internodal coupling capacitances, as illustrated in Figure 9. The keywords **overlap**, **sidewall**, **sideoverlap**, and **sidehalo** provide the parameters needed to do this.

Overlap capacitance is between pairs of tile types, and is described by the **overlap** keyword as follows:

overlap *toptypes bottomtypes cap* [*shieldtypes*]

where *toptypes*, *bottomtypes*, and optionally *shieldtypes* are type-lists and *cap* is a capacitance in attofarads per square lambda. The extractor searches for tiles

whose types are in *toptypes* that overlap tiles whose types are in *bottomtypes*, and that belong to different electrical nodes. (The planes of *toptypes* and *bottomtypes* must be disjoint). When such an overlap is found, the capacitance to substrate of the node of the tile in *toptypes* is deducted for the area of the overlap, and replaced by a capacitance to the node of the tile in *bottomtypes*.

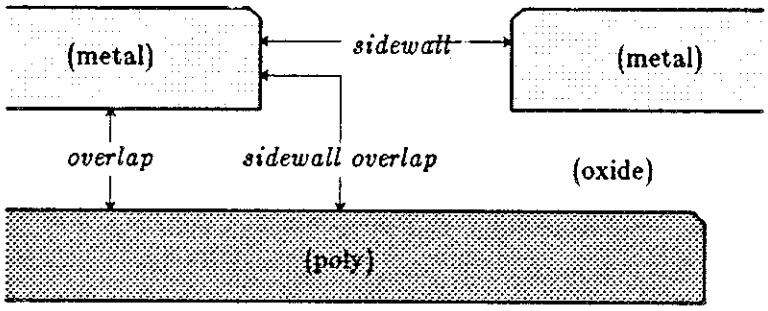


Figure 9. Magic extracts three kinds of internodal coupling capacitance. This figure is a side view of a set of masks that shows all three kinds of capacitance. *Overlap* capacitance is parallel-plate capacitance between two different kinds of material when they overlap. *Parallel wire* capacitance is fringing-field capacitance between the parallel vertical edges of two pieces of material. *Sidewall overlap* capacitance is fringing-field capacitance between the vertical edge of one piece of material and the horizontal surface of another piece of material that overlaps the vertical edge.

If *shieldtypes* are specified, overlaps between *toptypes* and *bottomtypes* that also overlap a type in *shieldtypes* are not counted. The types in *shieldtypes* must appear on a different plane (or planes) than any of the types in *toptypes* or *bottomtypes*.

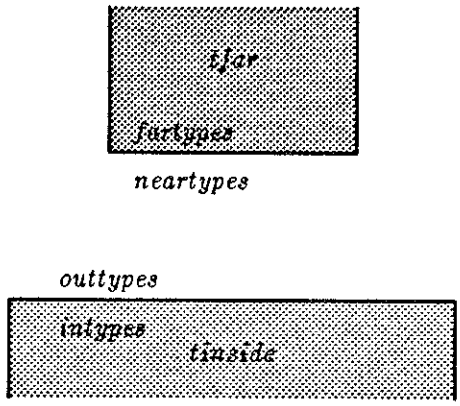


Figure 10. Parallel wire capacitance is between pairs of edges. The capacitance applies between the tiles *tinside* and *tfar* above, where *tinside*'s type is one of *intypes*, and *tfar*'s type is one of *fartypes*.

Parallel wire capacitance is between pairs of edges, and is described by the **sidewall** keyword:

sidewall *intypes outtypes neartypes fartypes cap*

where *intypes*, *outtypes*, *neartypes*, and *fartypes* are all type-lists, described in

Figure 10. *Cap* is half the capacitance in attofarads per lambda when the edges are 1 lambda apart. Parallel wire coupling capacitance is computed as being inversely proportional to the distance between two edges: at 2 lambda separation, it is equal to the value *cap*; at 4 lambda separation, it is half of *cap*. This approximation is not very good, in that it tends to overestimate the coupling capacitance between wires that are farther apart.

To reduce the amount of searching done by Magic, there is a threshold distance beyond which the effects of parallel wire coupling capacitance are ignored. This is set as follows:

sidehalo *distance*

where *distance* is the maximum distance between two edges at which Magic considers them to have parallel wire coupling capacitance. **If this number is not set explicitly in the technology file, it defaults to 0, with the result that no parallel wire coupling capacitance is computed.**

Sidewall overlap capacitance is between material on the inside of an edge and overlapping material of a different type. It is described by the **sideoverlap** keyword:

sideoverlap *intypes outtypes ovtypes cap*

where *intypes*, *outtypes*, and *ovtypes* are type-lists and *cap* is capacitance in attofarads per lambda. This is the capacitance associated with an edge with a type in *intypes* on its inside and a type in *outtypes* on its outside, that overlaps a tile whose type is in *ovtypes*. See Figure 9.

Transistors are represented in Magic by explicit tiletypes. The extraction of a fet (with gate, sources, and drains) from a collection of transistor tiles is governed by the information in a **fet** line. This line has the following format:

fet *types dtypes min-nterms name snode gscap gccap*

Types is a list of those tiletypes that make up this type of transistor. Normally, there will be only one type in this list, since Magic usually represents each type of transistor with a different tiletype.

Dtypes is a list of those tiletypes that connect to the diffusion terminals of the fet. Each transistor of this type must have at least *min-nterms* distinct diffusion terminals; otherwise, the extractor will generate an error message. For example, an **efet** in the nMOS technology must have a source and drain in addition to its gate; *min-nterms* for this type of fet is 2. The tiletypes connecting to the gate of the fet are the same as those specified in the **connect** section as connecting to the fet tiletype itself.

Name is a string used to identify this type of transistor to simulation programs. *Snode* is the name of the node to which the substrate of this transistor is connected. *Gscap* is the capacitance between the transistor's gate and its diffusion terminals, in attofarads per lambda. Finally, *gccap* is the capacitance between the gate and the channel, in attofarads per square lambda. *Currently, gscap and gccap are unused by the extractor.*

Often the units in the extracted circuit for a cell will always be multiples of certain basic units larger than centimicrons for distance, attofarads for capacitance, or milliohms for resistance. To allow larger units to be used in the **.ext** file for this technology, thereby reducing the file's size, the **extract** section may specify a scale for any of the three units, as follows:

cscale *c*
lambda *l*
rscale *r*

In the above, *c* is the number of attofarads per unit capacitance appearing in the **.ext** files, *l* is the number of centimicrons per unit length, and *r* is the number of milliohms per unit resistance. All three must be integers; *r* should divide evenly all the resistance-per-square values specified as part of **resist** lines, and *c* should divide evenly all the capacitance-per-unit values.

Magic's extractor breaks up large cells into chunks for hierarchical extraction, to avoid having to process too much of a cell all at once and possibly run out of memory. The size of these chunks is determined by the **step** keyword:

step *step*

This specifies that chunks of *step* units by *step* units will be processed during hierarchical extraction. The default is **100** units. Be careful about changing *step*; if it is too small then the overhead of hierarchical processing will increase, and if it is too large then more area will be processed during hierarchical extraction than necessary. It should rarely be necessary to change *step* unless the minimum feature size changes dramatically; if so, a value of about 50 times the minimum feature size appears to work fairly well.

```
wiring
contact pmc 4 metal 0 poly 0
contact dmc 4 metal 0 diff 0
contact bc 2 poly 0 diff 0
end
```

Table 14. **Wiring** section

12. Wiring section

The **wiring** section provides information used by the **:wire switch** command to generate contacts. See Table 14 for the **wiring** section from the nMOS technology file. Each line in the section has the syntax

contact *type minSize layer1 surround1 layer2 surround2*

Type is the name of a contact layer, and *layer1* and *layer2* are the two wiring

layers that it connects. *MinSize* is the minimum size of contacts of this type. If *Surround1* is non-zero, then additional material of type *layer1* will be painted for *surround1* units around contacts of *type*. If *surround2* is non-zero, it indicates an overlap distance for *layer2*.

During **:wire switch** commands, Magic scans the wiring information to find a contact whose *layer1* and *layer2* correspond to the previous and desired new wiring materials (or vice versa). If a match is found, a contact is generated according to *type*, *minSize*, *surround1*, and *surround2*.

13. Router section

The **router** section of a technology file provides information used to guide the automatic routing tools. The section contains four lines. See Table 13 for an example **router** section.

router						
layer1	metal	3	metal,pmc/metal,dmc/metal,glass	3		
layer2	poly	2	poly,efet,dfet,dcap,pmc,bc	2	diff,dmc	1
contacts	pmc	4				
gridspacing	7					
end						

Table 15. **Router** section

The first two lines have the keywords **layer1** and **layer2** and the following format:

layer1 *wireType wireWidth type-list distance type-list distance ...*

They define the two layers used for routing. After the **layer1** or **layer2** keyword are two fields giving the name of the material to be used for routing that layer and the width to use for its wires. The remaining fields are used by Magic to avoid routing over existing material in the channels. Each pair of fields contains a list of types and a distance. The distance indicates how far away the given types must be from routing on that layer. Layer1 and layer2 are not symmetrical: wherever possible, Magic will try to route on layer1 in preference to layer2. Thus, in a single-metal process, metal should always be used for layer1.

The third line provides information about contacts. It has the format

contacts *contactType size* [*surround1 surround2*]

The tile type *contactType* will be used to make contacts between layer1 and layer2. Contacts will be *size* units square. In order to avoid placing contacts too close to hand-routed material, Magic assumes that both the layer1 and layer2 rules will apply to contacts. If *surround1* and *surround2* are present, they specify overlap distances around contacts for layer1 and layer2: additional layer1 material will be painted for *surround1* units around each contact, and additional

layer2 material will be painted for *surround2* units around contacts.

The last line of the **routing** section indicates the size of the grid on which to route. It has the format

gridspacing *distance*

The *distance* must be chosen large enough that contacts and/or wires on adjacent grid lines will not generate any design rule violations.

plowing	
fixed	efet,dfet,dcap,bc,glass
covered	efet,dfet,dcap,bc
drag	efet,dfet,dcap,bc
end	

Table 16. **Plowing** section

14. Plowing section

The **plowing** section of a technology file identifies those types of tiles whose sizes and shapes should not be changed as a result of plowing. Typically, these types will be transistors and buried contacts. The section currently contains three kinds of lines:

fixed *types*
covered *types*
drag *types*

where *types* is a type-list. Table 16 gives this section for the nMOS technology file.

In a **fixed** line, each of *types* is considered to be fixed-size; regions consisting of tiles of these types are not deformed by plowing. Contact types are always considered to be fixed-size, so need not be included in *types*.

In a **covered** line, each of *types* will remain "covered" by plowing. If a face of a covered type is covered with a given type before plowing, it will remain so afterwards. For example, if a face of a transistor is covered by diffusion, the diffusion won't be allowed to slide along the transistor and expose the channel to empty space. Usually, you should make all fixed-width types covered as well, except for contacts.

In a **drag** line, whenever material of a type in *types* moves, it will drag with it any minimum-width material on its trailing side. This can be used, for example, to insure that when a transistor moves, the poly-overlap forming its gate gets dragged along in its entirety, instead of becoming elongated.

```
plot
type    gremlin
        poly,efet,dfet,bc,pmc/active        18
        diff,efet,dfet,bc,dmc/active        22
        metal,dmc/metal,pmc/metal           11
        pmc/metal,dmc/metal,bc
type    versatec
        poly,efet,dfet,bc,pmc/active        0808 0404 0202 0101 \
                                            8080 4040 2020 1010 \
                                            0808 0404 0202 0101 \
                                            8080 4040 2020 1010
        diff,efet,dfet,bc,dmc/active        0000 4242 6666 0000 \
                                            0000 2424 6666 0000 \
                                            0000 4242 6666 0000 \
                                            0000 2424 6666 0000
        metal,dmc/metal,pmc/metal           8080 0000 0000 0000 \
                                            0808 0000 0000 0000 \
                                            8080 0000 0000 0000 \
                                            0808 0000 0000 0000
        pmc/metal,dmc/metal,bc              X
end
```

Table 16. Sample **plot** section

15. Plot section

The **plot** section of the technology file contains information used by Magic to generate hardcopy plots of layouts. Plots can be generated in different styles, which correspond to different printing mechanisms. For each style of printing, there is a separate subsection within the **plot** section. Each subsection is preceded by a line of the form

style *styleName*

Right now, only **gremlin** and **versatec** styles are supported.

Within the **gremlin** subsection, lines must have one of three forms:

type-list stippleNumber
type-list **X**
type-list **B**

The first form of line associates a Gremlin stipple number with all Magic layers in *type-list*. When Gremlin files are generated, all areas covered by *type-list* will appear as stippled areas filled with stipple *stippleNumber* and bordered with thin solid lines. The second form is designed for contacts. It causes each tile in *type-list* to be outlined with a medium-thickness line with an additional medium-thickness "X" drawn between opposite corners. The **B** specification is identical to

X except that only the border is drawn, without the diagonal "X".

Within the **versatec** subsection, lines may also be in either of three forms:

type-list pat0 pat1 ... pat15
type-list **X**
type-list **B**

In the first case, the material of types *type-list* is rendered with a stipple pattern given by 16 hexadecimal numbers. Each number contains four hex digits; the result is a 16-by-16 bit pattern of 1's and 0's. A one means that the corresponding bit of the output file is set and a zero means that the bit is not modified when this layer is rendered (thus the patterns from different *type-lists* will OR together). *Pat0* specifies the top line of the stipple pattern; within each pattern, the most significant bit corresponds to the leftmost bit within the line of the stipple pattern. Stippled areas area also bordered by thin solid lines. The second and third forms (**X** and **B**) are similar to the second and third forms for **gremlin** lines: Magic outlines tiles in *type-list* with medium-thickness lines and also draws crosses through the tiles if **X** is given. Currently, the **X** option isn't complete, and comes out the same as **B**. With a little luck, the X'es will be completed in time for the 1986 VLSI Tools release.

For **versatec** plotting there are a number of parameters that can be set directly by users, such as the printer width. These parameters allow users to reconfigure the system for different kinds of plotters and different spooling mechanisms. See the manual page for details. If your normal printer is different from the one we use at Berkeley, you may want to modify your system **.magic** file to set up default parameters for your printer.

16. Installing a Technology File

As mentioned earlier, "raw" technology files cannot be read directly by Magic. The C preprocessor must first be used to eliminate comments and expand macros in a technology file before it gets installed. As a consequence, the full power of the C preprocessor is available to the writer of a technology file. Not only may macro definitions be made with **#define**, but "conditional compilation" using **#ifdef** and the ability to use other files via the **#include** mechanism are possible.

Technology files are installed as a file of the name *techname*.**tech***n*. The numeric version suffix *n* (currently **20**) is added to the final **.tech** when the file is installed, and allows multiple versions of the technology file to coexist in the same directory. There is a shell script, **tech/:techinstall**, to do all the necessary processing to install a new technology file.

Technology files can be installed in any directory. When Magic is run, it searches for a technology file first in the current directory and next in the system library directory, ~**cad/lib/magic/sys**. To install a new technology file whose source is *techname*.**tech**, run:

tech/:techinstall *techname*.**tech** *vers dir*

where *dir* is the directory in which the technology file is to be installed, and *vers* is the proper version suffix to insure that this technology file is readable by the latest version of Magic. See the Makefile in **tech** for the string **VERSION**, which defines the current version number.

Magic Maintainer's Manual #3:
Display Styles, Color Maps, and Glyphs

Robert N. Mayo
John Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

This tutorial corresponds to Magic version 4.

Tutorials to read first:

All of them.

Commands covered in this tutorial:

none

Macros covered in this tutorial:

none

1. Introduction

This document gives overall information about the files that tell Magic how to display information on the screen. There are three types of files that contain display information: display styles files, color-map files, and glyph files.

2. Display Styles

Display styles files describe how to draw rectangular areas and text. A single file contains a large number of display styles. Each display style contains two kinds of information: a) how to modify pixels (which bits of the pixel should be changed and what their new value(s) should be); and b) which pixels to modify. Part b) consists of things like "fill the entire area," or "modify only those pixels in the area that are given by a particular stipple pattern," or "draw a dashed-line around the area's outline." In the case of text, "which pixels to modify" is

determined by the font for the text, which is not part of the display style, so the display style information for this is ignored. See the manual page **dstyle(5)** for details on the format of display styles files.

Display styles are designed to take into account both the characteristics of certain technologies and the characteristics of certain displays. For example, a bipolar process may require information to be displayed very differently than a MOS process, and a black-and-white display will be used much differently than a color display. Thus there can be many different display styles files, each corresponding to a particular class of technologies and a class of displays. The names of styles files reflect these classes: each display styles file has a name of the form *x.y*.**dstyle5**, where *x* is the technology class (given by the **styletype** line in the **styles** section of the technology file), and *y* is the class of display. Each display driver knows its display class; the driver initialization routine sets an internal Magic variable with the display class to use. Right now we have two display styles files: **mos.7bit.dstyle5** and **mos.bw.dstyle5**. Both files contain enough different styles to handle a variety of MOS processes, including both nMOS and CMOS (hence the **mos** field). **Mos.7bit.dstyle5** is designed for color displays with at least seven bits of color per pixel, while **mos.bw.dstyle5** is for black-and-white displays (stipple patterns are used instead of colors).

