1983 VLSI Tools:
Selected Works by the Original Artists

Includes these smash hits:

Caesar	by John Ousterhout
Cifplot	by Dan Fitzpatrick
Crystal	by John Ousterhout
Eqntott	by Bob Cmelik
Esim	by Chris Terman (MIT)
Lyra	by Michael Arnold
Mextra	by Dan Fitzpatrick
Peg	by Gordon Hamachi
Quilt	by Robert Mayo
Slang	by John Foderaro
Tpack	by Robert Mayo
Tpla	by Robert Mayo

Plus many more of your old favorites...

March, 1983

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

Computer Science Division
EECS Department
University of California at Berkeley

TABLE OF CONTENTS

About this distribution.....

This manual describes the programs in the 1983 VLSI Tools Distribution put together by the CS Division of the Department of EECS, UC Berkeley. The distribution consists of about twenty programs for designing and analyzing VLSI circuits. The programs were designed to run on VAXes under the Berkeley 4.1 distribution of Unix, and are not known to run under any other systems. The tools 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

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

Highlights

We have several new tools on this distribution, as well as enhancements to old tools. Here's an overview of the major tools:

Caesar	Our graphical layout editor, with which many of you are familiar. This version includes macros, new short commands, an interactive interface to Lyra, and drivers for several graphics displays. Be sure to look in the tutorial manual for other features of Caesar version VII.
Cifplot	Plots CIF files. It can work with nMOS, CMOS, and other technologies.
Crystal	A timing analyzer that helps the designer find performance problems in his design. It is able to send its output back to Caesar, giving a graphical display of critical paths, high capacitance nodes, and other potential performance bugs.
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.
Lyra	A hierarchical, corner-based design rule checker that handles many sets of design rules. Rules are written in a LISP-like rules language, which is then compiled to produce a customized design rule checker. Lyra can be invoked interactively from Caesar.
Mextra	A manhattan circuit extractor. Its output can be fed into Crystal, Esim, or converted for input to Spice.
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.

Quilt	A generator of personalized arrays (built using Tpack).
Slang	A functional-level simulator. Designs may be simulated at any level of abstraction using a lisp-like language. The results may be compared to the **esim** simulation in order to validate the layout.
Tpack	A library of routines for generating semi-regular modules. These routines allow module generators to generate layouts by assembling tiles (which are small chunks of layout designed with Caesar). The end result is a module generator that can generate different styles of modules depending upon what set of tiles is used.
Tpla	A technology-independent PLA generator built using Tpack.

Currently only a few of our tools handle CMOS: Caesar, Cifplot, Lyra, Mextra, Tpack, and Quilt. We are busy converting some of the other tools (such as Crystal), and we plan to build our new tools so they are independent of any particular technology. Tpla doesn't have a set of CMOS tiles designed yet -- they are still under development.

Installation Instructions

The tape is written in Unix "tar" program format, 1600 BPI, 20 blocks per record. There are about 10000 kbytes of stuff on the tape. Installing the tools on a VAX running Berkeley 4.1 Unix is relatively easy. First, create a new user, "cad". Then change your current directory to ~cad, and load all the information from the tape into the ~cad area with the command "tar x", then type "INSTALL".

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, "cad83" for example, then move individual programs over to try them. Be careful if you use this approach, since many of the programs (like Caesar, Cifplot, Lyra, Tpack, Quilt, and Tpla) use library information in ~cad/lib.

Several things need to be set up for your specific machine. The cadman macros in ~cad/man/tmac/tmac.anc need to be modified to reflect the location of ~cad on your machine, or if your normal troff macros are not in the standard place (/usr/lib/tmac). Vlsifont expects the Berkeley fonts to be present in the standard place (/usr/lib/vfont).

Although the other programs should work immediately after loading them from tape, Caesar really needs to be tuned to your site. To do this, cd to ~cad/src/caesar and follow the instructions in the file ReadMe. 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. The manuals can be printed with troff and/or eqn and/or tbl. Along with the tape you should get one copy of each user manual, 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 most, but not all, of the programs.
doc	Contains tutorial-style user manuals for many of the more complicated programs.
examples	Contains examples of how to use the programs. This is a recent addition, and there isn't much in this directory as of now.

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.

Acknowledgements

An early version of this tape was sent to several beta test sites, who then tried out our tools and gave us feedback. Many bugs were discovered – we fixed most of them and documented the rest. We were impressed with the amount of effort put in by our test sites.

Many thanks to:

Aerospace Corporation (Mel Cutler was the contact but lots of other people helped)
UCLA (David Booth)
Carnegie-Mellon University (Carl Ebeling)
Microelectronics Center of North Carolina (Gary Nifong)
University of North Carolina (Gary Bishop)
University of Wisconsin (Randy Katz and Dan Schuh)

We've listed the name of the contact person at each site, but in many cases other people helped too.

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.

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. It may also be "cad", "new", "junk", or "public". 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 *ssp*(1) to crush out useless blank lines, *ul*(1) to create proper underlines for different terminals, and through *more*(1) to stop after each page on the screen. Hit a carriage return to continue, 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 Versatec.

FILES

~cad/doc/cadman/man?/*

SEE ALSO

Programmer's manual: more(1), ul(1), ssp(1), man(1), appropos(1)

AUTHOR

William Joy (modified by Jim Kleckner)

BUGS

The manual is supposed to be reproducible either on the phototypesetter or on a typewriter. However, on a typewriter some information is necessarily lost.

NAME

caesar - VLSI circuit editor

SYNOPSIS

caesar [**-n** **-g** graphics_port **-t** tablet_port **-p** path **-m** monitor_type **-d** display_type] [file]

DESCRIPTION

Caesar is an interactive system for editing VLSI circuits at the level of mask geometries. It uses a variety of color displays with a bit pad as well as a standard text terminal. For a complete description and tutorial introduction, see the user manual "Editing VLSI Circuits with Caesar" (an on-line copy is in ~cad/doc/caesar.tblms).

Command line switches are:

-n Execute in non-interactive mode.

-g The next argument is the name of the port to use for communication with the graphics display. If not specified, Caesar makes an educated guess based on the terminal from which it is being run.

-t The next argument is the name of the port to use for reading information from the graphics tablet. If not specified, Caesar makes an educated guess (usually the graphics port).

-p The next argument is a search path to be used when opening files.

-m The next argument is the type of color monitor being used, and is used to select the right color map for the monitor's phosphors. "Barco2" works well for most monitors, "barco" is for monitors with especially pale blue phosphor.

-d The next argument is the type of display controller being used. Among the display types currently understood are: AED512, UCB512 (the AED512 with special Berkeley PROMs for stippling), AED767, AED640 (an AED767 configured as 483x640 pixels), Omega440, R9400, or Vectrix.

When Caesar starts up it looks for a command file with the name ".caesar" in the home directory and processes it if it exists. Then Caesar looks for a .caesar file in the current directory and reads it as a command file if it exists. The .caesar file format is described under the long command *source*.

You generally have to log in a job on the color terminal under the name "sleeper" (no password required). This is necessary in order for the tablet to be useable. Sleeper can be killed either by typing two control-backslashes in quick succession on the color display keyboard (on the AED displays, control-backslash is gotten by typing control-shift-L), or by invoking the shell command *killsleeper* with the correct process id. On some systems you have to log yourself in and run sleeper as a shell command. On still other systems there is no login process for the color display port, so it isn't necessary to run sleeper at all.

The four buttons on the graphics tablet puck are used in the following way:

left (white)
Move the box so that its fixed corner (normally lower-left) coincides with the crosshair position.

right (green)
Move the box's variable corner (normally upper-right) to coincide with the crosshair position. The fixed corner is not moved.

top (yellow)
Find the cell containing the crosshair whose lower-left corner is closest to the crosshair. Make that cell the current cell. If the button is depressed again without moving the crosshair, the parent of the current cell is made the current cell.

bottom(blue)
Paint the area of the box with the mask layers underneath the crosshair. If there are no mask layers visible underneath the crosshair, erase the area of the box.

SHORT COMMANDS

Short commands are invoked by typing a single letter on the keyboard. Valid commands are:

a	Yank the information underneath the box into the yank buffer. Only yank the mask layers present under the crosshair (if there are no mask layers underneath the crosshair, yank all mask layers and labels).
c	Unexpand current cell (display in bounding box form).
d	Delete paint underneath the box in the mask layers underneath the crosshair (if there are no mask layers underneath the crosshair, the delete labels and all mask layers).
e	Move the box up 1 lambda.
g	Toggle grid on/off.
l	Redisplay the information on both text and graphics screens.
q	Move the box left 1 lambda.
r	Move the box down 1 lambda.
s	Put back (stuff) all the information in the yank buffer at the current box location. Stuff only information in mask layers that are present underneath the crosshair (if there are no mask layers underneath the crosshair, stuff all mask layers plus labels).
u	Undo the last change to the layout.
w	Move the box right one lambda.
x	Unexpand all cells that intersect the box but don't contain it.
z	Zoom in so that the area underneath the box fills the screen.
C	Expand current cell so that its paint and children can be seen.
X	Expand all cells that intersect the box, recursively, until there are no unexpanded cells intersecting the box.
Z	Zoom out so that everything on current screen fills the area underneath the box.
5	Move the picture so that the fixed corner of the box is in the center of the screen.
6	Move the picture so that the variable corner of the box is in the center of the screen.
^L	Redisplay the graphics and text displays.
.	Repeat the last long command.

LONG COMMANDS

Long commands are invoked by typing a colon character (":"). The cursor will appear on the bottom line of the text terminal. A line containing a command name and parameters should be typed, terminated by return. Each line may consist of multiple commands separated by semicolons (to use a colon as part of a long command, precede it with a backslash). Short commands may be invoked in long command format by preceding the short command letter with a single quote. Unambiguous abbreviations for command names and parameters are accepted. The commands are:

align <scale>
Change crosshair alignment to <scale>. Crosshair position will be rounded off to nearest multiple of <scale>.

array <xsize> <ysize>
: Make the current cell into an array with <xsize> instances in the x-direction and <ysize> instances in the y-direction. The spacing between elements is determined by the box x- and y-dimensions.

array <xbot> <ybot> <xtop> <ytop>
: Make the current cell into an array, numbered from <xbot> to <xtop> in the x-direction and from <ybot> to <ytop> in the y-direction. The spacing between array elements is determined by the box x- and y-dimensions.

box <keyword> <amount>
: Change the box by <amount> lambda units, according to <keyword>. If <keyword> is one of "left", "right", "up", or "down", the whole box is moved the indicated amount in the indicated direction. If <keyword> is one of "xbot", "ybot", "xtop", or "ytop", then one of the coordinates of the box is adjusted by the given amount. <amount> may be either positive or negative.

button <number> <x> <y>
: Simulate the pressing of button <number> at the screen location given by <x> and <y> (in pixels). If <x> and <y> are omitted, the current crosshair position is used.

cif -sblpx <name> <scale>
: Write out a CIF description of the layout into file <name> (use edit cell name by default; a ".cif" extension is supplied by default). <scale> indicates how many centimicrons to use per Caesar unit (200 by default). The -s switch causes no silicon (paint) to be output to the CIF file. The -b switch causes bounding boxes to be drawn for unexpanded cells. The -l causes labels to be output. The -p switch causes a CIF point to be generated for each label. The -x switch causes Caesar not to automatically expand all cells (they are expanded by default).

cload <file>
: Load the colormap from <file>. The monitor type is used as default extension.

clockwise <degrees> [y]
: Rotate the current cell by the largest multiple of 90 degrees less than or equal to <degrees>. <degrees> defaults to 90. If the command is followed by a "y" then the yank buffer is rotated instead of the current cell.

colormap <layers>
: Print out the red, green, and blue intensities associated with <layers>.

colormap <layers> <red> <green> <blue>
: Set the intensities associated with <layers> to the given values.

copycell
: Make a copy of the current cell, and position it so that its lower-left corner coincides with the lower-left corner of the box.

csave <file>
: Save the current colormap in <file> (the monitor type is used as default extension).

deletecell
: Delete the current cell.

editcell <file>
: Edit the cell hierarchy rooted at <file>. A ".ca" extension is supplied by default. If information in the current hierarchy has changed, you are given a chance to write it out.

erasepaint <layers>
: For the area enclosed by the box, erase all paint in <layers>. If <layers> is omitted it defaults to "*l".

fill <direction> <layers>
: <direction> is one of "left", "right", "up", or "down". The paint under one edge of the box (respectively, the right, left, bottom, or top edge) is sampled; everywhere that the edge touches paint, the paint is extended in the given direction to the opposite side of the box. <layers> selects which layers to fill; if omitted then a default of "*" is used.

flushcell
: Remove the definition of the current definition from main memory and reload it from the disk version. Any changes to the cell since it was last written are lost.

getcell <file>
: This command makes an instance of the cell in <file> (a ".ca" extension is supplied by default) and positions that instance at the current box location. The box size is changed to equal the bounding box of the cell.

gridspacing
: The grid is modified so that its spacings in x and y equal the dimensions of the box. The grid is set so that the box falls on grid points.

gripe The mail program is run so that comments can be sent to the Caesar maintainer.

height <size>
: The box's height is set to <size>. If <size> is preceded by a plus sign then the fixed corner is moved to set the correct height; otherwise the variable corner is moved. <size> defaults to 2.

identifycell <name>
: The current cell is tagged with the instance name given by <name>. This feature is not currently supported in any useful fashion. <name> may not contain any white space.

label <name> <position>
: A rectangular label is placed at the box location and tagged with <name>. <name> may not contain any white space. <position> is one of "center", "left", "right", "top", or "bottom"; it specifies where the text is to be displayed relative to the rectangle. If omitted, <position> defaults to "top".

lyra <ruleset>
: The program ~cad/bin/lyra is run, and is passed via pipe all the mask features within 3λ of the box. The program returns labels identifying design rule violations, and these are added to the edit cell. If <ruleset> is specified, it is passed to Lyra with the -r switch to indicate a specific ruleset. Otherwise, the current technology is used as the ruleset.

macro <character> <command>
: The given long command is associated with the given character, such that whenever the character is typed as a short command then the given command is executed. This overrides any existing definition for the character. To clear a macro definition, type ":macro <character>", and to clear all macro definitions, type ":macro"

mark <mark1> <mark2>
: The box is saved in the mark given by <mark1>. <mark1> must be a lower-case letter. If <mark2> is specified, the box is changed to coincide with <mark2>.

movecell <keyword>
: The current cell is moved in one of two ways, selected by <keyword>. If <keyword> is "byposition", then the cell is moved so that its lower-left corner coincides with the lower-left corner of the box. This also happens if no keyword is specified. If <keyword> is "bysize", then the cell is displaced by the size of the box (this means that what used to be at the fixed corner of the box will now be at the variable corner).

paint <layers>
: The area underneath the box is painted in <layers>.

path <path>
: The string given by <path> becomes the search path used during file lookups. <path> consists of directory names separated by colons or spaces. Each name should end in "/".

peek <layers>
: Display all paint underneath the box belonging to <layers>, even for unexpanded cells and their descendants.

popbox <mark>
: If <mark> is specified, then the box is replaced with the given mark. Otherwise the box stack is popped and the top stack element overwrites the box.

pushbox <mark>
: The box is pushed onto the box stack. If <mark> is specified then it is used to overwrite the box, otherwise the box remains unchanged.

put <layers>
: The yank buffer information in <layers> is copied back to the box location. If <layers> is omitted, it defaults to "*Sl".

quit If any cells have changed since they were last saved on disk, the user is given a chance to write them out or abort the command. Otherwise the program returns to the shell.

reset The graphics display is reinitialized and the colormap is reloaded.

return The current subedit is left, and the containing edit is resumed.

savecell <name>
: If <name> is specified then the current cell is given that name and written to disk under the name (a ".ca" extension is supplied by default). If <file> isn't specified then the cell is written out to the disk file from which it was read.

scroll <direction> <amount> <units>
: The current view is moved in the indicated direction by the indicated amount. <direction> must be one of "left", "right", "up", or "down", <amount> is a floating-point number, and <units> is one of "screens" or "lambda". <units> defaults to "screens", and <amount> defaults to 0.5.

search <regexp>
: Search labels and bounding boxes underneath the box for text matching <regexp>. See the manual entry for *ed* for a description of <regexp>. Push an entry onto the box stack for each match. Even unexpanded cells are searched.

sideways [y]
: Flip the current cell sideways (i.e. about a vertical axis). If the command is followed by a "y" then the yank buffer is flipped instead of the current cell.

source <filename>
: The given file is read, and each line is processed as one long command (no colons are necessary). Any line whose last character is backslash is joined to the following line.

subedit
: Make the current cell the edit cell, and edit it in context.

technology <file>
: Load technology information from <file>. A ".tech" extension is supplied by default.

upsidedown [y]
: Flip the current cell upside down. If the command is followed by a "y" then the yank buffer is flipped instead of the current cell.

usage <file>
: Write out in <file> the names of all the files containing cell definitions used anywhere in

the design hierarchy.

view <mark>
If <mark> is specified, set view to it, otherwise, change the view to encompass the entire edit cell.

visiblelayers <layers>
Set the visible layers to include just <layers>. Preface <layers> with a plus or minus sign to add to or remove from the currently visible ones.

width <size>
Set the box width to <size> (default is 2). Move variable corner unless width is preceded by "+ ", else move fixed corner.

writeall
Run through interactive script to write out all cells that have been modified.

yank <layers>
Save in the yank buffer all information underneath the box in <layers>. <layers> defaults to "*l".

ycell <name>
If <name> is specified, do the equivalent of ":getcell <name>". Then expand current cell, yank it, delete the cell, and put back everything that was yanked. This flattens the hierarchy by one level.

ysave <name>
Save the yank buffer contents in a cell named <name>. A ".ca" extension is provided by default.

LAYERS

nMOS mask layers are:

p or r	Polysilicon (red) layer.
d or g	Diffusion (green) layer.
m	Metal (blue) layer.
i or y	Implant (yellow) layer.
b	Buried contact (brown) layer.
c	Contact cut layer.
o	Overglass hole (gray) layer.
e	Error layer: used by design rule checkers and other programs.

CMOS P-well mask layers are (using technology cmos-pw):

p or r	Polysilicon (red) layer.
d or g	Diffusion (green) layer.
m	Metal (blue) layer.
c	Contact cut layer.
P or y	P+ implant (pale yellow) layer.
w	P-well (brown stipple) layer.
o	Overglass hole (gray) layer.
e	Error layer: used by design rule checkers and other programs.

Predefined system layers are:

*****	All mask layers.
l	Label layer.
S	Subcell layer.
C	Cursor layer.
G	Grid layer.
B	Background layer.

SYSTEM MARKS

C	The bounding box of the current cell.
E	The bounding box of the edit cell.
P	The previous view.
R	The bounding box of the root cell.
V	The current view.

FILES

~cad/new/caesar, ~cad/doc/caesar.tblms

SEE ALSO

cif2ca(1)

AUTHOR

John Ousterhout

BUGS

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

cif2ca – convert CIF files to CAESAR files

SYNOPSIS

cif2ca [**–l** *lambda*] [**–t** *tech*] [**–o** *offset*] ciffile

DESCRIPTION

cif2ca accepts as input a CIF file and produces a CAESAR file for each defined symbol. Specifying the **–l** *lambda* option scales the output to *lambda* centi-microns per lambda. The default scale is 200 centi-microns per lambda. The **–t** *tech* option causes layers from the specified technology to be acceptable. The default technology is nmos. For a list of acceptable technologies, see *caesar* (1). The **–o** *offset* option causes all CIF numbers to be incremented by *offset*. This is useful when the CIF numbers are used for Caesar file names, and when several CIF files with overlapping numbers are to be joined together in Caesar.

Each symbol defined in the CIF file creates a CAESAR file. By default, the files are named "symbol*m*.ca", where *m* is the CIF symbol number (as modified by the **-o** *offset*). Symbols can also be named with a user-extension "9" command, giving a name to the symbol definition which encloses it. CIF commands which appear outside of symbol definitions are gathered into a symbol called, by default, "project", and are output to the CAESAR file "project.ca".

SEE ALSO

caesar(1)

DIAGNOSTICS

Diagnostics from *cif2ca* are supposed to be self-explanatory. Each diagnostic gives the line number from the input file, an error class (informational, warning, fatal, or panic), the error message, and the action taken by *cif2ca*, usually to ignore the CIF command. Informational messages usually refer to limitations of *cif2ca*. Warning messages usually refer to inconsistencies in the CIF file, these will typically result in CAESAR files which do not accurately reflect the input CIF file. Fatal messages refer to fatal inconsistencies or errors in the CIF file. A fatal error terminates *cif2ca* processing. Panic messages refer to internal problems with *cif2ca*. If any diagnostics are produced, a summary of the diagnostics is produced.

AUTHOR

Peter B. Kessler, bug fixes and new features by John Ousterhout and Steve Rubin.

BUGS

"Delete Definitions" commands are not implemented. *cif2ca* also has certain restrictions due to restrictions of CAESAR: *e.g.* non-manhattan objects are not allowed.

Library cells are not automagically included.

Some care should be taken in naming symbols, since symbol names are used for CAESAR file names. Names which are not unique in the first 14 characters will attempt to create the same CAESAR file, and only the last one wins. Similarly, one should avoid trying to have two project.ca files in the same directory.

NAME

cifplot – CIF interpreter and plotter

SYNOPSIS

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

DESCRIPTION

Cifplot takes a description in Cal-Tech 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 programs 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 llb.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 poly-silicon:

```
"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/bin/vdump
/usr/lib/vfont/R.6
/usr/tmp/#cif*

ALSO SEE

mcp(cad1), vdump(cad1), cadrc(cad5)
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 semicolons 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 -l option to *cif2ca*.

NAME

cifstat – prints out statistics about a CIF file

SYNOPSIS

cifstat file.cif

DESCRIPTION

cifstat reads in a structured CIF file and prints out statistics about the file. If no file is specified then input is taken from standard input. The statistics includes a count of the number of rectangles in the CIF file and the total number of rectangles in the fully instanciated circuit. The sum of the lengths of the rectangles for each layer is also printed out.

AUTHOR

Dan Fitzpatrick

NAME

clipcif – clip CIF file to specified window

SYNOPSIS

clipcif [*options*] [file.cif]

DESCRIPTION

clipcif reads in a structured CIF file and outputs a structured CIF file to standard output that has been clipped against the specified window. If no CIF file is specified then input is taken from standard input. If no window is specified then the CIF file is not clipped at all, and the output file is equivelant to the input file. The window is specified in user specified units. (CIF units by default.) Units may be set by the **-u** option, or specified in your '.cadrc' file. Specifying units using **-u** option overrides the '.cadrc' specifier. The window may be set with with one of two options, the **-w** option which sets the window in absolute coordinates, or **-c** option which sets the window in relative coordinates. The form of these two options is:

-w *xmin ymin xmax ymax*
-c *xmin ymin deltax deltay*

ALSO SEE

cadrc(cad5)

AUTHOR

Dan Fitzpatrick

NAME

Crystal - VLSI timing analyzer

SYNOPSIS

crystal [**–s –S –d –D –l** nodelimit **–u** units **–b** threshold **–r** repeatlimit **-p -m**] file

DESCRIPTION

Crystal is a semi-interactive program for analyzing the timing characteristics of large integrated circuits. It reads in **file** (which must be in .sim format) and then processes commands from standard input. Diagnostic messages are output on standard output.

Command line switches are:

–s	When nodes get set to fixed values (because of the **set** command), print out information for all such nodes whose names don't begin with a digit.
–S	Print out all nodes that that get set to fixed values, regardless of name.
–d	When computing delay information, print information everytime a new worst time is found for a node whose name doesn't begin with a digit.
–D	Print delay information for all nodes, regardless of name.
–l	The next argument is an integer limit value. When propagating delay information, Crystal gives up in despair if it evaluates more than the limit number of stages (since the last **clear** command) without completing its job. The default is 200000.
–u	The next argument indicates the units to use when printing coordinates, in microns. The default is 2.0, which means that a printed coordinate of 1 corresponds to 2 microns.
–b	The next argument contains a capacitance value in picofarads. Any node with at least this much capacitance is considered to be a bus. The default threshold value is 1.0 pf.
–r	The next argument contains an integer repeat limit. Several of the printout commands attempt to eliminate lines that have the same value of something (capacitance, for example). The repeat limit specifies the maximum number of times that any one value can appear in a single such command. The default repeat limit is one.
–p	When propagating timing information during the "propagate" command, this switch causes information to be printed for each node from which delays are propagated.
–m	If this switch is specified, then during the "markdynamic" command information is printed for each dynamic node found.

COMMANDS

After reading in the .sim file, Crystal processes commands from the standard input, one per line (lines beginning with exclamation points are ignored). Unique abbreviations for commands are acceptable. The commands normally appear in three groups. The first commands are usually **bus**, **inputs**, **precharged**, and **set** commands; these give Crystal additional information to guide its timing analysis. The second group usually contains one or more **delay** commands to invoke the timing analysis. Then commands like **worstdelay** and **prgreater** are used to print out information about long paths. After all three groups, there may be a **clear** command followed by the groups again for a different analysis of the same chip (e.g. a different clock phase).

Where nodes 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.

Several of the commands have switches to generate Caesar command files. To use the 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.

The commands are:

bus node node ...
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 command line switch **-b**.

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.

clear [-t]
All node markings are removed, including those set with the **bus**, **inputs**, **precharged**, and **set** commands. If the **-t** switch isn't used, then all timing information is cleared. If the **-t** switch is given, then timing information is not affected. This command doesn't affect information set with the **capacitance**, **resistance**, or **markdynamic** commands.

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 entire network has settled. A -1 value for **risetime** or **falltime** means that there is no transition to that level.

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", or "ignore". If "ignore" is specified then no restrictions are enforced whatsoever.

help Print a short listing of the valid commands.

inputs node node ...
Mark each of the nodes as in 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, Crystal assumes that the timing of input nodes is fixed by the outside world and is not affected by anything in the circuit: if no **delay** command is given for the node, then Crystal assumes the value never changes.

manyworst [-m] [-c file] [-n number]
This command prints out information about several of the longest delay paths. The **-c** switch is used to specify a Caesar command file, if Caesar commands are desired. The **-n** switch is used to indicate how many paths should be printed out (default: 10). Then Crystal searches for the slowest nodes on the chip and prints out the paths to the indicated number of nodes. The repeat limit is used to ignore nodes when several have exactly the same total delay. If the **-m** switch is specified, then only paths ending in dynamic or static memory nodes are considered.

markdynamic
This statement causes Crystal to examine all nodes. If a node is not an input and cannot be driven to either zero or one (because all transistors connecting to the node are enhancement transistors that are forced off), it is marked as a dynamic memory node.

This statement should generally be preceded by **input** and **set** statements to turn off all the clocks in the circuit.

model name
Use **name** as the model for delay calculations. Currently, two models are available. The **unit** model assumes that it takes one nanosecond for a transistor to turn on or off, but that all resistances both for transistors and interconnect are zero. In other words, the unit model counts how many gates a signal must pass through. The **tau** model uses a simple RC view of the world to compute real delays. The default is **tau**.

parameter [name] [value]
This statement is used to see or change the parameters used in calculating delays. **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. Only the parameters of the current model may be printed or changed.

prcapacitance [-c 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 switch isn't present). If no node names are given, then all nodes in the the circuit are checked. The repeat limit is used to restrict the printing of many nodes with the same capacitance value. The **-c** switch can be used to generate a Caesar 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 marked as a bus.

prgreater [-c file] [-t threshold] node node ...
Print out information for each of the indicated nodes with a single stage delay to the node of at least **threshold** nanoseconds. Single stage delay refers to the delay from the node immediately preceding the given node on its worst-case timing path. If no threshold is given, 0 is used by default. If no node names are given, all nodes in the circuit are checked. The repeat limit is used to restrict the printing of nodes with exactly the same delay value. The **-c** switch causes the generation of a Caesar command file.

printallfets
Print out information about all transistors in the circuit. This is not very useful except for debugging Crystal.

propagate
This statement is used to propagate information between clock phases. The entire circuit is searched for enhancement transistors that were forced off before the last **clear** statement, but are not currently forced off. For each such transistor, delay information is propagated from each side of the transistor to the other.

prresistance [-c 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 repeat limit is used to restrict the printing of nodes with exactly the same delay value. The **-c** switch is used to generate a Caesar command file.

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.

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

spice [-m] file [node]
Generates a SPICE deck in **file**, describing the transistors and parasitics along the worst-case timing path to **node**. If **node** isn't specified, then the slowest node in the chip is used, unless **-m** is specified, in which case the slowest memory node is used. In the SPICE deck, no model parameters or bias voltages are output. Node 0 is ground, node 1 is Vdd, and node 2 is the body bias voltage.

worstdelay [-m] [-c file] node node ...
Print out the worst-case delay paths to each of the indicated nodes. If no nodes are given, print out the overall worst-case delay path for the circuit. If the **-c** switch is present, a Caesar command file is generated. If the **-m** switch is given, then only static and dynamic memory nodes are considered.

FILES
~cad/new/crystal, ~cad/bin/crystal

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 option seperately; 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 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 <newline>
```

where 'c' is a single letter specifying the command to be performed and the *argi* 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.

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 is 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.

R n
: Run the simulator through n cycles. 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
mextra(CAD1)

AUTHOR
Chris Terman

BUGS
This is an old version of *esim*, we should get an up-to-date version soon.

NAME

intro – introduction to the cad section of the manual

DESCRIPTION

The *cad* section of the manual describes programs to aid in the design and implementation of integrated circuits and computer hardware. Included are programs for circuit simulation, layout, plotting, compaction, placement, and logic minimization. All of the programs work on NMOS designs; however, the only tools capable of handling CMOS designs are caesar, mextra, cifplot (and other cif tools), lyra, tpack, quilt, and tpla.

This manual entry describes the directory structure which contains the binaries and sources of these programs and the conventions of their use. The root of the tree, which contains all of the *cad* binaries, sources and documentation is, by convention, the home directory of a pseudo-user called **cad.** This way, programs which use *csh(1)* can refer to *~cad* in pathnames to allow relocation from one machine to another. At the first level of the tree are 6 directories: **bin, src, man, doc, new,** and **old. Bin** contains binaries for all of the *cad* programs. **Src** contains sources for all of the *cad* programs in **bin,** and **man** contains manual entries which are exactly like those in **/usr/man** (see *man(1)*). User guides and other documentation which are too long for the short, one page manual entries are kept in **doc. New** is used only for binaries which are, as yet, untested or are not compatible with other tools.

Thus, a user who wishes to access these tools would set his search path (see *setenv* in *csh(1)*) through *~cad/bin*. Binaries in **new** should migrate fairly quickly over to **bin.** A daring user who wants to help out with debugging untested or variant versions might set his path to first search *~cad/new*. A person may wish, when migrating a binary from **new** into **bin,** to put the copy of the binary from **bin** into the **old** directory for two or three days. The rest of this note describes the conventions for using these directories when installing a new program or working on an existing one.

