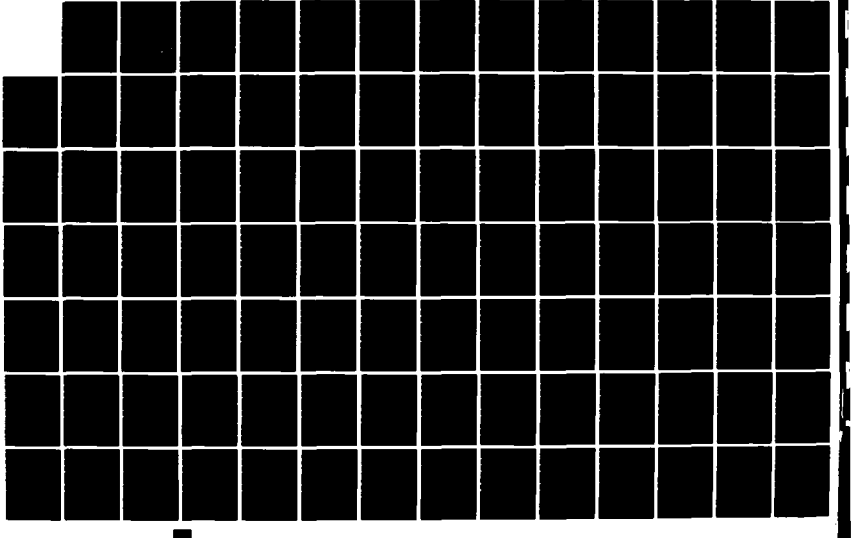
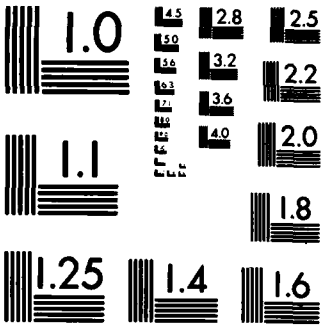
AD-A158 699 VLSI (VERY LARGE SCALE INTEGRATION) DESIGN TOOLS 1/5
REFERENCE MANUAL RELEASE 30(U) WASHINGTON UNIV SEATTLE
DEPT OF COMPUTER SCIENCE AUG 85 TR-85-07-03
UNCLASSIFIED MDA903-85-K-0072 F/G 9/5 NL





VLSI DESIGN TOOLS

REFERENCE MANUAL

RELEASE 3.0

June 1, 1985

UW/NW VLSI CONSORTIUM
University of Washington, Seattle, Washington 98195

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR-85-07-03	2. GOVT ACCESSION NO. AD-A158 699	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) VLSI DESIGN TOOLS REFERENCE MANUAL - RELEASE 3.0		5. TYPE OF REPORT & PERIOD COVERED Technical , Interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) UW/NW VLSI Consortium		8. CONTRACT OR GRANT NUMBER(s) MDA 903-85-K-0072
9. PERFORMING ORGANIZATION NAME AND ADDRESS UW/NW VLSI Consortium Department of Computer Science, FR-35 University of Washington Seattle, Washington 98195		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS N/A
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA/IPTO 1400 Wilson Boulevard Arlington, Virginia 22209		12. REPORT DATE August 1985 13. NUMBER OF PAGES 423
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) ONR University of Washington 315 University District Building 1107 N.E. 45th St., JD-16 Seattle, Washington 98195		15. SECURITY CLASS. (of this report) unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Distribution of this report is unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) very large scale integration, VLSI design tools, CAD tools, CMOS, nMOS, VLSI layout, VLSI circuit simulation, design rule checking		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This report describes the use of the University of Washington/Northwest VLSI Consortium's package of VLSI design tools. The tools described are: — (see reverse)		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

APPENDIX A. TOOL DESCRIPTIONS

1. Functional Design Tools

1.1 "PEG": Translates a finite state machine description into logic equations.
1.2 "EQNTOTT": Converts logic equations into a truth table format.
1.3 "PRESTO": Minimizes truth table attributes.

2. Layout Tools

2.1 "CFL": Library of coordinate free layout procedures.
2.2 "CAESAR": Graphical layout editor.
2.3 "TPLA": Technology independent pla generator.
2.4 "PADS": Padframe generator.