3. Color Maps

The display styles file tells how to modify pixels, but this doesn't completely specify the color that will be displayed on the screen (unless the screen is black-and-white). For color displays, the pixel values are used to index into a *color map*, which contains the red, green, and blue intensity values to use for each pixel value. The values for color maps are stored in color-map files and can be edited using the color-map-editing window in Magic. See **cmap(5)** for details on the format of color-map files.

Each display styles file uses a separate color map. Unfortunately, some monitors have slightly different phosphors than others; this will result in different colors if the same intensity values are used for them. To compensate for monitor differences, Magic supports multiple color maps for each display style, depending on the monitor being used. The monitor type can be specified with the **-m** command line switch to Magic, with **std** as the default. Color-map files have names of the form *x.y.z*.**cmap1**, where *x* and *y* have the same meaning as for display styles and *z* is the monitor type. Over the last few years monitor phosphors appear to have standardized quite a bit, so almost all monitors now work well with the **std** monitor type. The color map **mos.7bit.std.cmap1** is the standard one used at Berkeley.

4. Transparent and Opaque Layers

One of the key decisions in defining a set of display styles for a color display is how to use the bits of a pixel (this section doesn't apply to black-and-white displays). One option is to use a separate bit of each pixel (called a *bit plane*) for each mask layer. The advantage of this is that each possible combination of layer overlaps results in a different pixel value, and hence a different color (if you wish).

Thus, for example, if metal and poly are represented with different bit planes, poly-without-metal, metal-without-poly, poly-and-metal, and neither-poly-nor-metal will each cause a different value to be stored in the pixel. A different color can be used to display each of these combinations. Typically, the colors are chosen to present an illusion of transparency: the poly-and-metal color is chosen to make it appear as if metal were a transparent colored foil placed on top of poly. You can see this effect if you paint polysilicon, metal1, and metal2 on top of each other in our standard technologies.

The problem with transparent layers is that they require many bits per pixel. Most color displays don't have enough planes to use a different one for each mask layer. Another option is to use a group of planes together. For example, three bits of a pixel can be used to store seven mask layers plus background, with each mask layer corresponding to one of the combinations of the three bits. The problem with this scheme is that there is no way to represent overlaps: where there is an overlap, one of the layers must be displayed at the expense of the others. We call this scheme an *opaque* one since when it is used it appears as if each layer is an opaque foil, with the foils lying on top of each other in some priority order. This makes it harder to see what's going on when there are several mask layers in an area.

The display styles files we've designed for Magic use a combination of these techniques to get as much transparency as possible. For example, our **mos.7bit.dstyle5** file uses three bits of the pixel in an opaque scheme to represent polysilicon, diffusion, and various combinations of them such as transistors. Two additional bits are used, one each, for the two metal layers, so they are transparent with respect to each other and the poly-diff combinations. Thus, although only one poly-diff combination can appear at each point, it's possible to see the overlaps between each of these combinations and each combination of metal1 and metal2. Furthermore, all of these styles are overriden if the sixth bit of the pixel is set. In this case the low order five bits no longer correspond to mask layers; they are used for opaque layers for things like labels and cell bounding boxes, and override any mask information. Thus, for example, when metal1 is displayed it only affects one bit plane, but when labels are displayed, the entire low-order six bits of the pixel are modified. It's important that the opaque layers like labels are drawn after the transparent things that they blot out; this is guaranteed by giving them higher style numbers in the display styles files.

Finally, the seventh bit of the pixel is used for highlights like the box and the selection. All 64 entries in the color map corresponding to pixel values with this bit set contain the same value, namely pure white. This makes the highlights appear opaque with respect to everything else. However, since they have their own bit plane which is completely independent of anything else, they can be drawn and erased without having to redraw any of the mask information underneath. This is why the box can be moved relatively quickly. On the other hand, if Magic erases a label it must redraw all the mask information in the area because the label shared pixel bits with the mask information.

Thus, the scheme we've been using for Magic is a hierarchical combination of transparent and opaque layers. This scheme is defined almost entirely by the styles file, so you can try other schemes if you wish. However, you're likely to have problems if you try anything too radically different; we haven't tried any schemes but the one currently being used so there are probably some code dependencies on it.

For more information on transparent and opaque layers, see the paper "The User Interface and Implementation of an IC Layout Editor," which appeared in *IEEE Transactions on CAD* in July 1984.

5. Glyphs

Glyphs are small rectangular bit patterns that are used in two places in Magic. The primary use for glyphs is for programmable cursors, such as the shapes that show you which corner of the box you're moving and the various tools described in Tutorial #3. Each programmable cursor is stored as a glyph describing the pattern to be displayed in the cursor. The second use of glyphs is by the window package: the little arrow icons appearing at the ends of scroll bars are stored as glyphs, as is the zoom box in the lower-left corner of the window. We may eventually use glyphs in a menu interface (but don't hold your breath).

Glyphs are stored in ASCII glyph files, each of which can hold one or more glyph patterns. Each glyph is represented as a pattern of characters representing the pixels in the glyph. Each character selects a display style from the current display styles file; the display style indicates the color to use for that pixel. See the manual page **glyphs(5)** for details on the syntax of glyphs files.

The window glyphs are stored in two files, **windows7.glyphs** (which is used for 512-line displays where the scroll bars are 7 pixels wide) and **windows11.glyphs** (which is used for 1024-line displays that have 11-bit wide scroll bars). The positions of the various glyphs in these files is important, and is defined in the **window** module of Magic.

Programmable cursors are stored in files named *x*.**glyphs**, where *x* is determined by the device driver for the display. Displays capable of supporting full-color cursors use **color.glyphs**; displays that can only support monochrome cursors used **bw.glyphs**. The order of the various glyphs in these files is important. It is defined by the files **styles.h** in the **misc** module of Magic.

Magic Technology Manual #1: NMOS

John Ousterhout

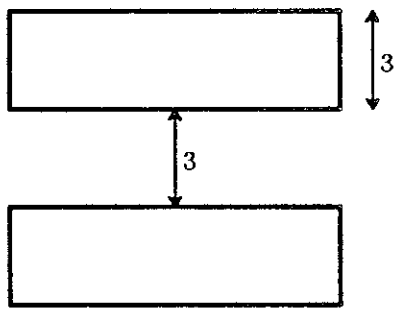
Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

1. Introduction

This document describes Magic's NMOS technology. It includes information about the layers, design rules, routing, CIF generation, and extraction. This technology is the default one in Magic, and is also available by the name **nmos** (run Magic with the shell command **magic -T nmos**). The design rules described here are for the standard Mead and Conway NMOS process with butting contacts omitted and buried contacts added. There is a single layer each of metal and polysilicon. If you've been reading the Mead and Conway text, or if you've already done circuit layout with a different editing system, don't forget that these are not the layers that actually end up on masks. Contacts and transistors are drawn in a stylized form that omits implants, vias, and buried windows.

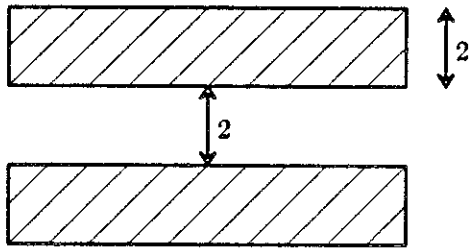
2. Layers and Design Rules

2.1. Metal



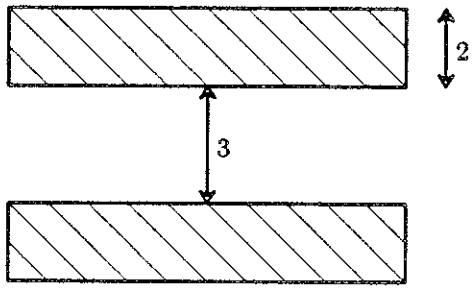
There is only one layer of metal, and it is drawn in blue. Magic accepts the names **metal** or **blue** for this layer. Metal must always be at least 3 units wide and must be separated from other metal by at least 3 units.

2.2. Polysilicon



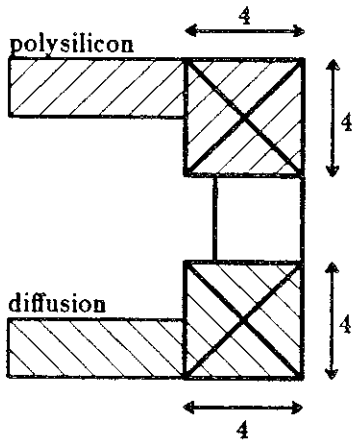
Polysilicon is drawn in red, and can be referred to in Magic as either **polysilicon** or **red**. It has a minimum width of 2 units and a minimum spacing of 2 units.

2.3. Diffusion



Diffusion is drawn in green, and can be referred to in Magic as either **diffusion** or **green**. It has a minimum width of 2 units and a minimum spacing of 3 units.

2.4. Contacts to Metal

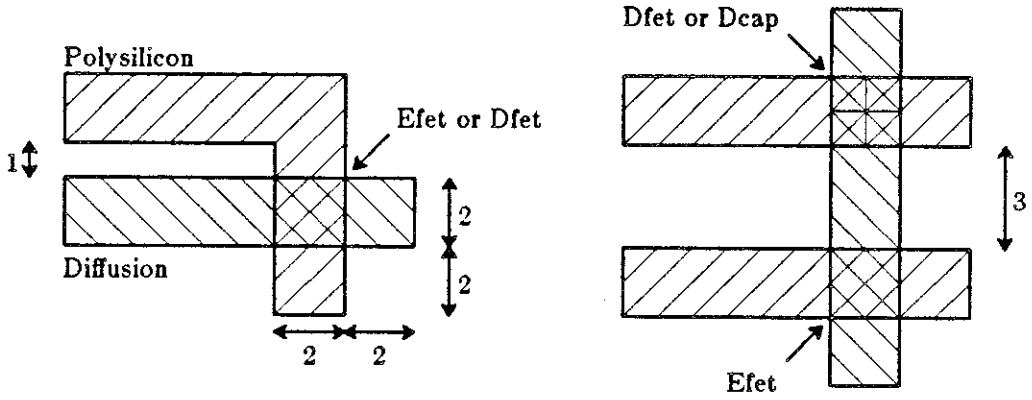


Contacts between metal and polysilicon, and between metal and diffusion, have similar forms. Poly-metal contacts can be referred to as **pmc** or **poly-metal-contact**; they are drawn to look like metal running on top of poly, with an "X" over the area of the contact. Diffusion-metal contacts are similar, except that they look like metal running on top of diffusion, and have names **dmc** and **diff-metal-contact**. Contacts are drawn differently in Magic than they will appear in the CIF: you do *not* draw the via hole. Instead, you draw the outer

area of the metal pad around the contact, which must be at least 4 units on each side. Magic will fill in the appropriate via when CIF is generated. If you draw contacts larger than 4 units on a side, Magic will fill in as many 2-by-2 CIF via holes (with 2-unit spacings) as it can. Contacts areas must be rectangular in shape: contacts of the same type may not abut.

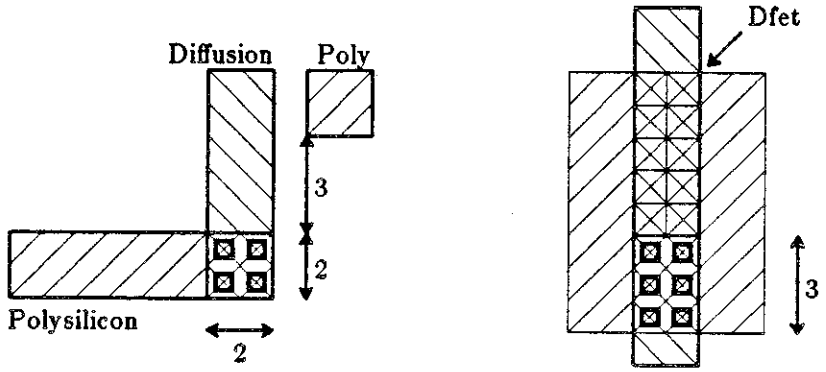
An additional kind of contact, called **glass-contact**, is used to generate holes in the overglass layer for use in bonding to pads. This layer is drawn as gray stripes over blue, and includes both metal and the overglass hole.

2.5. Transistors



There are three transistor structures in the NMOS technology. Enhancement transistors are known by the names **efet** and **enhancement-fet**, and are drawn to look like red over green, with green stripes. You get efet automatically when you paint poly over diffusion or vice versa. Depletion transistors are known by the names **dfet** and **depletion-fet**, and are drawn the same way, except with yellow stripes. A third type of material is called **depletion-capacitor** or **dcap**. It is displayed with yellow crosses over the transistor area, and is identical to dfet except that there are no overhang design rules for it since it is assumed to be used only as a capacitor. You do not drawn any implants in Magic, but just use a different material for the transistor. Magic will generate the implants automatically. All transistors must be at least 2 units on each side, and there must be a poly or diffusion overhang for 2 units on each side of efet or dfet (this is not required for dcap). Poly must be separated from diffusion by at least one unit except where it is forming a transistor. Dfet and dcap must be at least 3 units from efet in order to keep the implant from contaminating the enhancement transistor.

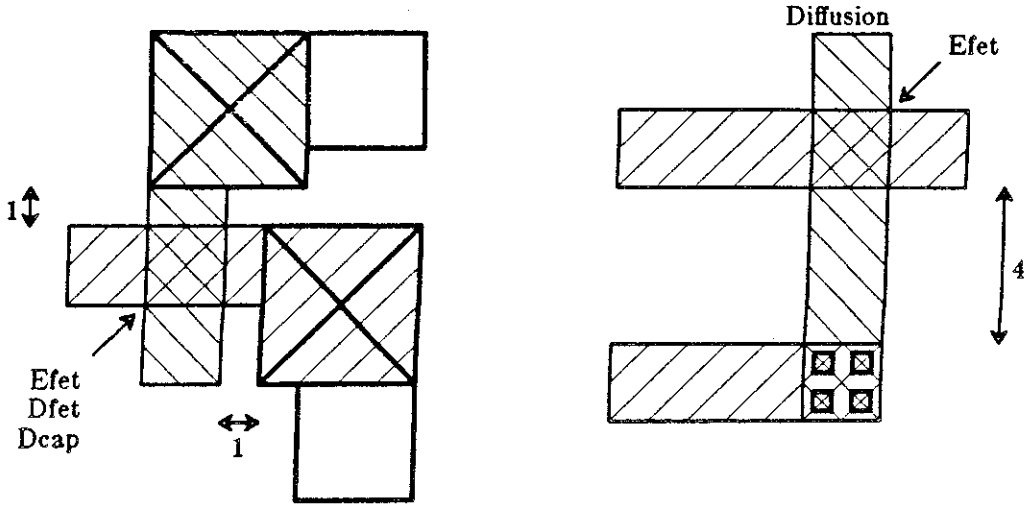
2.6. Buried Contacts



Buried contacts go by the names **bc** and **buried-contact**. They are drawn in a brownish color (the same as transistors), except with solid black squares over their area. As with other contacts, you draw just the area where the two connecting materials (poly and diffusion) overlap; Magic will generate the CIF buried window, which is actually larger than the overlap area. Buried contacts come in two forms. The normal form is 2 units on a side, and no poly or diffusion overhang is required. The second form is used only next to depletion transistors, and is a 3-by-2 structure abutting the depletion transistor. This form is a little controversial, since it results in larger-than-normal variations in the size of the depletion transistor. As a consequence, Magic reports design-rule violationsi wherever buried contacts abut depletion transistors less than 4 units long. In butting bc-dfet structure, you should measure the transistor length from the bc-dfet boundary.

WARNING: there is one additional rule for buried contacts that is NOT enforced by Magic. Where diffusion enters a buried contact, there must be no unrelated polysilicon for 3 units on that side of the buried contact. This rule is necessary because the buried window extends outward from the buried contact by one unit on the diffusion side, and polysilicon must be far enough away to avoid shorting to the diffusion through the buried window. Unfortunately, there is no way to check this rule in Magic without being extremely conservative (the rule would have to require no poly whatsoever on the diffusion side, even if the poly was connected to the buried contact). So, for now, this rule is not checked. Be careful!

2.7. Transistor Spacings



Transistors must be spaced at least 1 unit from any contact to metal, in order to keep the contact from shorting the transistor. In addition, buried contacts must be at least 4 units from enhancement transistors in the diffusion direction. This rule applies only to the side of buried contact where diffusion leaves the contact.

2.8. Hierarchical Constraints

The design-rule checker enforces several constraints on how subcells may overlap. The general rule is that overlaps may be used to connect portions of cells, but the overlaps must not change the structure of the circuit. Thus, for example, it is acceptable for poly in one cell to overlap poly-metal contact in another cell, but it is not acceptable for poly in one cell to overlap diffusion in another (thereby forming a transistor).