In order to allow programs at Berkeley both to be developed on different network sites and shared between them, certain conventions are used to reduce the complexity of the remote updating problem. The assumption made here is that any specific program is developed at only one network site which is then the reference site for that program. All updates for both program source and documentation originate from there. To reduce the difficulty of updating the sources on different machines and to help in automating the process, separate directories are created in **~cad/src** and **~cad/doc** for each originating site for which software is being shared. Thus, for example, there is an entry **~cad/src/pcad** which contains the sources for the programs developed at the site *ucbcad*. Updating all of the software, for example, from *ucbcad* on *ucbvax* then becomes a simple *tar(1)* of **~cad/src/pcad** and **~cad/doc/pcad.** Alternatively, if a network connection of sufficient bandwidth is available, an automatic scheme where the dates of the files in the remote directory are compared against the dates in the local directory and old copies replaced. This would then be followed by *make(1)*. Installing a program for distribution then becomes a simple matter of placing it in the local outgoing directory. For example, to install the program *splice(CAD)* from the **ucbcad** machine would require the creation of the directory **~cad/src/pcad/splice** and then copying the needed source files into it. It is then important to perform a *make* to insure consistency with **~cad/bin.**

Some simple conventions are needed to make the process work smoothly. First, except for brief periods when being installed locally, the source for a program in **~cad/src** should correspond exactly to the binary in **~cad/bin.** This implies that development copies should be kept outside of the **~cad/src** hierarchy. Second, whenever an update of sources occurs, the binaries in **~cad/bin** should be rebuilt with *make(1)*. Third, any site or machine specific code should be parameterized so that the same source can be compiled at different places without modification. For this use, several defined strings are passed to **C** programs and **makefiles:**

HOMEOFCAD	fully expanded path name for **~cad.**
NETSITE	See *netsite(5C)* for a list.
MACHTYPE	See *machtype(5C)* for a list.

SYSTYPE See *systype(5C)* for a list.

SEE ALSO

intro(5C) netsite(5C) machtype(5C) systype(5C) csh(1) make(1)

AUTHOR

Jim Kleckner

BUGS

The problem of the flow of updates from one development machine to another is solved by creating separate directories in ~cad/src and ~cad/doc. The problem of manual pages is not solved.

Should we bite the bullet and make separate hierarchies in ~cad/man for each machine?

NAME

lyra – Performs hierarchical layout rule check on caesar design.

SYNOPSIS

lyra [**-vz**] [**–o** output] [**–p** path] [**–r** ruleset] [**–t** technology] rootCaesarFile,
or
lyra –e [**–t** technology] [**–r** ruleset]

DESCRIPTION

Lyra has two modes of operation: it can be invoked directly to perform a batch hierarchical check of a caesar design, or from the *Caesar* (or *Kic*) layout editor to interactively check a portion of the design currently being edited.

In batch mode, a hierarchical check of the caesar design rooted at *rootCaesarFile* is done. A log, including a summary of errors is written to stdout, and a *lyra file* "name.ly" is created for every cell "name.ca" in which design rule violations are detected. The lyra files flag each design rule violation with a bright splotch of paint on the error layer, and a caesar label identifying the type of violation. The lyra file for a cell "name.ca" contains the original caesar file as a subcell, thus the caesar subedit command can be used to conveniently fix design rule violations reported by *Lyra*. Obsolete lyra files are removed by *Lyra* when a cell checks on the current run.

Lyra's violation messages have the form:

!< LayersOrConstructs >_< Type >.

Note that all violation messages begin with an exclamation mark ("!"). *LayersOrConstructs* gives the single character abbreviations for the layers involved in the violation. Circuit constructs such as transistors and buried contacts may also be indicated by short abbreviations (e.g. **tr** for transistor; **Bc** for buried contact). *Type* is given by one or two characters indicating the type of error as follows:

s = minimum spacing violation,
w = minimum width violation,
pe = parallel edge spacing violation,
x = insufficient extension or enclosure,
p = polarity, e.g. Dif. doping doesn't match well in CMOS,
f = malformed circuit construct.

For example, a spacing violation between **Polysilicon** and Diffusion would look like this:

!P/D_s.

Note that *Parallel Edge* checks are less restrictive than the corresponding *Width* and *Spacing* checks would be, since they ignore diagonal interactions.

The following *rulesets* are currently supported at Berkeley:

nmosBERK
Berkeley nMOS rules. Modified Mead & Conway rules. Buried contacts are supported; Butting Contacts are disallowed. The Lyon Implant rules are used.

cmos-pwJPL
CMOS rules (p well). An extension of the Mead and Conway nMOS rules to CMOS, worked out by Carlo Sequin in conjunction with JPL.

nmosMC

Mead & Conway nMOS rules as described in "Introduction to VLSI Systems" by Mead and Conway. Butting Contacts are allowed; buried contacts are not allowed.

If the **-r** option is not given, *Lyra* chooses a *ruleset* based on the *technology* specified in the *rootCaesarFile*. The correspondence between caesar *technologies* and default *rulesets* is specified in *~cad/lib/lyra/DEFAULTS*. If *Lyra* does not recognize the *technology* of the *rootCaesarFile*, it uses the default *ruleset* for **nmos**.

In *editor mode* standard input and standard output are used to communicate with the layout editor, no log is written to stdout!, and violations are flagged directly in the edit cell. The caesar *technology* or *ruleset*, if different from **nmos**, must be specified explicitly on the command line, since *Lyra* does not have direct access to the caesar database. Note that interactive checks are nonhierarchical and slow, thus it is a good idea to use this mode only to check small pieces of a design; complete designs are best checked in batch mode.

The options described below may be specified in a .cadrc file or as command line options. *Lyra* reads options from *~cad/.cadrc*, *~/.cadrc* and the command line, in that order. If an option is specified in more than one place, the later setting takes precedence. Capitalizing an option on the command line, or giving the keyword **unset<option>** in .cadrc causes the option to be reset to its default value (e.g. "lyra -R", resets any previous ruleset specification, forcing the default to be used).

-e **(edit mode)** Used by *Caesar* and *Kic*. In this mode *Lyra* reads rectangles etc. from standard input and reports violations on standard output.

-o <outputDir>
(output directory) Gives directory for lyra (-.ly) files. Defaults to current directory.

-p <path>
(search path for caesar files) Path gives a colon (":") separated sequence of directorys to be searched in order for caesar files. The default search path is just the current directory. As in caesar *~cad/lib/caesar* is searched as a last resort.

-r <ruleset>
(design rule set) Gives *ruleset* to use. *Rulesets* are stored in *~cad/lib/lyra*. A user can supply his own *ruleset* by giving the full pathname on the **-r** option (see rulec). If the **-r** option is not specified, *Lyra* determines which *ruleset* to use from the *technology* specified in the *rootCaesarFile* for the design.

-t <technology>
(caesar technology) Used to specify caesar *technology* in *editor mode*, or to override the *technology* given in the *rootCaesarFile*. *Lyra* uses the caesar *technology* to choose a default *ruleset*.

-v **(verbose mode)** Causes more detailed log information to be written to *stdout*. This option is primarily for debugging.

-s **(restart)** If Lyra dies abnormally, it leaves a **RESTART** file in the output directory which gives the cells which were completely checked. Lyra can then be restarted with the **-s** option, to resume checking with the first (sub)cell not already checked. Note that the **restart** option should only be used if the caesar database for the project has not been changed since the time the original *Lyra* run was started.

FILES

~cad/bin/lyra -- executable lyra.
~cad/lib/lyra -- rulesets (in symbolic and executable form).
~cad/lib/lyra/DEFAULTS -- gives default rulesets for caesar technologies.

SEE ALSO

Rulec (CAD)
Caesar (CAD)

KIC (CAD)
Cif2ca (CAD)
Cifplot (CAD)

AUTHOR

Michael Arnold.

NAME

mcp – Manhatten CIF Plotter

SYNOPSIS

mcp [*options*] file.cif

DESCRIPTION

Mcp reads a CIF file and produces a plot.

Command Line Options

The following is a list of command line options recognized by *mcp*. Several options may appear on the command line. Each option must be preceded by a dash (–).

-w *xmin xmax ymin ymax*
(**window**) set the window of the plot. By default the plot is set to the entire circuit.

-s *scale*
(**scale**) set the scale of the plot. *Scale* is in terms of magnification. Thus **-s 250** will make the plot 250 times actual size. By default the scale is set so that the window fill the entire page. If the scale is set so that the entire window will not fit on one page, then several strips are output.

-n (**no action**) *mcp* works by calling other programs to do the actual plotting. The **-n** flag causes *mcp* to print out the calls to these programs rather than execute them. By redirecting standard output with this options set, the command file can be edited.

-V (**Varian**) plot on the varian plotter.

-W (**Wide**) plot on the wide plotter.

-T (**Trilog**) plot on the trilog plotter.

.cadrc options

The following is a list of options that can be set in your '.cadrc' file. Many of these options will be set in the global '.cadrc' file, so you will not have to repeat them in your file.

mcp *options*
set up the default command line options.

tmpdir *dir*
set the temporary directory to *dir*. (The default is '/usr/tmp/'.)

device *name xmax ymax resol prog*
define a new device. *Name* is the command line option used to refer to the device. *xmax* and *ymax* are the width and height of the plotter. (Setting *ymax* to -1 indicates unlimited length.) *resol* is the resolution of the device in dots per inch.

patfile *file*
set *file* to be the stipple pattern file. *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 is made up of 4 bytes, where each byte represents in order the colors cyan, magenta, yellow, and black. Integers may be in decimal, octal, or hex. Hex numbers start with '0x', octal numbers start with '0'.

FILES

~cad/.cadrc
~/.cadrc
~cad/bin/clipcif
~cad/bin/cifwindow
~cad/bin/vdump
~cad/lib/mcp1
~cad/lib/vfill

~cad/lib/tfill
~cad/lib/log

ALSO SEE

cifplot(cad1), cadrc(cad5)

AUTHOR

Dan Fitzpatrick

BUGS

Works for manhattan simple-CIF only. Text is not plotted.

NAME

mextra – Manhattan Circuit Extractor

SYNOPSIS

mextra [**-g**] [**-u** *scale*] [**-o**] basename

DESCRIPTION

Mextra reads an intergrated circuit layout description in Caltech Intermediate Form (CIF) and creates a circuit description. From this circuit description various electical checks can be done on your circuit. The circuit description is directly compatible with *mossim*, *moserc*, and *powest*.

Names

Mextra uses the CIF label construct to implement node names and attributes. The form of the CIF label command is as follows:

94 *name x y* [*layer*];

This command attaches the label to the mask geometry on the specified layer crossing the point (x, y). If no layer is present then any geometry crossing the point is given the label.

Mextra interprets these labels as node names. These names are used to describe the extracted circuit. When no name is given to a node, a number is assigned to the node. A label may contain any ASCII character except space, tab, newline, double quote, comma, semi-colon, and parenthesis. To avoid conflict with extractor generated names, names should not be numbers or end in '#*n*' where *n* is a number.

A problem arises when two nodes are given the same name although they are not connected electrically. Sometimes we want these nodes to have the same names, other times we don't. This frequently happens when a name is specified in a cell which is repeated many times. For instance, if we define a shift register cell with the input marked 'SR.in' then when we create an 8 bit shift register we could have 8 nodes names 'SR.in'. If this happens it would appear as though all 8 of the shift register cells were shorted together. To resolve this the extractor recognizes three different types of names: *local*, *global*, and *unspecified*. Any time a local name appears on more than one node it is appended with a unique suffix of the form '#*n*' where *n* is a number. The numbers are assigned in scanline order and starting at 0. In the shift register example, the names would be 'SR.in#0' through 'SR.in#7'. Global names do not have suffixes appended to them. Thus unconnected nodes with global names will appear connected after extraction. (The **-g** causes the extractor to append unique suffixes to unconnected nodes with the same global name.) Names are made local by ending them with a sharp sign, '#'. Names are global if they end with an exclamation mark, '!'. These terminating characters are not considered part of the name, however. Names which do not end with these characters are considered unspecified. Unspecified names are treated similar to locals. Multiple occurrences are appended with unique suffixes. By convention, unspecified names signify the designer's intention that this name is a local name, but is connected to only one node. It is illegal to have a name that is declared two different types. The extractor will complain if this is so and make the name local.

It makes no difference to the extractor if the same name is attached to the same node several times. However, if more than one name is given to a node then the extractor must choose which name it will use. Whenever two names are given to the same node the extractor will assign the name with the highest type priority, global being the highest, unspecified next, local lowest. If the names are the same type then the extractor takes the shortest name. At the end of the log file the extractor lists nodes with more than one name attached. These lines start with an equal sign and are readable by *mossim* so that it will understand these aliases.

Attributes

In addtion to naming nodes *mextra* allows you to attach attributes to nodes. There are two types of attributes, *node attributes*, and*transistor attributes*. A node attribute is attached to a node using the CIF 94 construct, just the same way as a node name. The node attribute must end in an at-sign, '@'. More than one attribute may be attached to a node. *Mextra* does not interpret these attributes other than to eliminate duplicates. For each attribute attached to a node there appears a line in the **.sim** file in the following form:

A *node attribute*

Node is the node name, and *attribute* is the attribute attached to that node with the at-sign removed.

Transistor attributes can be attached to the gate, source, or drain of a transistor. Transistor attributes are must end in a dollar sign, '**$**'. To attach an attribute to a transistor gate the label must be placed inside the transistor gate region. To attach an attribute to a source or drain of a transistor the label must be placed on the source or drain edge of a transistor. Transistor attributes are recorded in the transistor record in the **.sim** file. A transistor description has the following form:

type gate source drain l w x y **g=***attributes* **s=***attributes* **d=***attributes*

Attributes is a comma-separated list of attributes. If no attribute is present for the gate, source, or drain the **g=**, **s=**, or **d=** fields may be omitted.

Capacitance

The .sim file also has information about capacitance in the circuit. The lines containing capacitance information are of the form:

C *node1 node2 cap-value*

cap-value is the capacitance betweens the nodes in femto-farads. Capacitance values below a certain threshold are not reported. The default threshold is 50 femto-farads.

The extractor reports capacitance from two sources. Capacitance between node and substrate, and capacitance caused by poly overlapping diffusion but not forming a transistor. Transistor capacitances are not included since most of the tools that work on the .sim file calculate them from the width and length information.

The capacitance for each layer is calculated separately. The reported node capacitance is the total of the layer capacitances of the node. The layer capacitance is calculated by taking the area of a node on that layer and multiplying it by a constant. This is added to the product of the perimeter and a constant. The default constants are given below. Area constants are in femto-farads per square micron. Perimeter constants are femto-farads per micron.

layer	area	perimeter
metal	0.03	0.0
poly	0.05	0.0
diff	0.1	0.1
poly/diff	0.4	0.0

Poly/diffusion capacitance is calculated similar to layer capacitance. The area is multiplied by constant and this is added to the perimeter multiplied by a constant. Poly/diffusion capacitance is not threshold, however.

The **-o** option supresses the calculation of capacitance, and instead, gives for each node in the circuit the area and perimeter of that node on the diffusion, poly, and metal layers. The lines containing this information look like this:

N *node diff-area diff-perim poly-area poly-perim metal-area metal-perim*

Node is the node name. *Diff-area* through *metal-perim* are the area and perimeter of the diffusion, poly, and metal layers in user defined units. (In addtion the **-o** option causes transistors with only one terminal to be recorded in the **.sim** file as a transistor with source connected to drain.)

Setting Options

By default, *mextra* reports locations in CIF units. A more convenient form of units may be specified either in the '.cadrc' file or on the command line. The form of the command line option is:

> **units** *scale*

To set units on the command line use the **-u** option.

The parameters used to compute node capacitance may be changed by including the following commands in your '.cadrc' file.

> **areatocap** *layer value*
> **perimtocap** *layer value*

value is atto-farads per square micron for area, and atto-farads per micron for perimeter. *layer* may be "poly", "diff", "metal", or "poly/diff". The threshold for reporting capacitance may set in the '.cadrc' file with the following line.

> **capthreshold** *value*

A negative value sets the threshold to infinity.

Mextra knows of two technologies, NMOS and CMOS p-well. NMOS is assumed by default. To set the technology to CMOS p-well, include the following line in your '.cadrc' file:

> **tech cmos-pw**

FILES

~cad/lib/extname
~cad/lib/log
~cad/.cadrc
~/.cadrc
/usr/tmp/#mext*

ALSO SEE

caesar(cad1), kic(cad1), powest(cad1), cadrc(cad5)

AUTHOR

Dan Fitzpatrick

BUGS

Accepts manhattan simple CIF only, use 'cifplot -O' to convert complicated CIF. The length/width ratio for unusually shaped transistors may be inacurate. Attributes for funny transistors are not recorded. Node attributes are ignored unless the **-o** switch is present.

NAME

mkpla – automatic PLA and FSM layout generator (version 3.1)

SYNOPSIS

mkpla [-options [-options] ...]

DESCRIPTION

The mkpla program takes a description of a logical function in truth table format such as is output by eqntott, plasort, or presto, and produces a CIF file specifying the mask layout geometry for a circuit which performs that logical function. CIF, the Caltech Intermediate Form for describing layout, is defined in Mead & Conway and clarified somewhat in Hon & Sequin. It has many options; in general, a lower case letter as a flag means "do" something and the corresponding upper case letter means "don't do" the same thing. The default may be either upper or lower case. Some flags take numerical arguments; in the descriptions that follow, -F *n* means the flag F followed immediately by an unsigned integer (spaces not allowed), e.g. "-W16". The options are:

-b	Use buried contacts instead of butting contacts. (This option is recognized but not yet implemented, so it has no effect at present. Implementation of buried contacts is awaiting clarification of the buried contact design rules. Special note late November 1981: a proposal for these rules has just been released and preliminary cell design is underway.)
-B	{default} Use butting contacts.
-c	{default} Minimize capacitance of metal lines in AND and OR planes. Use 3-lambda wide metal lines.
-C	Don't minimize capacitance. Use 4-lambda metal lines like in standard PLA described in [Hon1980].
-C n	Use metal lines that are *n* lambda wide, where 3 <= *n* <= 5. (Just implemented with this release, but not tested well.)
-d	{default} Include a call to the top level symbol, so that you can plot the PLA.
-D	Don't include a call to the top level symbol. If you try to plot the file it will appear empty since it will consist only of cell definitions.
-e n	{default *n* = 9} Name the cells with extension command *n*.
-E	Put the cell name in a comment following the DS command.
-f <filename> <space>	Write the PLA in the file specified. Note that since this option reads a *string* from the command line, it won't stop until it encounters a space (or tab or newline) character.
-F	{default} Use the default filename of 'mkplaout.cif'.
-g	{default} Compute the grounding requirements and add extra ground lines as needed.
-g n	Compute the grounding requirements and add extra ground lines as needed, but use at least one every *n* terms.
-G	Don't use any extra ground lines regardless of the result of the computations.
-G n	Don't compute the grounding requirements. Use an extra ground line every *n* terms.
-i	Clock the inputs by adding dynamic latches.
-i n	Clock the inputs by adding dynamic latches; there are *n* inputs regardless of what number the input file specifies.
-I	{default} Don't clock the inputs.
-I n	Don't clock the inputs; there are *n* inputs regardless of what number the input file specifies.
-j n	{default *n* = 1} Use *n* for the symbol number of the top level symbol. If *n* is missing or

	<= 0, uses the default.
-J	Use 1 for the symbol number of the top level symbol.
-k	{default *n* = 2} Use *n* for the symbol number of the lowest numbered subsymbol. If *n* is missing or <= 0, uses the default.
-K	Use 2 for the symbol number of the lowest numbered subsymbol.
-l n	Set lambda to *n* CIF units. If *n* is missing or 0, sets it to the default value of 200.
-L	{default} Set lambda to default value of 200 CIF units (= 2.0 microns).
-m	{default} Use only "manhattan" (i.e. 90 degree) features.
-M	Use non-manhattan features to slightly improve electrical properties. (All non-manhattan features have been removed from the program, so this option has no effect at present. It is included only for compatibility with previous versions of **mkpla** and **mkfsm** .)
-n	{default} Label the inputs, outputs, Vdd and ground with 94 commands.
-n n	Label the inputs, outputs, Vdd and ground with user extension command *n*.
-N	Don't label any nodes.
-o	Clock the outputs by adding dynamic latches.
-o n	Clock the outputs by adding dynamic latches; there are *n* outputs regardless of the information in the input file.
-O	{default} Don't clock the outputs.
-O n	Don't clock the outputs; there are *n* outputs regardless of the information in the input file.
-p	{default} Program the PLA from info on standard input.
-p n	Program the PLA from info on standard input. There are *n* product terms regardless of the information in the input file.
-P	Leave the PLA unprogrammed and "blank".
-P n	Leave the PLA unprogrammed and "blank". There are *n* product terms regardless of the information in the input file.
-s	Modify the labels from the -n option (if any) to specify NMOS layers.
-S	{default} Don't specify layers in labels.
-t	Put the outputs on the opposite ("trans") side from the inputs.
-T	{default} Put the outputs on the same ("cis") side as the inputs.
-v	{default} Be verbose; write lots of text to standard output. Also, write comments describing the cells in the output CIF file.
-V	Be terse; avoid all inessential messages to user. Don't put comments in CIF file.
-w	{default} Compute the required width of Vdd and ground lines; widen them if needed.
-w n	Compute the required width of Vdd and ground lines; use the greatest of the computed width, *n*, and 4.
-W	Use 4 lambda wide Vdd and ground lines without computing requirements.
-W n	Use *n* lambda wide Vdd and ground lines. If *n* is less than 4, uses 4.
-x	Extend all polysilicon lines through the full length of the AND and OR-planes.
-X	{default} Truncate all polysilicon lines to improve speed.
-y n	Make a finite state machine by feeding back *n* bits from the inputs to the outputs. Both inputs and outputs should be clocked for this to work right, i.e., you should also specify

the -i and -o options. Note that output 1 will be connected to input *i*, output 2 will be connected to input *i* - 1, etc.

-Y Clears any previous specification of FSM bits to 0.

-z Make a low-power PLA. Lengthen the pullups to reduce power consumption. (Only implemented for the pullup-pairs; has no effect at present on the pullups in the input and output drivers.)

-Z {default} Use the normal size pullups.

FILES

~cad/src/mkpla/*

The Pascal source code.

~cad/new/mkpla

Executable object file made by running **pi** on the above files. The program is so fast that the penalty for running it interpreted under **pi** and **px** instead of compiling it using **pc** is negligible, and worth paying to get the better error handling of **px** .

SEE ALSO

tpla (a newer pla generator)

ncifplot, smallcif, cifplot, cifprint

These are programs to plot CIF so you can look at your PLA or FSM.

eqntott

A program to convert logic equations to PLA-style truth tables.

presto

A heuristic truth-table minimization program.

blam, plaid

Some programs by Mark Hoffman that also lay out PLA-type structures, and perform topological minimization ("folding" and "splitting").

caesar, cif2ca

Caesar is an all-manhattan interactive color graphics editor, and cif2ca converts from CIF to caesar format.

kic, ciftokic

Kic is an interactive color graphics editor which allows arbitrary geometries, and ciftokic its input conversion program.

plasim

A program which will simulate a PLA from its truth table, allowing verification of functional correctness.

MPC79 Cell Library

The PLA structures that mkpla uses are derived from ones designed by Dick Lyon of Xerox PARC for the Dec. 1979 Multi-project Chips. These cells are described in detail in Hon & Sequin (see below).

Introduction to VLSI Systems

This book by Carver Mead and Lynn Conway contains an excellent description of an earlier, similar PLA structure (also due to Dick Lyon) in pages 80-84 and 102-108, and plates 7 and 8.

A Guide to LSI Implementation

This book by Bob Hon and Carlo Sequin describes the 8-lambda pitch PLA cells by Dick Lyon which are similar to what mkpla uses, and gives several examples of ways in which they can be used (pages 140-156).

Automatic Layout of Optimized PLA Structures

The master's thesis by Howard Landman (U.C.Berkeley 1982) describing **mkpla** and giving a user's guide to the Berkeley PLA Tools.

AUTHOR

Howard A. Landman

DIAGNOSTICS

Error messages are generated if unrecognized flags are found on the command line.

The program checks to make sure that all inputs and product terms are used (i.e., contribute to the function of the PLA), and that all product terms and outputs are set (i.e., can be affected by the inputs). If it discovers otherwise it issues a "WARNING:" or "ERROR:" message on standard output.

PORTABILITY NOTES

Earlier versions of the program (up to 2.6) have been successfully run under VAX/VMS with only minor changes. Here is a list of the known changes needed to do this.

(1) The main program should be renamed to "mkpla.pas" to follow the VMS naming convention for Pascal files.

(2) All the include statements, which under UNIX are of the form

```
#include "file.i";
```

should be changed to the VMS form

```
%include 'file.i';
```

(3) The Berkeley Pascal constructs

```
reset(filevar,filename);
```

and

```
rewrite(filevar,filename);
```

are not supported under VMS Pascal. All instances of the former should be replaced with pairs of statements of the form

```
open(filevar,filename,OLD);
reset(filevar);
```

and all instances of the latter should be replaced with pairs of statements of the form

```
open(filevar,filename,NEW);
rewrite(filevar);
```

(4) The procedures in file unixif.i, which contains all the UNIX interfaces for the program, need to be rewritten so that the options are read without using calls to argc and argv. Alternatively, one could write code for procedures of those names which mimicked their behavior under UNIX.

(5) Since VMS doesn't distinguish upper and lower case characters on the command line, the way options are set needs to be modified either to read the options from a file, or to not depend on case distinctions.

This version of the program (3.1) has also been ported to a DECSystem 20 running TOPS-20 by Patrice Frison of University of Santa Clara.

BUGS

Mkpla only supports butting contacts, use Tpla for other technologies.

If the '.f' option appears in the input then the PLA may be programmed incorrectly.

Mkpla generates complicated CIF, you may have to run the CIF through the '-O' option of cifplot before passing it to other tools.

Feedback terms create design rule violations, other design rule violations occur in the input and output drivers.

The program only can generate even numbers of product terms and outputs. If an odd number is specified, it creates an extra (blank) one. Also, if an extra pterm is created, it is the one closest to the inputs, even though making it the one farthest from the inputs would improve the speed of the PLA.

Clocking of outputs on trans PLAs doesn't work right yet so you can't use the -o and -t options together.

The -P ("don't program") option should automatically imply the -x ("extend poly lines") option. It doesn't, so that unprogrammed PLAs will be missing their poly lines.

When the -z ("low-power") option is specified, the AND- and OR-plane pullups are longer, so the program flashes could be modified to reduce the pulldown transistor size and hence lower the capacitances of all poly lines. They aren't.

Since pc and pi don't know about the standard error file, it is not possible to write error messages to it. This means that standard output has to be used for error messages, and thus the program cannot run as a filter.

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 *mkpla* or *tpla* thus:

```
peg infile | eqntott | mkpla -i -o -y n -foutfile
peg infile | eqntott | tpla -c -s Bcis -I -O -o outfile
```

Either of these command lines generates a PLA implementation of the finite state machine in the file *outfile.cif.* In the above command line for *mkpla*, *n* must be replaced by the integer number of state bits generated for the fsm by *peg*.

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 *mkpla* result will feed back *n* state bits from the PLA outputs to the PLA inputs; however, if *tpla* is used then the feedback lines 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*

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. The special state name *LOOP* specifies 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
mkpla(CAD1), tpla(CAD1), eqntott(CAD1)
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.

NAME

quilt - assemble tiles into a rectangular array

SYNOPSIS

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

DESCRIPTION

The user of Quilt first creates a Caesar 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). Quilt 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.

Quilt is a small program written with the Tpack system.

OPTIONS

-a produce Caesar format (this is the default)

-c produce CIF format

-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. A **.tp** suffix is added if no suffix was specified.

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

text_file

The name of quilt'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 tpack(CAD).

FILES

~cad/bin/quilt	– executable
~cad/src/quilt/*	– source
~cad/lib/quilt/q-*	– location of standard templates
~cad/examples/quilt/*	– quilt example

SEE ALSO
tpack(CAD)

AUTHOR
Robert N. Mayo

BUGS
This program inherits any bugs that may exist in tpack(CAD).

NAME

rulec – Compile design rules for Lyra

SYNOPSIS

rulec [–lo] rules

DESCRIPTION

Rulec is a shell script with the following processing steps:

i) The actual *Lyra* rule compiler is invoked to translate the symbolic rule description, *rules.r*, to lisp code, rules.l.

ii) The lisp compiler, *Liszt*, is invoked to compile *rules.l* to *rules.o*

iii) *rules.o* is loaded into *Lyra.proto* to generate an executable lisp *Lyra*, *rules.*

iv) The intermediate files *rules.l*, and *rules.o* are deleted.

The following options are supported:

–l **(load only)** No compilation is done. Previously compiled rules, *rules.o*, are loaded into *Lyra.proto* to generate an executable Lyra, *rules.* This option is useful mainly at Berkeley, where *Lyra.proto* changes frequently.

–o (save object) *Name.o is not removed. Enables* '**rulec -l** *rules*' in the future.

FILES

~cad/bin/rulec – rulec shell script.
~cad/lib/lyra/Rulec1 – lisp rule compiler
~cad/lib/lyra/Lyra.proto – Lyra sans compiled rules code.
~cad/lib/lyra/*.r – standard rulesets.
~cad/lib/lyra/DEFAULTS – gives default rulesets for Caesar technologies.

SEE ALSO

Lyra (CAD)
Liszt (1)

AUTHOR

Michael Arnold.

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.

ALSO SEE

mextra(CAD1), spice(1), cadrc(CAD5)

AUTHOR

Dan Fitzpatrick CMOS fixes by Neil Soiffer

BUGS

The only pre-defined technologies are 'nmos' and 'cmos-pw'. Only one definition file is allowed.

NAME

tpla – technology independent PLA generator

SYNOPSIS

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

DESCRIPTION

Tpla 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. **Tpla** does not handle split and folded PLAs.

Tpla is a program written with the Tpack 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.

Mcis Mead & Conway design rules. Butting contacts, nMOS, cis version (inputs and outputs on same side of the PLA). Clocked inputs and outputs are supported.

Mtrans
Mead & Conway design rules. Butting contacts, nMOS, trans version (inputs and outputs on opposite sides of the PLA). Clocked inputs and outputs are supported.

Tcis Just like **Bcis** except that it has protection frames and terminals added (a special mod for EECS at Berkeley).

Ttrans
Just like **Btrans** except that it has protection frames and terminals added.

It is easy to create a template for a new style of PLA, and tpla(CAD5) has information on how to do it. If you develop a particularly nice template and would like to share it, send it to "mayo@berkeley" or "ucbvax!mayo".

Tpla handles CIF symbol naming directives and input & output labels as described in pla(CAD5).

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 *num*
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 *num* 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 Caesar output) how the PLA was constructed from its basic components.

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

-a produce Caesar format (this is the default)

-c produce CIF format

-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 tpla to use the file **~cad/lib/tpla/p-*style*.tp** as its template).

-l *num* Set lambda to *num* centimicrons. (200 is the default)

-t The next argument specifies the template to use, this normally defaults to the standard library. A **.tp** suffix is added if no suffix was specified. 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).

other options
This program inherits several more options from Tpack(CAD).

FILES

~cad/bin/tpla	– executable
~cad/src/tpla/*	– source
~cad/lib/tpla/p*.tp	– standard templates for PLAs

SEE ALSO

eqntott(CAD), presto(CAD), plasort(CAD), pla(CAD5), tpla(CAD5), tpack(CAD), mkpla(CAD)

AUTHOR

Robert N. Mayo

BUGS

The defaults for 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 tpack(CAD).

NAME
ucfilt – unique call filter

SYNOPSIS
ucfilt file.cif

DESCRIPTION
Ucfilt reads in a CIF file and elimanates all but one call to any symbol. This is useful for gathering information about the regularity of a design.

AUTHOR
Dan Fitzpatrick

BUGS
Reads only a restricted subset of CIF. Does not preserve quoted CIF strings.

NAME

vdump – plot a raster dump file on the Varian or Versatec

SYNOPSIS

vdump [**-c** *ncopies*] [**-b** *banner*] [**-l** *loginname*] [**-r**] [**-W**] [**-**] [filenames]

DESCRIPTION

Vdump takes a raster dump file and sends it to be plotted on the Varain or Versatec plotter. More than one file may be specified on the command line. Plots normally go to the Varian plotter.

Dump files can be formatted or non-formatted. Non-formatted dump files are simply a stream of bytes to be dumped on the plotter. Formatted dump files contain a little more information, such as the number of bytes per line to be plotted, the resolution, and a banner which should be printed across the top of the plot.