3. Display Tools

3.1 "CIFPLOT": Plots CIF designs using stipple patterns.
3.2 "VIC": Displays designs on TEK 4010 compatible devices and drives penplotters.

4. Rule Checkers

4.1 "CDRC": Checks CIF designs against CMOS rules.
4.2 "LYRA": Performs hierarchical design rule check.

5. Circuit Extractor

5.1 "MEXTRA": Extracts circuit description from CIF design.

6. Simulation Tools

6.1 "SPICE2G6": Device level circuit simulator.
6.2 "RNL": Event driven timing simulator.
6.3 "MTP": Displays RNL and SPICE output on a line printer.
6.4 "SIMSCOPE": Displays RNL and SPICE output on a CRT.

7. Utilities/Miscellaneous

7.1 "NETLIST": Generates circuit description procedurally.
7.2 "PRESIM": Converts a circuit description to RNL input.
7.3 "PSPICE": Creates a complete spice input deck from a circuit description and user input.
7.4 "CELLIB": Layouts of a set of standard cells and pads in CMOS.

VLSI Design Tools

Reference Manual

Release 3.0

06/01/85

TR# 85-07-03

UW/NW VLSI Consortium
Department of Computer Science
Room 315 Sieg Hall
University of Washington FR-35
Seattle, Washington 98195

TABLE OF CONTENTS

1. Introduction
   - 1.1 Who We Are : The UW/NW VLSI Consortium
   - 1.2 Installation Instructions

2. Overview of VLSI Design Tools
   - 2.1 Enhancements since Release 2.1
   - 2.2 Layout Tools : Functional Chart
   - 2.3 Simulation Tools : Functional Chart
   - 2.4 Tool Descriptions

3. Manual Pages

4. Coordinate Free LAP Reference Manual

5. Editing VLSI Circuits with Caesar

6. Standard Cell Library Guide

7. NETLIST/PRESIM/RNL Users Guides and Tutorials
   - 6.1 NETLIST User's Guide
   - 6.2 PRESIM User's Guide
   - 6.3 RNL User's Guide
   - 6.4 NETLIST and RNL Tutorial for Beginners
   - 6.5 NETLIST/PRESIM/RNL - A Tutorial

8. SPICE User's Guide

9. Designing Finite State Machines with PEG

10. Specifying Design Rules for Lyra

INTRODUCTION

Who We Are : The University of Washington/Northwest VLSI Consortium

UW/NW VLSI Consortium members include the University of Washington, represented by the Computer Science Department, and five Pacific Northwest firms: Boeing Aerospace, Honeywell Marine Systems, John Fluke Manufacturing, Microtel Pacific Research of Canada, and Tektronix, Inc. The purpose of the Consortium is to advance the state of the art in VLSI technology and to transfer this technology between industry and the university.

Each corporate member of the Consortium has a full-time liaison on campus who cooperates with faculty and students on circuit design as well as filling the role of visiting faculty, working with graduate students and contributing valuable "real world" experience to the Department of Computer Science. This program serves as a demonstration of cooperative research and technology exchange among universities and industries.

The Consortium maintains a VLSI design system and plans to evolve its capability over time, as well as to provide training in its use. The Consortium validates the system by exercising it with complex design problems. Missing pieces are identified and used for guiding future research and development. The resulting system is made available to other universities and industry.

The research activities of the Consortium are focused in the area of VLSI design generators -- programs that create circuit design layouts as well as other circuit description forms. (Generators are being distributed with this release; others may be exported in the future.) The generator research will lead to a general methodology for building generators and incorporating them into a custom VLSI design environment.

Installation Procedure

The distribution tape contains VLSI design tools that run on a VAX with Berkeley 4.2 UNIX. Many of them will not run on other machines or other versions of UNIX. In addition, VLSI display tools require plotting or graphics devices:

Pen plotters (HP7221, HP7580)

Dot matrix printers (Versatec, Varian, Printronix)

Interactive graphics devices with bitpads (AED512, Metheus Omega 440)

The tape contains two 1600-bpi tar format files:

1) UW/NW VLSI Design Tools, Release 2.1.