For contacts, there are additional restrictions. A contact in one cell may not overlap a contact in any other cell unless the two contacts have same type and they occupy exactly the same area. Partial overlaps are not permitted, nor are abutting contacts of the same type (contacts of different types may abut, as long as the abutment doesn't violate any other design rules). The contact restrictions are necessary to guarantee that CIF can be generated correctly in a hierarchical fashion.

3. Routing

If you use Magic's automatic routing tools on an NMOS design, the routing will be run in metal and polysilicon, with metal as the primary layer. The routing will be placed on a 7-unit grid.

4. Reading and Writing CIF

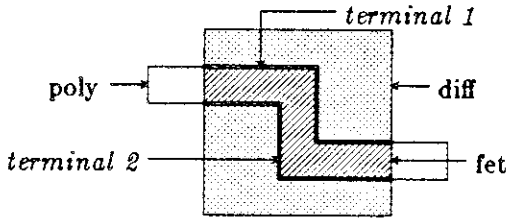
There is only one CIF output style available in the NMOS technology: **lambda=2**. The CIF layers in this style, and their meanings, are:

Name	Meaning
NP	polysilicon
NI	diffusion
NM	metal
NI	depletion implant: generated around depletion transistors and depletion contacts
NC	contact via: generated as small squares inside poly-metal contacts and diffusion-metal contacts
NB	buried window: generated around buried contacts
NG	overglass via: generated for overglass contacts

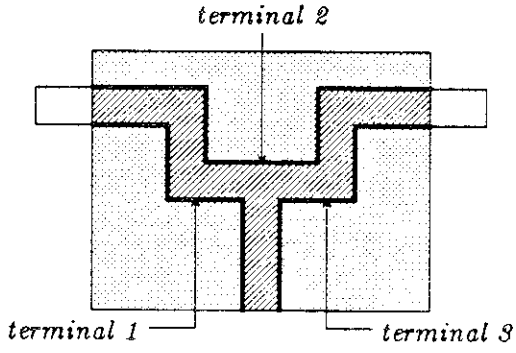
To see exactly where each CIF layer is generated for a particular design, use the **:cif see** command. There is also just one CIF input style. It is called **lambda=2** and can be used to read files written by Magic in the **lambda=2** style, or files written by Caesar using the standard NMOS technology with a scale factor of 200.

5. Extraction

Transistors of type **efet** or **dfet** in the NMOS technology must have at least two diffusion terminals. A diffusion terminal is a contiguous region along the perimeter of the transistor channel that connects to diffusion, as shown below:



2 diff terminals



3 diff terminals

A transistor may have more than two diffusion terminals, in which case it is modelled as a collection of two-terminal transistors. If only one diffusion terminal

is present, the the extractor flags this as an error and outputs a transistor with the source and drain shorted together.

Transistors of the special type **dcap** may have as few as one diffusion terminal. Although their normal use is as capacitors, the extractor will output them as though they were a **dfet**. It is up to simulation programs to compute the capacitance of a **dcap** from the area and perimeter of its channel.

The NMOS technology file currently contains no information on parasitic coupling capacitances. As a result, overlap capacitance, sidewall coupling capacitance, and sidewall overlap capacitance will always be zero.

Magic Technology Manual #2: Scalable CMOS

Shih-Lien Lu

Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90291

John Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

This manual corresponds to Revision 4 of the MOSIS scalable rules.

1. Introduction

This document describes the Magic technology corresponding to the MOSIS scalable CMOS design rules. It includes information about the layers, design rules, routing, CIF generation, and extraction. The Magic name for this technology is **scmos**: if it isn't the default technology at your site, you may have to type the switch **-T scmos** when running Magic from the shell.

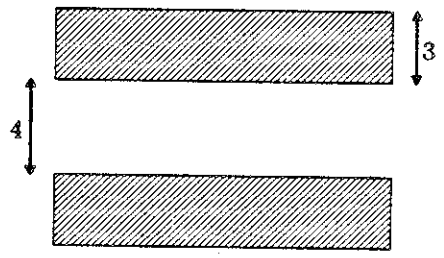
The most important characteristic of the SCMOS technology is that it is flavor-less and scalable: layouts designed using the SCMOS rules may be fabricated in either N-well or P-well technology at a variety of feature sizes. The lambda units used in Magic are dimensionless: MOSIS currently supports fabrication at .7 microns/lambda and 1.5 microns/lambda, and other scale factors will become available in the future. In order for SCMOS designs to be fabricated with either N-well or P-well technology, both p-well and n-well contacts must be placed, and where wells and rings are specified explicitly (e.g. in pads) both flavors must be specified. When the circuit is fabricated, one of the flavors of wells, rings, and substrate contacts will be ignored.

The SCMOS technology provides two levels of metal. All contacts are to first-level metal. There are no buried or stacked contacts.

Remember that the layers you draw in Magic are "abstract layers" or "logs", and do not correspond exactly to the mask patterns be used to fabricate your circuit. In general, the interconnect layers (metal2, metal1, poly, diffusions) will come out as you draw them here, but Magic will generate implants and wells automatically, and these may be bloated or shrunk versions of the various contacts and diffusions. You draw contacts in terms of the total overlap area between the layers being connected, not in terms of the via holes.

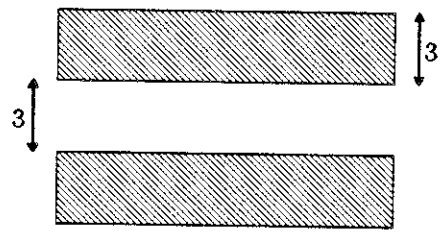
2. Layers and Design Rules

2.1. Second-level Metal



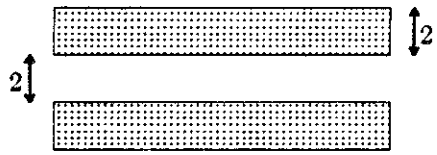
The top level of metal is drawn in a purple color, and has the names **metal2** or **m2** or **purple**. It must always be at least 3 units wide, and metal2 areas must be separated from each other by at least 4 units.

2.2. First-level Metal



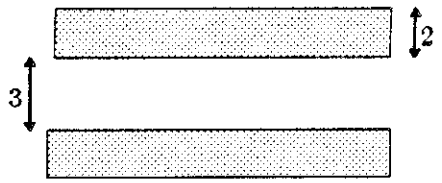
The lower level of metal is drawn in blue and has the names **metal1** or **m1** or **blue**. It has a minimum width of 3 units and a minimum spacing of 3 units.

2.3. Polysilicon



Polysilicon is drawn in red, and can be referred to in Magic as either **polysilicon** or **red** or **poly** or **p**. It has a minimum width of 2 units and a minimum spacing of 2 units.

2.4. Diffusion



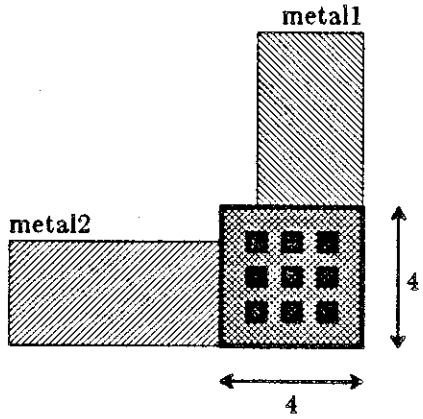
In the SCMOS technology, it is unnecessary for you to specify wells and implant selection layers explicitly. Instead, there are four different layers that correspond to the two kinds of diffusion in the two kinds of wells. Based on these four layers, Magic automatically generates the masks for active, wells, and implant select.

The most common kinds of diffusion are p-diffusion in an n-well and n-diffusion in a p-well; they are used for creating p-type and n-type transistors, respectively. P-diffusion in an n-well is drawn with a light brown color; Magic accepts the names **pdiffusion**, **pdiff**, and **brown** for this layer. N-diffusion in a p-well is drawn in green, and can be referred to as **ndiffusion**, **ndiff**, or **green**.

The other two kinds of diffusion are used for generating well contacts and guard rings; they consist of a strongly implanted diffusion area in a well of the same type. P-diffusion in a p-well is drawn with a light brown color and has stippled holes in it. It goes by the names **psubstratepdiff**, **psd**, **ppdiff**, or **ppd**. N-diffusion in an n-well is drawn in a light green color with stippled holes in it, and goes by the names **nsubstratendiff**, **nsd**, **nndiff**, or **nnd**.

The basic design rules for the four kinds of diffusion are the same: they must be at least 2 units wide and have a spacing (to the same kind of diffusion) of at least 3 units. Spacing rules between diffusions of different types are discussed below.

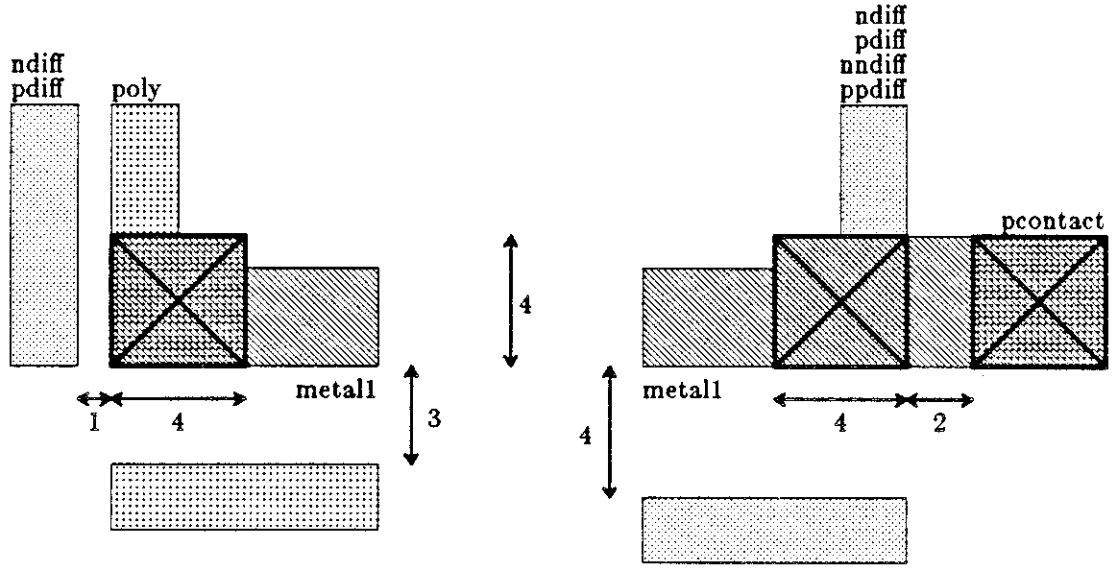
2.5. Metal2 Contacts



All contacts involve the metal1 layer. In Magic, contacts aren't drawn as two areas of overlapping material with a via hole in the middle. Instead, you just draw a single large area of a special contact type, **m2contact** in the case of metal2 contacts. This corresponds to the area where the two wiring layers overlap (metal and metal2 in the case of metal2 contacts). Magic will automatically output metal1, metal2 when it generates CIF or Stream output and will also generate the small via hole in the center of the contact area. For large contact areas Magic will automatically generate many small via holes in CIF or Stream output. All contacts must be rectangular: two contacts of the same type may not abut. Contacts from metal1 to metal2 are called **m2contact** or **m2c** or **via** or **v**. They appear on the screen as an area of metal1 overlapping an area of metal2, with a black waffle pattern over the contact area. Metal2 contacts must be at least 4 units wide.

There is an additional special rule for metal2 contacts: there must not be any polysilicon or diffusion edges underneath the area of the contact or within 1 unit of the contact. This rule is present because it is hard to fabricate a metal2 contact over the sharp rise of a poly or diffusion edge. It is acceptable for poly or diffusion to lie under a m2contact area, as long as it completely covers the area of the m2contact with an additional 1-unit surround.

2.6. Polysilicon and Diffusion Contacts

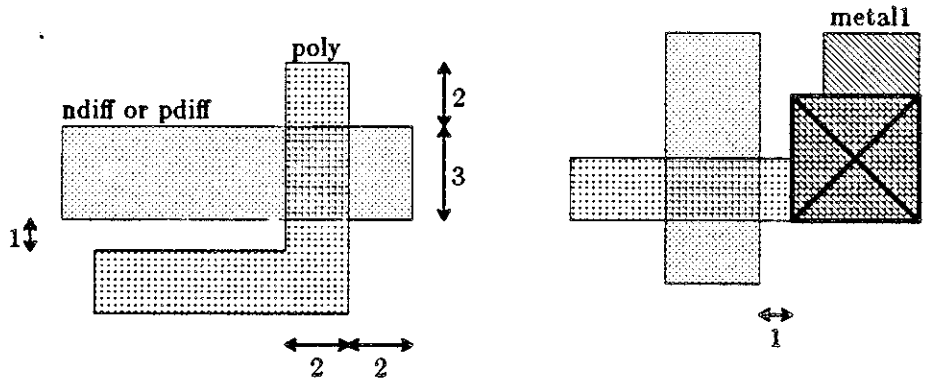


Contacts between metal1 and polysilicon go by the names **polycontact**, **pcontact**, and **pc**. As with all contacts, you only draw the outer boundary of the overlapping area of poly and metal1; Magic fills in the contact cut(s) at mask generation time. Pcontact areas must be at least 4 units wide and must be separated from unrelated polysilicon and pcontact by at least 3 units. This is one unit more than the normal poly-poly separation, and is required because MOSIS bloats the polysilicon around pcontacts.

There are four kinds of contacts between metal1 and diffusion, one for each of the kinds of diffusion. Contacts between metal1 and ndiffusion are called **ndcontact** or **ndc**. Contacts betwen metal1 and pdiffusion are called **pdcontact** or **pdc**. Contacts between metal1 and nsubstratendiff are called **nsubstratencontact** or **nncontact** or **nsc** or **nnc**. Lastly, contacts between metal1 and psubstratepdiff are called **psubstratepcontact** or **ppcontact** or **psc** or **ppc**. All diffusion contacts must be at least 4 units wide, and must be separated from unrelated diffusion by 4 units, one unit more than the normal diffusion-diffusion separation.

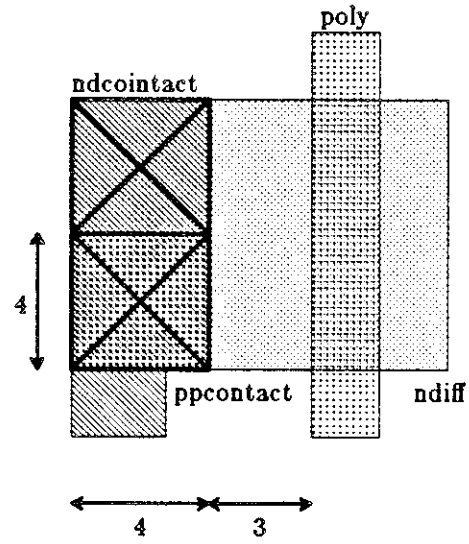
Both poly and diffusion contacts appear on the screen as an overlap between the two constituent layers, with a cross over the contact area. All contacts must be rectangular in shape. Pdc may abut nsc and and ndc may abut psc; all other contact abutments are illegal. Pcontact must be at least 2 units from any diffusion contact, even if the two contacts are electrically connected as on the right of the figure. See Section 2.8 for more rules on substrate contacts.

2.7. Transistors



P-type transistors are drawn as an area of poly overlapping pdiffusion, with brown stripes in the transistor area. Magic accepts the names **pfet** or **ptransistor** for this layer. N-type transistors are drawn as an area of poly overlapping ndiffusion, with green stripes in the transistor area. The names **nfet**, and **ntransistor** may be used. Transistors of each type can be generated by painting polysilicon and diffusion on top of each other, or by painting the transistor layer explicitly. The design rules are the same for both types of transistor: transistors must be at least 2 units long and 3 units wide. They must be surrounded by either poly or diffusion for 2 units on each side, and must be separated from nearby contacts by at least 1 unit. Polysilicon must be at least 1 unit away from diffusion, except where it is forming a transistor.