The command line options are:

–c ncopies
: Plot multiple (*ncopies*) copies of files.

–b banner
: Put the quoted string *banner* at the top of the plot.

–l loginname
: Identify plot with login name *loginname*.

–r　Causes the dump file to be removed after it is plotted.

–W　Puts the plot on the wide plotter (Versatec).

–　If the file name is a dash (–) then the file is assumed to come from standard input.

FILES

```
/usr/lib/vad                    /* Varian daemon */
/usr/lib/vpd                    /* Versatec daemon */
/usr/spool/vad/dfa*         /* Varian daemon command files */
/usr/spool/vpd/dfa*         /* Versatec daemon command files */
/usr/tmp/#vdmp*                 /* temporary spooling files */
```

BUGS

File names of the form ../foo and its derviatives are not handled correctly.

NAME
vlsifont - create text logos for VLSI chips

SYNOPSIS
vlsifont [-k key] [-f font] word | quilt -s vlsifont

DESCRIPTION
The 'word' on the command line is rasterized into a matrix of characters suitable for input to quilt 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 quilt, the user should use the standard template **~cad/lib/quilt/vlsifont.tp** by specifying the **-s vlsifont** switch, or else supply his own (see quilt(CAD) for how to do this using Caesar). 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/quilt/q-vlsifont.tp	– standard template for quilt
/usr/lib/vfont/*	– standard place for fonts
~cad/bin/vlsifont	– executable
~cad/src/vlsifont/*	– source

SEE ALSO
quilt(CAD), caesar(CAD), 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

tpack – routines for generating semi-regular modules

DESCRIPTION

Tpack (tile packer) 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 Caesar, a tpack 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 tpack library provides routines to aid in assembling tiles into a finished module.

MAKING AN EXAMPLE MODULE

The first step in using tpack 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 Caesar format and, by convention, end with a **.tp** 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*.**tp**. The *style* part of the filename interacts with the **-s** option (see later part of this manual).

WRITING A TPACK PROGRAM

A tpack 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 tpack library.

The tpack program must first include the file **~cad/lib/tpack.h** which defines the interface to the tpack 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 tpack library be sure to include it with your compile or load command (e.g. **cc** *your_file* **~cad/lib/tpack.lib**).

ROUTINES

Initialization and Output Routines

TPinitialize(*argc, argv, base_name*)

The tpack 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 **.tp** suffix if no suffix was provided. 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 Caesar 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.

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().

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 tpack 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.

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 **RECTANGLEs** and **POINTs** are defined as:

```
typedef struct {
   int x_left, x_right, y_top, y_bot;
} RECTANGLE;

typedef struct {
   int x, y;
} POINT;
```

The variable **ORIGIN_POINTER** 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 Caesar 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 tpack)

-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. (see TPinitialize)

-l *num* Set lambda to *num* centimicrons. (200 is the default)

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*
Normally tpack merges rectangles to form maximal horizontal strips, just like Caesar(CAD). If the **-M** option is present tpack will only look back through the last *num* rectangles on each layer when doing merges. A small value for *num* will make tpack run faster, but not all possible merges will be found. The **-v** option gives information about the number of merges done.

-D *num1 num2*
The *Demo* or *Debug* option. This option will cause **tpack** 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/quilt** be examined. Look at both the template and the 'C' code. A more complex example is in **~cad/src/tpla.**

FILES

~cad/lib/tpack.h	(definition of the tpack interface)
~cad/lib/tpack.lib	(linkable tpack library)
~cad/src/quilt/*	(an example of a tpack program)
~cad/lib/caesar/*.tech	(technology description files)

ALSO SEE

Caesar(CAD)
'C' Manual
Quilt(CAD)
Tpla(CAD)
Robert N. Mayo and John K. Ousterhout, *Pictures with Parentheses: Combining Graphics and Procedures in a VLSI Layout Tool,* Proceedings of the 20th Design Automation Conference, June, 1983.

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

intro – introduction to the cad formats section of the manual

DESCRIPTION

This section of the manual contains the definitions of site names, intermediate file formats and other common definitions.

SEE ALSO

intro(CAD)

BUGS

NAME

Template format for Tpla(CAD1)

DESCRIPTION

Making a template for **tpla** consists of first drawing a sample PLA in the desired style, and then labeling *tiles* using the Caesar(CAD) graphics editor. A *tile* is a rectangular area of paint, and is defined by a named label outlining the area. **Tpla** assembles these tiles to form a finished PLA.

There are 11 groups of tiles in a **tpla** 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. 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 impossible to stretch along a rectangle!

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, left-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 **left-and** tiles. 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.

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
tpla(CAD1), tpack(CAD)

AUTHOR
Robert N. Mayo

Editing VLSI Circuits with Caesar

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!ouster

This user manual corresponds to Caesar VII.

1. Introduction

Caesar is an interactive system for creating and modifying VLSI circuit designs. It is based on the Mead and Conway style of design, and produces CIF descriptions (Caltech Intermediate Form) suitable for chip fabrication. Caesar is not an "intelligent" design system in the sense of understanding design rules, electrical properties, or even connectivity. It is just a geometry editor that allows you to paint pictures of VLSI circuits and to combine pictures hierarchically into larger designs.

Caesar runs under the 4.1 Berkeley Distribution of VAX Unix. It is a two screen system. One screen, called the *text display,* may be any standard CRT terminal capable of running the screen editor vi. Caesar is invoked from this terminal; commands are typed at its keyboard and a command menu and several statistics about the chip design are displayed on the text display. The second screen is called the *graphics display* and is used to display in color a piece of the circuit being designed. The graphics display can be any of a variety of color displays. Caesar currently supports AED512, AED767, Metheus Omega440, Chromatics 7900, or Vectrix displays connected to the VAX via a 9600 baud RS232 line. It also supports Ramtek 9400 displays with DMA interfaces. A graphics tablet must be attached to the color display (except the Chromatics, which has a joystick). The tablet must have a four-button cursor, which is referred to here as the *puck*.

2. Getting Started

The command line for Caesar has the form

caesar -g<graphics port> -t<tablet port> -p<path> -n -m<monitor type> -d<display type> <file>

All of the switches are optional and have reasonable defaults. The -g switch indicates the name of a device in the */dev* directory that should be used as the graphics display port. Thus, if the display is connected to the machine as */dev/ttyh0,* then type *caesar -gttyh0* (you may leave spaces between the "-g" and the "ttyh0" if you wish). If the -g switch isn't supplied, Caesar will look in internal tables to locate the nearest AED display to the terminal from which the command is issued (if none can be found, /dev/null will be used). See Section 24 for details on how Caesar picks a default display. The -t switch is used to give the name of a port to use for reading tablet data. If not specified, Caesar makes an educated guess as to which port to use. If the -p switch is present it specifies a path to be used by Caesar for file lookups (see Section 15 for a discussion of paths). If the -p switch isn't specified, then a default path of "." is used. The -n switch causes Caesar to run in non-interactive mode; see Section 19 for more details on this. The -d and -m switches are used to select the type of display you are using (AED, Jupiter, Metheus, etc.), and the type of color monitor attached to the display; see Section 24 for details. If <file> is specified, it is the name of a cell to be edited. If <file> isn't specified, Caesar will assume you want to build a new cell from scratch.

When it starts up, Caesar attempts to read a command file from *.caesar* in the home directory. Whether or not it finds a file there, it next tries to read a command file from *.caesar* in the current directory. Command line options override specifications in the startup file. See Section 17 for detailed information on command files.

In order to use a tablet with the color display, you may have to go to extra effort so that Caesar can read characters from the display's port. If the display is hardwired to a machine, Caesar will work fine if you just arrange for there not to be a login process for its port. If there is a login process for the port, you'll have log in a job called "sleeper" on the color display. No password is required. Sleeper will run a special program to reset the terminal so Caesar can read from it. The sleeper program is extraordinarily durable (it has to be because of bugs in the AED display). The only way to kill it is by typing two control-backslash characters within ten seconds of each other. On the AED keyboard control-backslash is control-shift-L. On most Berkeley systems, you can also kill sleeper jobs from other terminals by typing "killsleeper <pid>" where <pid> is the process id of the sleeper process. On some systems, you have to log yourself in and then run sleeper as a shell command.

Although it shouldn't happen, the display will occasionally get itself into a mode where Caesar cannot run. When this happens, reset the color display. On AED's, this is done by hitting the "reset" key twice (it's a black key at the top left), after which you should hear a beep and see the screen go blank. To be absolutely safe, you may want to kill the sleeper program and log it in again. After resetting the display, type ":reset" to Caesar: this should fix things up again.

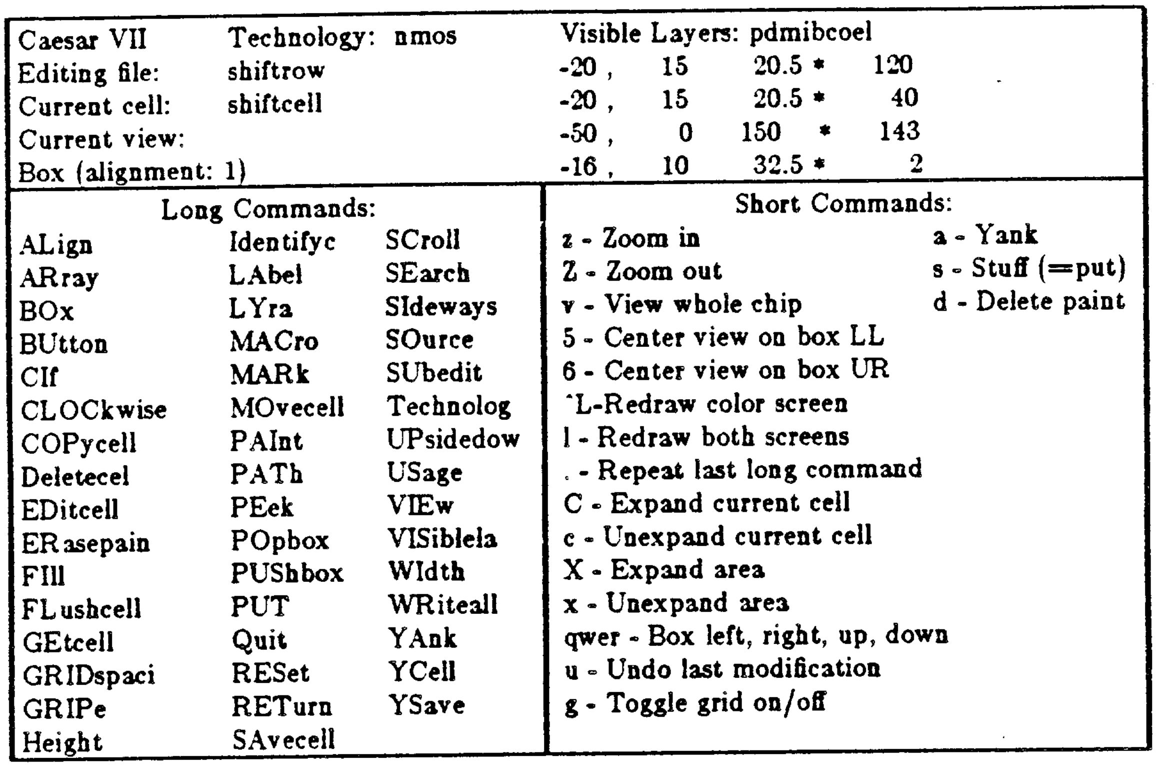
The rest of this manual is most easily understood if you can play with Caesar as you read. From now on, I assume that you are sitting in front of a terminal, and have run Caesar with the command "caesar shiftcell" (shiftcell is a cell in the Caesar library).

3. The Command Interface and Text Display

After Caesar starts running, the text screen will fill with characters (see Figure 1). The text display is divided into three sections. The lower right portion of the display is a list of all the *short commands* along with brief descriptions of their functions. Each short command is invoked by typing a single letter on the keyboard. Try typing the **g** short command (a grid should turn on and off).

The lower left portion of the text display gives the names of most of the *long commands* (the color map commands described in Section 23 were omitted for lack of space). A long command is invoked by typing a colon (":") followed by a line of text, followed by a return. The line of text contains the name of a command and any parameters that are needed by the command. To invoke a long command, you need only type enough characters to distinguish it from the other long commands. In the listing of long commands on the text display, the minimum set of characters that must be typed to invoke each command is shown in UPPER CASE. Try typing the commands **:align**, **:al**, **:a** (which is ambiguous), and **:badcommand** (which is not a command at all). The long commands appear on the bottom line of the text display, and you may edit these lines as you type them by using the standard Unix editing characters such as kill and backspace.

Each long command may actually contain several long commands separated by semi-colons.



```
Caesar VII          Technology:  nmos    Visible Layers: pdmibcoel
Editing file:       shiftrow             -20 ,   15     20.5 *    120
Current cell:       shiftcell            -20 ,   15     20.5 *     40
Current view:                            -50 ,    0    150   *    143
Box (alignment: 1)                       -16 ,   10     32.5 *      2

            Long Commands:                          Short Commands:
ALign       Identifyc    SCroll      z - Zoom in                      a - Yank
ARray       LAbel        SEarch      Z - Zoom out                     s - Stuff (=put)
BOx         LYra         SIdeways    v - View whole chip              d - Delete paint
BUtton      MACro        SOurce      5 - Center view on box LL
CIf         MARk         SUbedit     6 - Center view on box UR
CLOCkwise   MOvecell     Technolog   ^L-Redraw color screen
COPycell    PAInt        UPsidedow   l - Redraw both screens
Deletecel   PATh         USage       . - Repeat last long command
EDitcell    PEek         VIEw        C - Expand current cell
ERasepain   POpbox       VISiblela   c - Unexpand current cell
FIll        PUShbox      WIdth       X - Expand area
FLushcell   PUT          WRiteall    x - Unexpand area
GEtcell     Quit         YAnk        qwer - Box left, right, up, down
GRIDspaci   RESet        YCell       u - Undo last modification
GRIPe       RETurn       YSave       g - Toggle grid on/off
Height      SAvecell
```

Figure 1. A sample view of the Caesar text display.

This feature is particularly useful in writing macros (see Section 16). To include a semi-colon as part of a long command, rather than as a separator, type "\;" instead of ";". If a long command consists of a single quote followed by another character, the character is used as a short command. For example, the long command **:'a** is equivalent to the short command **'a**. This feature is useful for macros and command files.

The top portion of the text screen contains several statistics about the cell being edited. These will be explained in later sections.

All error messages are typed on the bottom line of the text screen. If several errors occur simultaneously, a mechanism like that of the *more* program is used to prevent one message from getting overwritten by subsequent ones. The first message is printed, then "--More--" appears at the end of the bottom line of the text screen. After you have read the first message, type a space character to see the next message. To see how this works, type the long command **:get xyzabc**.

Whenever Caesar makes any modifications to the cell database, it saves enough information to undo the effects of the most recent modification. If you discover that you have changed something you didn't really want to change, type the short command **u** to undo it. Undo applies to the last command that changed the database, including itself.

4. The Box and Crosshair

When you invoked Caesar, a picture of an NMOS shift register cell should have appeared in the middle of the color display, along with a white box and a blinking crosshair. If you move the puck around on the tablet, the crosshair will move around on the screen. The crosshair and the box are used as *tools* to invoke Caesar commands; they are not part of the circuit. Most of Caesar's commands operate in some way or other on the area selected by the box. The crosshair is used to position the box and to select mask layers and subcells.

The current position and size of the box are displayed in the upper portion of the text screen. The four numbers on the right side of the line labeled "Box" are, from left to right, the x-coordinate and y-coordinate of the lower-left corner of the box, and the x-size and y-size of the box. The units used in Caesar correspond to the lambda units of Mead and Conway. The precision of Caesar units is .5 lambda, which is in keeping with the smallest features of the Mead and Conway design rules.

Two of the buttons on the puck are used to position the box. When the left (white) button on the puck is depressed, the whole box is moved so that its lower-left, or fixed, corner coincides with the location of the crosshair. The right (green) puck button positions the upper-right, or variable, corner of the box without changing the lower-left corner.

The box can have zero (or even negative) size. When it has zero size it appears as a cross rather than as a rectangle.

When positioning the box with the crosshair, the crosshair position is rounded off to the nearest lambda unit. This alignment factor is displayed on the "Box" line of the text display. To change it, type the long command

:align <size>

which will set the alignment factor to <size> units. <size> is rounded off to the nearest power of two, and cannot be less than .5. As part of the action of **:align**, the box's coordinates are re-aligned to the units specified. <size> defaults to one lambda.

There are a few additional long commands that can be used to position the box. The command

:box <keyword> <amount>

will adjust the size and/or position of the box by <amount> units, according to <keyword>. If <keyword> is "up", "down", "left", or "right", then the box is moved in the specified direction. If the keyword is "xbot", "xtop", "ybot", or "ytop", then the size of the box is changed by adding <amount> to its lower or upper x- or y-coordinate. The amount may be negative. As usual, unique abbreviations for the keywords are acceptable. The short commands **q**, **w**, **e**, and **r** are equivalent, respectively, to **:box left 1**, **:box right 1**, **:box up 1**, and **:box down 1**.

The long commands

:height <size>
:width <size>

may be used to set the box's height and width to specific sizes. The upper-right (variable) corner of the box is moved so that the x- or y-dimesion is set to <size> units. If <size> is preceded by a "+" character then the box's size is changed by moving the lower-left (fixed) corner rather than the upper-right one. <size> defaults to 2 units.

5. Painting Commands

Caesar's mechanism for creating mask designs is much like painting. The two basic operations are to paint one or more mask layers over the area of the box, and to erase one or more mask layers from the area of the box. You should think of the mask information as paint: it has no structure other than its color and shape (for example, it doesn't make sense to think about mask information as objects such as rectangles or polygons; it is just a structureless blob).

There are several ways to paint and erase. The simplest way is to use the bottom (blue) button on the puck. First, use the left and right buttons to position the box over the area you wish to paint. Then move the crosshair over an existing piece of the design and press the bottom button. The area under the box will be painted in whatever mask layers are present underneath the crosshair. If there is no paint underneath the crosshair, then the area of the box is erased. Try painting a diffusion (green) rectangle, then erase a small hole in the middle of it. Remember that the painting commands, as well as any other commands that change the database, can be undone with the **u** short command.

Painting can also be invoked from the keyboard. The long command

:paint <layers>

will paint the area of the box with the layers given by <layers>. The parameter <layers> is a string of one or more single-letter mask layer abbreviations (see Table 1). The legal mask layers depend on what technology you are using. Only mask layers are valid for the **:paint** command: the usage of the other layers will be explained in later sections.

Besides using the blue puck button, there are two additional ways to erase paint. The long command

:erasepaint <layers>

will erase <layers> from the area of the box. <layers> defaults to "*l". Alternatively, you can use the short command **d**. This command will check the area underneath the crosshair to determine which layers are visible at that point, and will delete paint in those layers from the area of the box. If there are no layers visible underneath the crosshair, then the **d** command will erase the layers "*l".

Mask Layers for Technology "nmos"	
p or **r** -	Polysilicon layer (red).
d or **g** -	Diffusion layer (green).
m -	Metal layer (blue).
i or **y** -	Implant layer (yellow).
b -	Buried contact layer (brown).
c -	Contact cut layer (black cross).
o -	Overglass hole layer (grey).
e -	Error layer: used by design rule checkers and other programs.
***** -	All mask layers.
Mask Layers for Technology "cmos-pw"	
p or **r** -	Polysilicon layer (red).
d or **g** -	Diffusion layer (green).
m or **b** -	Metal layer (blue).
P or **y** -	P+ implant layer (yellow).
w -	P-well layer (brown).
c -	Contact cut layer (black cross).
o -	Overglass hole layer (grey).
e -	Error layer: used by design rule checkers and other programs.
***** -	All mask layers.
Layers Available in all Technologies	
l -	Label layer.
S -	Subcell layer.
X -	Box layer.
G -	Grid layer.
B -	Background layer.

Table 1. The single-letter layer mnemonics.

When specifying layers for long commands such as **:paint** and **:erasepaint**, the characters '+' and '-' may appear in the string as a convenience in typing. The '-' character causes subsequent layers to be omitted from the group rather than added, and '+ ' cancels the effect of an earlier '-'. Thus the layer specification '*-p' is synonymous with 'dmibcoe'. If '+ ' or '-' is the first character of the string, then "*l" are automatically included and subsequent letters add to or subtract from these layers. For example, **:erase -m** will erase labels and all mask layers except metal.

Caesar maintains a special collection of paint called the *yank buffer* that is used for shuffling around portions of the cell being designed. To enter information into the yank buffer, type the long command

:yank <layers>

This command will treat the box like a cookie cutter and will make an imprint of all the mask and label information underneath the box. The information is saved in the yank buffer, while leaving the original circuit unmodified. If <layers> is specified, then only those layers are yanked. The mask layers, as well as 'l' and 'S', are valid for **:yank** (later sections describe how to yank labels and subcells). The long command

:put <layers>

causes all the information in the yank buffer to be added back into the cell, such that the lower-left corner of the information is coincident with the lower-left corner of the box. After **:put** the yank buffer is still intact and may be **:put** again and again. <layers> has the same format as in the **:erasepaint** command. Only the layers selected by <layers> are put; <layers> defaults to "*lS". The yank buffer is loaded automatically as part of every **:erasepaint** command. To move a rectangular piece of the picture just **:erase** it, move the box over to the new location, and **:put** it back again.

There are also short commands to perform the same functions as **:yank** and **:put**. The short command **a** is equivalent to **:yank** and **s** is equivalent to **:put** (think of "s" as an abbreviation for the verb "stuff"). For each of these two commands, the crosshair selects the layers to be yanked or put. If there is no visible paint underneath the crosshair, then layers "*l" are affected in **a** and "*lS" are affected in **s**. If there is paint visible underneath the crosshair, then only the mask layers visible underneath the crosshair are used in the command.

The information in the yank buffer can be flipped and rotated. To flip the contents upside down (i.e. mirror about a horizontal line) use the long command

:upsidedown y

The **y** indicates that the yank buffer is to be flipped, rather than a cell. The long command

:sideways y

will flip the yank buffer contents sideways (i.e. about a vertical line), and the command

:clockwise <degrees> y

will rotate the yank buffer contents by <degrees>, which must be a multiple of 90. If <degrees> is omitted then it defaults to 90.

The long command

:fill <direction> <layers>

makes it relatively easy to stretch cells or extend busses. The <direction> parameter is one of the keywords "up", "down", "left", or "right" (unique abbreviations such as "u" or "l" are acceptable). <layers> has the same format as in **:erasepaint**; any of the mask layers are valid. If the layers are omitted then all mask layers are used. This command finds all paint crossing one edge of the box and extends that paint to the other edge of the box. For example, **:fill up** will extend all paint crossing the bottom of the box so that the paint reaches to the top of the box. Its effect is just as if the colors underneath the bottom edge of the box were a paintbrush; the brush is dragged up to the top of the box leaving a trail of paint behind it. Try this command on pieces of the shift register cell. To stretch the cell in the middle, delete one half of it (which puts the deleted paint into the yank buffer), then **:put** it back, leaving a gap between the two halves of the cell; use **:fill** to fill in the gap.

6. Viewing Commands

This section describes Caesar commands that change what is displayed on the graphics screen. The commands in this section don't have any effect on the actual circuit being designed. The information visible on the graphics display is called the *current view*. Its location and size are given in the upper portion of the text display in units in the same manner as the box location and size.

The **z** short command is used to zoom in on a small piece of the circuit. To use this command, first position the box over the area you want to fill the screen. After typing **z**, the view will be magnified so that the area under the box just barely fits on the screen. The short command **Z** does the opposite of **z**: it demagnifies the view such that what used to fill the screen just fits in the screen area given by the box. The **v** short command changes the view so that the whole chip just barely fits on the screen.

To shift the current view without changing its scale factor, use the long command

:scroll <direction> <amount> <units>

In this command, <direction> is one of "left", "right", "up", or "down," <amount> is a number, and <units> is one of "lambda" or "screens" (abbrevations are ok). The **:scroll** command shifts the view in the indicated direction by the indicated amount. The <units> parameter defaults to screens, and if both <amount> and <units> are defaulted, the default is 0.5 screen.

There are two additional short commands for shifting the current view. If **5** is typed, the view shifts so that the lower-left corner of the box is in the center of the screen. If **6** is typed, the view shifts so that the upper-right corner of the box is in the center of the screen. In both cases, the resulting view has the same scale as the initial view.

(The **:view** long command also changes the view; it is discussed in Section 13.)

It is possible to prevent some of the mask layers from being displayed, thereby making it easier to see the remaining layers. The long command

:visiblelayers <layers>

causes only <layers> to be displayed on the graphics screen. The **:visiblelayers** command makes it easier to verify the alignments between a few layers by eliminating the extraneous layers from view. <layers> has the same format as in the **:erase** command. For example, **:vis -p** will remove polysilicon from the set of layers that is displayed and won't affect any of the other layers. The visible layers are listed at the top of the text screen. Although it is possible to modify layers that aren't visible, Caesar will always issue a warning if there is a chance that invisible things may have been changed; **u** may be used to undo these effects if they weren't wanted.

7. Labels

A label consists of a rectangle and a piece of text. Caesar treats labels as comments but outputs them in CIF files so that other programs, such as circuit extractors, can use them. To place a label, type the long command

:label <text> <position>

A rectangle will be displayed on the graphics screen with the size and location of the box, and <text> will be displayed near the rectangle (if the rectangle has zero height or width then a line will appear, and if both dimensions are zero then a small cross will appear). <position> determines where the text is to be displayed: it must be one of the words "center", "right", "bottom", "left", or "top". If not specified, <position> defaults to "top".

For most purposes labels are considered just like a mask layer, with layer abbreviation l. A cell's labels are displayed only if the cell is expanded. Labels are made visible and invisible using the **:visiblelayers** command. Caesar will automatically turn off label visibility when the picture gets so large that labels are likely to clutter things up, but this decision can be overridden using **:visiblelayers**. The **:erase**, **:yank**, and **:put** commands can be used on labels just like any of the mask layers. The only difference between labels and paint is that if any of a label is affected, then the whole label is affected: it is not possible to erase or yank half of a label. When yanking or erasing, labels are ignored if they completely contain the box; to be yanked or erased, part of the rectangle of a label must be touching or contained in the box.

8. Grid Commands

The *g* command turns a grid on and off in toggle fashion. By default the grid is spaced on one unit centers. If the default grid spacing does not suit your fancy, Caesar allows you to use any spacing you wish. When you issue the command

:gridspacing

Caesar will recompute the grid so that the grid lines have the same x and y spacings as the x and y dimensions of the box and the box falls exactly on grid lines. This new grid spacing will be remembered across **g** commands until another **:gridspacing** command is typed. Note that the grid spacing and box alignment are independent.

9. Basic Cell Commands

The painting facilities described above allow you to paint pictures (cells) on the various mask layers. The cell commands described in this section allow you to save designs on disk and retrieve them later for further edits. These commands also permit to you to compose cells hierarchically into larger systems.

A cell is just a piece of the design that can be stored and retrieved by name. A separate disk file is used to hold the contents of each cell. At any given time in Caesar you are editing one cell; it is called the *edit cell*. The name (if any) and bounding box for the edit cell are displayed in the upper portion of the text screen. The bounding box is specified in terms of the x- and y-coordinates of its lower left corner and its x- and y-dimensions, in the same way as the box and current view.

To save the cell being edited, type the long command

:savecell <name>

This will change the name of the edit cell to <name> and write it out on disk in a file named *<name>.ca*. If <name> isn't specified, then the cell will be written to the file from which it was originally read. **NOTE: Caesar does not have any auto-save or checkpoint facilities. It is prudent to save cells periodically during long edits to safeguard against system crashes.**

To edit a different cell without restarting Caesar, the command

:editcell <name>

should be typed. This command destroys all the information related to the current cell and reinitializes the system to edit cell <name>. It will expect to find a file named <name>.ca containing the description of the cell. If the current cell has been modified since the last time it was written, Caesar will warn you and ask you if you wish to continue anyway. If you type 'yes' then the changed version of the cell will be lost. If you type 'no' or carriage return, then Caesar will give you a chance to save anything that has changed (see the **:writeall** command in Section 12 for details).

To include an existing cell as a subcell of a new cell, edit the new cell and type the command

:getcell <name>

Caesar will look on disk for a file named *<name>.ca* and will make that file a subcell of the edit cell. The paint of the subcell will appear on the graphics screen, and the subcell will be positioned such that the lower-left corner of its bounding box coincides with the lower-left corner of the box. The **:getcell** command causes the cell which is gotten to become the *current cell.* Its name and bounding box will then appear in the upper portion of the text screen. Most of the following commands operate on the current cell.

When a cell contains subcells, the subcells may appear in either of two ways. Most often, subcells will appear in *bounding box,* or *unexpanded,* form, in which case the subcell is displayed as a dark rectangle just large enough to contain all the components of the subcell. The mask designs painted as part of the subcell will not be shown, nor will the child's subcells, but the cell's name will be displayed in the upper half of its bounding box. The second form for subcells is *expanded* form. In expanded form, the bounding box of the subcell is not displayed. Instead, all of the cell's components (paint and expanded or unexpanded subsubcells) are shown. To modify the display mode of the current cell so that only its bounding box and name are displayed, type the short command **c**. To expand the cell again, type **C**.

When one cell is included in another using the **:getcell** command, Caesar does not copy information; it just stores in the parent a pointer to the child cell's file. If the child cell is edited, the new contents will appear in the parent cell the next time the parent is edited. Each parent cell maintains a *guess* about the bounding box for its children so that the definitions of the children need not be read in until they are expanded (this speeds up the editing of large designs). However, if the child has been changed then the bounding box as displayed in the parent may be incorrect. The guess will be corrected the next time the child cell is expanded.

To change the current cell, position the crosshair inside the cell you wish to select, then push the top (yellow) button on the puck. Of all the cells that contain the crosshair, Caesar will select the one whose lower-left corner is closest to the crosshair. If a child cell has the same lower-left corner as its parent then the child's corner is consdered to be slightly *inside* the corner of the parent. If two unrelated cells have the same corner, the choice between them will be made randomly. It is not possible to "find" the edit cell. Once a cell has been found, information for the new current cell will appear in the text display and the box will be changed to coincide with the bounding box for the selected cell. To select the parent of the current cell, press the yellow button again without moving the crosshair. This may be repeated many times to step up through the cell hierarchy.

To reposition the current cell, type the long command

:movecell <keyword>

where <keyword> is one of "byposition", or "bysize". If <keyword> is "byposition" or is omitted, then the cell is moved so that its lower-left corner coincides with the lower-left corner of the box. If <keyword> is "bysize" then the cell is displaced by the x- and y-dimensions of the box. Thus, what used to be at the lower-left (fixed) corner of the box will now be at the upper-right (variable) corner. This is especially useful for making fine adjustments on a large cell whose lower-left corner isn't on the screen.

The

:copycell

command makes a copy of the current cell and positions it at the lower left corner of the box (as explained above for the **:getcell** command, only a pointer to the subcell's file is copied). The copy is made the current cell. The

:upsidedown

command will flip the current cell upside down by mirroring its contents about a horizontal line. The

:sideways

command flips the current cell sideways by mirroring its contents about a vertical line. The

:clockwise <degrees>

long command will rotate the current cell clockwise by the nearest multiple of 90 degrees less than or equal to <degrees>. If <degrees> isn't specified, then the cell is rotated 90 degrees clockwise. Any of the **:upsidedown**, **:sideways**, or **:clockwise** commands may be followed by a "y": this causes the action to be performed on the yank buffer rather than the current cell. The long command

:deletecell

will delete the current cell (i.e. it removes the use of that cell from the edit cell; it does not affect the disk file containing the cell).