2) VLSI Tools for 4.2 Berkeley UNIX as distributed by Berkeley (1984 versions).

The first file includes complete, self-contained tools as well as modifications to several Berkeley tools contained in the second file. Organization of the first file is as follows:

bin	executables and shell scripts
doc	user manuals
include	source header files for some tools
lib	archived libraries and other stuff
man	UNIX programmer's manual entries for each tool
src	source code, make files, and installation scripts

To install the tools:

1. Copy the first file on the tape to a suitable directory (on our system it's /src/vlsi/vlsi-tools) using the *tar* command. If you do not already have the 4.2 Berkeley Tools you will need to copy the second file of the tape into some other directory.
2. Set environment variable UW_VLSI_TOOLS to the full path name of this directory (e.g. setenv UW_VLSI_TOOLS /src/vlsi/vlsi-tools). (Put this in your .login file.)
3. Set (.login file again) the following environment variables:

   PATH to some path that includes $UW_VLSI_TOOLS/bin
   PLAPPATH to some path that includes $UW_VLSI_TOOLS/lib/technology
   RNLPATH to some path that includes $UW_VLSI_TOOLS/lib/rnl

   Directory $UW_VLSI_TOOLS/bin should precede /usr/ucb and /usr/bin if you want to take advantage of the new *man* command that accesses $UW_VLSI_TOOLS/man, ~cad/man (Berkeley VLSI tools), and /usr/man.
4. Cd to $UW_VLSI_TOOLS/src and run 'MAKE man' to initialize the manual pages.
5. If you are running Berkeley 4.1 you will have to run MAKEALL (in $UW_VLSI_TOOLS/src) to recompile everything. Most of the tools have been run successfully on 4.1, but you should add a -Dbsd41 flag in src/lib/libutil/makefile when compiling on 4.1.

There are several improvements (including man pages) and bug fixes for the 4.2 Berkeley tools available in $UW_VLSI_TOOLS/src/ucb-cad.4.2. These may be installed using the INSTALL-UCB script in $UW_VLSI_TOOLS/src. You may wish to review the README file in $UW_VLSI_TOOLS/src/ucb-cad.4.2 and edit INSTALL-UCB appropriately if you want only some of the changes. Since only the changed source files are included in $UW_VLSI_TOOLS/src/ucb-cad.4.2, you must follow the procedures documented with the Berkeley VLSI tools to complete the installation after using INSTALL-UCB.

The following miscellaneous sources are included:

Sources for lpr that complement the changes to cifplot and allow cifplot to be used with a Printronix.

Sources for a parallel interface device driver for a Metheus omega 440 color graphics display. A DEC DR-11W is needed in addition to the Metheus.

2.0 OVERVIEW OF VLSI DESIGN TOOLS

ENHANCEMENTS SINCE RELEASE 2.1

A major improvement in the design system has been the addition Coordinate Free LAP (CFL), a library of "C" procedures that facilitates the layout of VLSI designs. CFL contains a variety of operators for juxtaposing, transforming and replicating hierarchies of cells. Although CFL has sufficient functionality to allow specification of arbitrary rectlinear mask geometries, it is mainly intended to be used in the chip assembly mode. Hence the typical application would be to use the graphical editor Caesar to generate lower level cells (or tiles) and then use CFL to to assemble these cells into higher level modules. Routing facilities are provided which generate a variety of planar and nonplanar wire patterns used to connect functional blocks. CFL replaces the aging Pascal-based layout facility PLAP.