2.8. Substrate Contacts

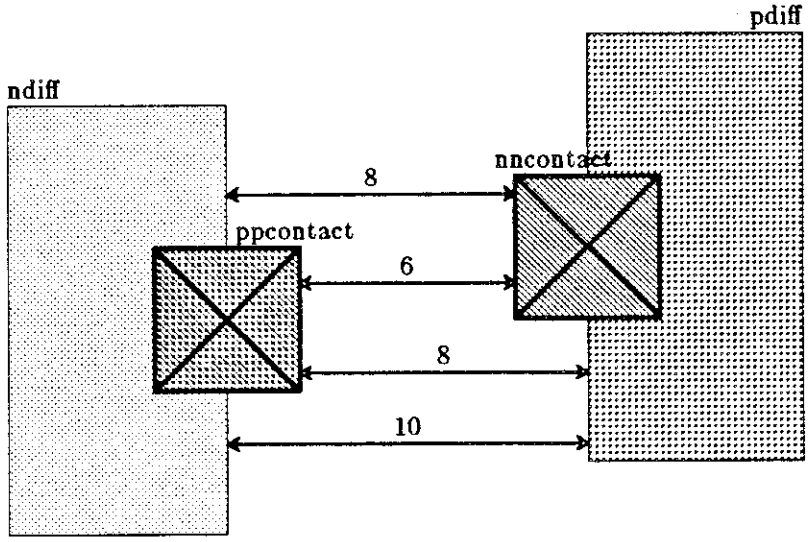


There are several additional rules besides those in Section 2.6 that apply to substrate contacts. Substrate contacts are used to maintain proper substrate voltages and prevent latchup. Nncontact is used to supply a Vdd voltage level to the n-wells (or n-substrate) around p-transistors, and ppcontact is used to supply a GND voltage level to the p-wells (or p-substrate) around n-transistors. Nncontact must be separated from p-transistors by at least 3 units to ensure that the n+ implant doesn't affect the transistor. Nncontact may be placed next to pdcontact

in order to tie a transistor terminal to Vdd at the same time. Because of diode formation, nncontact will not connect to adjoining pdiffusion unless there are contacts to metal1 to strap them together. Similar rules apply to ppcontact.

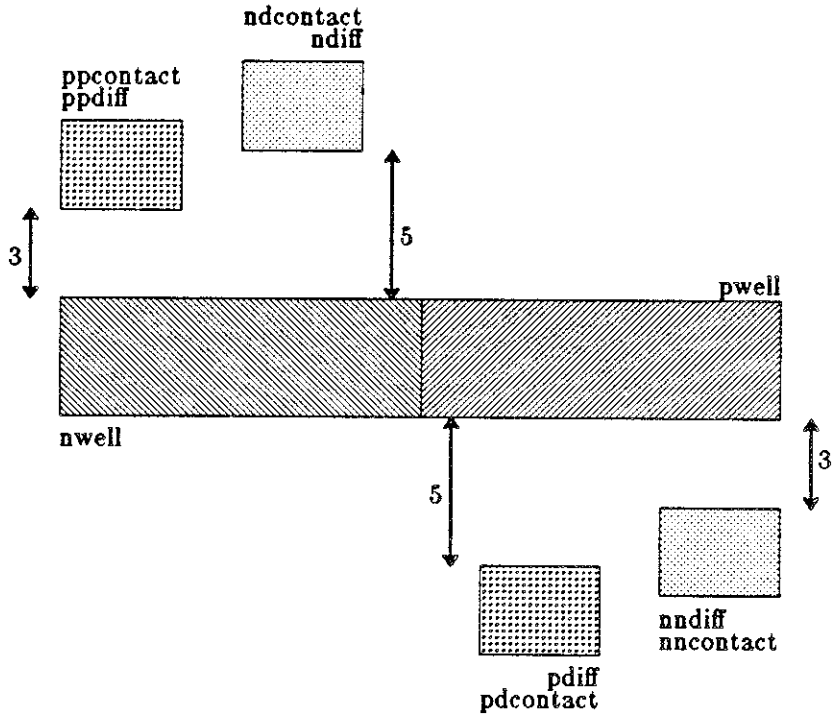
We don't have specific rules yet as to how many substrate contacts you should place, nor do we have any programs to verify that you've placed enough. If you place too few, you risk latch-up in your circuit, so a good rule of thumb is to place one nncontact for each ptransistor that has its source tied to Vdd, and one pwcontact for each ntransistor that has its drain tied to GND.

2.9. Spacings between P and N



Ndiffusion, ntransistor, ndcontact, and ppcontact must be kept far away from pdiffusion, ptransistor, pdcontact, and nncontact, in order to leave room for wells. Ndiffusion and pdiffusion must be 10 units apart. Substrate contacts can be 2 units closer to material of the opposite type (the wells needn't surround them by as many units), so nncontact need only be 8 units from ndiffusion, ppcontact need only be 8 units from pdiffusion, and nncontact and ppcontact need only be 6 units apart.

2.10. Wells and Rings



For the most part, you should not need to draw explicit wells. When writing out CIF or Stream files, Magic generates them automatically. Nwell is generated around pdiffusion, ptransistor, pdcontact, and nncontact. Pwell is generated around ndiffusion, ntransistor, ndcontact, and ppcontact. Magic merges nearby well areas into single large wells when possible. For example, two ndiffusion or pdiffusion areas will share a single well if they are within 16 units of each other. If you're curious about exactly what Magic will use as the well areas, you can use the Magic command **:cif see** along with the CIF layer names described in Section 5. There may be a few cases where you'd like to guarantee that certain areas are covered with wells, e.g. pads. For these cases you may paint the explicit layers **nwell** and **pwell**. Nwell appears on the screen as diagonal green stripes, and pwell appears as diagonal brown stripes. The explicit well layers that you paint will supplement the automatically-generated wells. Pwell must be at least five units from pdiffusion, ptransistor, or pdcontact, and three units from nncontact. Nwell must be at least five units from ndiffusion, ntransistor, or ndcontact, and three units from ppcontact. Nwell and pwell may abut but not overlap (if you paint one well on top of the other, the new one replaces the old one). All nwell and pwell areas that you paint must be at least six units wide and are separated from other wells of the same type by at least nine units.

Guard rings may be created using the nndiffusion and ppdiffusion layers. Nncontacts and ppcontacts should be used to strap the rings to Vdd and GND, respectively. Nndiffusion must be at least 3 units from pwell, 6 units from ppdifffusion or ppcontact, and 8 units from ndiffusion or ndcontact (these are the same rules as for nncontact). Similar rules apply to ppdiffusion. Guard rings are probably the only things that nndiffusion and ppdiffusion will be used for.

2.11. Overglass and Pads

Normally, everything in the layout is covered by overglass in order to protect the circuitry. If you do not wish to have overglass in certain areas of the layout, there are two Magic layers you can use for this. The Magic layer **pad** should be used for drawing pads. It generates a hole in the overglass covering and also automatically includes metal1, metal2, and m2contact. Pad is displayed as metal2 over metal1, with additional diagonal stripes. The rules for pads are in absolute microns, not lambda units: the pad layer must always be at least 100 microns on a side. Since this rule is in absolute units, it is not checked by the Magic design-rule checker. Pads will generally need to be modified in order to fabricate at different scale factors or for different flavors of CMOS.

An additional layer **glass** is provided to allow designers to make unusual glass cuts anywhere on the chip. This layer is drawn in dark diagonal stripes. Probe areas should generally be at least 75 microns wide, but Magic does not check this rule.

2.12. Hierarchical Constraints

The design-rule checker enforces several constraints on how subcells may overlap. The general rule is that overlaps may be used to connect portions of cells, but the overlaps must not change the structure of the circuit. Thus, for example, it is acceptable for poly in one cell to overlap pcontact in another cell, but it is not acceptable for poly in one cell to overlap ndiffusion in another, since that would form an ntransistor.

For contacts, there are additional restrictions. A contact in one cell may not overlap a contact in any other cell unless the two contacts have same type and they occupy exactly the same area. Partial overlaps are not permitted, nor are abutting contacts of the same type (contacts of different types may abut, as long as the abutment doesn't violate any other design rules). The contact restrictions are necessary to guarantee that the CIF via holes can be generated correctly in a hierarchical fashion.

3. Flavor Changes

The painting tables have been set up in the SCMOS technology to make it easy for you to turn n-flavored things into p-flavor, and vice versa. If you paint nwell over an area, any ndiffusion in the area will be turned into pdiffusion, nfet into pfet, ndc into pdc, psd into nsd, and psc into nsc. Similarly, if you paint pwell over an area, any pdiff in the area will be turned into ndiff, pfet into nfet, pdc into ndc, nsd into psd, and nsc into psc. Thus, if you'd like to make a symmetrical copy of something, except for the opposite well, you can copy it, paint the opposite well over it, then erase the well to leave just the basic layers.

4. Routing in CMOS

If you use Magic's automatic routing tools on an SCMOS design, the routing will be run in **metal1** and **metal2**. **Metal1** is the primary routing layer and will be used wherever possible. In order for Magic to route to terminals, they will have to be on layers that connect to either **metal1** or **metal2**. For example, terminals may be on the **pcontact** layer (since it connects to **metal1**) but not on the **polysilicon** layer. In this technology, the router will use an 8-unit grid.

5. Reading and Writing CIF and Calma

The SCMOS technology provides several styles of CIF and Calma output, corresponding to different flavors of CMOS and different scale factors. All of the output styles generate the following CIF layers:

CMS (Calma layer number 51) Corresponds to the metal2 and m2c Magic layers.

CMF (Calma layer number 49) Corresponds to the metal1 Magic layer, plus all contacts.

CPG (Calma layer number 46) Corresponds to the polysilicon and pcontact layers.

CAA (Calma layer number 43) This is the active mask. It is generated over the areas of Magic's four diffusion layers, plus transistors and diffusion contacts.

CVA (Calma layer number 50) This layer is generated as one or more small squares in the center of each m2contact.

CCP (Calma layer number 47) Generated as one or more small squares in the center of each pcontact.

CCA (Calma layer number 48) Generated as one or more small squares in the center of each contact to diffusion.

CWP (Calma layer number 41) P-well. This layer is generated automatically by bloating the ndiff, nfet, ndc, psd, and psc layers, and OR-ing in the pwell layer.

CWN (Calma layer number 42) N-well. This layer is generated by bloating the pdiff, pfet, pdc, nsd, and nsc layers, and OR-ing in the nwell layer.

CSP (Calma layer number 44) P-plus implant mask. This layer is generated by bloating the pdiff, pfet, pdc, psd, and psc layers.

CSN (Calma layer number 45) N-plus implant mask. This layer is generated by bloating the ndiff, nfet, ndc, nsd, and nsc layers.

COG (Calma layer number 52) Overglass holes: generated from the pad and overglass layers.

If you're curious to see exactly where these layers get generated for a particular design, read about the **:cif see** command in the Magic tutorials or man page.

There are currently 9 output styles supported for CIF and Stream. They are:

lambda=1.5(pwell)	Generates CIF for the MOSIS SCP technology, which uses p-wells and 3.0-micron feature sizes.
lambda=1.0(pwell)	Generates CIF for the MOSIS SCP technology, which uses p-wells and 2.0-micron feature sizes.
lambda=0.7(pwell)	Generates CIF for the MOSIS SCP technology, using p-wells and 1.2-micron feature sizes.
lambda=1.5(nwell)	Generates CIF for the MOSIS SCN technology, using n-wells and 3.0-micron feature sizes.
lambda=1.0(nwell)	Generates CIF for the MOSIS SCN technology, using n-wells and 2.0-micron feature sizes.
lambda=0.7(nwell)	Generates CIF for the MOSIS SCN technology, using n-wells and 1.2-micron feature sizes.
lambda=1.5(gen)	Generates CIF for the MOSIS SCE technology, which includes both p-wells and n-wells and has 3.0-micron minimum feature sizes.
lambda=1.0(gen)	Generates CIF for the MOSIS SCE technology, which includes both p-wells and n-wells and has 2.0-micron minimum feature sizes.
lambda=0.7(gen)	Generates CIF for the MOSIS SCE technology, which includes both p-wells and n-wells and has 1.2-micron minimum feature sizes.

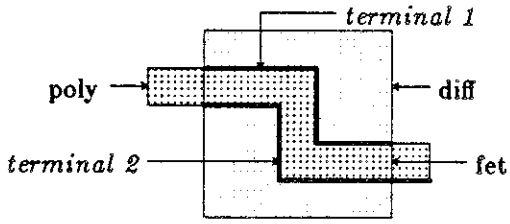
The default style is "lambda=1.0(gen)". Other styles may be selected with the Magic command **:cif ostyle**.

For reading CIF, there are ten styles. Nine of them correspond exactly to the styles listed above for output. The remaining one is "cbpm3u"; it can be used to read in cells designed under the old MOSIS 3-micron rules. Designs converted using style "cbpm3u" will probably have numerous design-rule violations. Whenever CIF is read, explicit wells are generated and left in the Magic files. You'll have to go in by hand and delete them if you don't want them, or modify the technology file locally so it doesn't generate them on read-in.

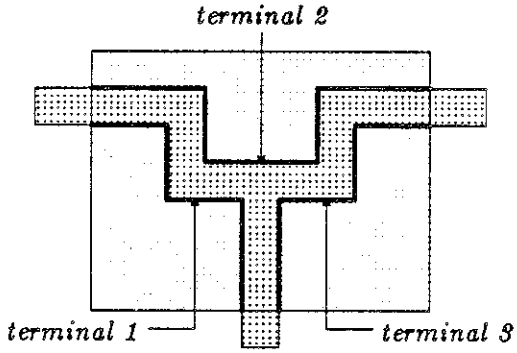
Be careful to select the correct style when reading CIF: if you use the wrong style you're likely to get many errors in the resulting Magic file, with very little warning from the CIF reader.

6. Extraction

The CMOS technology has only two types of transistor: **pfet** and **nfet**. Both must have at least two diffusion terminals. A diffusion terminal is a contiguous region along the perimeter of the transistor channel that connects to diffusion, as shown below:



2 diff terminals



3 diff terminals

A transistor may have more than two diffusion terminals, in which case it is modelled as a collection of two-terminal transistors. If only one diffusion terminal is present, the the extractor flags this as an error and outputs a transistor with the source and drain shorted together.

Using Crystal for Timing Analysis

John Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720
415-642-0865
Arpanet address: ousterhout@berkeley
Uucp address: ucbvax!ousterhout

This user manual corresponds to Crystal version 2.

1. Introduction

Crystal is a program that analyzes the performance of VLSI circuits. Its input consists of a circuit description extracted from the mask layout by the Mextra program. Users also supply a few lines of text to guide the analysis. Crystal then determines how long each clock phase must be and outputs information about the portions of the circuit that cause the worst delays.

Crystal helps in performance tuning by pointing out paths that limit clock speed. It is intended for circuits designed using multiple non-overlapping clocks. It will determine the length of each clock phase, but will not check clock skew or set-up and hold times. Circuits using more complex timing disciplines may require additional timing analysis besides what Crystal provides.

This manual is a tutorial on how to use the Crystal commands to get accurate timing information. It should be used together with the Unix *man* page, which provides detailed syntax information along with more concise descriptions of the commands, options, and built-in tables. Crystal is easiest to understand if you try it out on simple test cases while you read the manual sections.

2. Timing Analysis versus Simulation

Crystal's approach is very different from simulation, so the way you'll use it is quite different from the way you use a simulator. The difference is that Crystal does not consider specific data values. When you use a simulator like SPICE, you invoke a run by giving specific values for all the inputs to the circuit. The simulator then tells you exactly what will happen at each node at each point in time. When using Crystal, the goal is to specify as little as possible about your circuit. You only give Crystal vague information about a few nodes in the circuits (usually the clocks). Wherever Crystal doesn't have specific information from you, it chooses the worst possible alternative. Crystal combines all these worst possibilities to find the overall slowest path through the circuit, which it presents to you.

Simulation results are only as good as the specific choice of test cases: if your test cases don't exercise a particular portion of the circuit, bugs in that portion may go undetected. The advantage of Crystal's value-independent approach is that it is guaranteed to find the worst-case timing behavior of the circuit. Crystal tries all possibilities at each point and picks the worst, so it doesn't depend on designer input to find the critical paths. Furthermore, Crystal does all this in a single run. Simulation requires separate runs for the different test cases, which can be expensive for large circuits.

The disadvantage of Crystal's approach is that it may examine paths that could not occur in the actual chip. For example, Crystal may examine a path whose first portion can occur only when signal A is zero and whose second portion can occur only when A is one. Unless the value of A has been specified explicitly, Crystal will assume that A could be zero in the first portion of the path and one in the second portion. False paths like this result in camouflage that may hide the true critical paths. To eliminate false paths, you restrict Crystal's analysis by fixing a few node values, by restricting the way that signals can flow through transistors, and by giving Crystal specific information about which nodes to watch. Sections 10 and 11 show how to do this. You should try to get by with as little additional information as you possibly can: if you restrict the analysis too much, you may accidentally prevent Crystal from examining the true critical path. The best way to use Crystal is to start out with no additional information, and then add only the bare minimum that's needed to eliminate the false paths.

Crystal is not a replacement for a simulator. Since it ignores most data values, it doesn't give any information about whether your circuit is functionally correct; it will merely tell you how fast it will run. However, by analyzing the timing behavior for you, it allows you to use a fast high-level simulator that ignores timing behavior (ESIM, for example) instead of a slow circuit-level simulator like SPICE. Crystal's models for timing are much simpler than SPICE's. This makes the program run fast, but produces less accurate results in some situations. Section 14 discusses Crystal's models in detail.

3. Signal Flow, Stages, and Delay Analysis

Crystal analyzes your circuit in terms of *signal flow*. "Signals" means zero or one signals, not current or electrons. Signals flow from sources to targets. Signal sources are the chip inputs and the Vdd and GND supply rails. In addition, nodes of the circuit that are labelled as busses are also considered to be signal sources in some situations (see Section 9). Signal targets are places where information is used: gates of transistors and the chip outputs. A *stage* is a path leading from an signal source through transistor channels and other nodes to a target. If all the transistors in a stage are turned on, then a signal can flow from the source to the target. See Figure 1.