It's important to remember that at any one time you are editing one and only one cell. The things that you can change are the edit cell's paint, and the ways in which subcells are used in the edit cell. You are not permitted to make any modifications to the contents of subcells. Thus, you cannot erase paint in children, nor can you move a grandchild cell inside a child cell.

10. Cells and Painting

There are several commands that manipulate both subcells and paint, or turn one into the other. For purposes of the **:erase**, **:yank**, and **:put** commands, subcells may be thought of as a layer (abbreviation 'S') just like the mask layers or labels. For example, **:erase** S will delete all subcells that intersect the box, and **:yank** +S will yank all the visible layers and subcells too. Subcells are never included in **:erase**, **:yank**, and **:put** unless you specify them explicitly. Furthermore, **:yank** treats expanded and unexpanded subcells differently. In **:yank** S, only unexpanded subcells will be yanked. If a cell is expanded then Caesar assumes you want to yank the paint of the subcell, rather than the subcell itself. In **:erase**, subcells are deleted whether or not they are expanded. This distinction is a bit confusing but seems to do the right things in practice.

As mentioned above, **:yank** will grab all paint underneath the box, regardless of which cells contain the paint. Thus you can turn a subcell into paint by yanking its contents, deleting the subcell, and putting the paint back again. As a convenience, Caesar provides the long command

:ycell <name>

which does just that. If <name> isn't specified, this command performs exactly the sequence of operations listed above: it sets the box to the bounding box of the current cell, expands the cell if it isn't already expanded, yanks all the paint and labels of the cell, deletes the cell, and puts the paint and labels back into the edit cell. If <name> is specified, then the indicated cell is first read from disk and positioned at the box, just as if **:get <name>** had been typed. Since it collapses the cell hierarchy, I don't recommend using **:ycell** except for very small things such as contacts or transistors.

Caesar also provides a command to turn paint into a cell. The command

:ysave <name>.

causes all of the information in the yank buffer, including paint, labels, and subcells, to be written to disk as a cell named <name>.ca.

11. Arrays

Arrays provide an efficient mechanism for specifying and manipulating groups of identical cells. The long command

:array <xsize> <ysize>

will turn the current cell an array of cells. The array will be a rectangular one containing <xsize> instances in the x-direction and <ysize> instances in the y-direction. The instances will be spaced in x and y according to the x- and y-dimensions of the box at the time the **:array** command is issued. Once an array has been created, any manipulation of any element in the array will affect the entire array. For example, if one element is expanded, then all elements are expanded. Similarly, the entire array must be moved, unexpanded, and copied as a whole. The **:array** command may be typed when the current cell is an element of an array. When this happens the current array is replaced by a new array such that the lower-left corners of the old and new arrays coincide.

12. Subedits

When designing a large circuit with many subcells, subcells often must be changed in ways that depends on their usage in the chip as a whole (for example, a subcell might have to be modified to connect properly to its neighbors). To facilitate making such changes, Caesar provides a subedit facility that allows cells to be *edited in context*. The long command

:subedit

causes a subedit to be entered by making the current cell the edit cell. During a subedit, the child cell is displayed just as it appears in the larger cell (e.g. rotated or as an array), and all of the paint and other children of the larger cell continue to be displayed on the screen. During a subedit, as always, only the edit cell may be modified. It is impossible to select any information except that in the edit cell and the tree of subcells that it heads. To return from a subedit, type the long command

:return

Subedits may be nested. The term *root cell* refers to the topmost cell in the cell hierarchy, which differs from the edit cell if a subedit is in progress.

During a subedit the bounding box of the edit cell may change, and Caesar will automatically propagate this change to all uses of that cell, continuing up through the cell hierarchy until all bounding boxes are correct. This may mean that many cells need to be written to disk in order to reflect the changes. The long command

:writeall

will scan the database for all files that have changed since the last time they were written. For each cell that has changed, you are asked for one of three responses: "write" to write the cell to disk; "skip" to go on without writing this cell; or "abort" to return to command mode immediately. The **:writeall** command is invoked automatically as part of several other commands such as **:editcell**.

13. Marks and the Box Stack

The box gets used for many different functions, some of which conflict with each other. For example, if a long rectangle is to be painted to connect distant points, the most convenient way to do this is a) set the view to the area where the left end of the rectangle will be, and put the box's lower-left corner at the starting point for the rectangle; b) move the view to where the other end of the rectangle will be; c) set the box's upper-right corner and paint the rectangle. Unfortunately, it may be necessary to use the box to change the view; thus the position of the lower-left corner of the rectangle will be lost. Marks and the box stack are intended to facilitate such things as the drawing of long connections.

Caesar allows up to 26 user-settable *marks* to be stored during an editing session. Each mark is just a rectangle. To store a mark, type the command

:mark <mark1> <mark2>

<mark1> must be a single lower-case letter. The box will be stored in the indicated mark. If <mark2> is specified, then after setting <mark1> the box is set to the rectangle that is in <mark2>. If <mark2> isn't specified, then the box isn't changed.

When retrieving a mark, either in the **:mark** command or any of the additional commands discussed below, either a lower-case letter may be typed to specify a user mark, or one of several upper-case letters may be typed to specify a *system mark*. System marks are defined in Table 2. Instead of a single letter, it is also permissible to type either two or four integers, separated by spaces. This is referred to as an *absolute mark*. The first two integers specify the x- and y-coordinates of the lower-left corner of a rectangle, and the second two integers specify the x- and y-sizes. If the last two integers aren't specified then the rectangle is assumed to have zero size.

C -	The bounding box of the current cell.
E -	The bounding box of the edit cell.
R -	The bounding box of the root cell.
V -	The current view.
P -	The previous view.

Table 2. The system marks.

Although at any given time only one box is visible, Caesar maintains internally a stack of boxes. The current box is at the top of this stack. To save the current box on the stack, issue the long command

:pushbox <mark>

This command saves the current box on the stack, and provides a new one to be manipulated. If <mark> is present, it is a mark that is made the new box, and may be a user mark, system mark, or absolute mark. If <mark> isn't specified then the new box is made the same as the old one.

The command

:popbox

retrieves the last box pushed onto the stack by discarding the current box and making the one underneath it on the box stack the current box. If the command is typed as

:popbox<mark>

Then the current box is discarded and a new current box, indicated by the given mark, replaces it at the top of the box stack.

Marks may be used in the long command

:view <mark>

This command will set the current view to contain the area indicated by <mark>. <mark> may be a user mark, system mark, or absolute mark, and defaults to "R".

14. Searching

To assist you in finding a label or subcell of a given name, Caesar provides the long command

:search <regexp>

where <regexp> is a regular expression in the same form as those suitable for *ed* (see the manual entry for *ed* (1) for details). The **:search** command clears the box stack, then scans all of the information in the database that lies underneath the box (in subedits only information in the subtree of the edit cell is examined). For each label that matches <regexp> the label's box is pushed onto the top of the box stack and a message is output on the terminal. Similarly, for each cell whose name matches <regexp> the cell's bounding box is pushed onto the box stack and a message is output. After the command has finished, the various matches can be found merely by popping them from the box stack. For example, **:search the** will find all labels and cells that intersect the box and contain the string "the".

15. Filenames and Paths

In order to make it easy to identify how files are to be used, and in order to prevent accidental misuse of files, Caesar uses a standard set of file name extensions. For example, it expects all files that are edited using Caesar to have names ending in the characters ".ca". By convention, all colormap files use a monitor type as extension, and all CIF files have the extension ".cif". Caesar doesn't absolutely require you to abide by these conventions, but it makes it easy for you to do so and difficult for you to do otherwise. For example, in the **:getcell** command, Caesar will first try to get the cell by appending ".ca" to the name you type. If this fails, then it will try the unextended name. Similar things happen for colormap and CIF files; Caesar will first append the standard extension and will only try the unextended name if the extended one doesn't work.

There is one way to get around the default extensions. If a name that you supply contains a "." character, then Caesar will assume that you have your own (crazy) scheme for name extensions and will not tack on any of its own.

Caesar also implements a *search path* mechanism that makes it easier to work on large designs where the component files are spread over many directories. The search path contains the names of one or more directories that Caesar will examine in order when opening files for reading. Whenever Caesar attempts to open a file for reading, it searches for the file in each of the directories in the path until the open succeeds. If no directory in the path contains the file, Caesar will make one last attempt by looking in a system library directory. On the VAX'es at Berkeley, the library area is ~cad/caesar/lib. The library directory contains standard technology and color map files, shiftcell, and other files. If the original file name begins with a "~" or "/" then the path mechanism isn't used.

The initial search path is set to "." (the working directory) when Caesar begins execution, unless overridden by the -p switch or a *.caesar* file. To change the path once Caesar is running, type the long command

:path <string>

where <string> contains one or more directory names separated by blanks or colons. From then on, Caesar will search for files by looking in each of the directories in string in the order of their appearance in <string>. Directories may be specified using the "~" notation, and "::" is equivalent to ":.:". Typing **:path** with no parameters will cause Caesar to print out the current search path. Paths may also be specified when Caesar is invoked by using the -p<path> switch, with <path> having the same format as <string> above.

The search path mechanism is only used for reading files. When writing out files, one of two mechanisms is used. Normally, files are written into the current directory unless the file name starts with '~' or '/'. The one exception to this rule occurs when **:savecell** is invoked without specifying a file name. In this case, Caesar will write try to write the cell back to the place from which it read it, regardless of where that may be.

16. Macros

Caesar has a very simple facility for defining macros. A macro is just a short command defined by the user, such that whenever a particular character is typed as a short command, a long command is executed instead. The long command

:macro <character> <long command>

will set up a macro such that <long command> is executed whenever <character> is typed. For example, the command

:macro l paint p\; box up 1\; box left 1

defines a new short command **l** that will paint polysilicon and move the box diagonally up and to the left one unit. The backslashes are be used to prevent the semi-colons from terminating the macro definition (without the backslashes, the command would have defined a macro that just paints; then the box moving commands would have been executed). Macros override the system definitions of short commands. To remove a particular macro and restore the system definition, type

:macro <letter>

To remove all macro definitions, type

:macro

Note that macros may include short commands by using the single-quote notation defined in Section 3. Thus,

:macro l 'a\; box up 1\; box left 1

is the same as the macro definition in the previous paragraph except that it yanks all the layers visible underneath the crosshair. When short commands are invoked using the single-quote notation, macro expansion is NOT performed. Thus in the above example, any macro definition for the short command **a** is ignored.

17. Command Files

The command

:source <file>

will read <file> and execute each line of the file as a long command. If the last character of a line is a backslash, then the backslash is removed and the line is joined to the following line. When Caesar starts up it attempts to read two command files. First, it looks for a file named *.caesar* in the home directory of the user; if this file exists then it is processed as a command file. Then Caesar attempts to read *.caesar* in the current directory. The startup command files are useful for setting paths, technologies, and macros.

18. CIF Output

The format in which Caesar stores its cells on disk is not Caltech Intermediate Form, the standard representation used to fabricate chips. However, the long command

:cif -sblpx <name> <scale>

will cause Caesar to write out in CIF format a file that describes the edit cell. The parameters and switches may be specified in any order and are all optional. <name> is the name of a file in which to write the CIF (a .cif extension is appended automatically). If <name> is not specified, then the name of the edit cell is used by default. <scale> is a number that is used for conversion from Caesar units (lambdas) to CIF units (centimicrons); it specifies how many centimicrons there are in one lambda. If <scale> is not specified then it defaults to 200 (i.e. lambda = 2 microns).

Warning: if your design is made on a 1/2-lambda grid, then round-off errors will occur in the CIF file if the scale is not an even multiple of 4 centimicrons. If your design is entirely on a lambda grid, then round-off errors will occur if the scale is not an even multiple of 2 centimicrons. When round-off errors occur, pieces of the design may appear to move by as much as one centimicron. Although this movement will not cause any noticeable effect during fabrication, it may cause CIF-based analysis tools to misinterpret the circuit. To be safe, always use a scale factor that is a multiple of 4 centimicrons, e.g. for lambda = 2.5 microns, specify a scale of 252 or 260.

The CIF information may be used for many different purposes: a) for getting hardcopy plots of the circuit; b) for input to circuit extractors, design rule checkers, and simulators; and c) for fabricating chips. The switches control what information is to be output into the CIF file, according to the way the CIF file will be used. As many as four different kinds of information may be output in the CIF file:

Silicon	What actually gets fabricated: rectangles specifying the mask layers.
Bounding Boxes	When CIF representation is being used as a means for getting checkplots, Caesar can output commands that cause unexpanded cells to be plotted in bounding box form. This is done by outputting vectors ("0V" user extension) and text ("2" user extension) so that a bounding box will appear along with the cell's name and id when the CIF file is plotted. This makes it possible to get block diagrams of circuits.
Labels	For each label in an expanded cell, Caesar will output vectors and text to make the label appear in plots just as it appears on the screen (except that labels that appear as crosses on the screen will appear as dots in the plot). This feature is only useful for getting hardcopy.
Points	CIF provides the "94" construct to give names to various points on the masks. Caesar will generate "94" commands for each label. These commands are used by several of the circuit extraction and simulation programs.

If no switches are specified, Caesar will output none of the above information except silicon. Furthermore, cells will implicitly be expanded as CIF is being output so that all the silicon in the edit cell and its descendants will appear in the CIF output. The -b, -l, and -p switches will enable bounding boxes, labels, and points, respectively, and the -s switch will disable silicon output. If the -x switch is specified, then cells will not be automatically expanded: silicon will appear only for cells that are currently expanded. This switch can be used in conjunction with the -b switch to get block diagrams. Some useful combinations of switches are: a) to get a plot of things just as they appear on the screen, use **:cif -xbl**; to generate CIF files suitable for manufacturing, use **:cif**; to get CIF files for circuit extraction and/or simulation, use **:cif -p**.

19. Non-Interactive Use of Caesar

If the -n switch is present on the command line, then Caesar will execute in non-interactive mode. In this mode, it does not use a color display at all, nor does it display the normal menus and statistics on the text display. Instead, it merely reads long commands from its standard input (the single-quote notation described in Section 3 can be used to invoke short commands). This mode is useful for running the **:cif** command in background, for example. If an end-of-file is encountered on the standard input when in non-interactive mode, Caesar exits immediately, without saving anything.

20. Identifiers

This command is not well supported, and hence is not likely to be very useful.

In addition to its name, which refers to the file containing its definition, each cell may be given an *instance identifier*, or ID. The ID distinguishes a subcell from all the other children of its parent, particularly those siblings that share the same definition file. Caesar does not currently use the ID information and does not output it to CIF files, so it serves only to document the circuit. At some future date additional design tools may take advantage of the ID information. To give a cell an instance identifier, type the long command

:identifycell <id>

The identifier will become the instance identifier for the current cell, and will appear in the lower half of the cell's bounding box when the cell is unexpanded. The same identifier may not be used in two subcells of the same parent.

When an element of an array is given an identifier, Caesar will give IDs to all the elements of the array by taking the name and appending "[x,y]" where x and y are the indices of the element within the array. Normally, the indices start from 0 at the lower-left corner. To change this, the array should be generated using the command

:array <x1> <x2> <y1> <y2>

This command generates an array with elements indexed from <x1> to <x2> in the x-direction and from <y1> to <y2> in the y-direction.

21. Miscellaneous Commands

Caesar can communicate with the Lyra layout rule checker. To invoke Lyra, first use the box to select the area you wish to check. Then type

:lyra <ruleset>.

The parameter <ruleset> is optional and is passed to Lyra with the -r switch. If <ruleset> is omitted, an appropriate rulest is picked based on the current technology. Design rule violations returned by the layout rule checker are displayed as labels in the edit cell.

The short command **.** (period) causes the most recent long command to be repeated.

The long command

:quit

causes Caesar to cease execution and return to the shell.

The long command

:reset

will re-initialize the graphics display. This command is needed if the display should become fouled up or if the sleeper job should die. First reset the color display. Make sure that a sleeper job is still logged in, if necessary. Then invoke the **:reset** command.

A long command is provided whose function is equivalent to pressing a puck button. The command syntax is

:button <number> <x> <y>

This long command simulates the pressing of button <number> at the screen location given by <x> and <y> (in pixel coordinates). <number> must be 0, 1, 2, or 3, and the coordinates must lie on the screen. If <x> and <y> aren't specified, then the crosshair position is used. This command is useful for macros and for displays without buttons on their puck/mouse/joystick.

A new technology may be loaded with the long command

:technology <file>

where <file> is the name of a technology file (see Section 22). A default ".tech" extension is supplied. This command changes Caesar's current technology, regardless of the technology of the cells being edited, and may thereby produce bizarre and undesirable effects (for example, if the existing cells are saved on disk, they will be marked with the new technology). Normally **:technology** should only be invoked when the edit cell is null.

The short command **^L** (control-L) causes the graphics display to be erased and redrawn, and the command **l** causes both the text and graphics displays to be redrawn. These commands shouldn't be necessary very often. Nonetheless, one or the other of the screens will occasionally get trashed, and this provides a recovery mechanism.

X is an "expand all" command. Any cell that intersects the box is expanded, then all the subcells that intersect the box are expanded, and so on until there is nothing but paint left underneath the box. The short command **x** is the inverse of **X**: all of the cells that intersect the box, but do not completely contain the box, are unexpanded to be drawn in bounding box form.

The long command

:peek <layers>

provides another form of "expand all". It causes all the paint lying underneath the box to be displayed, including paint in unexpanded cells. However, the expanded/unexpanded state of cells is not changed, so the effects of the command are temporary: the next time the area is redrawn, information will appear as it did before the **:peek** command. <layers> has the same format as in the **:erase** command. Only the layers given by <layers> are displayed (if <layers> isn't specified then all visible layers are shown) and only the area underneath the box is affected. The **:peek** command is somewhat faster than **X** and **x** since it doesn't require any modifications to the database and involves only the area underneath the box. Information drawn by **:peek** is not "officially" visible and hence is ignored by commands such as **:yank** and **:fill**.

The long command

:flushcell

simply unloads the current cell from main memory. This command has two uses. First, if there are several people using different workstations to edit different cells of the same chip at the same time, **:flushcell** provides a mechanism to pass back and forth updated versions. If one person changes a cell and saves it on disk, then the other person can see the latest version by flushing his current version. Thus it isn't necessary to leave Caesar and restart. Flushing is also useful if you edit a cell and then decide that you don't want the edits after all. **:flushcell** will throw away the changes and reload the disk version.

The long command

:usage <filename>

can be used to figure out which files in your directory area are part of a design. The **:usage** command will write out in <filename> a list of all files containing definitions that are part of the cell hierarchy.

Caesar is still undergoing development, so you may stumble across bugs and unpleasant features as you use it. Hopefully this won't happen too often, but when it does you can use the

:gripe

command to give feedback to whomever is maintaining the system. When you type **:gripe** the mail program is run and Caesar will supply the address of the system maintainer. Just type in your message as you would if you had run mail yourself. Please put the word "Caesar" in the subject or first line. Feel free to suggest enhancements as well as report problems. When you have typed in the message, type ^D and control will return to Caesar.

22. Technologies

This section is intended for system maintainers only.

Caesar versions 6 and later are technology independent: they permit you to define new technologies of your own design. For Caesar's purposes, technology information merely contains layer names and information about how to display them. A technology is defined in a technology file, which usually has a ".tech" extension. For example, the standard NMOS technology is defined in a file called "nmos.tech" in the system library. To create a new technology or an extended version of an existing technology you need only create a new technology file. Table 3 contains the Berkeley technology file for NMOS.

```
nmos
nmos
polysilicon pr 0 solid 1
L NP
diffusion dg 0 solid 2
L ND
metal mb 0 solid 4
L NM
implant iy 0 solid 10
L NI
cut c 377 cross 40
L NC
overglass o 377 ll-ur 41
L NG
errors e 0 solid 42
L NZ
buried_contact x 0 stipple 20
L NB
210 42 210 42 210 42 210 42
```

Table 3. The Berkeley NMOS technology file.

The first line of each technology file is the name of the technology e.g. "nmos". Every cell is also marked with a technology name; the technology names in the .ca and .tech files must agree. Caesar does not permit cells of more than one technology to be edited at one time. The second line of the technology file contains the name of the color map to be used for that technology. When looking up the color map file, Caesar will supply the monitor type as extension (see Section 23 for a detailed discussion of color maps).

Lines after the first two are grouped in pairs or triplets; each group describes one mask layer. There may be up to 16 layers. The order of the layers makes no difference. The first line of each group has the syntax

<longname> <shortnames> <outlinestyle> <fillstyle> <layer>.

<longname> is a descriptive name for the layer, and is used by Caesar to identify mask layers in cell files. <longname> must not be either "labels" or "end". <shortnames> consists of one or more characters that will be used as abbreviations for the layer in commands such as **:erase** and **:yank**. <shortnames> entries for all layers must be distinct, and must not repeat any of the predefined layer names (those in the lower half of Table 1). <outlinestyle> and <fillstyle> describe how rectangles in the layer are to be displayed. Each rectangle is drawn in two stages: first an outline is drawn, then the contents of the rectangle are filled. <outlinestyle> is an eight-bit octal number whose bits give a pattern indicating how the outline is to be drawn. All ones (377) means draw the outline as a solid line, zero means don't draw any outline at all, 360

means draw a dashed line, 252 means draw a dotted line, and so on. <fillstyle> indicates how the box is to be filled, and must be one of the keywords listed in Table 4. The solid style is the most efficient one. <layer> is an octal layer number, which will be explained below. The second line for each layer contains the CIF command used to switch to that layer. This information is used when generating CIF files. If the <fillstyle> is "stipple", then there is a third line in the group (after the CIF command line) that contains eight octal numbers giving an 8-by-8 array of ones and zeroes used for stippling that layer. Stippling is only available on Chromatics, AED767, and specially microcoded AED512 displays (display type "UCB512")at present.

empty	Don't draw anything inside the rectangle.
solid	Fill the rectangle with solid color.
cross	Draw diagonal lines between opposite corners.
horizontal	Cross-hatch with horizontal lines.
vertical	Cross-hatch with vertical lines.
ll-ur	Cross-hatch with lines running from lower left to upper right.
ul-lr	Cross-hatch with lines running from upper left to lower right.
stipple	Use stipple pattern given in third line.

Table 4. Fill styles for technology files.

The <layer> entry must be an octal number that is either 1, 2, 4, 10, 20, or between 40 and 52 inclusive. No two <layer> entries may be the same. <layer> determines whether or not the corresponding mask layer is *opaque* or *transparent*. The distinction between transparent and opaque layers is necessary because the color displays don't have enough memory to allocate a separate bit plane for each mask layer. Transparent layers are those with <layer> values 0-4. They have two nice properties: first, it is possible to see transparent layers even when they lie underneath other transparent layers; second, Caesar can perform screen operations on transparent layers more efficiently than for opaque layers. Opaque layers have the property that they blot out everything underneath them. If one opaque layer is colored at a point, it is impossible to see transparent layers or other opaque layers underneath it. Higher-numbered opaque layers blot out lower-numbered opaque layers. Cross-hatching was implemented for use with opaque layers: only where the outline or cross-hatching is drawn does other information get blotted out. A good rule of thumb when assigning layer numbers is to make the densest and most frequently manipulated layers transparent.

Cells edited under one technology can be edited under another technology with no side effects as long as the two technologies agree on the <longname> values for each mask layer and the two technology files have the same first line. However, strange things may happen if you switch technologies while a cell is loaded into Caesar: the layers of the old technology will be mapped into those of the new technology according to their <layer> values, rather than their <longname> values. This will generally NOT produce the desired effects, although it can be used to move information from one layer to another. Normally, the cell to be edited should be reloaded (using the :editcell command) after a switch of technology.

23. Color Maps

This section is intended for system maintainers only.

Color maps are tables that indicate what color to display for each of the various layers. Caesar allows you to change the color choices and to save your own color map files. Each color is specified by means of red, green, and blue intensities that may range from 0 to 255. To read out the current color values for a particular layer or layer combination, type the command

:colormap <layer>

where <layer> is any combination of the layer mnemonics from Table 1 (if you are using a different technology then the mask layer mnemonics will be different). For example, **:colormap p** will print out the red, green, and blue intensities for the color that is displayed where polysicilon appears by itself, and **:colormap pm** will print out the intensities for the color used to represent overlaps between polysilicon and metal. The command

:colormap <layer> <red> <green> <blue>

will set the colors for <layer> to those given. If the first character of <layer> is a "*", then the indicated colors are stored for *all* layer combinations that contain the selected layers. For example, **:colormap *X 255 255 255** will cause the color white to be displayed anywhere that the box appears, no matter what other layers may be present. The layer may also be specified as an octal number.

There is a different color for each possible combination of transparent layers. In existing color maps, the colors are chosen to make certain layers appear on top of other layers. For example, the colormap entry for "pm" is different from the entries for "p" and for "m", and is intended to make the metal layer appear on top of the polysilicon layer while still permitting underlying details to be distinguished. There is only one color table entry for each opaque entry: in NMOS, for example, "cm", "cp", and "*c" all refer to a single entry. The G, S, and l layers are all the same as far as the color map is concerned; changing any one of them will change all of them. However, the G/S/l layer is transparent with respect to the mask layers: a separate color exists for each combination of mask layer and G/S/l. The box layer is also transparent with respect to mask layers and the grid/subcell/label layer. Layer name "B" is used to select the color of the background. This layer is blotted out by any of the other layers.

Modified colormaps may be saved on disk and retrieved. The command

:csave <name>

causes the current colormap values to be saved in file <name>. Caesar uses the monitor type as extension to the name. Thus, if you are working on a monitor of type "std", the command ":csave cmos-pw" will create a file named "cmos-pw.std". The command

:cload <name>

causes Caesar to reload its colormap from the named file, once again using the monitor type as extension.

24. Locating the Correct Display

This section is intended for system maintainers only.

When Caesar starts up, it tries to figure out what kind of display it should use by consulting the displays file. At Berkeley, the displays file is located in ~cad/lib/displays. Each line in the displays file describes one workstation and contains up to five strings. The first string gives the file name of the text terminal of the workstation. The second string gives the file name of the device to use for I/O to and from the color display. The third string gives the type of monitor attached to the display. The fourth string gives the type of display, and the fifth string gives the file name to use for reading characters from the display's tablet. Table 5 lists the display types understood by all versions of Caesar. Some sites may also have support for display types not listed. The "display type" indicates what kind of electronics is used to hold the raster memory, e.g. "AED512" or "Omega440". The "monitor type" indicates the type of color monitor that is attached to the display. The monitor type is used to select the right color map to use (phosphors on different monitors may be slightly different and hence require different color maps). At Berkeley we use several different types of monitors with different color characteristics. Caesar understand two general kinds of monitors: "std" and "pale". The monitor type is used by Caesar to select a color map that will make your circuits look nice on that particular monitor. The "std" colormaps work well with most monitors. Some monitors with long-persistence phosphors have a blue phosphor that is especially pale. With these monitors the "pale" colormaps work well. If you have a monitor with unusual colors, you'll probably have to make a new colormap by modifying one of the standard maps. If any of the strings are omitted, default values are used. In the case of the tablet file, the default is to use the same file as for display output.

Values from the displays file are overriden by command line switches.

Display Type	Manufacturer	Notes
AED512	Adv. Electr. Design	
UCB512	Adv. Electr. Design	(AED512 with UCB microcode for stipples)
AED767	Adv. Electr. Design	(This display type can also be used for some Jupiter displays)
AED640	Adv. Electr. Design	(AED767 configured as 640x483 pixels)
Omega440	Metheus	(Courtesy Metheus Corp.)
R9400	Ramtek	(Courtesy Gary Bishop, UNC-Chapel Hill)
Vectrix	Vectrix	(Courtesy Gary Bishop and Eric Vook, UNC-Chapel Hill)
Chr7900	Chromatics	(Courtesy Dan Schuh, Univ. Wisc.)

Table 5. Supported display types.

25. Known Bugs and Quirks

1. The cell expansion facilities have a quirk stemming from the fact that if the same subcell is used in two places Caesar only keeps a single copy of the definition of the cell in order to save memory space. What this means is that if you are editing a cell with two identical child cells, each with a child of its own, then if one of the grandchildren is expanded the other grandchild will be expanded as well. This quirk only affects grandchildren and more distant descendants of the edit cell: children may be expanded and unexpanded independently.

2. If Caesar should crash, the text terminal will be left in a weird state. To escape this state, type "reset" followed by linefeed (control-J), NOT carriage return.

3. Caesar expects that label text will fall within the bounding boxes of the cells they belong to. This results in much greater efficiency when moving cells around, since Caesar only worries about the area inside the cells' bounding boxes. However, if label text falls outside the bounding box, it will not be properly erased when the cell is moved. To clean up the screen it will be necessary to type ^L.

4. Caesar handles interrupts (e.g. rubout) but in a stilted fashion. When the interrupt key is typed, all searches in progress will be stopped immediately, but no other computations are affected. This will escape from long redisplays and finds, but it may still take a while for Caesar to finish whatever else it was doing.

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: ouster@berkeley
Uucp address: ucbvax!ousterhout

This user manual corresponds to Crystal version 1.

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.

Crystal differs from circuit simulators like SPICE and Relax in two ways. First, its device models are much simpler: transistors are characterized by fixed resistances. This makes the analysis fast enough for Crystal to handle circuits with 50000 to 100000 transistors, whereas SPICE becomes impractical for even a few hundred transistors. Unfortunately, the simple models are also less precise: there are situations where Crystal's results are much less accurate than those of a circuit simulator. Where Crystal errs, it usually underestimates the delays.

The second difference between Crystal and SPICE is that Crystal's analysis is independent of specific data values. What this means is that Crystal considers all possible combinations of data values in a single analysis, whereas SPICE examines just one situation at a time, which is determined by the input values supplied by the user. The advantage of Crystal's mechanism is that a single analysis is sufficient to find all the slow paths. However, Crystal's value-independence causes it to make grossly pessimistic delay calculations in several situations. When pessimistic calculations occur, you'll have to supply additional information to improve Crystal's analysis. The extra information is provided partly in the mask layout and partly with commands to Crystal.

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 and switches. As with most programs, Crystal is easiest to understand if you try it out on simple test cases as you read the manual sections. Warning: Crystal is a relatively new program and has only been used on a few chips. Don't be suprised if you run into bugs. Please let me know about any weird behavior you see.

2. Crystal's Basic Mechanism

In order for you to provide Crystal with the information it needs to make accurate timing analyses, you need to know about how Crystal works. Crystal's function is to examine *consequences*: you tell it a little bit about your circuit, and Crystal tries to infer as many consequences from that as possible. Furthermore, Crystal generally tries to infer the most pessimistic possible consequences, in order to make sure that you know the worst that can happen.

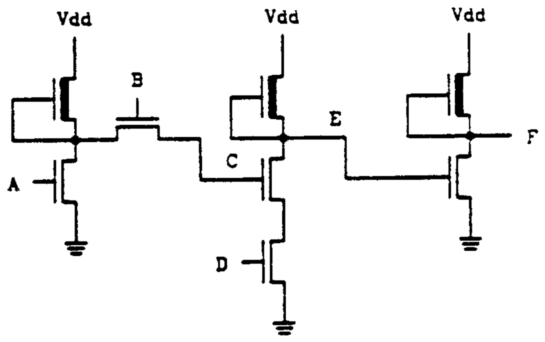


Figure 1. If you tell Crystal that node A can fall at a certain time, Crystal will infer that node C might rise at a later time, node E might fall at a still later time, and node F rise latest of all.

The consequences analyzed by Crystal are delays. To start a delay analysis, you give Crystal the latest time when some signal in your circuit changes (usually this is an input pad). See Figure 1. Given this information, Crystal determines what other nodes might change as a consequence of the change you indicated. For example, in Figure 1, if node A changes from 1 to 0, then node C might change 0 to 1 at a later time. Crystal computes the time when C will settle at 1 and stores this with the node. Of course, if B is zero then the pass transistor is off and C will not actually change value. Unless you explicitly tell Crystal that B is zero, it will assume that the pass transistor *might* be on and compute the delay from A to C.