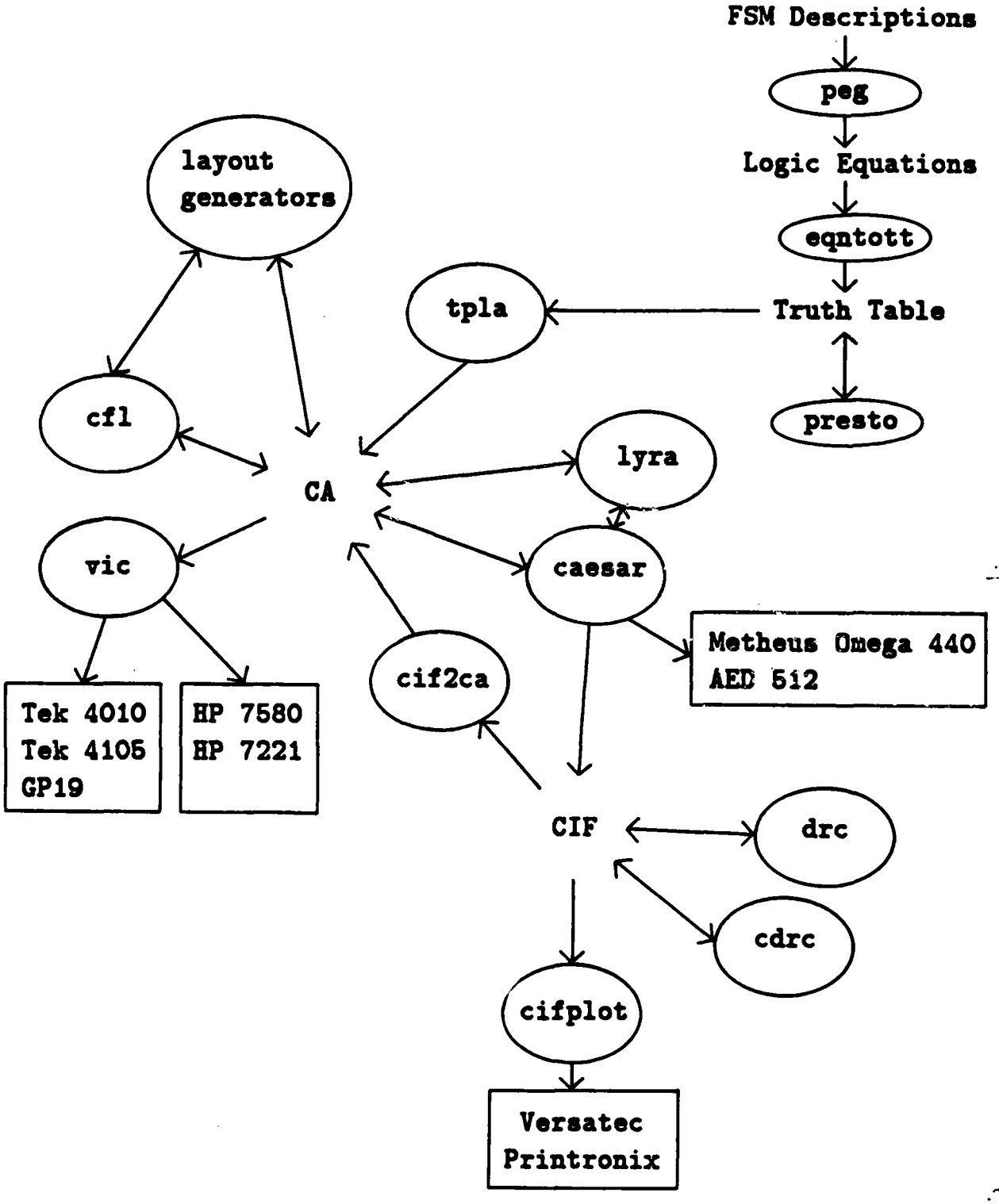
A number of layout generators have been written with the CFL facility. Each produces a layout consistent with the MOSIS 3 micron bulk CMOS specification. The multiplier generator *mult* produces a NxM two's complement multiplier with ripple carry addition. The generator *decNor* produces a dynamic nor form decoder with an arbitrary number of address bits and banks. A padframe generator *pads* produces a MOSIS - acceptable padframe with input, output and tristate pads instantiated according to user specifications.

Several new programs have been added to allow easier construction of RNL control files. These include *gen_control* and *gen_time*. A new display program, *simscope*, will display signal behavior derived from *rnl* or *spice* on several different graphics terminals.