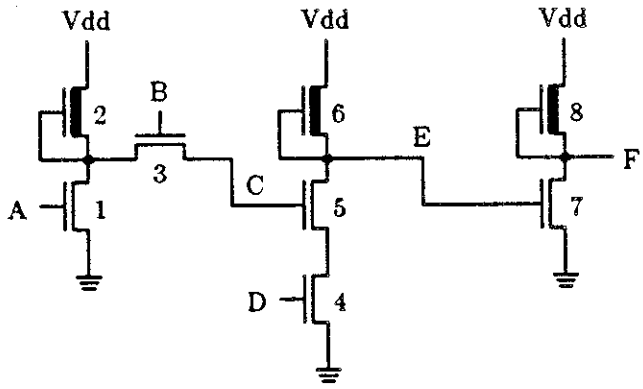


Figure 1. The path from Vdd to C through transistors 2 and 3 is a stage. If you tell Crystal that node A can fall at a certain time, Crystal will infer that node C might rise at a later time, node E might fall at a still later time, and node F rise latest of all.

To start a delay analysis, you give Crystal the time when some signal in your circuit rises or falls (usually this is the input pad for a clock signal). Crystal finds all the signal targets that can be reached from that node. For each target that is a gate, Crystal looks for stages that might be activated by the change in the gate. For each stage that it finds, Crystal computes the time when the stage's target will change value. Then if the target is a gate, Crystal repeats the whole analysis recursively by finding other stages that the gate change might activate. This continues until all possible consequences have been examined.

For example, in the circuit of Figure 1, if you tell Crystal that node A can rise at time 0, Crystal will realize that this change could activate a stage from GND to C through transistors 1 and 3. Whether or not this happens in the actual circuit depends on the value of B; if you haven't specified that value, Crystal will assume that it might be 1, so it will examine the stage. Using the information in the stage, Crystal will compute the delay to C, and use it to update the worst-case fall time for C. Since C connects to the gate of transistor 5, Crystal will then realize that when C falls, it could turn off the pulldown stage from GND through transistors 4 and 5 to E. This could activate the pullup stage from Vdd to E through transistor 6, so Crystal will examine that stage (of course, if node D is 0 then the pulldown stage was already turned off and the change in C has no effect; if you haven't explicitly told Crystal that D is 0, it will assume that it might be 1). Finally, when E rises it could activate the stage from GND through transistor

7 to F, so the worst-case fall time for F will be updated.

If the circuit has many input signals, you invoke the delay analysis again for each of them. Crystal remembers the worst delays seen in any of the analyses. After all the delay analysis has been done, you tell Crystal to print out the worst-case paths through the circuit. A path is a sequence of stages, each causing a change in the next. The worst-case path is the one whose final target reaches its final value later than any other node in the circuit. For example, the path from A to C to E to F is the worst case path in Figure 1. Information about the worst-case paths is recorded by Crystal as part of the delay analysis; you can control how many paths Crystal records.

4. Naming Nodes

Many of the Crystal commands take node names as parameters. A name can either refer to a single node or to a group of nodes. There are two forms for group names. The first form selects nodes whose names form a numerical sequence. The limits of the sequence are delimited by angle brackets (which are not part of the name). Thus, **Bit<1:4>** selects the nodes with names **Bit1**, **Bit2**, **Bit3**, and **Bit4**. To select a node whose name contains an angle bracket, use a backslash character in front of the bracket. For example, type **TrueIfX\<Y** to select the node whose name is **TrueIfX<Y**. To get a backslash in a node name, use two backslashes in a row.

The second form of group name selects all nodes whose names contain a given pattern. The name is specified as a star follwed by the pattern. Thus, ***abc** selects all nodes containing the pattern **abc**. Only simple pattern matching is done. The name * selects all nodes in the circuit.

5. How to Run Crystal

Invoke Crystal with the shell command

crystal *file*

where *file* is the name of a .sim file. If you want to modify Crystal's timing models, then you should not specify *file* on the command line; use the **build** command to read the file in after changing the models. The .sim file should have been created by Mextra. If Mextra was run with the **-o** switch (thereby generating "N" lines in the .sim file), then Crystal will know about parasitic capacitances and resistances associated with wires. If the **-o** switch wasn't specified to Mextra, then there will be "C" lines in the .sim file instead of "N" lines and Crystal will only know about parasitic capacitances. A .sim file shouldn't contain both "N" and "C" lines. Note: Crystal will not work with .sim files generated by Cifplot using its **-x** option.

Crystal reads its commands from standard input and writes its output to standard output. Each input line consists of a command name followed by arguments. The fields are separated by spaces or tabs. Any unique abbreviation for a command name is acceptable. If the first character of a command line is an

exclamation point, then the whole line is treated as a comment and ignored.

Commands are divided into seven groups, which should appear in the following order:

Model commands — These commands modify the timing models that Crystal uses to compute delays, and must appear before the circuit is read in. The model commands are **model**, **parameter**, and **transistor**. See Section 14 for information on how the models work and how to change them.

Circuit commands — Circuit commands are used to input the circuit and provide additional information about it, such as inputs and outputs. The circuit commands are **build**, **bus**, **capacitance**, **inputs**, **outputs**, and **resistance**.

Dynamic node command — This group includes the single command **markdynamic**, used to find and mark the dynamic memory nodes in the circuit. Section 13 describes how to use this command.

Check commands — There are two commands in this group, **check**, and **ratio**. They are used to examine the circuit's structure for suspicious looking electrical features, and may be useful in pointing out places where you need to provide extra information to Crystal. See Section 12.

Setup commands — Setup commands are used to restrict the paths that Crystal can examine in any given delay analysis. This group includes the **flow**, **precharged**, **predischarged**, and **set** commands.

Delay command — This group contains only a single command, **delay**, which performs the actual delay analysis.

Miscellaneous commands — These commands are used to set internal options and print out results and statistics. They can be invoked at any time. Commands in this group are: **alias**, **critical**, **dump**, **fillin**, **help**, **options**, **prcapacitance**, **prfets**, **prresistance**, **source**, **statistics**, and **undump**.

The only command outside these groups is the **clear** command, which resets information that was set by setup and delay commands. After **clear**, input may resume with anything except model commands. **Clear** is used to perform several different timing analyses (for example, for different clock phases) without having to read in the circuit again.

6. Simple Runs on Combinational Circuits

The simplest use of Crystal is for combinational (unclocked) circuits, where you are interested in knowing how long it takes for a change in an input to propagate throughout the circuit. Only four commands need be used: **inputs**, **outputs**, **delay**, and **critical**. First, you must identify to Crystal the circuit inputs (nodes that are driven by the outside world, such as input pads) and the circuit outputs (nodes whose values are used by the outside world). This information is used by Crystal in figuring out how signals can flow. For example,

inputs Bus<31:0> Select
outputs Overflow

identifies the 32 bus bits and the **Select** signal as inputs, and the **Overflow** signal as an output. See Section 9 for more on the **inputs** and **outputs** commands.

Delay commands are used to tell Crystal when input signals change value. For example,

delay BusBit 0 2

indicates that the latest time when **BusBit** will rise is time 0ns and the latest time when **BusBit** will fall is time 2ns. Crystal will then examine the consequences of this change to determine the latest possible rise and fall times for all other nodes affected directly or indirectly by **BusBit**. A negative time in a **delay** statement means that the transition never occurs:

delay Select -1 0

means that **Select** is initially 1, and will become 0 no later than time 0. Thus, only the falling transition of **Select** will be considered in the delay analysis. Many consecutive **delay** statements can be used where there are many inputs that change at different times.

After the **delay** commands, all that is needed is to print out the critical path. The **critical** command can be used for this. It requires no arguments.

7. Simple Runs on Clocked Circuits

Clocked circuits are handled like combinational circuits, except that there is a separate group of **delay** and **critical** commands for each clock phase. Typically, things in the circuit happen in response to the rising edges of clocks, and we'd like to know how long it takes for everything to stabilize once the clock phase has begun. Thus, there is usually a **delay** command of the form

delay Phi1 0 -1

in the group for each clock phase. If no other **delay** commands are given, it is assumed that all other input signals stabilize long before the clock rises.

The command

clear

is used between the commands for the different clock phases; it clears out old

delay information. An alternate way to handle different clock phases is with a completely separate Crystal run. However, for large chips it takes a long time to read in the circuit so it is usually faster to process all clock phases in a single run.

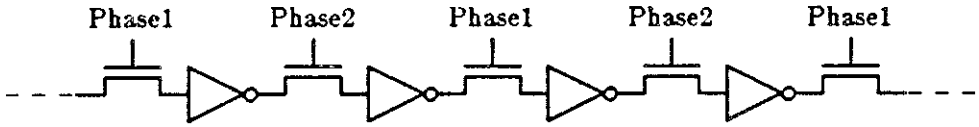


Figure 2. If Crystal doesn't know that Phase2 is zero, then during Phase1 analysis it will consider a path from the left end of the shifter all the way to the right end. If a **set** command is used to tell Crystal that Phase2 is zero, then Crystal won't propagate delays through the pass transistors that are turned off.

In addition to the **clear** commands between clock phases, **set** commands will be needed just before the **delay** commands for each phase. A **set** command indicates that a particular node will always have a particular value during the ensuing delay analysis. For example,

set 0 Phi<2:3>

indicates to Crystal that **Phi2** and **Phi3** will be 0 during the analysis. Crystal uses **set** information to avoid delay paths that cannot occur, as illustrated in Figure 2. The **clear** command will erase information from previous **set** commands; see Section 10 for more details on **set**.

Although the simple set of commands described above will work for many circuits, there are other circuits where it won't work very well. In particular, circuits with networks of transistors used for multiplexors or shifters require additional information that is discussed in Section 11. If only the simple commands are used for these circuits, Crystal will either produce pessimistic results or it will never finish. The sections below describe how to get more information out of Crystal and how to feed additional information into Crystal to produce more accurate results more quickly.

8. More on the Printing Commands

Besides the simple usage of the **critical** command, there are several additional ways that Crystal can print information. All of the printing commands are in the "miscellaneous" command group, so they can be invoked at any time.

8.1. Graphical Command Files

The printing commands will generate graphical command files if you wish. The command files can be used to highlight nodes and transistors using layout editors like Caesar, Magic, and Squid. The default for such files is Caesar format (the **options** command can be used to change the format to Squid or Magic style). The **-g** switch is used to generate the command files. For example,

critical -g dum

will generate in file **dum** a list of Caesar commands that will highlight the critical path. To use a Caesar command file generated in this way, do the following: first,

edit the circuit in Caesar; second, select a view that contains the entire circuit (using the **v** short command if necessary); third, use the **:source** long command to process the command file. The commands will place splotches of the error layer along with labels to identify "interesting points" on the circuit. Boxes are pushed on the box stack so that you can step from one interesting point to another using the **:popbox** long command. The interesting points and labels are different for different Crystal commands. In the **critical** command, for example, the points are the gates of transistors along the worst-case timing path, and the label for each point shows the delay to that point.

8.2. Critical Paths

The **critical** command prints out delay paths through the circuit and has the following form:

critical [**-g** *graphicsFile*] [**-s** *spiceFile*] [*textFile*] *number number* ...

For each *number* given, information about the *number*th slowest path in the circuit is output (Crystal only records a small number of the slowest paths; to change this number use the **options** command). If the **-g** switch is given, graphics output is generated. If the **-s** switch is given, a SPICE deck is generated for the critical path. If *textFile* is given, a textual description of the critical path is written to that file. If none of *graphicsFile*, *spiceFile*, or *textFile* is given, a textual description is output on standard output.

SPICE decks generated by Crystal contain circuit description cards and transient analysis cards, but no model cards; you should add your own model cards to the beginning of the deck. The circuit contains all the transistors and parasitic resistances and capacitances along the path, including gate-source and gate-channel capacitances for transistors that aren't part of the path but connect to it. Node 0 is used for GND, node 1 for Vdd, and node 2 for the substrate body. Crystal generates a card for Vdd, but it doesn't know what the body bias voltage is, so you must add your own card to the deck to generate it.

Crystal actually records three separate lists of slow paths, corresponding to different categories of nodes. The first list is for all nodes. The second list is for paths leading to memory nodes, and the third list is for paths leading to nodes that you have specially requested to be watched, using the **watch** command. Normally the *number*s in the **critical** command refer to the overall list. However, if you end the number with the letter "m", then the the *number*th slowest path to a memory node is printed. For example, "1m" refers to the slowest path to a memory node. Similarly, the suffix "w" is used to refer to the list for watched nodes: "2w" refers to the next-to-slowest path leading to a watched node. The lists for memory and watched nodes are explained in Section 13.

8.3. Capacitance and Resistance Information

Prcapacitance and **prresistance** have similar syntax and are used to print out nodes with large capacitances or resistances:

prcapacitance [**-g** *graphicsFile*] [**-t** *threshold*] *node node ...*
prresistance [**-g** *graphicsFile*] [**-t** *threshold*] *node node ...*

For **prcapacitance** the threshold is in picofarads, and for **prresistance** the threshold is in ohms. The thresholds default to zero. If no nodes are specified, then the entire circuit is searched for nodes whose capacitance or resistance is greater than the threshold. If nodes are specified, then only those nodes are considered. A line of output is generated for each node exceeding the threshold. For example, **prcap abc** will print out the capacitance at node **abc**, and **prres -t 10000** will print out all nodes with lumped resistance greater than 10 kohms. The **-g** switch is used to generate a graphical command file. If Crystal encounters several nodes with exactly the same resistance or capacitance, only the first is printed. At the end of the printout, Crystal lists how many duplicate values were discarded.

8.4. Information about Transistors

The command

prfets *node node ...*

will print out lots of information about each transistor whose gate attaches to one of the *nodes*. If no *node* is given, then information is printed about all transistors.

9. Circuit Commands

Commands in the "circuit" group are used to read in the circuit and give Crystal additional information about it. Information from circuit commands lasts for the entire Crystal run, and isn't affected by **clear** or any other commands.

9.1. Reading in the Circuit

The command

build *file*

is used to read in the circuit. *File* is the name of a file in .sim format. If you type a filename on the command line when you invoke Crystal, then the **build** command is automatically invoked. However, if you wish to modify the circuit models, you must do so before reading in the circuit. In this case, don't give a filename on the command line, but use **build** instead. See Section 14 for information on changing the models.

If a node has been labelled several times, then Mextra picks one of those labels to identify the node. The other names are recorded in an alias file but are not used in the .sim file. If you'd like to use one of the aliases to refer to a node, rather than the name Mextra chose, you can use the command

alias *file*

to read in the ".al" file produced by Mextra and add the aliases to the Crystal's

name table.

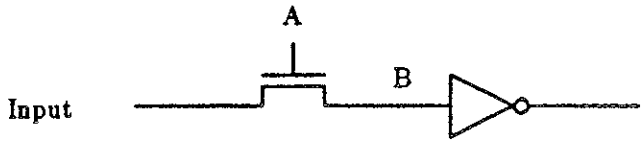


Figure 3. If **Input** isn't marked as an input, Crystal will not realize that it is a source of signals, and will mistakenly assume that a change at A has no effect on B.

9.2. Inputs and Outputs

These two commands were introduced in Section 6. They have the form

inputs *node node ...*
outputs *node node ...*

Crystal uses information about inputs and outputs to determine how signals can flow around the circuit: inputs and Vdd and GND are assumed to be sources of either a logic one or logic zero, and outputs and gates are assumed to be signal targets (places to which signals flow). If you forget to tell Crystal which nodes are inputs and/or outputs, it may miss some signal flows and overlook the critical path (see Figure 3). The **check** command can help to find nodes that should be marked as inputs.

Any input node that is not also an output node is assumed to be driven entirely from off chip. Crystal assumes that nothing on the chip can affect the value of the node, so if the node isn't used in a **delay** command, then Crystal will assume that its value never changes during the timing analysis. However, if a node is marked as both an input and an output, then Crystal will calculate delays to the node from the rest of the circuit. Usually only pads are marked as inputs, but this need not necessarily be the case. Marking a node as in input is roughly equivalent to applying a probe to the circuit at that point.

9.3. Changing Parasitic Values

Two commands are available to override Crystal's computation of parasitic capacitance and resistance:

resistance *ohms node node ...*
capacitance *pfs node node ...*

These commands will replace Crystal's computed value for the parasitic resistance or capacitance of one or more nodes with the specified value. There are at least two situations where this may be useful. For pads, there is relatively little capacitance on-chip, compared to the off-chip capacitance that must be driven. The **capacitance** command can be used to simulate the presence of the off-chip capacitance. The **resistance** command is used primarily to compensate for errors in the way Crystal computes resistances. To compute the internal resistance of a node, Crystal sums all of the internal resistances of all the wires connected to the node. All of the transistor gates attached to the node are assumed to be driven through all of the resistance. If a node has no branches this will give an accurate

result, but if the node has many branches then Crystal will substantially overestimate the resistance (this happens commonly for clock lines). The **resistance** command should be used to correct such situations. Since Crystal's resistance calculation is conservative, I suggest that you not use the **resistance** command until you discover that a bad resistance value is causing Crystal to overestimate the critical path.