Once Crystal discovers that node C can change value, it will examine the consequences of this change too: Crystal will see that if C changes to 1, this might close a circuit from node E through the two pulldown transistors to ground. If Crystal knows that D is 0, then then pulldown path is broken, so Crystal will conclude that C has no effect on E. Normally, though, D will not be known to be zero, so Crystal will compute how long it takes for E to fall to zero, given the time when C turns on. Finally, when node E changes to zero, node F will eventually rise to 1, so Crystal calculates this delay.

If the circuit has many input signals, you invoke the delay analysis again for each of them. 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 chain of nodes, each causing a change in the next. The worst-case path is the one whose last node 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. In this manual, the term *stage* refers to a path passing through no more than one transistor gate. For example, the path from A to C is a *stage*, but the path from A to E contains two stages. To calculate the settling time for a node, Crystal uses the information in the stage leading to the node, along with the time when the stage is activated (see Section 10 for more information on delay calculations).

3. 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 if **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.

4. Simple Runs on Combinational Circuits

Invoke Crystal with the shell command

crystal *file*

where *file* is the name of a .sim file. 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

Crystal reads its commands from standard input and writes its output to standard output (except when processing a **source** command; see Section 11). Each input line consists of a command name followed by parameters. 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.

A Crystal run contains three kinds of commands: information commands, delay commands, and printing commands. The first commands give Crystal additional information about the circuit, identifying inputs, noting signals whose values are fixed, and so on. Commands in the second group invoke delay analysis. The third group of commands is used to print out information about the delays that Crystal has calculated.

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 two commands need be used: **inputs** and **delay**. First, you must identify to Crystal the circuit inputs (the nodes that are driven externally, such as input pads). For example,

inputs Bus<31:0> Select

identifies the 32 bus bits and the select signal as inputs (see Section 7.1 for more on the **inputs** command).

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 results. The **worstdelay** command can be used for this. It requires no parameters. **Worstdelay** will locate the node with the latest possible rise or fall time, and print on standard output the path through the circuit that caused the worst case time.

5. Simple Runs on Clocked Circuits

Clocked circuits are handled like combinational circuits, except that there is a separate group of **input**, **delay**, and **worstdelay** 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 all information about delays and inputs. An alternate way to handle different clock phases is with a completely separate Crystal run. However, for large chips it takes a long time just to read in the circuit so it is usually faster to process all clock phases in a single run.

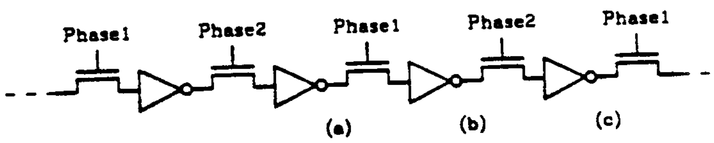


Figure 2. If Crystal doesn't know that Phase2 is zero, then during Phase1 analysis it will consider a path from the left end of the shifter all the way to the right end. If a set command is used to tell Crystal that Phase2 is zero, then Crystal won't propagate delays through the pass transistors that are turned off.

In addition to the **clear** commands between clock phases, **set** commands will be needed just after the **input** 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 (the **clear** command will erase this information). Crystal uses **set** information to avoid considering delay paths that cannot occur, as illustrated in Figure 2 (**set** commands are described in more detail in Section 7.2).

Although the simple set of commands described above will work for many circuits, there are other circuits where it won't work at all. In particular, circuits with networks of transistors used for multiplexors or shifters require additional information that is discussed in Section 8. 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 **worstdelay** command, there are several additional ways that Crystal can print information. Some features are common to all of the printing commands.

8.1. Caesar Command Files

All printing commands except **printallfets** will generate Caesar command files if you wish. The **-c** switch is used for this purpose. For example,

worstdelay -c dum

will generate a list of Caesar commands in the file **dum**. To use a 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); fourth, 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 **worstdelay** 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. Repeat Limits

Since VLSI circuits typically contain arrays of identical logic, Crystal incorporates a mechanism to avoid printing out exactly the same information for every member of an identical array. The mechanism is based on a repeat limit. If several lines of output contain an identical field, then only a few of those lines are printed. For example, in the **prresistance** command, if many nodes have the same resistance then only the first few will actually be printed. The repeat limit, which is set with the **-r** command line switch, determines how many duplicates will be printed. By default the repeat limit is one: only one item will appear with each distinct value. Whenever output is suppressed because of the repeat limit, Crystal indicates how many duplicates were suppressed.

8.3. Worst-case Paths

The **worstdelay** and **manyworst** commands print out delay paths through the circuit and are similar in form:

worstdelay [**-m**] [**-c** *file*] *node node ...*
manyworst [**-m**] [**-c** *file*] [**-n** *number*]

In each case the output is one or more delay paths through the circuit and possibly a Caesar command file with the same information. Each delay path is printed as a series of stages starting at the node that changes last. Each stage is printed as a path from a node to a source of logic 1 or 0. The source is usually Vdd or GND, but occasionally can be an input or bus. The transistor that causes the change is referred to in the output as the "driver" for that stage.

Worstdelay will either print out the delay paths to each *node*, or if no *node* is given it will print out the delay path to the slowest node in the circuit. **Manyworst** will print out paths to the *number* slowest nodes in the circuit (if the **-n** switch is omitted then 10 paths are printed). In **manyworst**, the repeat limit is used to avoid printing nodes with identical delays.

Manyworst uses more processing time than **worstdelay** because it requires the entire circuit to be searched for the slowest nodes (Crystal always remembers the single slowest node in order to make **worstdelay** run fast). The **-m** switch causes Crystal to consider only paths that end in a memory node. This feature is discussed in Section 9.

6.4. Slow Stages

The **prgreater** command will print out information about slow single stages:

prgreater [**-c** *file*] [**-t** *threshold*] *node node ...*

Threshold is a delay time in nanoseconds and defaults to zero. Only single stages with delays greater than *threshold* are printed out. Normally, no nodes are specified, and Crystal searches the entire circuit for single stage delays greater than *threshold.* If nodes are specified, then only delays ending at those nodes are considered. The repeat limit is used to eliminate stages with duplicate delays. The **-c** switch is used to generate a Caesar command file.

6.5. SPICE Decks

The command

spice [**-m**] *file* [*node*]

generates a SPICE deck describing a path. The deck is written into *file* and describes the path to *node* (if *node* is specified), or to the slowest node in the circuit if *node* isn't specified. The **-m** has the same function as in the **worst** command.

The SPICE deck contains 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. You must add your own cards to the deck to generate the Vdd and body bias voltages.

6.6. General Information

Prcapacitance and **prresistance** have similar syntax and are used to print out nodes with large capacitances or resistances:

prcapacitance [**-c** *file*] [**-t** *threshold*] *node node ...*
prresistance [**-c** *file*] [**-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 **-c** switch is used to generate a Caesar command file, and the repeat limit is used to suppress nodes with the identical capacitances or resistances.

The **printallfets** command takes no parameters, and merely prints out Crystal's information for each transistor in the circuit. This command is intended for Crystal debugging; for circuits of any size it will generate enormous amounts of output.

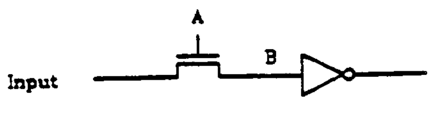


Figure 3. If Input isn't marked as an input, Crystal will assume that it cannot be driven to either zero or one, and it will thus assume that a change at A has no effect on B.

7. Information Commands

7.1. Inputs

The commands described in this section provide Crystal with additional information to control the delay analysis. They are used to limit the range of consequences that Crystal will consider. The **input** command was introduced in Section 4. It serves two purposes. First, input nodes are assumed to be sources of both zero and one signals (unless a **set** command is used to force a particular value). If a node isn't marked as an input then Crystal will assume that it is floating unless it can be driven from within the chip. Crystal does not consider delays that involve floating nodes (see Figure 3). Second, input nodes are assumed to be driven from off-chip: Crystal will calculate delays *to* the node (to handle pads that are used for both input and output), but will not consider delays *from* the node to other points in the circuit unless the node is used in a **delay** command. If a node is used for both input and output, and you want to consider the case where the circuit provides data to itself, then don't use the node in an **input** command. 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.

7.2. Set

The **set** command indicates that a node is fixed in value. Crystal uses this information in a static logic simulation to infer the values of other nodes. For example, if an input of a NAND gate is fixed at 0, then Crystal will deduce that the output must be fixed at 1. If an input of a NOR gate is fixed at 1, then the output must be fixed at 0. When processing **delay** commands, Crystal ignores stages containing enhancement transistors whose gates are turned off. Thus, if a clock signal is set to zero, any logic enabled by that clock signal will be ignored during timing analysis. Furthermore, any node that has been set to a value is assumed to have a worst-case rise or fall time of zero and to be highly capacitive, as if a probe had been placed on chip.

The following rules are used to determine what value a node has when it is driven from several places. A path to ground through enhancement transistors is considered to be stronger than any pullup path, regardless of transistor sizes. Pullups of any type (enhancement or depletion) are considered to be stronger than depletion pulldowns but weaker than enhancement pulldowns. Pullups will drive a node to 1 if there is no possible way for the node to be pulled to zero through an enhancement path. A path to ground through a depletion device is considered weakest of all, regardless of transistor size: it will drive the node to ground only if there is no possible way for the node to be driven to 1.

7.3. 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 8 describes another one that is even more serious. Crystal has two mechanisms to detect when it is doing too much work. First, if it keeps passing through the same node over and over again, then it prints a message to that effect; second, if the total number of stages examined reaches a certain limit (which can be changed with the **-l** command line switch) then Crystal aborts the delay analysis and prints out a backtrace of the path it was examining when it gave up. Whenever the second of these two scenarios occurs, and often when the first scenario occurs, you must provide Crystal with guidance so it can avoid chasing irrelevant paths.

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

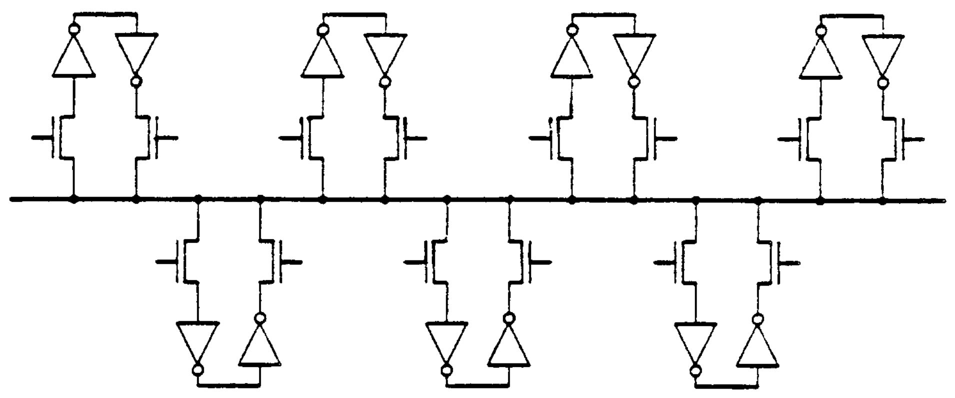


Figure 4. Without any additional information, Crystal will make a separate examination of every path from an output in one cell to an input in another cell. If Crystal knows about the presence of the bus, it first examines all paths from outputs to the bus, then examines paths from the bus to inputs. This makes the analysis much faster.

following syntax:

bus *node node ...*

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 may be underestimated. Crystal automatically marks all nodes with more than 1 pf of capacitance as busses. However, the threshold value can be changed with the **-b** command line switch. To prevent Caesar from automatically marking busses, use a very high threshold.

7.4. Precharged

The command

precharged *node node ...*

indicates to Crystal that the nodes are precharged. 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.

7.5. 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 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 wires that make up a node are in a star configuration, then Crystal will substantially overestimate the resistance. The **resistance** command should be used to correct such situations.

7.8. Clear

The command

clear [-t]

will clear out most of the user-supplied information stored in the database while leaving the circuit intact. In particular, it clears information provided by **inputs**, **set**, **bus**, and **precharged** commands. If the -t switch isn't specified, then all the timing information is also cleared. The -t switch is used when propagating information between clock phases; this is described in Section 9. **Clear** does not affect information set with the **resistance** or **capacitance** commands, nor does it clear information set by the **markdynamic** command, which is discussed in Section 9.

8. Attributes and Pass Transistor Flow

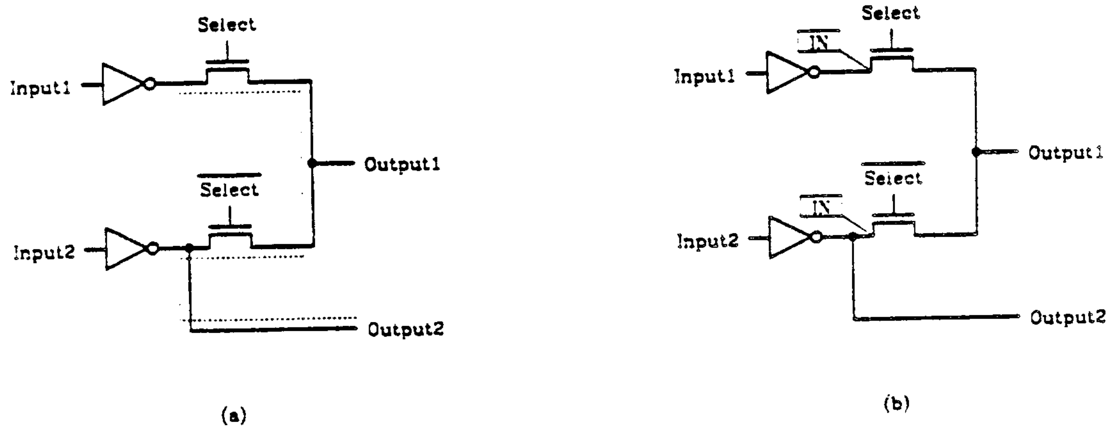


Figure 5. If Crystal doesn't know about pass transistor flow, it will consider the impossible path shown in (a). If the pass transistor flow is labelled with attributes, as in (b), then Crystal will consider paths from **Input1** to **Output1** and from **Input2** to both outputs, but it will not consider the path from **Input1** to **Output2**.

As mentioned in Section 7.3, there are a few situations where Crystal does much 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 will 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 the multiplexor. In Figure 6, there are 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 the paths from occurring in the acutal circuit.

To prevent Crystal from chasing impossible paths, you must tell it which way information flows through pass transistors. By information flow, I mean which way the 1 or 0 value flows, not which way the actual currents flow. Flow is indicated using *transistor attributes* in the CIF files that are input to Mextra and then Crystal. 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

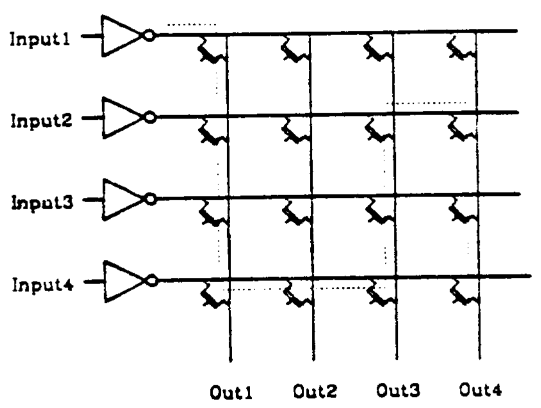


Figure 6. Crystal will consider long snake-like paths through this barrel shifter structure unless pass transistor flow information is provided.

indicate the direction of information 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 desired terminal (attributes placed entirely within the gate region are attached to the gate of the transistor; Crystal does not currently use gate attributes).

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.

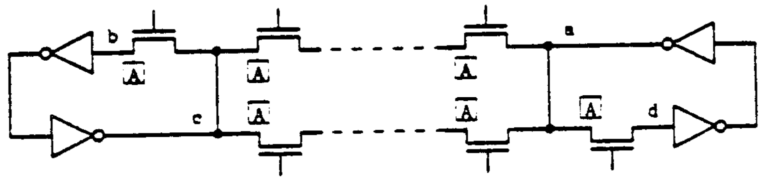


Figure 7. Named attributes can be used to control flow in bidirectional structures. In this case, paths from a to b and from c to d will be considered, but the path from a to c to d will not be considered (flow must be unidirectional with respect to tags of a given name).

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 information may flow through the array: Crystal only allows information to flow unidirectionally with respect to the attributes. This means that Crystal will consider any path through the array as long as the information 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 information enters one transistor from the labelled side and another from the unlabelled 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.

8.1. 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**, or **ignore**. 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 **ignore** is specified, Crystal will pretend that the attribute doesn't exist.

9. Multi-phase Signals and Memory Nodes

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. This section describes how to get Crystal to split the cost of the delay between several phases. These features have only been tested a little bit, and are probably buggier than other Crystal features.

There are two things that must be done to handle multi-phase signals correctly. First, Crystal must know how much of the delay to bill to the clock phase currently being processed. Second, Crystal must be able to keep track of the remaining delay so that it gets charged against the next clock phase.

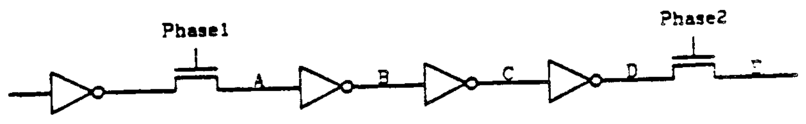


Figure 8. Because node **A** is a memory node, it must be allowed to settle during **Phase1**. Nodes **B**, **C**, and **D** need not settle during **Phase1**: they can settle during **Phase2**. However, if they don't settle during **Phase1**, enough time must be allowed during **Phase2** for them to settle and for the value at **E** to settle also.

Memory nodes determine the delays that are counted in each clock phase. That is, Crystal assumes that if a memory node can possibly be modified in a clock phase, then the phase must be long enough to permit that modification to complete. Nodes containing purely combinational information need not settle during a clock phase as long as all the memory nodes settle. See Figure 8.

9.1. Markdynamic

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 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 **set** commands to electrically isolate the dynamic memory nodes. Then, invoke the

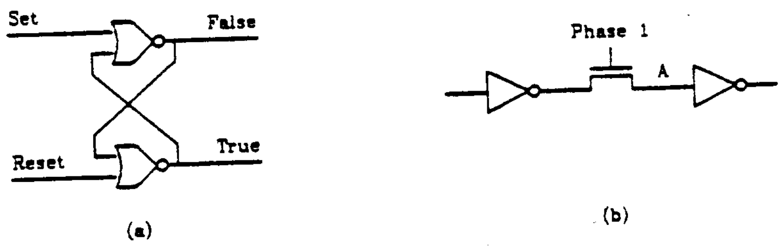


Figure 9. In (a), nodes **False** and **True** are static memory nodes. In (b), **A** is a dynamic memory node.

markdynamic

command. During this command, Crystal searches its database and marks as dynamic memory every node that is electrically isolated (i.e. all of the transistors attached to it are enhancement transistors that are turned off). Typically, dynamic memory nodes are those that are isolated when all of the clock phases are off. If this is the case, then **set** all clock phases to 0 and invoke **markdynamic**. Dynamic node markings are not affected by **clear** commands; only another **markdynamic** command will change them.

Once Crystal knows which nodes are memory, use the **-m** switch in the printing commands **worst**, **manyworst**, and **spice**. When the **-m** switch is used, Crystal will only print-out paths ending in memory nodes. The **-m** switch does not have any effect on delay calculations; it only affects what information is printed out.

9.2. Propagate

Two commands, **clear -t** and **propagate**, are used to pass residual timing information from one clock phase to another so that delays can span the two. The way this works is as follows. The very first commands given to Crystal should mark the dynamic nodes as described above. Then analyze the first clock phase. After the **delay** commands for the first clock phase, print out worst case timing information using the **-m** switch.

To advance to the next clock phase, issue a **clear -t** command. This will clear settings from previous **set**, **input**, **bus**, and similar commands, but will not affect timing information, because of the -t switch. Then issue **input**, **set**, and similar commands to set up for the analysis of the next clock phase. Just before issuing the first **delay** command for the new clock phase, issue a **propagate** command. The **propagate** command allows delays that began under the previous **settings** to continue through the new **settings**.

After the **propagate** command, issue **delay** commands in the usual way, followed by printing commands with the **-m** switch. For additional clock phases the process can be repeated with more **clear -t** and **propagate** commands.

Warning: **propagate** doesn't work very well right now. In fact, I suggest not using it at all. Instead, I suggest invoking first **worst -m** and then **worst**. The first command will tell how long the current clock phase must last, and the second will tell how long the next clock phase must last. This is not strictly accurate, and should be considered 3/4 part kludge, 1/4 part convenience.

10. The Models

The construction of Crystal permits the definition of different delay models for the circuit, each of which may have several user-settable parameters. Currently there are just two models. The normal model is called **rc**, and uses a simple RC model for delay calculations. The second model is called **unit**. In this model, it costs 1ns to turn a transistor on or off, but all other delays are zero.

Parameter Name	Meaning
DiffCPerArea	Capacitance per square micron for diffusion (pfs).
DiffCPerPerim	Capacitance per micron of diffusion perimeter (pfs).
DiffResistance	Resistance per square of diffusion (ohms).
MetalCPerArea	Capacitance per square micron for metal (pfs).
MetalResistance	Resistance per square of metal (ohms).
PolyCPerArea	Capacitance per square micron for polysilicon (pfs).
PolyResistance	Resistance per square for polysilicon (ohms).
FETCPerArea	Gate capacitance per square (pfs).
FETCPerWidth	Gate-source/drain overlap capacitance per micron of gate width (pfs).

Table 1. The parameters that are valid for all models.

Two commands are provided to deal with the models. The command

model [*name*]

will set the current 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

parameter [*name*] [*value*]

is used to see and set parameters for the current model. 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. Several parameters are used in all models; they are listed in Table 1.

10.1. The Unit Model

In the unit model, it takes one nanosecond to turn any transistor on or off. Everything else in the circuit is assumed to be instantaneous. Although several parameters are available under this model, they are all ignored.

Parameter Name	Meaning
EnhRDown	Resistance per square (ohms) of an enhancement transistor pulling to GND.
EnhRUp	Resistance per square (ohms) of an enhancement transistor pulling to Vdd.
LoadRUp	Resistance per square (ohms) of a load transistor (gate tied to source) pulling to Vdd.
DepRDown	Resistance per square (ohms) of a depletion transistor pulling to GND when gate could be at GND.
DepRUp	Resistance per square (ohms) of a depletion transistor pulling to Vdd when gate could be at GND.
SuperRDown	Resistance per square (ohms) of a depletion transistor pulling to GND when gate is known to be at Vdd.
SuperRUp	Resistance per square (ohms) of a depletion transistor pulling to Vdd when gate is known to be at Vdd.

Table 2. The parameters that are valid only for the RC model.

10.2. The RC Model

In the RC model each piece of interconnect is assigned a capacitance and resistance value based on its area and perimeter. Each transistor is assigned a resistance and a capacitance, based on its type, area, perimeter, and whether it is being used to pull high or low. Delays are calculated by summing all resistances and capacitances in a stage and then using the RC time constant as the delay for the stage. The parameters specify how to compute resistances and capacitances. Table 2 lists the parameters that are specific to the RC model.

11. Additional Commands: Help and Source

The **help** command prints out a list of the commands and their parameters. For information on the commands that is more detailed the **help**, and more concise than this document, see the manual page.

The command

source *file*

will cause Crystal to read further commands from *file* until its end is reached. Upon end-of-file, Crystal continues reading from the standard input. Source files may be nested.

12. 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 should add more information commands to help guide Crystal's analysis. Each of the following subsections describes one such command.

12.1. Funny-looking E FET....

This message is issued while reading the .sim file whenever Crystal finds an enhancement transistor whose gate is connected to its source or drain. It is usually caused by a lightning arrestor on an input pad, but you should check to make sure this is the case. The message is just a warning and doesn't damage the timing analysis.

12.2. 100 times through xyz

There are usually several paths through the circuit to each node. During delay analysis Crystal counts how many times it has reached each node, and issues a warning message when it reaches a node for the 100th time. Another message is issued at 1000 times, and still another at 10000 times. These messages are warnings that perhaps more pass transistor flow control is needed. In each circuit there will probably be a few nodes that can be reached 100 different ways (particularly in bus or barrel shifter structures). However, there are rarely any nodes that can be reached through 1000 different valid paths.

12.3. Aborting: no solution after 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 **-l** command line switch 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 **worstdelay** and similar commands may not be accurate, since the delay analysis wasn't completed. However, the path provided by **worstdelay** may indicate the place where more flow control is needed.

12.4. There is no path to x

This message appears occasionally in **worstdelay** and similar commands. It means that Crystal could not find a delay path to the given node. It usually signifies that the delay to that node is instantaneous, or if you are using **propagate** commands it means that the delay to the node was determined by the previous clock phase.

12.5. 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.

13. Bugs

1. Because node capacitances and resistances are computed as the circuit is read in, and because this happens before any model parameters can be changed, it is currently impossible to change the parameters that determine capacitance and resistance for metal, poly, and diffusion.

2. The Crystal bug message "Couldn't trace back from ...". occasionally appears in **worst**, **manyworst**, and **spice** commands. This is the result of a nasty bug that I understand but haven't found a fix for. I know of no way to get around this one, except to disable the offending path with **set** commands. Unfortunately, this bug seems to occur with the standard nMOS pads.

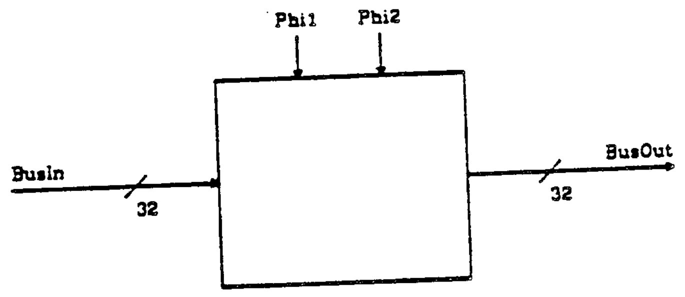


Figure 10. A simple circuit with two non-overlapping clock phases, 32 data inputs, and 32 data outputs.

14. An Example

For the circuit of Figure 10, 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
set Phi2 0
delay Phi1 0 -1
delay BusIn<0:31> 20 20
worst -c phi1cmds

clear
inputs BusIn<0:31> Phi1 Phi2
set Phi1 0
delay Phi2 0 -1
worst -c phi2cmds
```

Specifying Design Rules for Lyra

Michael H. Arnold
Computer Science Division
University of California
Berkeley, CA 94720

Abstract

The Lyra layout rule checker can be retargeted to new design rules, permitting Lyra to be used with multiple technologies and processes and also making it possible to track design rules as they change over time. To adapt Lyra to a new ruleset, a symbolic ruleset specification is written and then compiled with the Lyra rule compiler to generate an executable module for checking the new rules. Rules are specified in terms of Lyra's corner based paradigm: Each rule gives a context specifying corners where it applies, and a set of constraints to be applied at these corners. Complex or unusual rules can be coded directly in terms of a primitive rule construct. Common checks, such as width and spacing, are coded more concisely using rule macros.

This manual gives the details of writing rulesets for Lyra. All the basic constructs for rule specification are explained, and the rule macros are presented. Compiling, testing and installation of rulesets are also discussed.

The work described here was supported in part by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 3803, monitored by the Naval Electronic System Command under Contract No. N00039-81-K-0251

1. Introduction

The Lyra layout rule checker can be retargeted to new design rulesets. This is important because design rules for many different technologies and processes are in use, and because rules change over time.

To retarget Lyra for a new ruleset, it is necessary first to prepare a ruleset specification in the Lyra format. This rule specification can then be compiled, by the Lyra rule compiler, to generate an executable file for checking the specified rules. This executable file is invoked by the Lyra frontend to check designs against the ruleset.

This document explains how to write design rule specifications for Lyra. The next section sketches the basic paradigm for rules in Lyra. The following three sections give all the details (syntax and semantics) of rule specifications. The final section gives suggestions for writing, compiling and testing Lyra rulesets.

2. The Idea: Corner Based Rule Checking

In Lyra, rules are given as constraints to be applied at corners. Each rule gives a context describing the corners it applies to and a set of constraints to be checked at these corners. Constraints are rectangular regions where some combination of layers are required or prohibited. A rule to check spacing on a metal layer, M, would give constraints to be applied around the outside of corners on layer M disallowing M in the vicinity. This is illustrated in Figure 1. In the figure the hatched constraints have been violated indicating the two geometries are spaced too closely. Figure 2 suggests graphically the form the spacing rule takes in Lyra. The left hand side characterizes convex corners on layer M, namely M is present in one quadrant (the upper right) but not in adjacent corners. The right hand side of the rule gives a cluster of constraints to be applied at all corners matching the left hand side of the rule. The other corner orientations, obtained by rotating this rule, are implied.

When complete rulesets are considered, several complications arise. It is not sufficient to deal only with corners on single mask layers. Some rules refer to corners defined by the interaction of mask layers, for example transistor rules in nMOS involve the corners generated by crossing polysilicon and diffusion features. Some rules do not apply at all corners on a layer, but only if some additional layer pattern is present at the corner. For instance the Lyon implant rules used in the Berkeley nMOS ruleset require one set of constraints to be applied at transistor corners in the direction of polysilicon and another in the direction of diffusion. Such rules require the specification of additional context information in the left hand side of the rule. Finally, for completeness, it is necessary to consider concave corners in addition to convex ones. However all of these situations can easily be cast in terms of the rule format suggested by Figure 2. Surprisingly, this simple paradigm can be used to accurately express almost all design rules. Exceptions are a few rules involving connectivity, and some of the more involved industrial rules.

In Lyra rule specifications, composite layers are defined for rules which do not apply to corners on actual mask layers, and the left hand sides of rules are split into two parts: a *corner specification* giving the layer and convexity of corners to which the rule applies, and an *also clause* which gives any additional pattern of layers required for the rule to apply. A set of rule

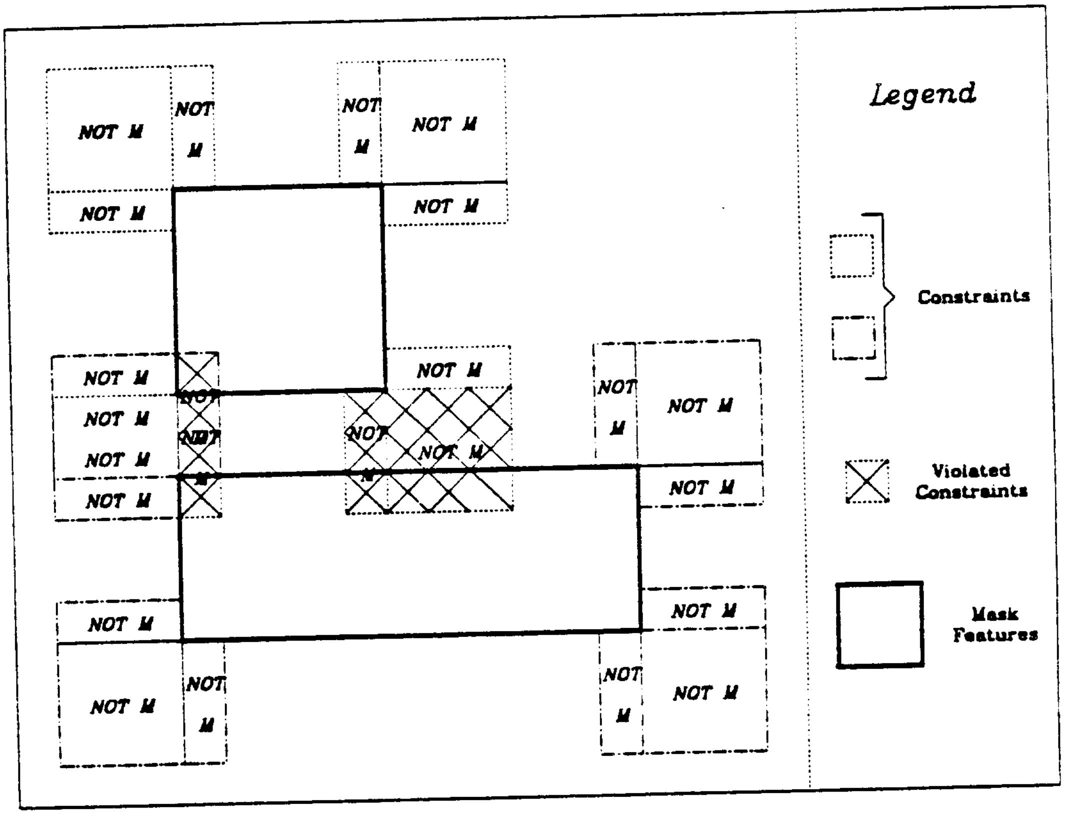


Figure 1. Metal Spacing Constraints.

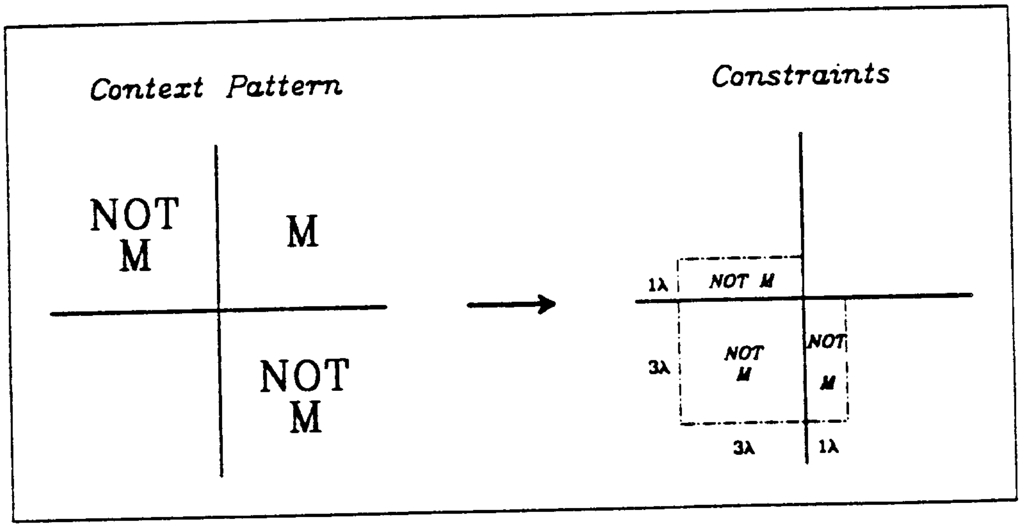


Figure 2. Metal Spacing Rule.

macros is used for concise coding of the more common rules, e.g. width and spacing.

This is the idea of the corner based paradigm; the following sections give the details.

3. Definitions

This section defines the basic constructs involved in rule specifications for Lyra. The first part of the section deals with syntax, and dimensional units. Then the layer declaration are defined. These preceed the rule specifications in Lyra rulefiles. The last part of the section develops the rule construct which is used directly to specify the more complex or unusual design rules, and to which the macros given in the next section expand. See section 5 for the overall organization of rulefiles.

3.1. Lisp Syntax

Lisp syntax is used in the Lyra rule files. This means constructs have the general form,

(<*construct name*> <*expr1*> <*expr2*> ... <*exprn*>)

The expressions, <*expr1*>, ..., <*exprn*> may be numbers, layer names, text, or subconstructs. The use of constructs within constructs leads to nested parenthesized lists. For example, a pair of rules concerning transistor form in nmosMC, the Mead & Conway nMOS rules are shown below (the details are not important).