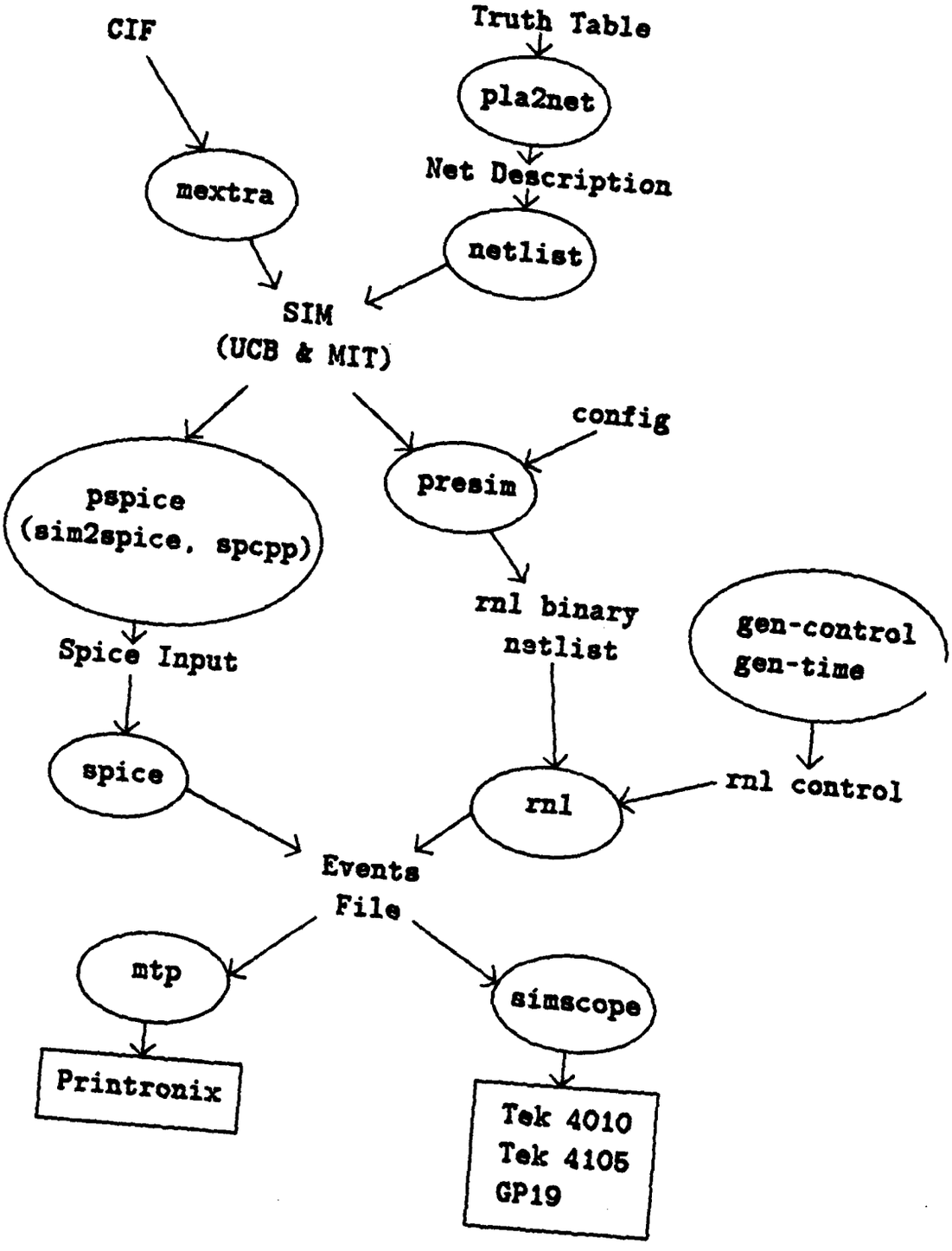
The design system described in the rest of this manual contains a number of tools from the 1984 Berkeley VLSI Tools Distribution. Since March 1985, Berkeley has distributed a 1985 toolset that includes *magic*, *mpack*, *cifplot*, *crystal*, *esim*, and various PLA tools. For information on these tools contact

Prof. John K. Ousterhout
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720
(415 642 0865)

LAYOUT TOOLS : FUNCTIONAL CHART



SIMULATION TOOLS : FUNCTIONAL CHART



TOOL DESCRIPTIONS

The following is a brief overview of the vlsi-tools we are distributing. An asterisk (*) appears after the name of tools that are contained in the 1984 Berkeley Distribution.