9.4. Bus

There are a few occasions where, without guidance from the user, Crystal will chase around the circuit almost endlessly during a **delay** command without getting anywhere. This section describes once such scenario, and Section 11 describes another one that is even more serious.

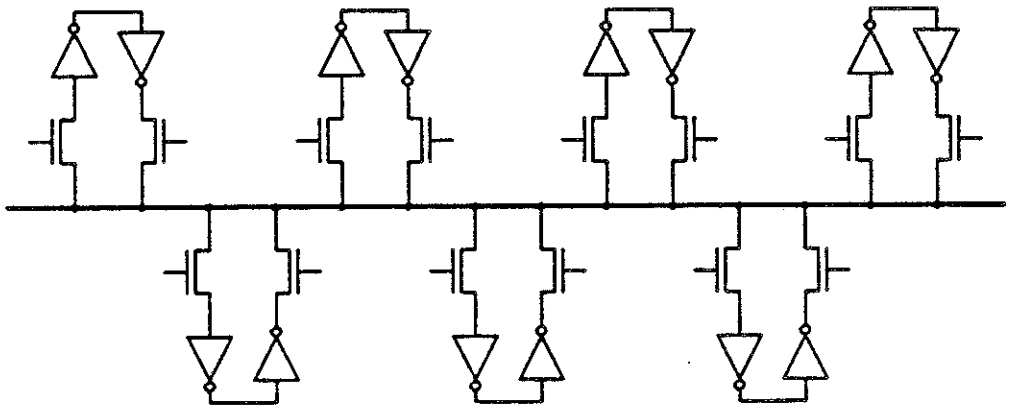


Figure 4. Without any additional information, Crystal will make a separate examination of every path from an output in one cell to an input in another cell. If Crystal knows about the presence of the bus, it first examines all paths from outputs to the bus, then examines paths from the bus to inputs. This makes the analysis much faster.

One situation where Crystal works too hard is the case of a bus with many elements attached to it. Figure 4 shows such a situation. During delay analysis, Crystal will check separately each path from the output of each bus element to the input of each other bus element, resulting in total work proportional to the square of the number of elements on the bus. If Crystal is told that the connecting node is a bus, then it breaks up the paths into separate stages from the elements onto the bus and from the bus to the inputs of the elements. For N elements on the bus, this results in 2N stages to examine instead of N^2. The **bus** command has the following syntax:

bus *node node ...*

Nodes marked as busses are treated both as signal sources and as signal targets.

It is only safe to mark a node as a bus if its capacitance is much greater than the internal capacitances of its elements. If this is not the case, then delays through the supposed bus will be underestimated. Crystal automatically marks all nodes with more than 2 pf of capacitance as busses (the threshold value can be changed with the **options** command; to prevent Crystal from automatically marking busses, use a very high threshold).

10. Setup Commands

Commands in the "setup" group are used to give Crystal additional information to restrict the paths it examines in **delay** commands. The **clear** command will erase any information provided by setup comands.

10.1. Set

The **set** command indicates that a node is fixed in value. Its syntax is

set 0/1 *node node ...*

When you tell Crystal that a node is fixed in value, Crystal performs a simple logic simulation to see if that fixed value causes other nodes to be fixed as well. For example, if an input of a NAND gate is set to 0, Crystal will deduce that the output is fixed at 1. If an input of a NOR gate is fixed at 1, then the output must be 0, and so on. See Section 14 for a description of how Crystal does the logic simulation. When processing delays, Crystal checks transistors to see if their gates are fixed in value. If a transistor's gate is forced to the value that turns the transistor off, then no signals can flow through the transistor.

If a node is forced to a value by a **set** command, then Crystal assumes that its value can never change during the timing analysis; that node will never appear in a critical path. Because of this, you should use **set** sparingly, lest you accidentally mask the critical path. Normally, **set** is used only to turn off all clock phases but one and to disable diagnostic circuitry such as scan-in-scan-out loops.

10.2. Precharging

The commands

precharged *node node ...*
predischarged *node node ...*

indicate to Crystal that the nodes are precharged to 1 or 0, respectively. When a node is precharged, Crystal assumes that it has an initial value of 1 and can only change to 0. Delays that would pull the node to 1 are ignored. When a node is predischarged, Crystal assumes that it has an initial value of 0 and can only change to 1. Delays that would pull the node to 0 are ignored. Precharged nodes are assumed to be highly capacitive, so they are treated like busses.

11. Pass Transistor Flow

As mentioned in Section 9.4, there are a few situations where Crystal can end up doing more work than necessary. The most severe examples of this concern pass transistors. Because Crystal does not generally have information about specific data values, it may examine impossible paths through pass transistors. Figures 5 and 6 show two cases. In Figure 5, Crystal will produce a pessimistic delay to Output2 by examining a path that passes forward and backward through

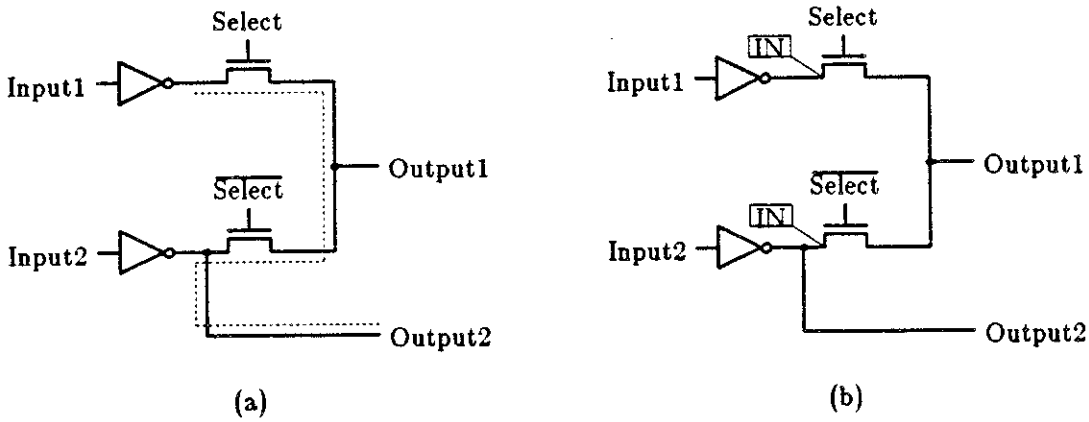


Figure 5. If Crystal doesn't know about pass transistor flow, it will consider the impossible path shown in (a). If the pass transistor flow is labelled with attributes, as in (b), then Crystal will consider paths from **Input1** to **Output1** and from **Input2** to both outputs, but it will not consider the path from **Input1** to **Output2**.

the multiplexor. In Figure 6, there is an enormous number of contorted paths through the shifter array. Crystal will attempt to examine every distinct path, even though the values on the control lines will prevent most of the paths from occurring in the actual circuit.

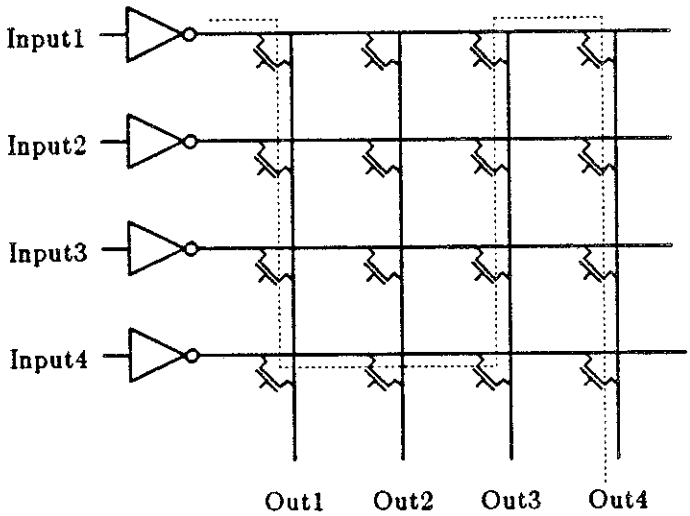


Figure 6. Crystal will consider long snake-like paths through this barrel shifter structure unless pass transistor flow information is provided.

To keep Crystal from chasing impossible paths, you must give it additional information about which way signals flow through pass transistors. Flow is indicated using *transistor attributes* in the CIF files that are input to Mextra. A transistor attribute is a label (CIF "94" construct, or a standard Caesar label) that touches the gate region of a transistor and ends in the character "$". Crystal ignores all attributes unless their first characters are either **Cr:** or **Crystal:**. To indicate the direction of signal flow, attach an attribute to a transistor's source or drain; this is done by placing the label exactly on the line between the gate and the source or drain (attributes placed entirely within the gate region are attached to the gate of the transistor and are used to identify the type of the transistor; see Section 14 for details).

For pass transistors that are unidirectional, two special attributes, **In** and **Out**, may be used. To use the **In** attribute, place a label of the form **Cr:In$** or **Crystal:In$** on the source or drain edge of a transistor gate. This indicates that whenever a 0 or 1 signal passes through the transistor, the source of the 0 or 1 is on the same side of the transistor as the **In** attribute (i.e. the 0 or 1 flows into the transistor from that side). The **Out** attribute indicates just the opposite, namely that 0's and 1's flow out of the transistor at that side.

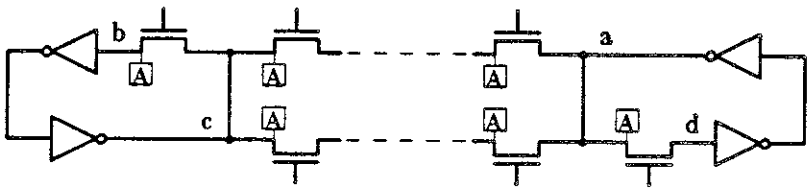


Figure 7. Named attributes can be used to control flow in bidirectional structures. In this case, paths from **a** to **b** and from **c** to **d** will be considered, but the path from **a** to **c** to **d** will not be considered (flow must be unidirectional with respect to tags of a given name).

Bidirectional pass transistors cause special problems. To handle bidirectional structures, one terminal of each pass transistor in the structure should be labelled with an attribute other than **In** or **Out**. See Figure 7. These attributes limit the way that signals may flow through the array: Crystal only allows signals to flow unidirectionally with respect to the attributes. This means that Crystal will consider any path through the array as long as the signal either a) flows into each transistor from the labelled side, or b) flows out of each transistor from the labelled side. A path will be ignored if a signal enters one transistor from the labelled side and leaves another from the labelled side. This allows signals to cross the structure in either direction, but will not allow them to criss-cross back and forth.

If different bidirectional structures are labelled with different attributes, then they are treated independently by Crystal. For example, Crystal will consider a path that enters at one transistor at a side labelled **Cr:A$**, and leaves another transistor at a side labelled **Cr:B$**. However, if the attribute **Cr:A$** is used for both transistors then the path is ignored.

Only a small number of transistors in any design should need to have flow attributes. These transistors can be identified in either of two ways. The easiest way is to use the **check** command, described in Section 12 below, to identify candidates for flow tagging. The hard way is just to run Crystal: if you haven't placed enough tags, then either Crystal will suggest impossible critical paths, or it will abort the delay analysis because it found too many paths. In the first case, it will be easy to identify the transistors that need flow tagging by looking at the critical path. In the second case, you'll have to examine the backtrace information printed after the abort to try to identify the transistors that need tagging (see Section 16).

11.1. Flow

The **flow** command allows you to restrict flow through named attributes, and has the form

flow *direction attribute attribute ...*

Direction must be one of **in**, **out**, **off**, **ignore**, or **normal**. If *direction* is **in** then Crystal treats each of the attributes as if it was an **In** attribute, and if *direction* is **out** then the attributes are treated as if they were **Out**. If **off** is specified then no flow is allowed through any transistors with the given attributes. If **ignore** is specified, Crystal will pretend that the attributes don't exist. If **normal** is given, the flow is reset to do the normal thing. All flow attributes are reset to **normal** by the **clear** command. The **flow** command has no effect on attributes **In** or **Out**.

12. Checking Commands

Two commands are provided by Crystal to perform a static electrical analysis of the circuit. They are only indirectly related to timing analysis, but are useful to find problems such as improper ratios, nodes that aren't marked as inputs, and transistors that should have flow attributes.

12.1. Check

The command

check

makes a series of static electrical checks on the circuit. It prints out information about nodes with no transistors connected to them, nodes that are not driven from anywhere, nodes that don't drive anything, transistors that are permanently forced off, and transistors connecting Vdd and GND directly. Each of these situations is probably an error. The **check** command also identifies transistors that are bidirectional (each side of the transistor has both a signal source and a signal target), but do not have any flow attributes attached. In most cases, bidirectional transistors should have flow attributes to keep Crystal from examining impossible paths.

12.2. Ratio

The **ratio** command has the form

ratio [*limit value*] [*limit value*] ...

and may be used for nMOS circuits to detect improper pullup/pulldown ratios. Normal logic gates are expected to have pullup/pulldown ratios between 3.8 and 4.2, while logic gates driven through pass transistors must have ratios between 7.8 and 8.2. Any ratios outside this range are printed out. If the same erroneous ratio occurs more than 20 times, only the first 20 are printed. The acceptable range may be changed using *limit-value* pairs. *Limit* is one of **normalhi,**

normallow, passhi, or **passlow**.

13. Multi-phase Signals, Memory Nodes, and Watched Nodes

Crystal treats clock phases in a very simple fashion: each clock phase is assumed to be long enough for the circuit to completely settle. The **critical** command indicates how long this takes. Although this approach will produce correct circuits, it is an overly pessimistic view of how clocks are used. In most clocked designs, some signals will settle over more than one clock phase. For example, the input latch for an ALU might be loaded during phase 1, and the output of the ALU might not be used until phase 2. In situations like this, Crystal will normally charge the ALU delay entirely to phase 1, leading to a pessimistic timing estimate.

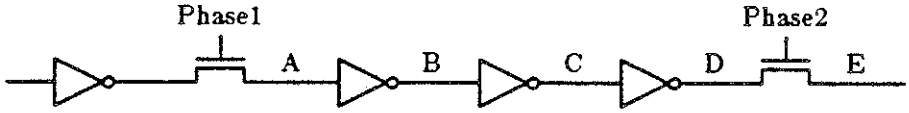


Figure 8. Because node **A** is a memory node, **Phase1** must be long enough for **A** to settle. Nodes **B**, **C**, and **D** need not settle during **Phase1**: they can settle anytime during **Phase1** or **Phase2**. However, if they don't settle during **Phase1**, enough time must be allowed during **Phase2** for them to settle and for the value at **E** to settle also.

For a circuit to function correctly, it isn't really necessary for everything to stabilize during each clock phase. All that matters is that clock phases are long enough for memory cells to be loaded correctly. This means that there can be some tradeoff between the lengths of the various clock phases: see Figure 8 for an example. Ideally, Crystal should deal only with memory nodes: when analyzing clock phase 1, Crystal should compute delays to memory cells loaded in phase 1, memory cells loaded in phase 2, and so on. Then, instead of outputting a single time and critical path, there would be separate times and critical paths for delays between the leading edge of phase 1 and the trailing edges of phase 1, phase 2, and so on. Unfortunately, Crystal doesn't provide this much detail. Instead, it uses memory nodes to provide a first-order approximation to this.

During the **delay** command, Crystal keeps three separate records of worst-case delays: one for all nodes, one just for memory nodes, and one for watched nodes. In the **critical** command, you can use the "m" suffix to print out memory nodes. For example, **critical 1m** will print out the path to the slowest memory node. This simple facility allows you to ignore signals that need not settle during the current clock phase. However, if a signal starts settling in one clock phase and is loaded into a memory cell in the next clock phase, Crystal will not check that the sum of the two phases is enough for this to happen safely. I suggest that you examine critical paths both for memory nodes and for all nodes: check to see that memory nodes will settle before the end of the current clock phase, and that all nodes will settle before the end of the next clock phase.