```
; malformed green-transistor abuttment
(rule
 (corner: (a G))
 (also: nil (not D) (or D (not P)) (and P D (not X)))
 (constraints: (build-constraints (quad3 1 1) false "tr f")))

;malformed red-transistor abuttment
(rule
 (corner: (a R))
 (also: nil (not D) (or P (not D)) (and P D (not X)))
 (constraints: (build-constraints (quad3 1 1) false "tr f")))
```

Note the use of semicolons to delimit comments. The lisp parser treats all text between a semicolon and the end of the line as a comment.

The rule files are actually executed as lisp code by the rule compiler. Care must be taken to balance parentheses properly. If parentheses are not balanced, cryptic error messages from the lisp parser will occur when compilation is attempted.

Many editors provide special functions for editing lisp files. In Vi it is useful to set *lisp* and *showmatch* mode:

:set lisp sm

When *showmatch* is set, typing a closing parenthesis causes the cursor to momentarily jumps to the matching open parenthesis. In *lisp* mode, lisp indentation style is supported (e.g. with the *open* command), you can move from any parenthesis to its matching parenthesis by typing '**%**', and the indentation of an expression can be made to correspond to the parenthesis structure by placing the cursor at the beginning of the expression and typing

'%='.

3.2. Units

Distances are specified in the same units used in the input Caesar files, and should be integral. Caesar's internal units are half as big as the units apparent to the user. Thus normally one unit corresponds to 1/2 lambda.

Since Lyra is not raster based, distances can be scaled up, say for greater precision, without significantly affecting performance. Note however that numbers with absolute value less than 1024 are stored more efficiently in Franz Lisp, and thus in Lyra, than larger numbers.

3.3. Layers and Predicates

A *rule* in Lyra consists of a left hand side (*LHS*) specifying the context in which the rule applys and a right hand side (*RHS*) specifying *constraints* which apply wherever the LHS holds. Both the LHS and RHS of rules involve *layers* and combinations of layers. Layers come in three varieties: *primary layers*, *grown layers*, and *composite layers*. *Predicates* are used to specify layer combinations.

3.3.1. Primary Layers

Primary layers are just the mask layers of the technology to be checked. Primary layers are specified as follows:

```
(primary-layers
 (<internal name> (<Caesar name> <cif name>))
 ...
)
```

It is convenient to choose short *internal names* (one or two characters), and to capitalize the first character. The *cif name* is provided so that Lyra can accept *cif names* in place of *Caesar names* in the input file. This simplifys the interface between Lyra and programs not using the Caesar data format, such as KIC. The primary layer specification for nmosMC looks like this:

```
(primary-layers
 (P (polysilicon NP))
 (D (diffusion ND))
 (M (metal NM))
 (I (implant NI))
 (C (cut NC)))
```

3.3.2. Grown Layers

A *grown layer* is generated from a *primary layer* by expanding each rectangle on the specified primary layer by a specified amount. Here is the form of the grown layers specification:

```
(grown-layers
 (<grown layer name> <primary layer> <amount>)
 ...
)
```

<Grown layer name> gives a name for the new layer. *<Primary layer>* is the internal name of a primary layer. Each edge of each rectangle on *<primary layer>* is shifter out by *<amount>* units to create the corresponding

rectangle on layer *<grown layer name>*.

Grown layers are created prior to all other processing in Lyra, and once created they are treated exactly as primary layers are. Note that grown layers are internal to Lyra only: they are never output.

Grown layers should be used cautiously since they increase the memory requirements for processing designs substantially. Also, the largest *amount grown* is added to the largest *constraint size* in a ruleset to determine the *design rule interaction distance*. If the design rule interaction distance is large, hierarchical processing will suffer.

Currently only the nmosMC rules make use of a grown layer. In the nmosMC rules the contact cut layer is grown out by one lambda, to provide sufficient context to distinguish certain corners in butting contacts from similar corners in badly formed transistors. The specification of this grown layer in the nmosMC rules is:

```
(grown-layers
 (X C 2)) ; X = cut grown by 1 lambda = cut-context layer
```

Remember that two units correspond to one lambda.

3.3.3. Predicates

Predicates define combinations of layers. They are used in specifying *Composite Layers*, in specifying the *context patterns* in the LHS of rules, and in defining *constraints* in the RHS of rules.

Predicates have the following forms:

<Primary or Grown Layer>,
(not *<Predicate>*),
(and *<Predicate> <Predicate>* ...), and
(or *<Predicate> <Predicate>* ...).

Arbitrarily complex layer combinations can be defined in this way. Some examples of predicates are,

```
M              ;Presence of layer M
(not M)        ;Absence of M
(and P D (not C))  ;P and D both present, but not C.
```

3.3.4. Composite Layers

In Lyra each rule applies to corners on a specific layer. Many rules apply to corners on primary layers, but some rules refer to corners resulting from the interaction of layers. These rules occur at corners on a composite layer defined as a combination of primary (and grown) layers. These composite layers must be defined at the front of the rules file by a specification of the following form:

```
(composite-layers
 (<composite layer name> <defining predicate>)
 ...
 )
```

For example the composite layers specification for the nmosMC rules looks like this:

```
(composite-layers
 (R (and P (or X (not D))))     ; red
 (G (and D (not P)))            ; green
 (T (and P D (not X)))          ; transistor
 (E (and P D (not I) (not X)))  ; enhancement mode transistor
 (Z (and P D I (not X)))        ; depletion mode transistor
 (PC (and P C))                 ; poly-cut
 (GC (and D C (not P)))         ; dif-cut
 (Xbad (and X (or (not M) (not (or P D)))))) ; bad cover over contact
```

Note that unlike grown layers, composite layers are not actually constructed in the data base; they simply refer to combinations of primary and grown layers.

3.3.5. Layers

Once specified, primary-layers, grown-layers and composite-layers are all used in identical fashion. When the term *layer* is used below it is meant to encompass all three types of layers.

3.4. LHS

The *LHS* of a rule defines the context in which the rule applies. It gives the relevant *corner layer* (the layer on which corners are to be examined) a *corner type* (convex or concave) and any additional pattern of layers that must occur at a corner for the rule to apply. This section gives the details involved in specifying the LHS's of rules.

3.4.1. Corners

Each rule refers to corners on a specific layer (primary, grown or composite) and of a specific type (either convex or concave). Convex corners, such as those on a square are designated by the letter '**a**' (read acute — though technically a misnomer). Concave corners, such as those inside a U-shape, are designated '**o**' (read obtuse). Corner specifications in rules have the form,

(corner: <*type*> <*layer*>)

Where <*type*> is '**a**' for convex corners, and '**o**' for concave corners as explained above. Examples:

```
(corner: a M) ; rule is to apply to convex corners on layer M.
(corner: o X) ; rule is to apply to concave corners on layer X.
```

3.4.2. Canonical Orientation and Quadrant Numbers In addition to the corner layer and type explained above, it is sometimes desirable to specify an additional pattern of layers which must occur at a corner. To express this additional pattern information, and also to express the location of the constraints in the RHS of the rule, horizontal and vertical lines are drawn through the corner under consideration, dividing space into four quadrants. Following standard convention, the quadrants are numbered, by designating the upper right quadrant the first quadrant and counting counter clockwise from there (see Figure 4).

Since a corner can be oriented in four ways with respect to these quadrants, it is necessary to choose a fixed *canonical orientation*, so that there

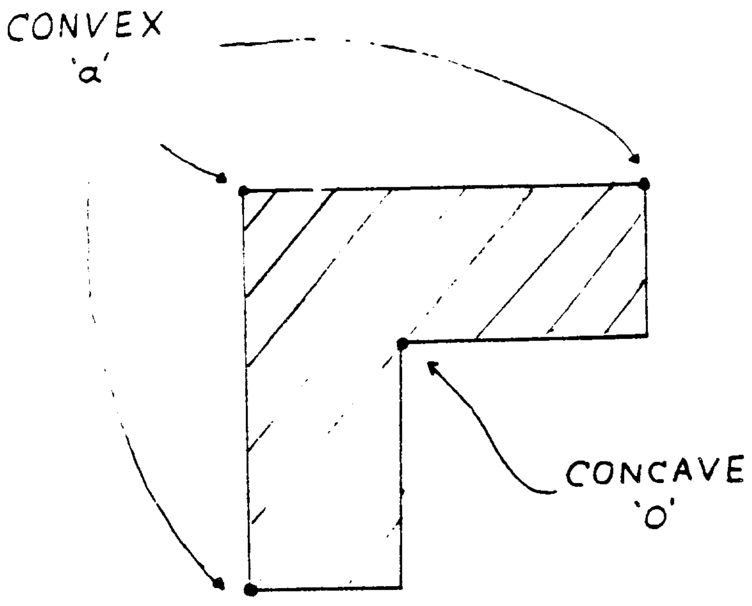


Figure 3. The Two Corner Types.

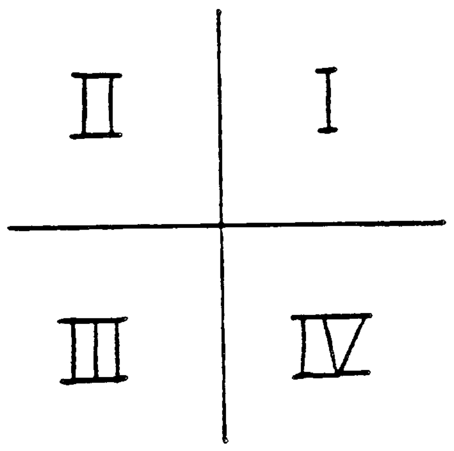


Figure 4. Quadrant Numbering.

is a definite relationship between corners and the information specified in terms of the quadrant system. In Lyra rule specifications, corners are always assumed to be centered on the first quadrant, as illustrated in Figure 5. The canonical orientation is used only to facilitate the specification of rules: the rules apply equally to corners of all four orientations.

3.4.3. Also

The *also* clause can be used to specify combinations of layers which must be present (or absent) in each of the four quadrants immediately adjacent to a corner. A rule applies to a corner only if all also conditions are satisfied. The *also* **clause has the form,**

```
(also: <predicate for quadrant 1>
       <predicate for quadrant 2>
         <predicate for quadrant 3>
       <predicate for quadrant 4>)
```

Predicates are defined in a previous section. If there are no additional context requirements on a quadrant, **'nil'** can be used in place of a predicate. If the conditions on the second and fourth quadrant are distinct, a mirrored version of the rule is automatically generated by the rule compiler, so that all eight symmetries (orientations) of the rule will be checked. The also clause is optional; in many cases a rule applies to all corners of the layer and type specified in the corner clause, and an also clause is not needed.

The corner and also clauses together constitute the LHS's of rules.

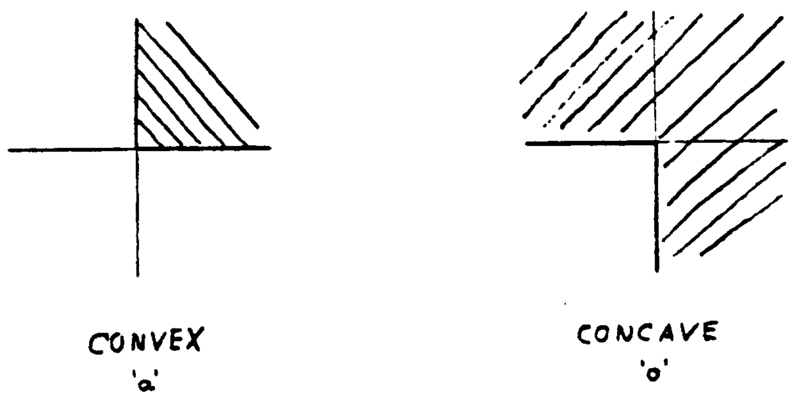


Figure 5. Canonical Corner Orientation.

3.5. RHS

The RHS of a rule gives one or more constraints which are checked at all corners which match the LHS of the rule. Constraints are defined in terms of a rectangular region and a predicate. One corner of the constraint rectangle always coincides with the corner in the design that the rule is being applied to. This corner is called the generating corner for the constraint. If the predicate does not hold throughout the interior of the constraint rectangle, the constraint is violated, and a design rule violation is reported. Each constraint also has an associated text string which is output with violation reports to identify the nature of the violation.

Constraint rectangles can be specified directly by giving the quadrants in which they are located and their dimensions. Several macros are also provided for the most common constraint configurations. The details of specifying constraints are given below.

3.5.1. Violation Text

By convention Lyra's violation messages have the form:

"< *Layers or Constructs* >_< *Type* >"

<*Layers or Constructs*> gives the single character abbreviations for the layers involved in the violation. Circuit constructs such as transistors and buried contacts may also be indicated by short abbreviations (e.g. **tr** for transistor; **Bc** for buried contact). <*Type*>'s are one or two characters indicating the type of error as follows:

s = minimum spacing violation,

w = minimum width violation,

pe = parallel edge spacing violation,

x = insufficient extension or enclosure,

p = polarity, e.g. diffusion doping doesn't match well in CMOS,

f = malformed circuit construct.

For example, the text string for a spacing violation between polysilicon and diffusion would look like this:

"P/D_s".

New types can of course be invented if none of the ones listed fits. We have found it best to keep violation messages short, since long messages tend to overlap each other in the graphical output and become illegible.

The quotation marks delimiting the text string are for the benefit of the lisp parser, they do not appear in violation reports. Violations are actually reported as Caesar labels with the label texts corresponding to the violation messages, and the label boxes corresponding to the violated constraint regions. An explanation mark, '!', is automatically prepended to the beginning of the all violation messages so that violations can be easily located with the Caesar *search* command.

3.5.2. Direct Specification of Constraints

If the RHS of a rule contains only one constraint, it can be specified using the *build-constraints* construct as follows:

```
(constraints:
 (build-constraints
  (quad<i> <x dimension> <y dimension>)
  <predicate>
  <text>))
```

Here *<i>* gives the quadrant number of the constraint: 1,2,3 or 4. *<X dimension>* and *<y dimension>* give the dimensions of the constraint. *<Predicate>* is the predicate required to hold inside the constraint region. An always-violated constraint can be coded by using '**false**' for *<predicate>*. This is used in rules where the LHS gives a corner pattern which should not occur. *<text>* is a quoted text string describing the design rule violation, as explained above. *Append* is used to specify multiple constraints for a rule:

```
(constraints:
  (append
   (build-constraints ...)
   (build-constraints ...)
   ...
   ))
```

Where each *build-constraints* construct is structured as above. The following example is from the CMOS rules, cmos-pwJPL.

```
(constraints:
 (append
  (build-constraints (quad1 6 1) C "sc f")
  (build-constraints (quad2 6 1) C "sc f")))
```

3.5.3. Constraint Specification using Macros

Some frequently occurring constraint configurations can be specified using macros. These specifications take the general form,

```
(constraints:
 (<constraint macro> <dimension> <predicate> <text>))
```

Where *<constraint macro>* is one of: *inside*, *outside*, *e-inside*, *e-outside*, *s-inside*, and *s-outside*. All constraint dimensions are either *<dimension>* or one unit. *<Predicate>* and *<text>* specify a common predicate and text for the generated constraints. Figure 6 shows the constraint configurations corresponding to the six macros. Remember that the canonical corner orientation is assumed when specifying constraints.

3.6. Rules

We have now defined all the pieces necessary to specify rules in Lyra. As we have seen, the LHS's of rules consist of corner and also constructs which define the context in which the rules apply. The RHS's of rules consist of constraint constructs which define constraints on corners where the rules apply. The general form of the rule construct is:

```
(rule
 (corner: ...)
 [(also: ...)]
```

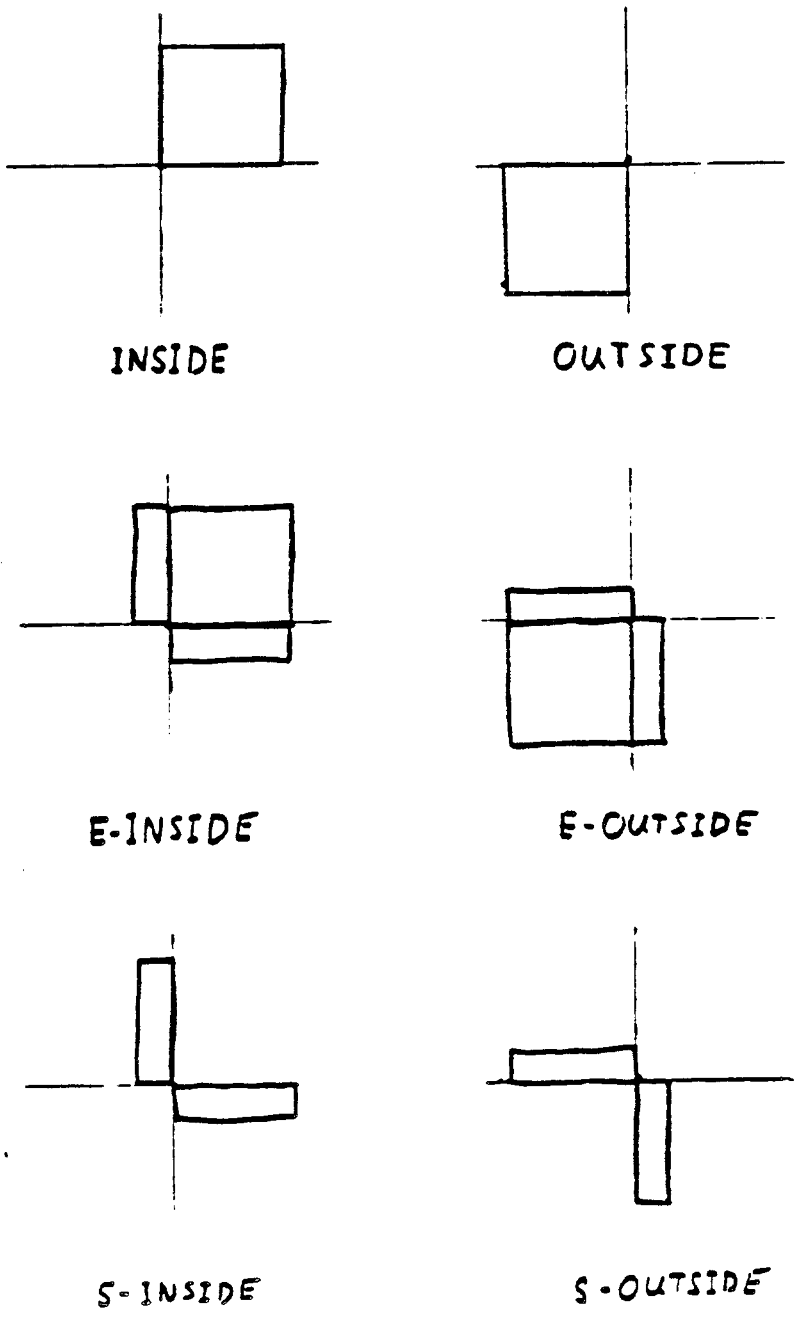


Figure 6. Constraint Macros.

(constraints: ...))

Where the *also* construct is optional as suggested by the brackets. For example, three rules from the nmosMC rules concerning the form of poly/metal contacts look like this:

```
;c. Poly-Cut extension (complicated to handle butting contacts)
(rule
        (corner: (o P))
        (constraints: (e-inside 2 (not PC) "P/C x")) )
(rule
        (corner: (a PC))
        (also: nil (not GC) nil (not GC))
        (constraints: (e-outside 2 P "P/C x")) )
(rule
        (corner: (a PC))
        (also: nil nil nil GC)
        (constraints: (build-constraints (quad2 2 1) P "P/C x")))
```

4. Rule Macros

This section gives the rule macros provided with the Lyra system. These macros allow the easy and concise coding of many common rules. The expansion of each macro is given both in terms of the rule construct of the last section and graphically. In addition to defining the macros precisely, these expansions are good examples of the use of the rule construct and the corner based paradigm.

Each macro takes one or two layers as arguments. These layers can be of any type: primary, grown or composite.

4.1. width

The width macro has the form:

(width *<layer> <dimension> <text>***)**

This macro checks that *<layer>* is at least *<dimension>* wide everywhere. The macro expands into two rules, one for concave corners on *<layer>* and one for convex corners on *<layer>*. The second rule is necessary to check that a pair of closely placed holes in *<layer>* do not result in a narrow strip. *(width P 4 "P_w")* expands to:

```
(rule
  (corner: (a P))
  (constraints: (inside 4 P "P_W")))

(rule
  (corner: (o P))
  (constraints: (e-inside 4 P "P_W)))
```

These rules are shown graphically in Figure 7.

4.2. ss

Single layer spacing can be specified with the *ss* macro. This macro has the form:

(ss *<layer> <dimension> <text>***)**

This macro checks for a spacing of <dimension> between mask features on

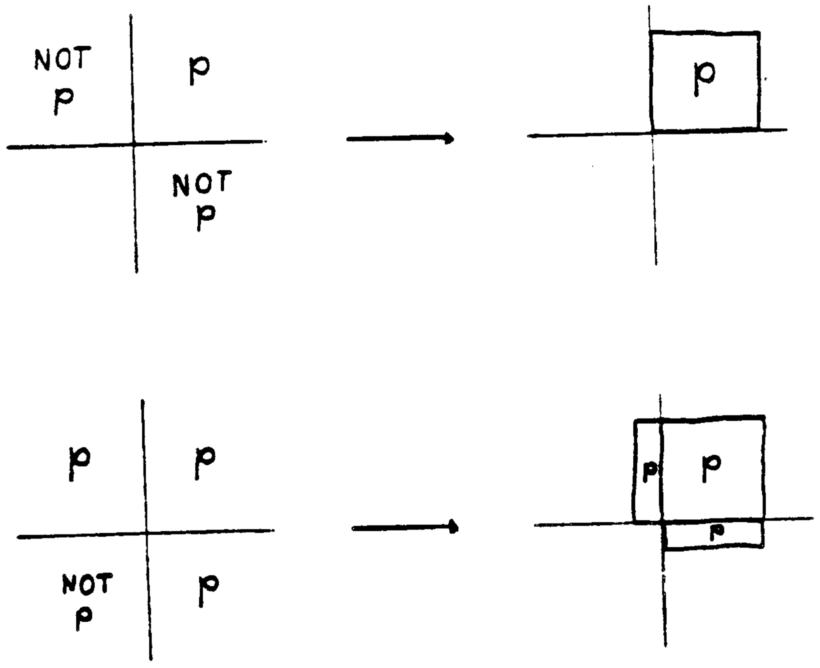


Figure 7. Width Rules. (width)

<layer>. Like the *width* macro, *ss* expands to two rules: one for convex corners and one for concave. The second rule checks for small holes in *<layer>*.

(ss P 4 "P_s") expands to:

```
(rule
 (corner: (a P))
 (constraints: (e-outside 4 (not P) "P_w")))

(rule
 (corner: (o P))
 (constraints: (outside 4 (not P) "P_w")))
```

This is shown graphically in Figure 8.

4.3. pe

Parallel edge checks can be done with the *pe* macro.

(pe *<layer> <dimension> <text>***)**

This macro checks spacing between adjacent feature edges on a layer. The difference between parallel edge checks and spacing or width checks is that the parallel edge checks are not concerned with diagonal spacing. These checks are used to guard against thin slivers of resist during fabrication. Such thin slivers could break off and be deposited somewhere else on the design causing damage.

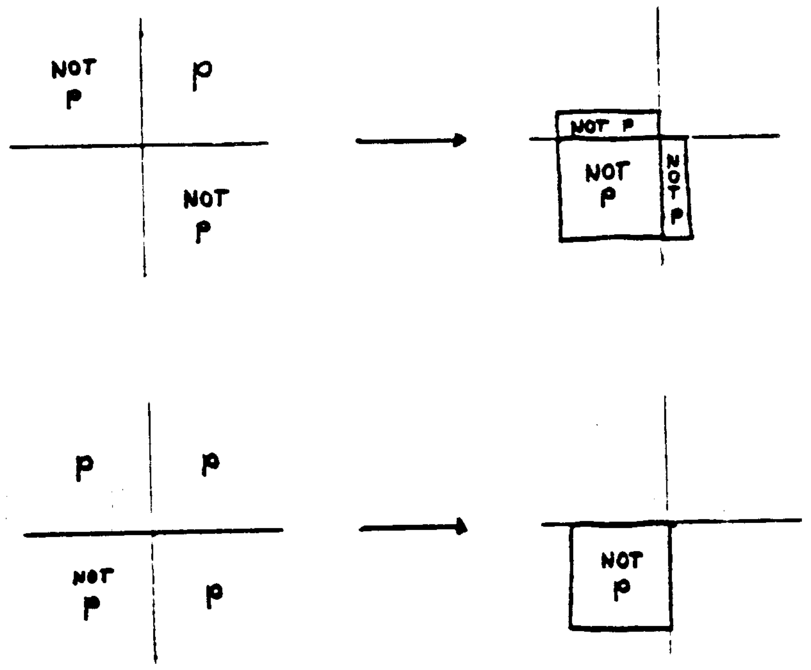


Figure 8. Single Layer Spacing Rules. (ss)

The *pe* macro expands to two rules, one for convex corners, and the other for concave corners. For example, **(pe I 4 "I_pe")** has the following expansion:

```
(rule
 (corner: (a I))
 (constraints:
  (append
   (build-constraints (quad1 4 1) I "I_pe")
   (build-constraints (quad1 1 4) I "I_pe")
   (build-constraints (quad2 4 1) (not I) "I_pe")
   (build-constraints (quad4 1 4) (not I) "I_pe"))))

(rule
 (corner: (o I))
 (constraints:
  (append
   (build-constraints (quad2 1 4) I "I_pe")
   (build-constraints (quad4 4 1) I "I_pe")
   (build-constraints (quad3 4 1) (not I) "I_pe")
   (build-constraints (quad3 1 4) (not I) "I_pe"))))
```

This is shown graphically in Figure 9.

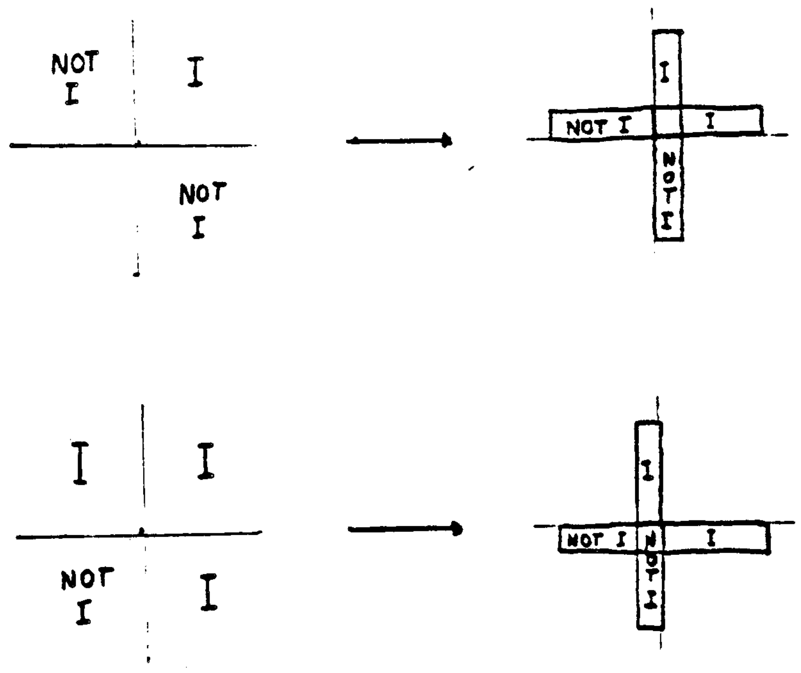


Figure 9. Parallel Edge Rules. (pe)

4.3.1. sep

The sep macro can be used to check interlayer spacing.

(sep *<layer 1> <layer 2> <dimension> <text>***)**

This checks that *<layer 1>* and *<layer 2>* are spaced at least *<dimension>* apart, and where this is not so reports violations with message *<text>*. *Sep* assumes *<layer 1>* and *<layer 2>* do not overlap. In some cases the two layers are logically disjoint and this condition need not be checked. This is true, for example, if *<layer 1>* is nonimplanted gate regions, and *<layer 2>* is implant. If the two layers are not logically disjoint, the *sep* test must be augmented with a disjointness test. An easy way to check that for this is to define a composite layer to be the overlap of *<layer 1>* and *<layer 2>*, e.g. **(and** *<layer 1> <layer 2>***)**, and then write a rule that generates violations at all convex corners of the composite layer.

The expansion of **(sep A B 4 "AB_s")** is:

```
(rule
 (corner: (a A))
 (constraints: (outside 4 (not B) "AB_s")))

(rule
 (corner: (a B))
 (constraints: (s-outside 4 (not A) "AB_s")))
```

This is shown graphically in Figure 10. Note that the constraints on layer A and layer B are not symmetrical.

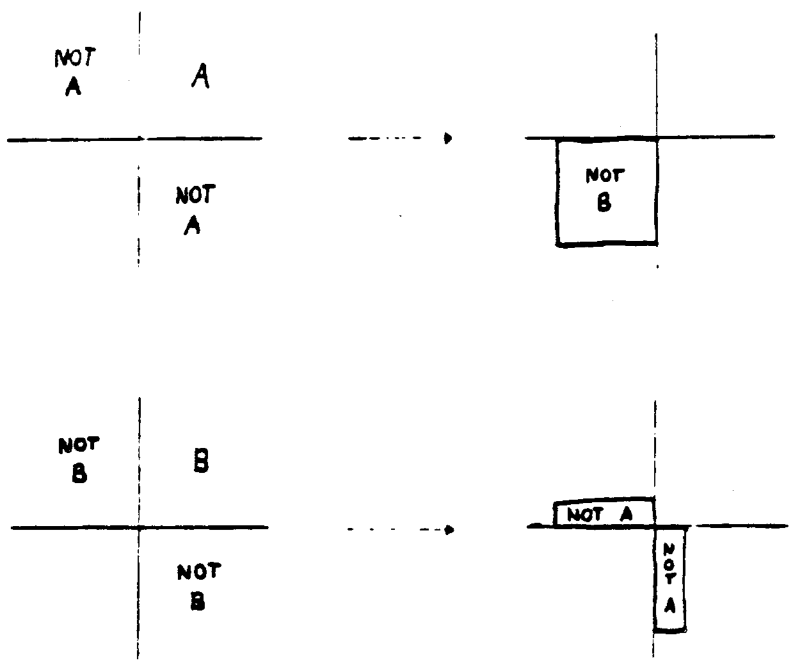


Figure 10. Interlayer Spacing Rules. (sep)

4.3.2. ext

Enclosure rules can be written using the *ext* macro.

(ext *<layer 1> <layer 2> <dimension> <text>***)**

This specifies that *<layer 1>* must extend beyond *<layer 2>* by *<dimension>* in all directions. **The** *ext* macro assumes *<layer 1>* covers *<layer 2>* everywhere. If this is not logically required by the definitions of *<layer 1>* and *<layer 2>*, then it must be checked for separately. This can be done by looking for the existence of a composite layer **(and** *<layer 1>* **(not** *<layer 2>***))**. The expansion of **(ext A B 4 "AB_x")** looks like this:

```
(rule
 (corner: (a B))
 (constraints: (outside 4 A "AB_x")))

(rule
 (corner: (o A))
 (constraints: (s-inside 4 (not B) "AB_x")))
```

This is shown graphically in Figure 11.

5. Rule File Organization and Naming

The overall organization of Lyra rule files is as follows:

1. Primary layer specification
2. Grown layer specification

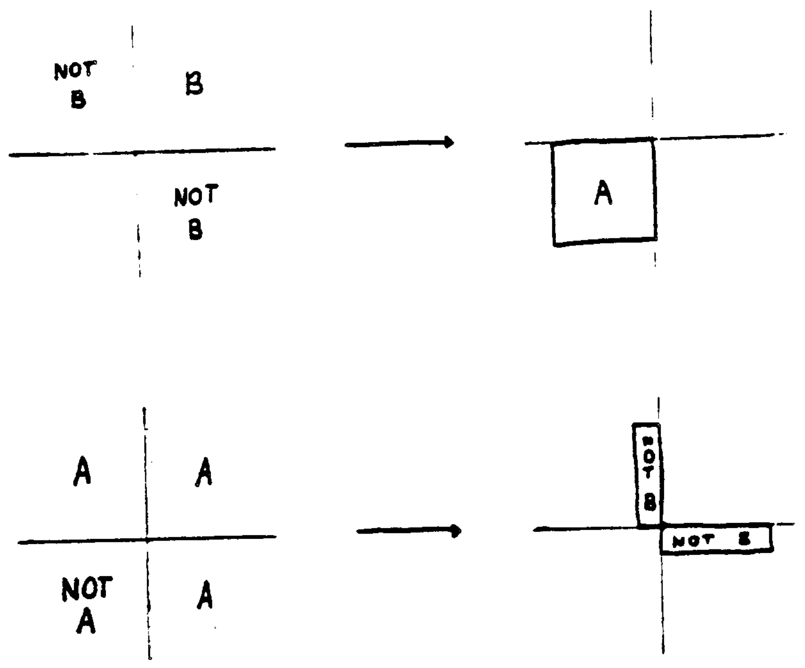


Figure 11. Enclosure Rules. (ext)

3. Composite layer specification
4. Rule constructs and rule macros

All but the primary layer specification is optional.

By convention rulesets are named by appending an indication of the source (in capital letters) to the name of the technology (as known to Caesar). The symbolic rule files are given extension '.r'. Thus: nmosMC.r, nmosBERK.r, and cmos-pwJPL.r are the source files for the Mead & Conway nMOS rules, the Berkeley nMOS rules, and the JPL CMOS rules respectively. These rulesets can be found in ~cad/lib/lyra.

6. Writing, Compiling, and Testing Rule Files

The Lyra rule compiler, *Rulec*, is used to generate an executable Lyra from a ruleset file. Rulec is a shell script which invokes three processing steps:

(1) Compile rule file to lisp code (Rulec1)

(2) Compile lisp code to object code (Liszt)

(3) Link rule specific object code with a Lisp containing rest of the Lyra code to generate an executable Lyra.

Together, the three steps take about 10 cpu minutes for a typical ruleset. Syntax errors in the input file will be caught by the lisp parser during step 1, and will result in strange error messages. Many text editors have special features to help balance parentheses properly in lisp code. The section on

Lisp Syntax above describes such features in Vi. The executable Lyra produced in the third step will have the same name as the input file, but without the '**.r**' extension.

The best way to go about writing a new ruleset is to copy and then modify an existing ruleset. The nmosMC, nmosBERK and cmos-pwJPL rulesets can be found in **~cad/lib/lyra**. It would be helpful to read through these rulesets carefully to see how some of the more complicated rules are handled. You can work in **~cad/lib/lyra** or in your own area. Note however that each executable Lyra produced by Rulec is about one megabyte big!

After a ruleset is compiled , it can be tested in batch mode by giving Lyra the **-r** option with the full pathname of the executable for your ruleset.

```
lyra -r ~me/myrules testfile.ca
```

Your ruleset can also be invoked interactively from Caesar by giving the full pathname of your file as an argument to the Caesar *lyra* subcommand.

```
:lyra ~me/myrules
```

It is a good idea to exercise each new rule both with cases that should pass the rule and cases that should violate it. It is useful to keep around the test files you create for your rules. They can be rerun at any time to see if anything has been broken. One way to quickly generate test cases is to make multiple copies of some structure and then modify each copy in a different way.

A ruleset can be made the default for a given Caesar technology by adding an entry to ~cad/lib/lyra/DEFAULTS. You can also create a personal default by setting the Lyra '**r**' option in the .cadrc file in your home directory.

More details on *Rulec*, *Lyra*, and *Caesar* are given in the (cad)man pages for these programs and in the Caesar manual.

Designing Finite State Machines with PEG

Gordon Hamachi
University of California at Berkeley

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 (PLA Equation Generator) 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 *tpla* [Mayo], *mkpla* [Landman], *presto* [Fang], and *plasort* [Kleckner&Landman]. As an option, *PEG* will also print the unminimized truth table from which the logic equations are derrived.

2. A Simple Example

Designing finite state machines using *PEG* is introduced with a very simple example. Figure 1 shows the state diagram for a four-state machine implementing a 2-bit binary counter. The *PEG* program implementing this design is shown in figure 2.

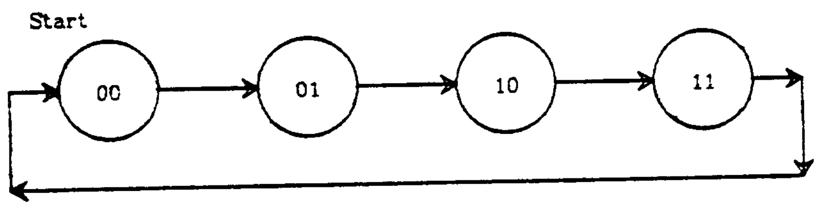


Figure 1: State Diagram for Example 1

```
--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 *PEG* program in figure 2 consists of four state descriptions. The program has no inputs. The outputs of the state machine are its *next state* bits, which are automatically generated by *PEG*.

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.

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 | mkpla -i -o -y 2
```

The digit *2* appears on the command line as an argument to ***mkpla*** to indicate that there are 2 state bits to be fed back from the output to the input of the PLA. In order specify the number of state bits required, it may be necessary to run *PEG* twice: once to determine the number, and another time to actually generate the PLA.

```
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 0	s 1	n 1	n 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 grammar (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

```
peg prog
```

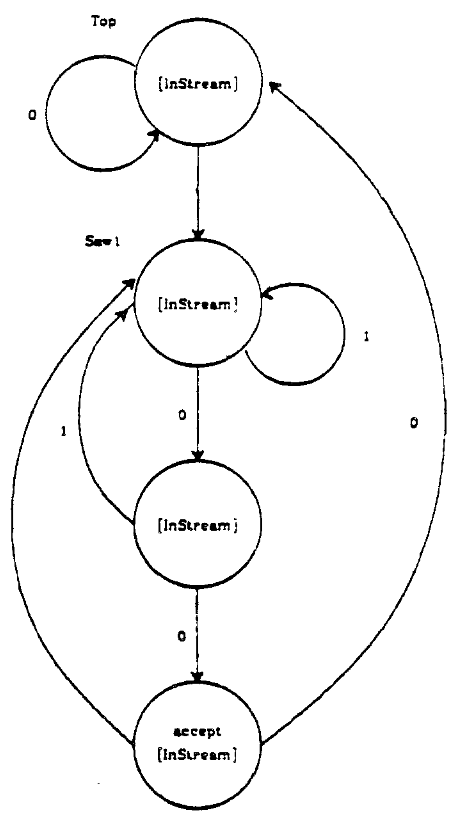


Figure 5: State Diagram Recognizing (1|0)*100

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 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

```
--Simple FSM example:  Accepts the grammar (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.

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.

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>*] ;

INPUTS:	i00:	RESET
	i01:	InStream
	s00:	InSt0* (msb)
	s01:	InSt1* (lsb)
OUTPUTS:	n01:	OutSt1* (lsb)
	n00:	OutSt0* (msb)
	o00:	Accept

State Table	i	i	s	s	n	n	o
	0	1	0	1	1	0	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.

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.

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.

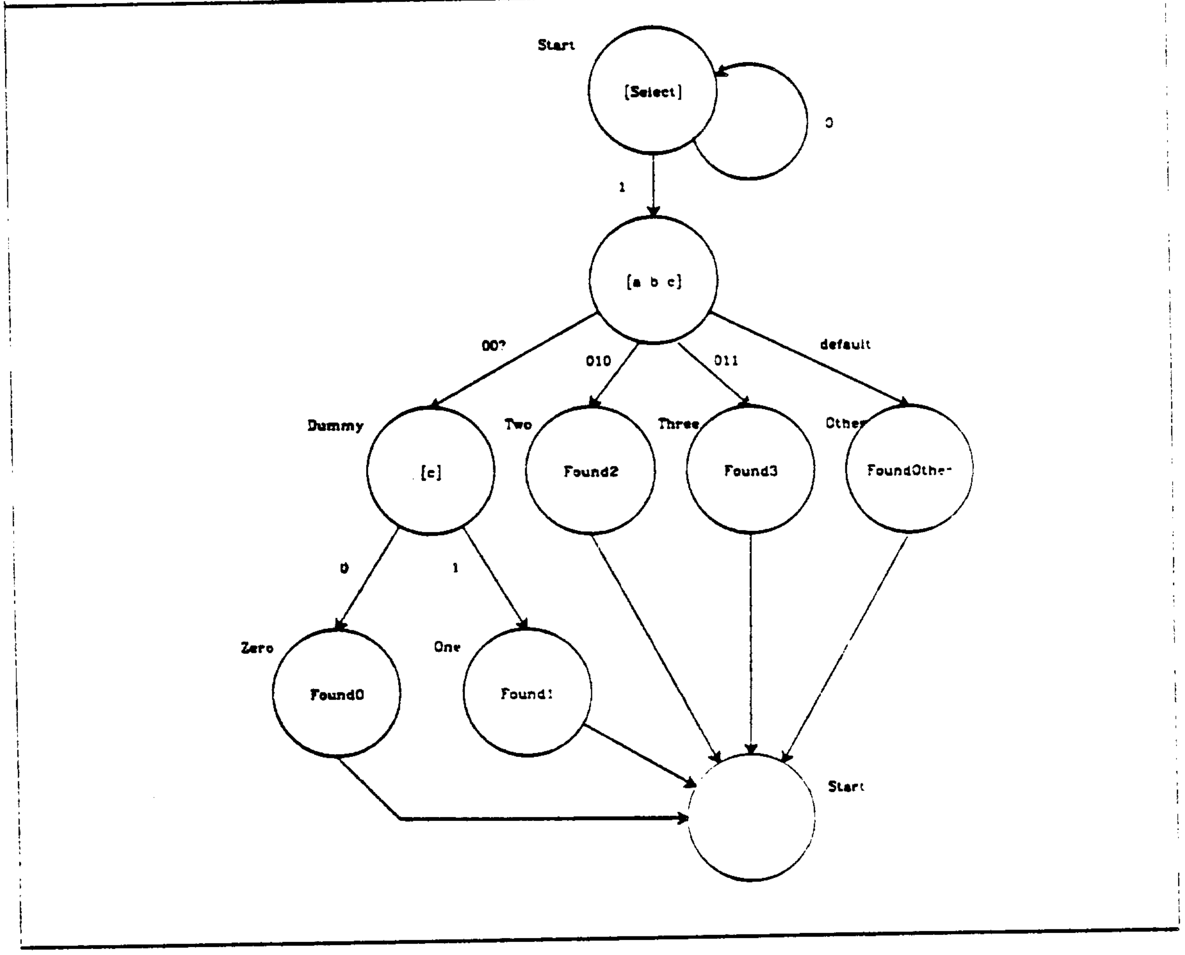


Figure 9: State Diagram for Example 3

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 ?.

```
--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

6. Peg Grammar

```
<program>          : <InputList> <OutputList> <StateList>

<InputList>        : INPUTS : <IdentifierList> ; | /*NULL*/

<OutputList>       : OUTPUTS : <IdentifierList> ; | /*NULL*/

<StateList>        : <State> | <StateList> <State>

<IdentifierList>   : <IDENTIFIER> | <IdentifierList> <IDENTIFIER>

<State>            : <IDENTIFIER> : <Signals> <Control> | : <Signals> <Control>

<Signals>          : /*null*/ | <ASSERT> <IdentifierList> ;
```

```
<Control>          : CASE ( <IdentifierList> ) <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*/

<CaseStatement>    : <BitList> => <IDENTIFIER> ;

<Cases>            : <Cases> <CaseStatement> | <CaseStatement>

<Bit>              : 0 | 1 | ?

<BitList>          : <BitList> <Bit> | <Bit>

<DefaultCase>      : ENDCASE => <IDENTIFIER> ; | ENDCASE ;

<NOT>              : "!" | "NOT" | "-"

<Comment>          : "--".*$

<IDENTIFIER>       : [A-Za-z][A-Za-z0-9._]*
```

7. References

[CADMan]
CAD Manual, Online Unix documentation.

[Danford]
Peggy Danford, Private communication with author, June 1982.

[Landman]
Howard Landman, *Automatic Layout of Optimized PLA Structures*, Masters Project Report, University of California at Berkeley, June 1982.

[Unix]
Unix Programmer's Manual, Seventh Edition, Virtual VAX-11 Version, Computer Science Division, University of California at Berkeley, November 1980.

Ain't is a word in SLANG

Korbin S. Van Dyke

Computer Science Division
University of California
Berkeley, California 94720

1. Introduction

This document is a tutorial primer for SLANG, a **S**imulation **LANG**uage for digital logic, and its event driven simulator. The term SLANG will be used interchangeably to refer to both the description language and the simulator itself. SLANG is a multi-level system for logic specification and simulation. It allows a user to specify a system at an arbitrarily high level (functional), or at an arbitrarily low level (gates), and to intermix these levels of description at will. It allows convenient comparison of these levels and the switch level simulator ESIM.

SLANG was written by John Foderaro in 1981 to describe and simulate a 44K transistor NMOS processor, RISC I. It has also been used to simulate, at a very high functional level, some parts of an ECL system. Currently it is being used to simulate another NMOS processor, RISC II. SLANG is just a superset of Lisp and provides:

(1) a hardware specification language -- documentation and evaluation
(2) testing of hardware specification through simulation
(3) generation of PLA equations
(4) interactive signal description helpful during routing and low level logic design
(5) aid in ESIM simulation of modules and complete systems
(6) vectors for testing the resulting hardware

Following are informal terminal (the CRT type) sessions with SLANG to instruct the beginning user. For more exacting information, see the *SLANG Slinger's Cyclopedia*. In the following transcriptions the SLANG output is in small print, with user input in a **bold** font. Before proceeding to the examples, a brief introduction to Lisp will be given.

2. A Few Words About Lisp

Since SLANG is embedded in Franz Lisp, (see the *Franz Lisp Manual**) all the things about Lisp apply to SLANG, and the functions available in Lisp are usable in SLANG. (SLANG is really a Lisp system with added functions and some pre-defined variables.)

A Lisp program is a sequence of expressions. An expression is either *atomic* or non-atomic. An atomic expression is a symbol (like a variable in other programming languages) or is a number. A non-atomic expression or *list* is a left

* If you live in Berkeley this can be purchased in 515 Evans Hall.

parenthesis followed by zero or more expressions separated by blanks, tabs or newlines, and then a right parenthesis. Everything between a semicolon and the end of line (including the semicolon) is ignored and this provides comments in program text.

In a list expression the first expression inside the parentheses is usually the name of a function and the rest of the expressions are arguments to the function call. Other times the first element is a tag describing what the rest of the expressions are. There are both kinds of list expressions in a SLANG expression.

To make it easier to type in Lisp (or SLANG) expressions, the editor should be given appropriate commands. In *vi* use this set command:

```
:set lisp ai sm
```

where ai is short for autoindent, and sm is short for showmatch. Alternatively, the set command can be put in the file itself at either the beginning or end. The form is:

```
; vi:set lisp ai sm:
```

3. A Simple Boolean Network

To start with, consider this simple boolean network with inputs s0, s1, Q, Ql, Qr, and Din; the output is mux4to1. Shown in Figure 1, the circuit has four **and** gates and a single **or** gate. We will describe it and then simulate it with SLANG.

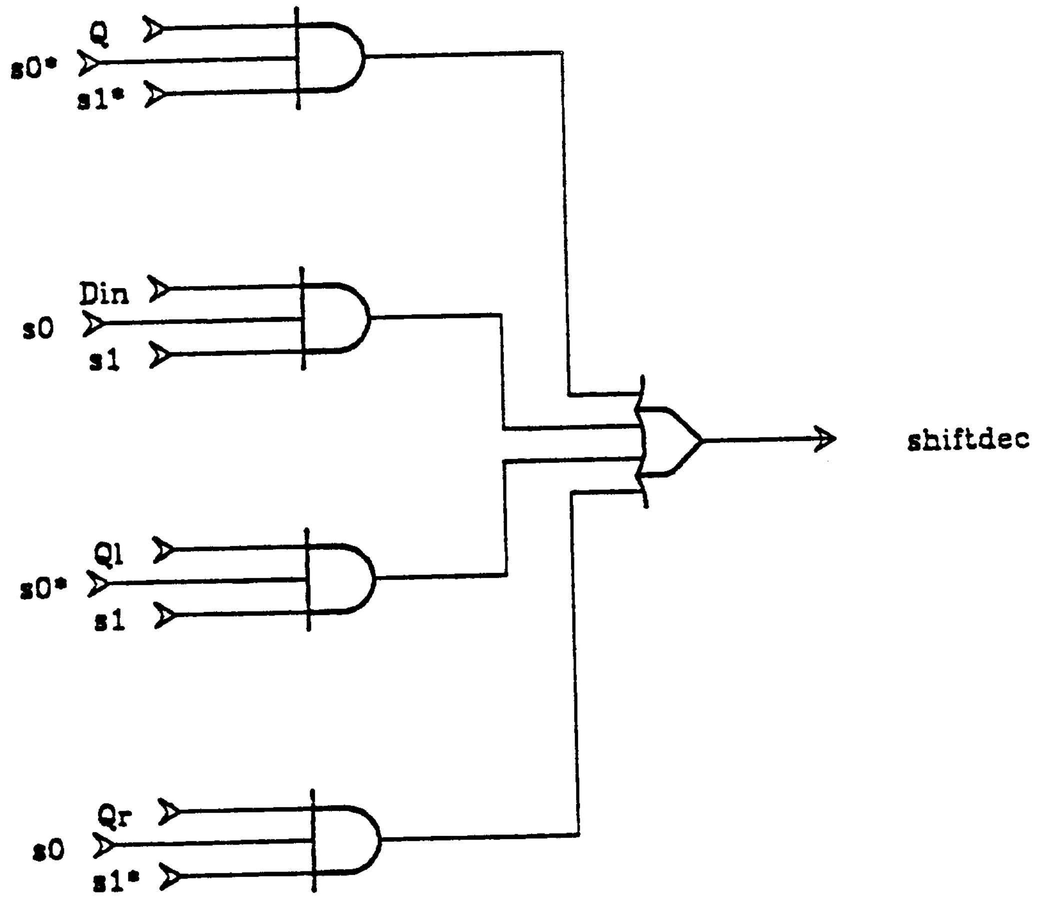


Figure 1: A simple boolean network.

Let's jump right in and start up SLANG.

```
% slang
Slang Simulation: Blank    v0   Technology: Unknown
0 nodes defined
->
```

SLANG responds with a heading message and the "->" prompt. Since we have not yet entered a description into the system, the simulator tells us that the name of this simulation is "Blank", that is ready for a fresh specification. It also tells us that this is "v0" or version zero, since we have not informed it otherwise. It further reminds us of its ignorance by informing us that the technology of the system is "unknown".

Let's proceed by telling SLANG that our system has a node called *mux4to1* and at the same time tell SLANG how to calculate the node's new value given the value of the inputs to the system. To turn the logic diagram in Figure 1 into something SLANG understands just type:

```
(defnode mux4to1
 (update (Or
        (And A0 (Not s0) (Not s1))
        (And A3 s0 s1)
        (And A2 (Not s0) s1)
        (And A1 s0 (Not s1))
        ))
 )
```

and SLANG responds with:

```
(Or (And A0 (Not s0) (Not s1)) (And A3 s0 s1) (And A2 (Not s0) s1) (And
A1 s0 (Not s1)))
```

Notice that the SLANG (or Lisp) reader allows input across multiple lines. It will continue reading until a matching parenthesis is found. The construct *(defnode mux4to1 ...)* tells SLANG: "Here is a node named mux4to1. Give it the following properties." The *(update ...)* clause of the defnode expression returns the new value of the node. Notice that it uses the SLANG functions **And**, **Or**, and **Not**. Although we might guess what these functions mean, we can test the functions directly by trying some cases with ON and OFF:

```
-> (And OFF OFF)
OFF
-> (And OFF ON)
OFF
-> (And ON ON)
ON
-> (Or ON ON)
ON
-> (Not OFF)
ON
```

You probably thought the universe is binary; you are wrong. SLANG's universe also contains the value UNKNOWN or UNK. This is required since in real systems there may be sets of inputs for which the output is undefined, especially when an input value is not defined. Let's test this out:

```
-> (And UNK ON)
UNK
-> (And OFF UNK)
OFF
-> (Or ON UNK)
ON
-> (Not UNK)
UNK
```

What this limited demonstration shows is that the And, Or, and Not operations behave exactly as expected for single bit inputs which can be ON, OFF, or UNK. These types of inputs correspond to *simple nodes* in SLANG.

Back to the mainstream of the example, we'll query the simulator about the node that has been defined.

```
-> (desc mux4to1)
mux4to1:
     value: (unbound)
     update: ((A0{UNBOUND} & !s0{UNBOUND} & !s1{UNBOUND}) | (A3{UNBOU
ND} & s0{UNBOUND} & s1{UNBOUND}) | (A2{UNBOUND} & !s0{UNBOUND} & s1{UNBO
UND}) | (A1{UNBOUND} & s0{UNBOUND} & !s1{UNBOUND}))
nil
```

The *(desc node)* construct tells SLANG to respond by showing us what it knows about the node – in this case mux4to1. There are two parts to this: the current state (**value**) and the method of computing a new state (**update**). Because we are not simulating mux4to1 yet, it has no value (unbound). The update clause defines the logical function using | (logical or), & (logical and), and ! (logical negation). Note that when the update function is printed the current value of an input is shown inside {}'s. Since we have not yet done any simulation the values are all UNBOUND.

Let's tell SLANG some more information about our system.

```
-> (defnode mux4to1
  (doc "4 to 1 multiplexor")
  )
("4 to 1 multiplexor")
```

Now let's see what happens:

```
-> (desc mux4to1)
mux4to1: 4 to 1 multiplexor
     value: (unbound)
     update: ((A0{UNBOUND} & !s0{UNBOUND} & !s1{UNBOUND}) | (A3{UNBOU
ND} & s0{UNBOUND} & s1{UNBOUND}) | (A2{UNBOUND} & !s0{UNBOUND} & s1{UNBO
UND}) | (A1{UNBOUND} & s0{UNBOUND} & !s1{UNBOUND}))
nil
```

Notice that we add information by using the *(defnode mux4to1 ...)* command. The *(doc ...)* clause takes a string as an argument and SLANG just prints it out following the node name when the *(desc ...)* function is used. This information is helpful when debugging a system.

So far we have left out an important piece of information about our system. One way to simulate is to update every node on each simulation step. A much more efficient way, used by SLANG, is to only update the nodes whose inputs have changed. The easiest way (for SLANG) to know the dependencies between nodes is for the user to specify this explicitly. We use the *depends* clause to do this. In other words, SLANG should recompute a new value for this node when any of the nodes in the *(depends ...)* list change.

```
-> (defnode mux4to1
 (depends s0 s1 A3 A2 A1)
 )
(s0 s1 A3 A2 A1)
```

Again we will ask SLANG to print everything it knows about the node mux4to1.

```
-> (desc mux4to1)
mux4to1: 4 to 1 multiplexor
     value: (unbound)
     depends: (s0 s1 A3 A2 A1)
     update: ((A0{UNBOUND} & !s0{UNBOUND} & !s1{UNBOUND}) | (A3{UNBOU
ND} & s0{UNBOUND} & s1{UNBOUND}) | (A2{UNBOUND} & !s0{UNBOUND} & s1{UNBO
UND}) | (A1{UNBOUND} & s0{UNBOUND} & !s1{UNBOUND}))
nil
```

Now it appears SLANG has enough information to do useful work for us. Let's summarize the data SLANG knows about our system:

(1) There is a single node -- mux4to1.

(2) The value of the mux4to1 node should be recomputed when the inputs s0, s1, Ql, Qr, or Din change.

(3) The value of the node is computed with the update clause, a simple boolean function of the inputs s0, s1, Q, Ql, Qr, and Din.

I got tired of typing this description into SLANG for each session, so I put the information into the file *try1.l*. (See page A.1.) Normally information is recorded in a file and then loaded into SLANG -- see the *SLANG Slinger's Cyclopedia* for more information about how to do this. In any case, we will now exit SLANG, start it up again, read in the description file, and finally ask SLANG to describe our node to see if it matches the earlier description.

```
-> (exit)
% slang
Slang Simulation: Blank    v0  Technology: Unknown
0 nodes defined
-> (load 'try1)
t
-> (desc mux4to1)
mux4to1: 4 to 1 multiplexor
     value: (unbound)
     depends: (s0 s1 A3 A2 A1)
     update: ((A0{UNBOUND} & !s0{UNBOUND} & !s1{UNBOUND}) | (A3{UNBOU
ND} & s0{UNBOUND} & s1{UNBOUND}) | (A2{UNBOUND} & !s0{UNBOUND} & s1{UNBO
UND}) | (A1{UNBOUND} & s0{UNBOUND} & !s1{UNBOUND}))
nil
```

The *(load 'try1)* forced SLANG to read input from the file try1.l. Remember, this contained a summary of the description of our system. When we again asked SLANG to describe our node, we see that the output provided matches what we had earlier, so the information in the file must be correct. Occasionally Lisp will return values such as "t" or "nil." Don't worry. It has to return something, even if it doesn't make sense in your context.

Now we are ready to try to do a simulation -- that is:

(1) drive the inputs of the system
(2) simulate until the results of changing the inputs have propagated throughout the system
(3) print information about the system
(4) continue (1) - (3) until told to stop

We do this with the *(simulate ...)* command. When this function is used the first argument is the name of a variable containing commands to use during the simulation. If you have no special commands, use the Lisp symbol for "nothing" – *nil*. The second argument, *i*, tells the simulator to run *interactively* – stop at each step and talk to the user.

```
-> (simulate nil i)
*** Warning [clock=0] Node not defined s0
*** Warning [clock=0] Node not defined s1
*** Warning [clock=0] Node not defined A3
*** Warning [clock=0] Node not defined A2
*** Warning [clock=0] Node not defined A1
Error: Undefined function called from compiled code  extsymenable
```

What are the results of this simulation? Five **Warnings** and a single **Error**. Something must be wrong.

First we must recover from the error. If we look at the terminal we see:

```
<1>:
```

instead of "->" for a prompt. In general, when the SLANG (or Lisp) system encounters an error, it changes the prompt to <number>:, where number begins with 1 and increases as more errors are found. Thus it contains a nested error stack. The reset command resets this stack.

```
(reset)

[Return to top level]
```

We have not supplied SLANG with enough information about *how* our simulation is to be performed. SLANG is a very flexible and complex system with variables and functions the user must define before a simulation can be run. Never fear, however, because the system comes complete with a file containing reasonable defaults for those variables and functions. This information is contained in the file *~cad/lib/sproject.l*. (See page A.2.) We can load it into the simulator and proceed.

```
-> (load 'sproject)
t
Slang Simulation: library.project.file v-1.0     Technology: nmos
6 nodes defined
```

After the project file was loaded SLANG printed the heading message again, but now the simulation name is **library.project.file**, the version is **-1.0**, and the technology is no longer unknown, but **nmos**. Now that we have defined the user definable functions, we will again try to run a simulation. (Remember: try1 is still loaded in the system.)

```
-> (simulate nil i)
 *** Warning [clock=0] Node not defined s0
 *** Warning [clock=0] Node not defined s1
 *** Warning [clock=0] Node not defined A3
 *** Warning [clock=0] Node not defined A2
 *** Warning [clock=0] Node not defined A1

clock = 0, upds = 2

==>
```

SLANG does something useful! After the warnings, it prints the clock number and tells us how many *upds* (updates) there were during the simulation step. The clock number is the simulator's notion of time; it numbers the simulation steps sequentially, starting from 0. The number of updates is how much work the simulator did, indicating the activity of the system. With an empty simulation variable (or simulation script) all the simulation does is initialize nodes (usually to UNK). Observe that during a simulation SLANG changes the prompt to "==>" to remind us that we are in the middle of a simulation. We instruct it to continue to the next step by typing a ^D.

```
^D

—

clock = 1, upds = 1
```

This particular simulation is not very interesting; on clock 1 there is a just single update.