Functional Design Tools

peg * Translates a language description of a finite state machine into logic equations compatible with *eqntott*. (Gordon Hamachi, UCB)

eqntott * Converts logic equations into a truth table format to be used as input to *mkpla* or *tpla*. (Bob Cmelik, UCB)

presto Tries to minimize the number of product terms in a truth table. (UCB)

Layout Construction Tools

caesar * A display editor for manhattan designs written at Berkeley. Runs on AED512 or Metheus Omega 440 color displays. Requires a design to be inputted in *caesar* format but will optionally output in CIF. (John Ousterhout, UCB)

cfl A library of C procedures for assembling *caesar* formatted cells into modules. Employed by *pads*, *decNor* and *mult*. (William Beckett, UW/NW VLSI Consortium)

pads A generator for constructing a MOSIS padframe with user-specified pad types. (Wayne Winder, UW/NW VLSI Consortium)

mult A generator for constructing an NxM multiplier in MOSIS CMOS. (Wayne Winder, UW/NW VLSI Consortium)

decNor A generator for constructing a MOSIS CMOS "nor" form decoder with an arbitrary number of inputs and banks. (David Morgan, UW/NW VLSI Consortium)

tpla * Technology independent pla artwork generator. Employs a Caesar-generated template and a truth table. (Robert Mayo, UCB)

Layout Display Tools

cifplot * Berkeley program that plots a CIF design in stipple patterns on Versatec or Printronix dot-matrix printers. (Dan Fitzpatrick, UCB)

vic Display program for a Tektronix 4010 compatible device with penplot options for HP 4- and 8-pen plotters. Takes Caesar files. (Bruce Yanagida, Boeing; Larry McMurchie, UW/NW VLSI Consortium)

Design Rule Checkers

drc Design rule checker from Carnegie-Mellon. Checks MOSIS NMOS (buried contact) rules on Manhattan CIF designs. (Dorothea Haken, CMU)

drcscript Merges the design rule violation files created by *drc* with the CIF design file for display purposes. (Dorothea Haken, CMU)

cdrc Carnegie-Mellon design rule checker with MOSIS CMOS rules. (Jagannathan Ramanujam)

cdrcscript Merges *cdrc* violation files with the CIF file.

lyra * Performs hierarchical design rule check on a Caesar-formatted design using a corner based algorithm. NMOS and CMOS rulesets are available. (Michael Arnold, UCB)

Circuit Extractors

mextra * Extracts a **.sim** file from a CIF input file for use in *presim* and *sim2spice*. (Dan Fitzpatrick, UCB)

Simulation Tools

spice2g6 The well known device level circuit simulator with minor mods to output an additional file that allows multiple time series plots to be made. (Lawrence Nagel, UCB)

rnl An event driven "timing" simulator. It uses logic levels and a simplified circuit model to estimate timing delays through digital circuits. (Chris Terman, MIT)

mtp Produces signal behavior plots on a Printronix printer, the simulation output from either *spice* or *rnl*. (William Beckett, UW/NW VLSI Consortium)

simscope Displays signal behavior plots on a Tek 4010 or Tek 4105. Input can come from either *spice* or *rnl*. (Rudolf Nottrott)

Filters and Utilities

cif2ca * Converts from CIF format to *caesar* format. (Peter Kessler, UCB)

netlist Generates circuit descriptions in the form of a **.sim** file. (Chris Terman, MIT)

presim Converts a **.sim** file into the binary format required by *rnl*. In the process *presim* simplifies the circuit by identifying gates in the circuit. (Chris Terman, MIT)

pspice Runs *sim2spice* and *spcpp*. In addition to running those programs it concatenates various files so as to create a complete Spice input deck. (Rob Fowler, UW/NW VLSI Consortium)

sim2spice * Reads a **.sim** file containing a description of a circuit and writes a **.names** file and a **.spice** file. The **.names** file contains a translation from node names in the **.sim** file to the Spice node numbers. The **.spice** file contains a description of the devices in the circuit in a form acceptible to Spice. (Dan Fitzpatrick, UCB)

spcpp Facilitates the writing of Spice input by allowing the user to refer circuit nodes using mnemonic node labels rather than Spice node numbers. (Rob Fowler, UW/NW VLSI Consortium)

spice A *csh* script for running *spice2g6*. (Lawrence Nagel, UCB)

gen_time A program for specifying input signal patterns to *rnl*. (Riekus Koeman, Fluke Manufacturing)

gen_control Constructs rnl control files. (Riekus Koeman, Fluke Manufacturing)

pla2net Creates a PLA description for use in *netlist*. Input is a truth table in the format of *presto* and *eqntott*. (Riekus Koeman, Fluke Manufacturing)