There are two kinds of memory nodes in a MOS circuit, static and dynamic (see Figure 9). Static memory nodes are those like cross-coupled NAND gates

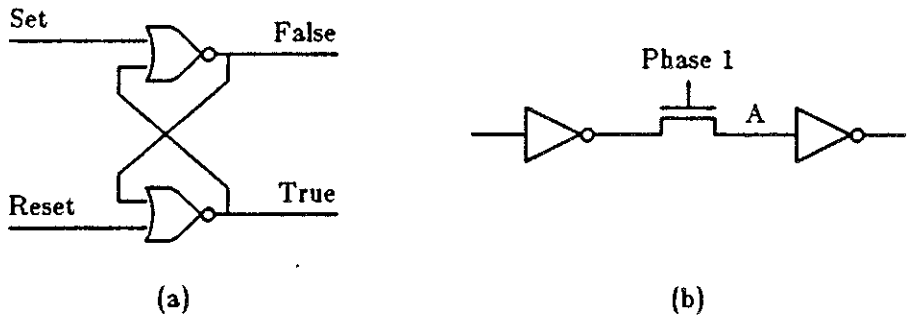


Figure 9. In (a), nodes **False** and **True** are static memory nodes. In (b), **A** is a dynamic memory node.

where there is an ever-present feedback path. Crystal detects such feedback paths during delay analysis and marks the memory nodes. However, Crystal cannot identify dynamic memory nodes without help from the user. At the beginning of analysis, you should use the **markdynamic** command to tell Crystal which nodes are dynamic memory. The command has the form

markdynamic *node value node value ...*

During the **markdynamic** command, Crystal sets each *node* to the given *value* just as if the **set** command had been used. Any nodes that are electrically isolated by these settings (i.e. all transistors connecting to them are forced off) are marked as dynamic memory. Normally, **markdynamic** is used by turning off all of the clock phases.

If Crystal's memory mechanism isn't discriminating enough to pick out all the important paths, there is one more mechanism available as a last resort. You can indicate certain nodes to be handled specially. These nodes are called "watched nodes" because you select them with the command

watch *node node ...*

A special third list of slow memory nodes will be used to record the slowest delays to watched nodes. This allows you to select key nodes and see the delays to those nodes, even if those delays aren't great enough to make the nodes appear on the overall list or the memory list. The danger of the watch mechanism is that it forces you to pick out the key nodes. If you forget a key node then you may end up missing the critical path. I recommend that you work as much as possible with the overall and memory lists, and only use the watch mechanism as a last resort.

14. The Models

Crystal's model of circuit behavior has two parts: one part is used to do logic simulation during the **set** command, and the other part is used to do delay calculations during the **delay** command. Both the simulation and delay models are based on transistor types: there are several types of transistors in the circuit, and each is parameterized by several values. The *man* page lists the predefined transistor types and the fields associated with each type. The subsections below tell how this information is used by Crystal, how to change the predefined

information, and how to define new transistor types.

14.1. Simulation

In order to do logic simulation, each type of transistor is given two integer strength values: **histrength** tells how strongly the transistor can pull to logic 1, and **lostrength** tells how strongly the transistor can pull to logic zero. The strength values are the same for all transistors of a given type, and are independent of the geometry of the transistor. For example, all nMOS enhancement transistors have a **lostrength** greater than the **histrength** of all nMOS depletion pullups.

During the **set** command, the nodes listed in the command are forced to a given value. Then Crystal sees if these settings cause any transistors to be forced on or off. If this happens, nodes on either side of the forced-on or forced-off transistors may be forced to a value. The strength values are used to see if this is the case. For a node to be forced to 1 in this way, two conditions must be met. First, there must be a path from the node to a source of logic level 1, all of whose transistors are forced on. Second, all paths from the node to sources of logic 0 must either contain a forced-off transistor or be weaker than the path to logic 1. The strength of a path is the strength of its weakest transistor.

This simple simulation model is powerful enough to handle a variety of nMOS and CMOS structures. Its weakness is that it doesn't take account of the sizes of transistors, so it may behave incorrectly if improper ratios are used.

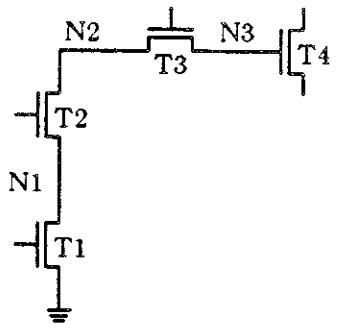


Figure 10. To calculate the delay along this path with transistor T2 as the trigger device, the resistances from T1, N1, T2, N2, T3, and N3 will be summed, and the capacitances from N2, T3, N3, and T4 will be summed. The delay will be the product of the two sums.

14.2. Delay Calculation: the RC Model

Crystal has been designed to include several different delay models and to permit the user to switch between them. At present, there are two delay models, **rc** and **slope**. In the rc model each transistor type is characterized by two resistances, **rup** and **rdown**. The transistor is assumed to have a fixed resistance value **rup** per square whenever it is used to transmit a 1 signal, and **rdown** per square whenever it is used to transmit a 0 signal.

To calculate the delay in a stage, the RC model divides the stage into two portions, separated by the transistor that turned on or off to activate the stage (this transistor is called the trigger for the stage). See Figure 10. All the resistances along the stage are summed, including **rup** or **rdown** for each transistor, plus the resistance of the interconnect. All the capacitances between the trigger and the target are also summed, including the gate-channel capacitance of each transistor along the stage, the parasitic capacitances of the interconnect, and the gate-source or gate-drain capacitances of unrelated transistors that connect to nodes along the stage. Crystal assumes that the trigger is the last transistor in the stage to turn on or off, so that all the charge between the trigger and the signal source has already been drained. The total delay for the stage is computed by multiplying the total capacitance by total resistance.

14.3. Delay Calculation: the Slope Model

The RC model is simple and efficient, but it often produces optimistic delay estimates. It assumes that the effective resistance of a transistor is independent of the waveform on the transistor's gate, and this simply isn't true in reality. If the gate voltage of a transistor rises or falls very slowly, the transistor has a much higher effective resistance than if the gate voltage changes instantaneously. The same transistor may vary in effective resistance by an order of magnitude or more, depending on the exact waveform on its input.

In the RC model, the waveform at a node is characterized solely by the time at which it rises or falls. In the slope model, an additional parameter is added: the rate at which the signal rises or falls. This is called the *edge speed*, and is measured in ns/volt at the instant in time when the signal crosses its logic threshold voltage (the logic threshold voltage is a model parameter and can be changed with the **parameter** command). Although this is only a first-order approximation to the actual waveforms, in Mead-Conway style digital circuits the waveforms tend to have about the same shape except for slope, so this characterization is fairly accurate.

The slope model characterizes the effective resistance of a particular type of transistor in terms of the ratio of two edge speeds: the input edge speed, and the output native edge speed. The output native edge speed is the edge speed that would occur on the output if the input rose or fell infinitely fast (edge speed 0). If the edge speed ratios are small (inputs much faster than output), or if they are uniform across the whole circuit, then the RC model is accurate.

Two tables are used to characterize each transistor type. One table is used when the transistor is pulling up, and the other is used when the transistor is pulling down (these are the **slopeparmsup** and **slopeparmsdown** fields in the transistor models). Each table consist of several triplets. Each triplet contains three values: an edge speed ratio, the transistor's effective resistance per square when that edge speed ratio occurs, and the output edge speed (per pf of capacitance driven and per square of transistor), when that edge speed ratio occurs. The table entries must be in increasing order of edge speed, and the first

entry must have a zero edge speed ratio. If Crystal ever encounters a ratio larger than the largest in the table, it issues a warning message and extrapolates from the largest values. To simplify the task of gathering all this model information, use the *Mkcp* ("make Crystal parameters") program.

Delay calculation in the slope model proceeds in much the same way as for the RC model, except that for the trigger transistor, Crystal interpolates in the tables to find the effective resistance. For transistors other than the trigger, the native resistance is used. In addition, the slope model computes an output edge speed contribution from each component along the path (transistor or node resistance), and sums these to compute the edge speed at the target. The edge speed contributions are computed for each componenet as if that component were driving the capacitance all by itself.

The slope model appears to be fairly accurate. Initial measurements suggest that it is usually within 5% of the times that SPICE predicts for the same circuits, and is rarely worse than 20% off. In contrast, the rc model often produces estimates that are optimistic by 40% or more. The slope model is almost as fast as the rc model, so there is little reason to use the rc model anymore, except for comparison.

14.4. Changing the Models

Crystal provides three commands that you can use to change its internal models. The command

model [*name*]

will set the current delay model to *name*, if it is specified. If *name* is omitted, then the command will print out the valid model names with two stars next to the current model.

The command

transistor [*name* [*field value(s)*] [*field value(s)*] ...]

is used to see and modify the values used to characterize each transistor. If **transistor** is invoked with no arguments, all the transistor types and their current values are printed. If only *name* is supplied with no fields or values, all the transistor type information for *name* is printed. Otherwise, fields for transistor type *name* are changed to the given values. The *man* page lists the predefined transistor types and the field names. If *name* isn't one of the predefined transistor types, then a new transistor type is created with the given field values.

The third command is used to see and set the model parameters that don't have to do with specific transistor types. At present, these parameters are used only for computing the parasitic resistance and capacitance of interconnect. The command has the form

parameter [*name*] [*value*]

If both *name* and *value* are specified, then the selected parameter is set to the given value. If *value* is omitted, then the value of the parameter is printed. If

neither *value* or *name* is given, then the values of all parameters for the current model are printed. See the *man* page for a listing of the parameter names.

14.5. Defining New Transistor Types

The **transistor** command can be used to define new transistor types besides the standard ones. To get Crystal to treat transistors in your circuit as one of the new ones you've defined, use transistor attributes. Normally, Crystal decides the type of each transistor based on its type in the .sim file (enhancement, depletion, p-channel, etc) and how it is used in the circuit. For example, depletion transistors with source or drain connected to Vdd and the other two terminals connected together are given type **nload**. If you want Crystal to use a type of your choosing for a transistor, place an attribute inside the gate area of the transistor. The name of the attribute will be taken by Crystal as the type of that transistor. For example, if you have defined a new transistor type **bootstrap**, then each of these devices should have an attribute **Cr:bootstrap$** on its gate.

15. Miscellaneous Commands

The **help** command prints out a list of the commands and their parameters. For information on the commands that is more detailed than **help**, and more concise than this document, see the *man* page.

The command

source *file*

will cause Crystal to read commands from *file* until its end is reached. Upon end-of-file, Crystal continues reading from the standard input. Source files may be nested.

The **options** command is provided so that you can change internal thresholds and switch settings used by Crystal. For example, one of the options is the threshold capacitance value at which Crystal automatically marks nodes as busses. Normal users shouldn't need to use this command very frequently. See the *man* page for details on its syntax and on the available options.

The **statistics** command prints out a variety of statistics gathered by Crystal as it runs. This information is probably not useful except to system maintainers.

The **quit** command causes Crystal to return to the shell.

16. Deciphering Crystal's messages

Crystal outputs a huge variety of error messages, bug messages and hints. Most of them are in response to syntax errors in the .sim file or errors in commands: these are relatively easy to understand. You should never see a message beginning with the words "Crystal bug:". If you do, report it to me or to your local Crystal wizard. There are several other messages whose meaning is not obvious. They generally indicate that something not-quite-right happened and

are hints that either you are not issuing the right commands or you need to use flow tags or **set** commands to restrict Crystal's analysis. Each of the following subsections describes one such message.

16.1. Aborting: no solution after examining 200000 stages

Crystal has a limit on how many stages it will examine in delay calculations. If the limit is reached, Crystal gives up in despair. When it gives up, it usually means that you need to add more flow control to pass transistors to restrict the set of paths Crystal has to analyze. Occasionally, the built-in limit isn't sufficient for a particular clock phase, even after all the necessary flow control has been added. In this case, use the **options** command to increase the limit.

When the limit is reached, Crystal outputs many messages, the first of which is the "Aborting:" message. Following this will be many messages of two forms: "ChaseVG giving up at xyz", and "ChaseGates giving up at abc". ChaseVG and ChaseGates are the two internal routines that trace out paths through the circuit during delay analysis. The messages indicate the path Crystal was examining when it gave up in despair, in backwards order from the node where it gave up to the node in the **delay** command. Often, the node names in the messages will identify the area where more flow control is needed.

If Crystal aborts a delay calculation, then the information in **critical** and similar commands may not be accurate, since the delay analysis wasn't completed. However, the path provided by **critical** may indicate the place where more flow control is needed. Another way to locate transistors that need flow tagging is to use the **check** command.

16.2. More than 8 transistors in series

During delay analaysis, if Crystal finds a single stage containing more than a certain number of transistors in series, it prints this message. The stage is also ignored (usually such stages cannot occur in practice anyway). A typical place where this might occur is in carry-chain precharging schemes where there are both parallel and serial paths to each node in the chain.

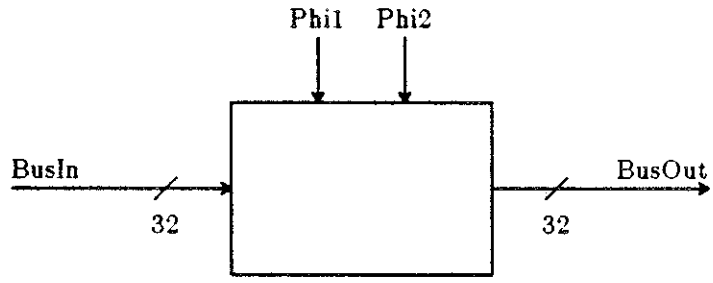


Figure 11. A simple circuit with two non-overlapping clock phases, 32 data inputs, and 32 data outputs.

17. An Example

For the circuit of Figure 11, the following Crystal commands might be used to do timing analysis, assuming that data is read into the circuit only during **Phi1** and that it stabilizes no later than 20ns into the clock cycle. The **BusIn** signals are unidirectional (if they could also be driven from on-chip then it would not be necessary to specify them in the **inputs** command). As a result of this set of commands, two Caesar command files will be created: phi1cmds and phi2cmds.

```
inputs BusIn<0:31> Phi1 Phi2
outputs BusOut<31:0>

set Phi2 0
delay Phi1 0 -1
delay BusIn<0:31> 20 20
critical -g phi1cmds

clear
set Phi1 0
delay Phi2 0 -1
critical -g phi2cmds
```

Designing Finite State Machines with PEG

Gordon Hamachi

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

ABSTRACT

PEG is a finite state machine compiler. It translates high level language descriptions of finite state machines into the logic equations needed to implement state machine designs. Since the output format is compatible with *eqntott*, PEG may be used as a front end for Berkeley PLA tools.

1. Introduction

PEG (**P**LA **E**quation **G**enerator) is a design tool for finite state machines. It compiles high level language descriptions of finite state machines into the logic equations needed to implement a design.

PEG programs are isomorphic to Moore machine state diagrams. There is a one-to-one correspondence between states in a state diagram and state definitions in the corresponding *PEG* program. The translation from state diagrams to *PEG* programs is simple and straightforward.

Designing with PEG provides a number of advantages over the traditional pencil-and-paper approach method of FSM design. PEG's high level language enables designs and design changes to quickly be implemented. PEG programs provide easy-to-understand documentation with clear control flow. PEG does the tedious and error-prone bookkeeping task of generating *output* and *next state* bits as a function of current state bits. It checks for design errors and eliminates redundant terms in logic equations.

As output PEG generates logic equations in the *eqn* format accepted by *eqntott* [Cmelik], another Berkeley design tool. By piping the output of PEG

through *eqntott*, PEG may be used as a front end for Berkeley PLA tools such as *mpla* [Mayo], and *espresso* [Rudell]. As an option, *PEG* will also print the unminimized truth table from which the logic equations are derrived.

2. A Simple Example

Figure 1 shows the state diagram for a four-state finite state machine implementing a 2-bit binary counter. The *PEG* description of this design appears in Figure 2. The program has no inputs besides an implicit clock. The outputs of the state machine are its *next state* bits, which are automatically generated by *PEG*.

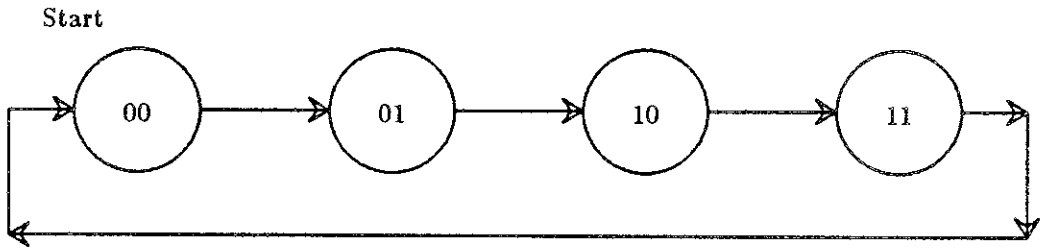


Figure 1: State Diagram for Example 1

In its most simple form, a *PEG* program consists of a list of state descriptions. The sample program has four states. Each state has four parts: an optional label, a colon, an optional signal assertion part, and and optional control part.

```
--Simple PEG program for 2-bit counter
--State transition on every clock
--No reset => starts in a random state

Start       :           --This is state 0
            :           --This is state 1
            :           --This is state 2
            :           --This is state 3
                          GOTO Start;
```

Figure 2: *PEG* Program for Example 1

The first state in the example is labeled with the identifier *Start*. The label is necessary only because of the GOTO from state 3 back to state 0.

States 1 and 2 are examples of the minimal state description. These states are completely defined by a colon, which acts as a place holder for the state. Empty states, in which no branching or signal assertions occur, are sometimes used to introduce necessary delays in FSMs.