In order for us to run a more interesting simulation, (doing such things as forcing nodes to values and watching nodes) we need to define a simulation variable containing a simulation script. As before, this could be done interactively by typing the information directly to SLANG, but for many lines of input it is easier to use a file and the editor. The file *simscr.l* (shown in Table 1) contains the lines to set a simulation variable to a list of commands.

Perusing this file the *(setq variable 'variable_value)* form is just an assignment statement. Not surprisingly, the variable gets the variable_value as its value. Therefore the variable sim1 contains a list of commands for the simulator. Also in this file we give the *(watch nodelist)* command, telling SLANG to print the values of the nodes s0, s1, Q, Ql, Qr, Din, and mux4to1 after each simulation step. Now let's load this file into SLANG and try a simulation.

Table 1: The file containing the simulation script.

```
;Simulation script for the 4 to 1 multiplexor
;
;A simulation script (sim1 in this example) is a list of
;lists. The first member of the list is an expression
;which evaluates to t or nil (t or (eq clock 1) in the
;example). The following member of the list is a
;sequence of commands that are done when the first
;expression evaluates to t.
;

(setq sim1 '(
            (t (forcevalue 's0 OFF))
            (t (forcevalue 's1 OFF))
            (t (forcevalue 'A0 OFF))
            (t (forcevalue 'A1 OFF))
            (t (forcevalue 'A2 OFF))

            ((eq clock 1) (forcevalue 's0 ON))
            ((eq clock 1) (forcevalue 's1 ON))
            ((eq clock 1) (forcevalue 'A0 OFF))
            ((eq clock 1) (forcevalue 'A1 OFF))
            ((eq clock 1) (forcevalue 'A2 ON))

            ((eq clock 2) (forcevalue 's0 ON))
            ((eq clock 2) (forcevalue 's1 ON))
            ((eq clock 2) (forcevalue 'A0 ON))
            ((eq clock 2) (forcevalue 'A1 OFF))
            ((eq clock 2) (forcevalue 'A2 ON))

            ((eq clock 3) (forcevalue 's0 OFF))
            ((eq clock 3) (forcevalue 's1 ON))
            ((eq clock 3) (forcevalue 'A0 OFF))
            ((eq clock 3) (forcevalue 'A1 OFF))
            ((eq clock 3) (forcevalue 'A2 ON))

            ((eq clock 5) (simulationend))
            )
  )

 (watch s0 s1 A0 A1 A2 A3 mux4to1)
```

```
==>(load 'simscr)
A0 is not a node
t
```

Rats! Just after loading the simulation script file we get an error message that Q is not a node. What's going on? Looking at the file try1.l we see that while Q is used in the update clause, it is missing from the depends clause. This must be fixed. The correction is made in *try2.l*. (See page A.6.) We can now load this description into the simulator and try again.

```
==>(load 'try2)
t
```

At last we are getting somewhere. On clock tick 0 the node mux4to1 is OFF. It should be, since from the sim1 variable defined in the file simscr.l the inputs s0, s1, Q, Ql, and Qr are OFF, so mux4to1 should also be OFF (look again at figure 1). On clock tick 1 the s0, s1, and Qr inputs are turned ON, but since Din is UNK mux4to1 is also UNK. Finally, on clock 2 Q is ON instead of OFF, but mux4to1 should still be UNK, and it is.

However, we are still receiving warning messages. The simulator thinks that the following "nodes" are not defined: s0, s1, Ql, Qr, Din, and Q. The simulator is, of course, correct. We have half-way implicitly defined them as nodes by using them as inputs for the mux4to1 node, but we have not explicitly declared them to be nodes in their own right. This situation can be remedied by adding def-node expressions for each of these nodes to our description file. This is done in the file *try3.l*. (See page A.7.) Notice that since these new nodes are external inputs, they do not depend on any other nodes in the system. (No depends clause is present.) Also, the update clause is trivial; each node just gets its own value. The doc expressions are present to aid in debugging. Now our system consists of 7 full fledged nodes, and we can try to simulate it again.

```
==>(load 'simscr)
t
==>(simulate sim1 i)
 *** Warning [clock=0] Node not defined s0
 *** Warning [clock=0] Node not defined s1
 *** Warning [clock=0] Node not defined A3
 *** Warning [clock=0] Node not defined A2
 *** Warning [clock=0] Node not defined A1
 *** Warning [clock=0] Node not defined A0

clock = 0, upds = 7
A0:OFF; mux4to1:OFF; A3:UNK; A2:OFF; A1:OFF; s1:OFF; s0:OFF;

==>^D

—

clock = 1, upds = 6
A0:OFF; mux4to1:UNK; A3:UNK; A2:ON; A1:OFF; s1:ON; s0:ON;

==>^D

—

clock = 2, upds = 6
A0:ON; mux4to1:UNK; A3:UNK; A2:ON; A1:OFF; s1:ON; s0:ON;

==>(load 'try3)
t
==>(simulate sim1 i)
 *** Warning [clock=0] Node not defined Q
 *** Warning [clock=0] Node not defined Ql
 *** Warning [clock=0] Node not defined Qr
 *** Warning [clock=0] Node not defined Din

clock = 0, upds = 7
A0:OFF; mux4to1:OFF; A3:UNK; A2:OFF; A1:OFF; s1:OFF; s0:OFF;

==>^D

—

clock = 1, upds = 6
A0:OFF; mux4to1:UNK; A3:UNK; A2:ON; A1:OFF; s1:ON; s0:ON;
```

Success is finally ours. We can perform two clock steps in the simulation without error messages and with correct results.

Continuing with the example, we will explore a little more about how to use certain features of SLANG. When a simulation is in progress and SLANG gives us the "==>" prompt, we can find out the value of a node by simply typing its name. The following shows this being done, and the two nodes in question described by SLANG. Finally ^D is typed to continue on to the next step of the simulation.

```
==>s0
ON
==>s1
ON
==>(desc s0 s1)
s0: Function select input
    value: ON
    update: s0{ON}
    affects: (mux4to1)
s1: Function select input
    value: ON
    update: s1{ON}
    affects: (mux4to1)
nil
==>^D

—

clock = 2, upds = 6
A0:ON; mux4to1:UNK; A3:UNK; A2:ON; A1:OFF; s1:ON; s0:ON;
```

Now we will interactively force these two nodes ON using the *(force ...)* function. Then we will proceed to the next step of the simulation. However, since the simulation script turns s0 OFF, after the next clock step it is OFF, not ON as we forced it to. Typically nodes are forced to values when they are not being driven, either externally or internally.

```
==>(force 's0 ON)
(mux4to1)
==>(force 's1 ON)
(mux4to1 mux4to1)
==>^D

—

clock = 3, upds = 6
A0:OFF; mux4to1:ON; A3:UNK; A2:ON; A1:OFF; s1:ON; s0:OFF;
```

Finally we will run the entire simulation non-interactively and exit SLANG. After clock 4 the simulator reads the *(simulationend)* command from the simulation script and terminates.

```
==>(simulate sim1)
 *** Warning [clock=0] Node not defined Q
 *** Warning [clock=0] Node not defined Ql
 *** Warning [clock=0] Node not defined Qr
 *** Warning [clock=0] Node not defined Din

clock = 0, upds = 7
A0:OFF; mux4to1:OFF; A3:UNK; A2:OFF; A1:OFF; s1:OFF; s0:OFF;


clock = 1, upds = 6
A0:OFF; mux4to1:UNK; A3:UNK; A2:ON; A1:OFF; s1:ON; s0:ON;


clock = 2, upds = 6
A0:ON; mux4to1:UNK; A3:UNK; A2:ON; A1:OFF; s1:ON; s0:ON;


clock = 3, upds = 6
A0:OFF; mux4to1:ON; A3:UNK; A2:ON; A1:OFF; s1:ON; s0:OFF;


clock = 4, upds = 1
A0:OFF; mux4to1:ON; A3:UNK; A2:ON; A1:OFF; s1:ON; s0:OFF;

End of simulation

[Return to top level]
-> (exit)
%
```

This concludes the example of the simple boolean network.

In this tutorial we have seen the amazing power of the interactive SLANG system. Written in Lisp, it provides the capability to describe and simulate digital systems. Such systems are modeled in terms of nodes using the defnode command; each node has several properties. The update property calculates the new value for a node. Documentation for a node is done with the doc clause. Finally, information about interaction between nodes is recorded with the depends expression.

The simulator itself can provide the user with all known information about nodes. Of course it can also perform simulations, using predetermined commands if desired. Most importantly, it provides interactive feedback during simulations and allows convenient interrogation of a system's status during a simulation.

SLANG Slinger's Cyclopedia

Korbin S. Van Dyke, John K. Foderaro

Computer Science Division
University of California
Berkeley, California 94720

Table of Contents

1. Introduction

This document describes SLANG, a Simulation LANGuage for digital logic, and its event driven simulator. The term SLANG will be used interchangeably to refer to both the description language and the simulator itself. SLANG is a multi-level system for logic specification and simulation. It allows a user to specify a system at an arbitrarily high level (functional), or at an arbitrarily low level (gates), and to intermix these levels of description at will. It allows convenient comparison of these levels and the switch level simulator ESIM.

SLANG was written by John Foderaro in 1981 to describe and simulate a 44K transistor NMOS processor, RISC I. It has also been used to simulate, at a very high functional level, some parts of an ECL system. Currently it is being used to simulate another NMOS processor, RISC II. SLANG is actually a superset of Lisp and provides:

(1) a hardware specification language -- documentation and evaluation

(2) testing of hardware specification through simulation

(3) generation of PLA equations

(4) interactive signal description helpful during routing and low level logic design

(5) aid in ESIM simulation of modules and complete systems

(6) vectors for testing the resulting hardware

The following sections describe the SLANG circuit model, provide a brief introdction to Lisp, highlight SLANG built in primitives, describe how to get started using SLANG, and give a simple example of the use of SLANG.

2. SLANG circuit model

The SLANG model of a circuit is a collection of objects called *nodes*. Each node is driven by one or more modules, having several other nodes as inputs. The module is described in terms of a procedure, and its inputs are specified so that an interconnection database is generated internally. In many cases a node corresponds to a single physical point in the circuit, but a node may also correspond to the collective value of a group of points in the circuit. Normally these groups represent a *bus*.

2.1. Node Model

A node has these properties:

(1) **name** - a string of characters of any length with upper and lower case alphabetics distinct. Certain characters are treated specially by the SLANG parser and should be avoided in names. These characters are "()'"[]\#:" SPACE and TAB. If you must use one of these characters you should precede the character with the quoting character BACKSLASH.

(2) **value** - The value of a node is determined by the simulator and assigned to the node. There are different sets of values depending on whether the node corresponds to a single point in the circuit (a *simple* node) or to a collection of points (a *complex* node). Simple nodes have one of the three values: ON, OFF or UNK (meaning unknown). Complex nodes may either have the value UNK or may have an integer as their value. Currently the integers are limited to 32 bits.

(3) **depends** - a list of nodes whose values are used in computing the value of this node (other nodes may be included in this list if circumstances require it). Normally these correspond to inputs to the module driving that node.

(4) **update** - an expression calculating the current value of this node. For certain nodes (e.g. buses) this clause isn't provided in the SLANG description but is generated automatically using information in affects clauses.

(5) **affects** - an expression describing how this node affects buses in the circuit.

(6) **class** - just some characteristic of the node. The class is looked at by such programs as the pla generation program. Otherwise it is ignored.

(7) **doc** - a documentation string describing what this node is. This information is helpful when debugging the circuit.

(8) **init** - an expression defining how the node is to be initialized. This is especially useful for complex nodes.

2.2. System Model

In the SLANG model a circuit is composed of a collection of blocks with one output and one or more inputs. Blocks are implicitly connected if the name on the output node of one block is the same as the name on the input of another block. To describe a circuit, the SLANG user must first decide which portions of the circuit are to be considered blocks, and then must describe how the value (output) of each block is computed from the values of the input nodes. The decision about what corresponds to a block can be changed at any time.

There is a single exception to this simple rule of defining blocks and their interconnections -- buses. A bus and its update function are implicitly defined by the affects clause of modules that connect to it. Note that a bus is still a node and has a value associated with it.

To describe the function of a block the SLANG user may use some of the predefined SLANG functions or may use any Lisp function (the language in which SLANG is embedded). We now take a brief break from describing SLANG to introduce the Lisp language.

3. Lisp

Since SLANG is embedded in Franz Lisp, (see the Franz Lisp Manual) all of the things about Lisp apply to SLANG, and the functions available in Lisp are usable in SLANG. (SLANG is actually a Lisp system with added functions and some pre-defined variables.) Therefore a short section of the syntax of Lisp and some of the available functions follows.

3.1. Syntax

A Lisp program is a sequence of expressions. An expression is either *atomic* or non-atomic. An atomic expression is a symbol (like a variable in other programming languages) or is a number. A non-atomic expression or *list* is a left parenthesis followed by zero or more expressions separated by blanks, tabs or newlines, and then a right parenthesis. Everything between a semicolon and the end of line (including the semicolon) is ignored and this is used to include comments in program text.

In a list expression the first expression inside the parantheses is usually the name of a function and the rest of the expressions are arguments to the function call. Other times the first element is a tag describing what the rest of the expressions are. There are both kinds of list expressions in a SLANG expression.

To make it easier to type in Lisp (or SLANG) expressions, the editor should be given appropriate commands. In *vi* use this set command:

```
:set lisp ai sm
```

where ai is short for autoindent, and sm is short for showmatch. Alternatively, the set command can be put in the file itself at either the beginning or end. The form is:

```
; vi:set lisp ai sm:
```

3.2. Lisp Functions

This section describes some of the more important functions available from the Lisp interpreter. This is only a sample, containing just the minimum functions that are necessary to use the system with SLANG. For more detailed information on these functions or other functions available, consult the Franz Liszt Manual.

3.2.1. load

To enter new information into the system, use the load function.

```
(load 'filename)
```

This reads filename.o into the system if it exists, and if not it reads in filename.l. Filename.l is the conventional name for Lisp (and SLANG) source code, and filename.o is the name for compiled source code. To force loading and subsequent interpretation of filename.l, explicitly use filename.l.

3.2.2. reset

If you are stuck in the Lisp system and keep getting error messages, you can return to the top level of interaction, losing all accumulated errors by using the reset function.

```
(reset)
```

Note that if a simulation was in progress, the reset function throws it away.

3.2.3. exit

When you are finished using Lisp and wish to end the session, use exit.

```
(exit)
```

This may be used at any place in the session.

3.2.4. return

Lisp keeps track of nested errors on an error stack. The prompt becomes

```
<number>:
```

where number begins with 1 and continues up as the nested errors occur. To move one level up in the error stack use return.

```
(return)
```

Note that if the error level is just 1, then return will take you back to the top level.

4. SLANG Built In Functions

Besides having all the available Lisp functions, SLANG also has its own functions. There are many of these functions available, and for a complete list and brief usage description see the SLANG Reference Manual. Below some of the more important functions are described in detail.

4.1. simulate

SLANG performs a simulation of a system when it is given the simulate command. When this command is executed the following occurs:

(1) All nodes are initialized to UNK.

(2) If required, nodes (usually complex) are initialized by a user specified function (the init clause in the defnode expression).

(3) Nodes that are to be set to a value externally are forced to that value and the required changes are propagated throughout the system until a steady state is reached.

(4) The special node *masterclock* is put on the event queue and changes are again propagated until a steady state is reached.

(5) Nodes that are being watched are printed and if desired the simulator pauses for user interaction before going to the next simulation step.

(6) Items (3) to (5) are repeated until the simulation terminates.

User interaction for a particular step can be ended by typing EOF<cr> or ^D. The simulator then proceeds to the next simulation step. There are two main conditions which completely terminate a simulation:

- The maximum number of steps (specified by the variable clockmax) may have been performed.
- There may have been an explicit simulationend command given.

There are many options to the simulate command. They specify the simulation script to be used, if and when interaction should occur, and if nodes that are being watched should be printed. Using these options it is possible to perform a number of "set up" steps in the system being simulated and then be ready to interactively debug it without a large amount of terminal output.

4.2. force

To drive the system being simulated the force command is used. This command may be in a simulation script or it may be given interactively from the terminal. It takes a node value pair and forces the node to the given value. A synonym for this command is forcevalue. An example of this command is:

```
(force 'input ON)
```

4.3. watch

During a simulation it is necessary to know if nodes contain the correct values. This can be done by having the values of nodes printed after each simulation step. This is done with the various watch commands:

- watch -- watch the given nodes constantly
- watchon -- watch the given nodes only when ON
- watchoff -- watch the given nodes only when OFF
- watchunk -- watch the given nodes only when UNK
- watchch -- watch the given nodes when they change
- watchwho -- if you are using an H19, watch the given nodes constantly on the who line (this requires the h19sys program to be running in the background)

Note that the printing of the watched items may be prevented by using the watchmute function or the mute option to the simulate command.

4.4. desc

One of the powerful capabilities of SLANG is the online accessing of data about the system. Invoking the desc function will provide all the information SLANG knows about a node. This includes the documentation clause for a node, the update function for a node, what it affects, and the current value.

4.5. helpsim

Built into the SLANG system is a help function called helpsim. When information about a command is needed use the helpsim function to get a brief online description of the command. The function can also be used to get a list of all available commands or to get a complete "manual" of commands.

4.6. split

During debugging of a system, it is useful to try something out for a clock step or two, and then go "back" in time to try out something else from the same starting position. The SLANG function split allows a user to conveniently do this. A copy of the state of the current Lisp system is built and forked. You can try out the forcevalues and do a ^D to see what happens. This will change masterclock, and do a complete simulation step. This can be repeated if desired. When you are ready to "return" to the original simulator type (exit) and you will be back to where you were before the split function.

5. Getting Started

SLANG is a rather complex and flexible system, and requires some non-trivial set-up operations to be used effectively. There are certain functions, specifications, and default values that are left for the user to define and set; these things are handled in the *project file*. When a simulation or set of simulations is to be run, it is convenient to put the simulation steps and parameters into a file; this is called a *simulation script*. Finally, and most importantly, there must be a description (in SLANG) of the system to be simulated; this is contained in the *description file*. These items are described below in more detail.

5.1. Project File

Associated with each system being simulated is a project file. This file contains header information for personalizing the simulator in the Lisp variables sim-name, tech, and ver. This data is printed by SLANG when loaded and at the beginning of SLANG sessions. It also contains some user-definable functions which must be defined somewhere for proper operation of the simulator: *extsymenable*, *breakpoint*, and *busupdate*.

The extsymenable function tells SLANG if an external simulator is being used. This function is evaluated at each clock tick of the simulation. If it returns non-nil then SLANG continues to the next clock tick, otherwise the external simulator is stepped and a comparison made between the SLANG nodes and the corresponding external simulator nodes.

During a simulation it is convenient to check for something specific happening and begin a user interaction if desired. This capability is provided by a call to the user-defined function breakpoint at every clock tick of the simulation. If the function returns non-nil, then a message is sent to the user and interaction is begun by SLANG.

Because different technologies differ in how buses act, and because users may desire to modify the behavior of buses, the busupdate function is user-defined. Currently there are two versions of this function available, one for nmos and one for ecl.

Finally, the project file should contain the magical line

```
(setq user-top-level 'sim-top-level) ; Print info when loaded
```

so SLANG will print the header information provided in the project file when it is loaded.

5.2. Simulation Scripts

To enable convenient use of the simulator, the commands required to run a simulation may be gathered up together and stored in a Lisp variable. A simulation using this set of saved commands may be performed using the SLANG function *simulate*. There may be more than one such set of commands stored in different variables; the file containing the values for these variables is called a simulation script. The term simulation script will also be used loosely to refer to the Lisp variables containing the sequences of commands. Examples of things which may be in a script are commands to set nodes to certain values, print the values of nodes, and end the simulation.

The format for the variables containing the commands is:

```
( (expr1) (expr2) ... (exprn) )
```

where each expr is of two parts. When a simulation step is done expr1 is taken and the first part of it is evaluated. If it evaluates to nil then nothing is done. If it evaluates to non-nil then the second part of the expression is evaluated and then expr1 is thrown away. Then expr2 moves to the head of the list and the first part of it is evaluated. The evaluation/throw away process is continued until the first non-nil value.

What this apparently confusing arrangement means in practice is quite simple. SLANG has a variable that is the number of the next clock tick to occur. A simulation script would involve comparisons to that clock variable in the order things are to be done:

```
(
        (t (junk done at beginning of simulation) )
        ( (eq clock 1) (junk done for clock tick 1) )
        ( (eq clock 2) (junk done for clock tick 2) )
        ...
        ( (eq clock n) (simulationend) )
)
```

Note that if the commands are in the wrong order, (clock 4 before clock 3, for example) then the script will not work correctly.

5.3. Description File(s)

One or more description files describe the system being simulated by the SLANG simulator. These files (normally organized with one major module each) usually consist of many defnode expressions defining the system's nodes to the simulator. The files may also contain Lisp functions for application by the user or for reference in the defnode clauses, perhaps in a complicated update function. In any event, there are no restrictions on what the description files may contain.

5.3.1. defnode

Nodes are made known to the simulator using the defnode clause. It defines the name, depends, update, affects, class, doc, and init properties of a node. See the SLANG Reference Manual for a generic defnode example. The use of the defnode expression is straightforward and the example shows its syntax. However, the update (defines new value for a node), affects (determines if and how a node affects buses), and init (gives how a node is to be initialized) clauses are much easier to write with the aid of several builtin SLANG constructs.

5.3.2. If

The system has a macro available allowing the writing of conditional tests and sequencing in a convenient and conventional way. This is the If macro. It is used in the following way:

```
(If (condition) then (do this) (and this) ... else (do this instead) ... )
```

The condition clause is evaluated and if it is non-nil then the (do this) and following clauses are executed up to the next else. If the condition clause is is nil, then the (do this instead) and following clauses are evaluated instead.

The else portion may be omitted, or it may be substituted with an elseif section. There may be multiple elseif statements. Also, the then may be substituted with a thenret, which returns immediately, without evaluating anything else, but returning the value of the predicate.

5.3.3. If3way

The If3way macro is somewhat similar to the If macro, but it splits the condition three ways: ON, OFF, or UNK. This is useful for describing updates dependent upon the value of a simple node in the system. It takes the following form:

```
(If3way pred true false unknown)
```

The pred is evaluated and if it is ON then true is done; if it is OFF then false is done; if it is UNK then unknown is done. See the generic defnode example to see a typical use of this in describing a three-state connection to a bus.

5.3.4. conflict

The description of a module may contain a specification that a certain set of signals (usually inputs to the module) never have more than one active member, where active means not OFF. This is done with the conflict function, which is normally a clause within the update specification:

```
(update (conflict name1 name2 ... namen) (other update clauses ...) )
```

When a simulation step occurs, after changes have been propagated, the signals in the conflict list are checked to make sure that only one of them is ON or UNK. (This is a static check.) If this is not the case then an error message is printed, listing the clock number, the signal names, and their values.

5.3.5. logical support functions

Since nodes may be UNK, as well as being a 32 bit integer, ON, or OFF, it is desirable to have a set of functions that deal specifically with these types of values. There is a set of functions written for simple nodes (ON, OFF, or UNK) that do the "proper" thing when the node is UNK. Examples of these are the And, Or, and Not functions. Corresponding to these functions is a similar set for complex nodes (UNK or 32 bit integer) which operate in a bitwise manner. These work correctly for an UNK value, as well as the special values of all 0's (0) and all 1's (-1). Examples of these are the Logand, Logor, and Comp functions. For a complete list of these functions, as well as details on using them, see the SLANG Reference Manual.

6. Examples

This section is a transcript of an uninterrupted terminal session with SLANG, simulating a prototype system consisting of a simple boolean network of four **and** gates and a single **or** gate. The user's responses are in **bold**.

In this first part of the transcription, SLANG is invoked, the first effort at describing the system is loaded, an example of the desc function is shown, and an attempt is made to simulate the system. The file try.l is the description file and shiftdec is the single node defined in it. The "->" is the prompt from the SLANG system when no simulation is in progress. Note that without the project file and the functions defined in it, the simulation will not run.

```
Script started on Tue Jan  5 08:25:43 1982
Warning: no access to tty; thus no job control in this shell...
% /vc/risc/gold/slang/slang
Slang Simulation: Blank    v0   Technology: Unknown
0 nodes defined
-> (load 'try1)
t
-> (desc shiftdec)
shiftdec: Simple boolean function
      value: (unbound)
      depends: (s0 s1 Ql Qr Din)
      update: ((Q{UNBOUND} & !s0{UNBOUND} & !s1{UNBOUND}) | (Din{UNBOUND} & s0
{UNBOUND} & s1{UNBOUND}) | (Ql{UNBOUND} & !s0{UNBOUND} & s1{UNBOUND}) | (Qr{UNBO
UND} & s0{UNBOUND} & !s1{UNBOUND}))
nil
-> (simulate nil i)
 *** Warning [clock=0] Node not defined s0
 *** Warning [clock=0] Node not defined s1
 *** Warning [clock=0] Node not defined Ql
 *** Warning [clock=0] Node not defined Qr
 *** Warning [clock=0] Node not defined Din
Error: Undefined function called from compiled code  extsymenable
<1>: (reset)

[Return to top level]
```

Now the project file (sproject.l) is loaded and the minimum interactive simulation (which does nothing except initialize nodes) can be run. Note that there are still error messages, but now the simulator can print the header information.. The "==>" is the prompt from SLANG when a simulation is in progress and user interaction is occurring. The user tells SLANG to continue to the next step by typing EOF <cr> (as in this example) or ^D.

```
-> (load 'sproject)
t
Slang Simulation: example  v-1.0     Technology: ttl
6 nodes defined
-> (simulate nil i)
 *** Warning [clock=0] Node not defined s0
 *** Warning [clock=0] Node not defined s1
 *** Warning [clock=0] Node not defined Ql
 *** Warning [clock=0] Node not defined Qr
 *** Warning [clock=0] Node not defined Din

 clock = 0, upds = 2
```

```
==>EOF

—

clock = 1, upds = 1
```

To make possible a simulation which does more than just initialize nodes, a simulation script is loaded. This is the file simscr.l. The simulation variable defined is sim1, which specifies values to be forced to nodes and which nodes are to be watched. The first step of a simulation is performed using that script. Note that there are still error messages, resulting from the incomplete description being used.

```
==>(load 'simscr)
Q is not a node
t
==>(simulate sim1 i)
 *** Warning [clock=0] Node not defined s0
 *** Warning [clock=0] Node not defined s1
 *** Warning [clock=0] Node not defined Ql
 *** Warning [clock=0] Node not defined Qr
 *** Warning [clock=0] Node not defined Din

clock = 0, upds = 6
shiftdec:OFF; Din:UNK; Qr:OFF; Ql:OFF; s1:OFF; s0:OFF;
```

The error message "Q is not a node" is eliminated in the second iteration at describing the system in the file try2.l. Q has been added to the depends clause for the shiftdec defnode. A simulation is done after this new description is loaded into SLANG. The simulation script is then reloaded (note no error message about Q now) and the simulation is performed again.

```
==>(load 'try2)
t
==>(simulate sim1 i)
 *** Warning [clock=0] Node not defined s0
 *** Warning [clock=0] Node not defined s1
 *** Warning [clock=0] Node not defined Ql
 *** Warning [clock=0] Node not defined Qr
 *** Warning [clock=0] Node not defined Din
 *** Warning [clock=0] Node not defined Q

clock = 0, upds = 7
shiftdec:OFF; Din:UNK; Qr:OFF; Ql:OFF; s1:OFF; s0:OFF;
```

```
==>EOF

—

clock = 1, upds = 6
shiftdec:UNK; Din:UNK; Qr:ON; Ql:OFF; s1:ON; s0:ON;



==>EOF

—

clock = 2, upds = 6
shiftdec:UNK; Din:UNK; Qr:ON; Ql:OFF; s1:ON; s0:ON;



==>(load 'simscr)
t
==>(simulate sim1 i)
 *** Warning [clock=0] Node not defined s0
 *** Warning [clock=0] Node not defined s1
 *** Warning [clock=0] Node not defined Ql
 *** Warning [clock=0] Node not defined Qr
 *** Warning [clock=0] Node not defined Din
 *** Warning [clock=0] Node not defined Q

clock = 0, upds = 7
Q:OFF; shiftdec:OFF; Din:UNK; Qr:OFF; Ql:OFF; s1:OFF; s0:OFF;



==>EOF

—

clock = 1, upds = 6
Q:OFF; shiftdec:UNK; Din:UNK; Qr:ON; Ql:OFF; s1:ON; s0:ON;



==>EOF

—

clock = 2, upds = 6
Q:ON; shiftdec:UNK; Din:UNK; Qr:ON; Ql:OFF; s1:ON; s0:ON;
```

The last error messages are eliminated in the complete description, try3.l, by defnode clauses for the inputs to the shiftdec module. This is loaded into the simulator and then a simulation is done (note that there are no error messages now). After clock 0 and 1 the simulator is interrogated about the nodes s0 and s1. Typing the name of the node returns its value, and the desc function is also invoked. Then the two nodes are forced to the value ON, and the simulation is continued from where it left off before the interaction began.

Finally the simulation is done in its entirety without interaction, the simulator exited, and the script ended.

```
==>(load 'try3)
t
==>(simulate sim1 i)

clock = 0, upds = 7
Q:OFF; shiftdec:OFF; Din:UNK; Qr:OFF; Ql:OFF; s1:OFF; s0:OFF;



==>EOF

_

clock = 1, upds = 6
Q:OFF; shiftdec:UNK; Din:UNK; Qr:ON; Ql:OFF; s1:ON; s0:ON;



==>s0
ON
==>s1
ON
==>(desc s0 s1)
s0: Function select input
     value: ON
     update: s0{ON}
     affects: (shiftdec)
s1: Function select input
     value: ON
     update: s1{ON}
     affects: (shiftdec)
nil
==>EOF

_

clock = 2, upds = 6
Q:ON; shiftdec:UNK; Din:UNK; Qr:ON; Ql:OFF; s1:ON; s0:ON;



==>(forcevalue 's0 ON)
```

```
(shiftdec)
==>(forcevalue 's1 ON)
(shiftdec shiftdec)
==>EOF

—

clock = 3, upds = 8
Q:OFF; shiftdec:OFF; Din:UNK; Qr:ON; Ql:OFF; s1:ON; s0:OFF;



==>(simulate sim1)

clock = 0, upds = 7
Q:OFF; shiftdec:OFF; Din:UNK; Qr:OFF; Ql:OFF; s1:OFF; s0:OFF;



clock = 1, upds = 6
Q:OFF; shiftdec:UNK; Din:UNK; Qr:ON; Ql:OFF; s1:ON; s0:ON;



clock = 2, upds = 6
Q:ON; shiftdec:UNK; Din:UNK; Qr:ON; Ql:OFF; s1:ON; s0:ON;



clock = 3, upds = 6
Q:OFF; shiftdec:OFF; Din:UNK; Qr:ON; Ql:OFF; s1:ON; s0:OFF;



clock = 4, upds = 1
Q:OFF; shiftdec:OFF; Din:UNK; Qr:ON; Ql:OFF; s1:ON; s0:OFF;



End of simulation

[Return to top level]
-> (exit)
%
```

```
script done on Tue Jan  5 08:27:02 1982
```