A VLSI DESIGN STRATEGY

This document describes a strategy for creating VLSI designs with the tools of this release. Although this strategy is a sequence of operations, the designer should realize that iterating between them may yield higher performance designs. It begins by partitioning the project into modules of manageable size, then for each module:

1) Define and partition the design with a high level description.

   Block diagram, schematics or text files.

2) Model the design as a network of transistors.

   2a) Create the network of transistors.
   2b) Create stimuli for this network and simulate.
   2c) Repeat steps 2a and 2b until the design is stable.

3) Implement the design as a collection of integrated circuit layers.

   3a) Create this layout.
   3b) Verify that the layout complies with the design rules.
   3c) Extract a transistor network model from the layout.
   3d) Create stimuli (modified from 2b) and simulate.
   3e) Repeat steps 3a,3b,3c,3d until the design is stable.

4) Fabricate the design.

5) Test the design.

   5a) Exercise the integrated circuit with simulation stimuli.
   5b) Document yield, performance and simulation discrepancies.

Step 1 is an abstract description of the design's operation. Design problems can be rectified at this point far cheaper than at any other. A clear and concise description enhances the chance of finding design problems and eliminating interface problems with other designs. Investing time in this step is consistent with the "topdown" design style common in software development.

Step 2 can identify errors which would take many hours to fix if discovered in a completed layout. Some designers eliminate this step believing that design time will be reduced, but the validity of this assumption is dependent on the design's complexity and the designer's experience. The novice should approach this shortcut with great caution. Commercial designers avoid this tradeoff with schematic capture programs that translate the high level description directly into the transistor network model.

The following outline describes the appropriate tool for each step in the design cycle:

1) High Level Description
2) Transistor Network Model
    Create Network
        Transistor Modeled as Switch
            By Hand: NETLIST, PRESIM
            Generate PLAs: PEG,EQNTOTT, PRESTO, PLA2NET, PRESIM
        Detailed Transistor Model: NETLIST, PSPICE
        Define Stimulus and Simulate
            Transistor Modeled as Switch: RNL, SIMSCOPE, MTP
            Detailed Transistor Model: PSPICE, SPICE, SIMSCOPE, MTP

3) Integrated Circuit Layout
    Create Layout
        By Hand: CAESAR, CFL
        Generate PLAs: PEG, EQNTOTT, PRESTO, TPLA
        Generate Pad Frames: PADS
        Generate Multipliers: MULT
        Hardcopy: CIFPLOT, VIC, PENPLOT
    Verify Design Rules: LYRA, CDRC
    Extract Transistor Model
        Transistor Modeled as Switch: CAESAR, MEXTRA, PRESIM
        Detailed Transistor Model: CAESAR, MEXTRA, PSPICE
    Define Stimulus and Simulate
        Transistor Modeled as Switch: RNL, SIMSCOPE, MTP
        Detailed Transistor Model: PSPICE, SPICE, SIMSCOPE, MTP

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. "std" works well for most monitors, "pale" 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 semi-colons (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 "*".

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 "*SI".

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

cdrc, cdrcscript, drc, drcscript - CMOS-BULK 3 micron and NMOS VLSI design rule checkers

SYNOPSIS

cdrc [-ku] *basename*.cif
cdrcscript *basename*.cif **drc [-ku]** *basename*.cif [*lambda*]
drcscript *basename*.cif

DESCRIPTION

Cdrc analyzes a CMOS CIF file for geometric rule violations using MOSIS cmos-bulk 3 micron process rules. *Drc* analyzes an NMOS CIF file for geometric rule violations using MOSIS (buried contact) rules. Both *cdrc* and *drc* are limited to rectilinear, orthogonal geometry. Wires are taken apart into rectangles, and round flashes are approximated by squares. Polygons and non-manhattan rectangles are simply ignored.