Flow of control in *PEG* programs is sequential unless otherwise specified. Since no control information is present for states 0, 1, and 2, the program steps sequentially through the states 0, 1, 2, and 3. State 3 has control information specifying a jump back to the state labeled *Start*.

Since it has no sequential *next state*, control must always be defined for the last state in the program. *PEG* generates an error message and quits if control is not defined for the last state.

Although state transitions are performed on clock ticks, no clock is mentioned in the program. It is the user's responsibility to implement the state machine with synchronous logic to latch input and output signals.

Comments begin with a double dash "--" and terminate at the end of the line on which they appear. The first three lines of the program are comments. Comments also appear on lines 5 through 8.

Input is free-format. White space may appear anywhere in a program to enhance readability.

3. Interpreting the Output

Assuming that the *PEG* program for example 1 is in a file called *counter*, the following Unix command line may be used to invoke *PEG*:

peg counter

The resulting output is shown in figure 3. Generating a PLA from the same input file is accomplished with the command line:

peg counter | eqntott | mpla –I –O

Mpla will not automatically connect *next-state* outputs to *current-state* inputs. After generating the PLA the state outputs must be manually wired to the state inputs.

```
INORDER    =    InSt0* InSt1*;
OUTORDER   =    OutSt1* OutSt0*;
OutSt1*    =    (!InSt1*);
OutSt0*    =    ( InSt0*&!InSt1*)| (!InSt0*& InSt1*);
```

Figure 3: PLA Equations for Example 1.

3.1. Equations

PEG generates the two input variables *InSt0** and *InSt1** which are the state inputs for the finite state machine. It also generates two output variables *OutSt0** and *OutSt1**, the next-state outputs. Any signal name ending with an

asterisk was generated by *PEG*.

The INORDER and OUTORDER statements specify that the resulting PLA inputs and outputs, from left to right, are InSt0*, InSt1*, OutSt1*, and OutSt0*.

Following the OUTORDER statement are the logic equations for the two output variables, OutSt1* and OutSt0*. The exclaimation mark "!" indicates logical negation. The ampersand "&" signifies the logical *AND*, while the vertical bar " | " signifies a logical *OR*.

3.2. Truth Table

The **-t** option generates a truth table for the finite state machine. This truth table is written to the file *peg.summary*. The truth table for example 1 is shown in figure 4.

INPUTS:		s00:	InSt0* (msb)	
		s01:	InSt1* (lsb)	
OUTPUTS:		n01:	OutSt1* (lsb)	
		n00:	OutSt0* (msb)	
State Table	s	s	n	n
	0	1	1	0
	0	0	1	0
	0	1	0	1
	1	0	1	1
	1	1	0	0

Figure 4: Truth Table for Example 1.

Labels across the top of the truth table identify its columns. The mapping from column labels to actual signal names is given in the lists of input and output signals which preceed the truth table. To the right of the truth table are the names of the states described by the rows of the table.

4. Another Example

The second and more complex example shows the state diagram and corresponding *PEG* program for a FSM which recognizes the regular expression (1|0)*100. The state diagram for this FSM is shown in figure 5.

The PEG program which implements this design is given in figure 6. Figure 6 describes a state machine with four states. The state machine has two inputs, *RESET* and *in*, and one output, *accept*.

Assume the text of figure 2 is in a file called *prog*. Logic equations for the state machine are generated by running the command

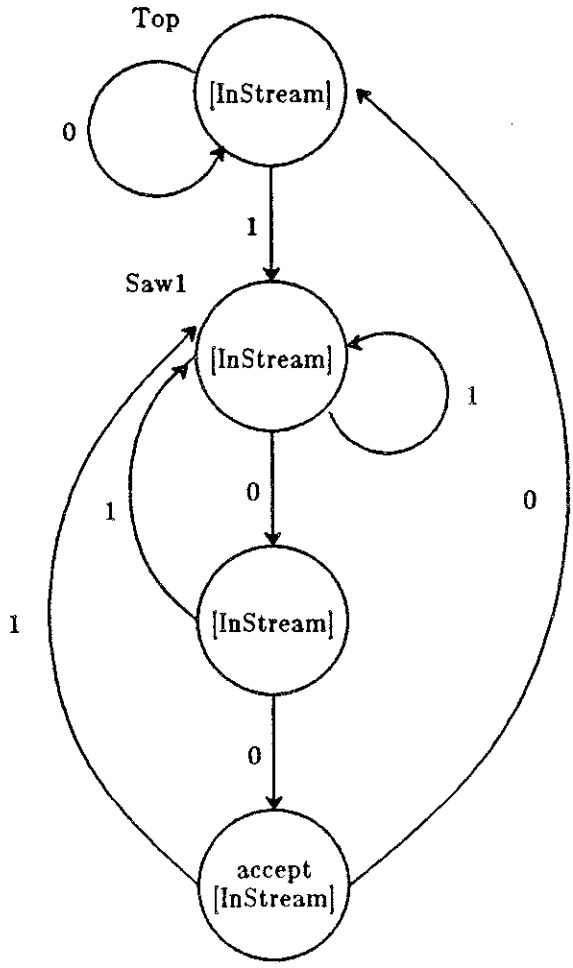


Figure 5: State Diagram Recognizing (1|0)*100

peg prog

Since this program has two inputs, they are declared in the *INPUTS* statement. If a *PEG* program has any inputs they must be declared in an *INPUTS* statement which must be the first statement in the program. The input *RESET* is a special keyword input. The other program input, *InStream*, is used to generate the *next state* for the FSM.

RESET indicates that when the *RESET* signal is asserted the state machine jumps to the top of the program, which in this case is named *Top*. When this keyword is present, conditional branches to the first state are automatically added to the *next state* expressions for each state. If *RESET* is not listed as an input, the program initializes in a random state.

IF the FSM designer does not want to pay the penalty of a larger and slower finite state machine, *RESET* may be omitted as it was in example 1. In this case

```
--Simple FSM example:  Accepts the regular expression (1|0)*100

INPUTS      :     RESET InStream;
OUTPUTS     :     accept;

Top         :     IF NOT InStream THEN LOOP;              --0*

Saw1        :     IF InStream THEN LOOP;                  --1

            :     IF InStream THEN Saw1;                  --10

            :     ASSERT accept;
                  IF InStream THEN Saw1 ELSE Top;         --100
```

Figure 6: PEG Program Recognizing (1|0)*100

```
INORDER     =     RESET InStream InSt0* InSt1*;
OUTORDER    =     OutSt1* OutSt0* Accept;
OutSt1*     =     (!RESET& InStream) |
                  (!RESET&!InStream& InSt0*&!InSt1*);
OutSt0*     =     (!RESET&!InStream& InSt0*&!InSt1*) |
                  !InStream&!InSt0*& InSt1*);
Accept      =     ( InSt0*& InSt1*);
```

Figure 7: Equations for Example 2.

the reset function may be external to the *PEG* program by implementing the FSM in such a manner that the *next state* feedback lines are pulled low when the *RESET* signal is asserted. This method will work because the top state in a *PEG* program is always assigned to state zero.

The *OUTPUTS* statement declares that this program has a single output called *accept*. The FSM asserts this signal high if a string in the given grammar is recognized. If any outputs are generated by a *PEG* program, they must be declared in an *OUTPUTS* statement which immediately follows the *INPUTS* statement. If no *INPUTS* statement is present, then the *OUTPUTS* statement is the first program statement.

INPUTS:	i00: RESET i01: InStream s00: InSt0* (msb) s01: InSt1* (lsb)
OUTPUTS:	n01: OutSt1* (lsb) n00: OutSt0* (msb) o00: Accept

State Table	i 0	i 1	s 0	s 1	n 1	n 0	o 0
1	-	0	0	0	0	-	Top
0	0	0	0	0	0	-	Top
0	1	0	0	1	0	-	Top
1	-	0	1	0	0	-	Saw1
0	0	0	1	0	1	-	Saw1
0	1	0	1	1	0	-	Saw1
1	-	1	0	0	0	-	Saw1+1
0	0	1	0	1	1	-	Saw1+1
0	1	1	0	1	0	-	Saw1+1
1	-	1	1	0	0	1	Saw1+2
0	0	1	1	0	0	1	Saw1+2
0	1	1	1	1	0	1	Saw1+2

Figure 8: Truth table for Example 2.

Example 2 introduces the *IF-THEN-ELSE* control construct. This construct is used to provide two-way branches *based only on a single input signal.* Branches based on more than one input signal are handled by the *CASE* statement which has not yet been presented. *IF* statements do not nest: Statements of the form *IF-THEN-ELSE-IF* are not allowed. The syntax of the *IF* is:

IF [NOT] <signal> THEN <state name> [ELSE <state name>] ;

The *ELSE* clause is optional: If it is omitted, the *ELSE* defaults to the next sequential state in the program. Thus, in state *Top*, if *InStream* is high, then the condition in the *IF* is false and the program takes the default branch to state *Saw1.*

The alert reader will have noticed that the state name *LOOP* is used but not defined. This is intentional. *LOOP* is a keyword which means to stay in the current state. It is an error to define a state with the label *LOOP*.

The final state in example 2 shows the first use of the *ASSERT* statement. The *accept* signal is asserted only in the accepting state of the FSM. If an *ASSERT* statement is present in the definition of a state, it must preceed the state's control statement.

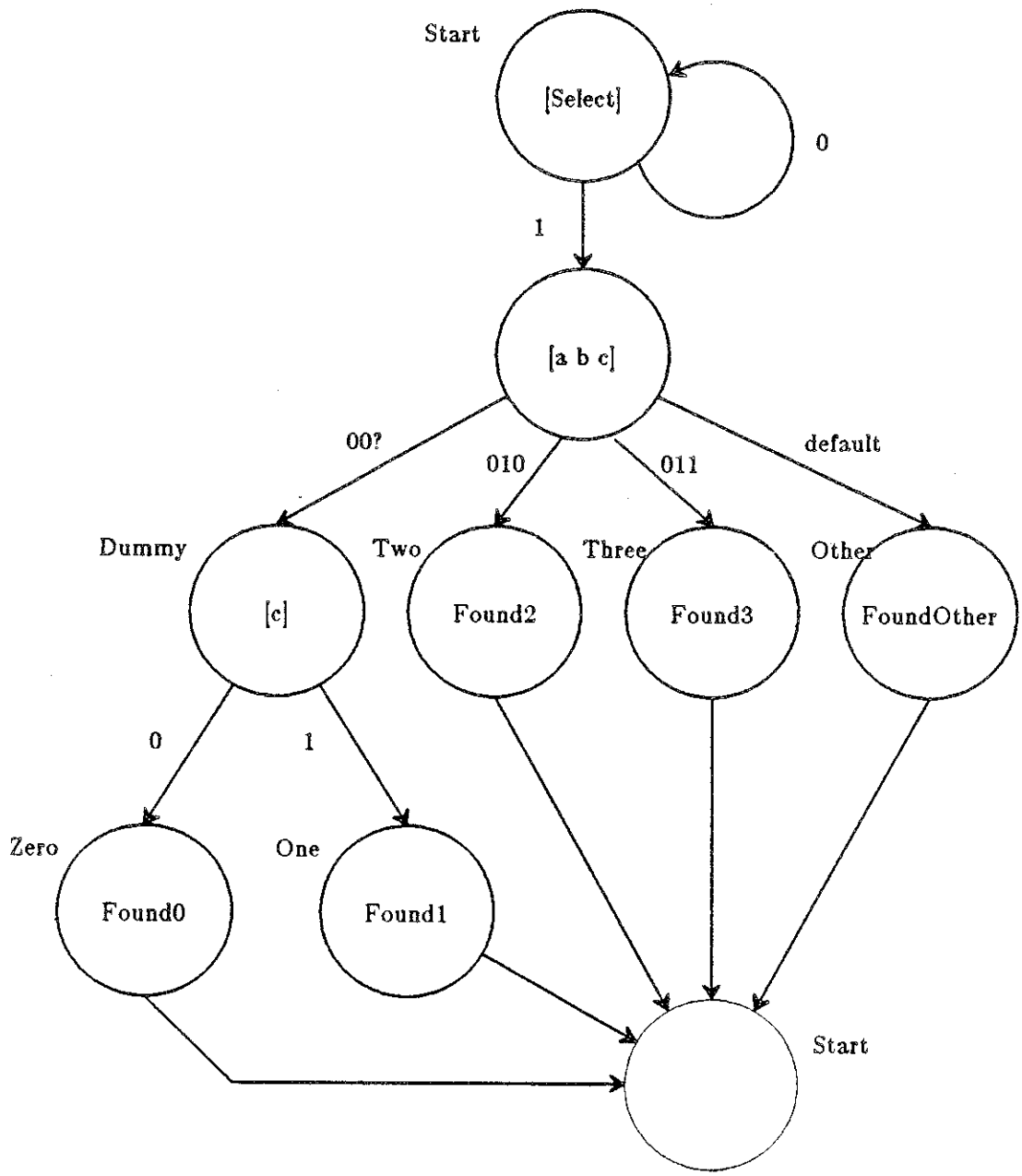


Figure 9: State Diagram for Example 3

Figure 11 shows an ambiguous case specifier. It is ambiguous because more than one case selector applies to the input (0 1 0). In such cases *PEG* processes

```
--Decode inputs a, b, and c into
--0, 1, 2, 3, or "other".

INPUTS      :   RESET Select a b c;
OUTPUTS     :   Found0 Found1 Found2 Found3 FoundOther;

Start       :   --This is the reset state
                IF NOT Select THEN LOOP;

            :   CASE (a b c) --Second state
                  0 0 ? => Dummy; --A don't-care
                  0 1 0 => Two;
                  0 1 1 => Three;
                ENDCASE=>Other;

Dummy       :   IF c THEN One;

Zero        :   ASSERT Found0; GOTO Start;

One         :   ASSERT Found1; GOTO Start;

Two         :   ASSERT Found2; GOTO Start;

Three       :   ASSERT Found3; GOTO Start;

Other       :   ASSERT FoundOther; GOTO Start;
```

Figure 10: PEG Program for Example 3

the list of case selectors from top to bottom, using the first one that applies to the inputs. Since the case specifier for State2 comes first, it defines the next state for inputs (0 1 0) and (0 1 1). The case specifier for State3 defines the next-state only for the case (1 1 0).

```
State1      :   CASE (a b c)
                  0 1 ? => State2;
                  ? 1 0 => State3;
                ENDCASE=>State4;
```

Figure 11: Ambiguity Resolution in Don't-Cares

5. Final Example

Figures 9 and 10 show the state diagram and *PEG* program for a state machine which decodes 3 bits into 0, 1, 2, 3, and "other". Example 3 shows the use of multiple inputs, multiple outputs, and multi-way branches.

Multi-way branches and branches based on two or more inputs are handled by the *CASE* statement. The *CASE* statement consists of the keyword *CASE* followed by an input signal list, a list of case selectors, and an *ENDCASE*.

A case selector specifies two things: a bit pattern corresponding to the input signals, and a *next-state* for that combination of inputs. Bit patterns are strings composed of the characters '0', '1', and signals in the input signal list. Don't-cares are specified with *?*.

The *ENDCASE* statement optionally specifies the default next-state if none of the other case selectors applies to the input. In keeping with the model of sequential execution, if the *ENDCASE* does not specify a next-state, the next-state defaults to the state following the one in which the *CASE* statement appears.

6. References

[CADMan]
CAD Manual, Online Unix documentation.

[Danford]
Peggy Danford, Private communication with author, June 1982.

[Unix]
Unix Programmer's Manual, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, Computer Science Division, University of California at Berkeley, November 1980.

7. Peg Syntax

```
<program>       : <InputList> <OutputList> <StateList>

<InputList>     : INPUTS : <IdentList> ; | /*NULL*/

<OutputList>    : OUTPUTS : <IdentList> ; | /*NULL*/

<StateList>     : <State> | <StateList> <State>

<IdentList>     : <Identifier> | <IdentList> <Identifier>

<State>         : <Identifier> : <Signals> <Control> | : <Signals> <Control>

<Signals>       : /*null*/ | <ASSERT> <IdentList> ;

<Control>       : CASE ( <IdentList> ) <Cases> <DefaultCase>
                | IF <Identifier> THEN <Identifier> ;
                | IF <Identifier> THEN <Identifier> ELSE <Identifier> ;
                | IF <NOT> <Identifier> THEN <Identifier> ;
                | IF <NOT> <Identifier> THEN <Identifier> ELSE <Identifier> ;
                | GOTO <Identifier>
                | /*NULL*/

<Cases>         : <Cases> <CaseStmt> | <CaseStmt>

<CaseStmt>      : <BitList> => <Identifier> ;

<Bit>           : 0 | 1 | ?

<BitList>       : <BitList> <Bit> | <Bit>

<DefaultCase>   : ENDCASE => <Identifier> ; | ENDCASE ;

<NOT>           : "!" | "NOT" | "-"

<Comment>       : "--".*$

<Identifier>    : [A-Za-z][A-Za-z0-9._]*
```