The options are as follows:

-k Keep around all intermediate files.

-u Keep around files of unfiltered error messages.

For large files, *cdrc* or *drc* should be run in batch mode, as a 7000 transistor chip takes over 2 11/780 cpu hours.

When *cdrc* or *drc* find violations, they create CIF files of rectangles marking the geometric edges involved. These markers are placed on the error layer (CZ) for cdrc and on the glass layer for drc. Separate files are created for each class of error, named err.*errortype*.*basename*.

To abort *cdrc* or *drc* hit the *BREAK* key and wait while it outputs some error messages until it eventually quits.

{*C*}*drcscript* will merge {*c*}*drc* output files, labels indicating error type, and the original CIF file into a single file, drc*basename*.cif. If this file is processed by *cif2ca* the results may be viewed with *caesar*. Errors show up as light blue boxes in the error layer for *cdrc* or as orange boxes in the glass layer for *drc*. Each pair of boxes involved in an error will have an associated *errortype* label which will be located at the midpoint between the centers of the two boxes.

MOSIS CMOS/BULK 3 micron process rules checked by *cdrc*:

Errortype	Microns	Rule
wWp	3	P-Well width
sWp	9	P-Well to P-Well spacing assuming all p-wells are connected to vss
dW	4	Diffusion size
dS	4	P+ diffusion to P+ diffusion spacing
	4	N+ diffusion to N+ diffusion spacing
	4	N+ diffusion to P+ diffusion spacing outside P-well
	4	N+ diffusion to P+ diffusion spacing inside P-well
pWp+DS	8	P+ diffusion in N-substrate to P-well edge spacing
Wpn+WnS	7	N+ diffusion in N-substrate to P-well edge spacing
pWn+DS	4	N+ diffusion in P-well to P-well edge spacing
pW	3	Poly width
pS	3	Poly to poly spacing
pSd	2	Field poly to diffusion spacing
pOg	3	Poly gate extension over field
gpSd	3	Gate poly to diffusion spacing
p+Od	2	P+ mask overlap of diffusion
	2	N+ mask to P+ diffusion spacing

Tn+S	3.5	P+ mask overlap of poly in diffusion
p+S	3	P+ mask to P+ mask spacing in diffusion
	3	N+ mask to N+ mask spacing in diffusion
Errortype	Microns	Rule
Dp+s	2	P+ mask to N+ diffusion spacing
	2	N+ mask overlap of diffusion
Tp+s	3.5	N+ mask overlap of poly in diffusion : P+ mask to poly spacing in diffusion inside P-Well
bcut		Cut must have metal and (poly or diffusion) underneath
wC	3	Contact width
cL	8	Maximum contact length
cS	3	Contact to contact spacing
pOc	2	Poly overlap of contact
PMCx	2.5	Poly overlap of contact in direction of metal
cSpc	3	Contact to poly channel spacing
mOc	2	Metal overlap of contact
dOc	2	Diffusion overlap of contact
cp+s	3	Contact to P+ and N+ mask spacing
scfp+	4	Shorting contact extension into P+ diffusion
scfn+	4	Shorting contact extension into N+ diffusion
mW	3	Metal width
mS	4	Metal to metal spacing
m2w	5	Metal2 width
m2S	5	Metal2 to metal2 spacing
c2w	3	Via width
c2S	3	Via to via spacing
m2Oc2	2.5	Metal2 extension over via
mOc2	2.5	Metal extension over via
CC2s	3	Via-cut separation
dOv	3	Diffusion overlap of via
dSv	3	Diffusion to via spacing when they dont overlap
pOv	3	Poly overlap of via
pSv	3	Poly to via spacing when they dont overlap
M2Pst	1	Metal2, metal and poly intersection 1 width
MPMM2x	4	Metal extension over the above intersection
PPMM2x	3	Poly extension over the above intersection
PM2st	5	Metal2 metal intersection(no poly) to metal2 poly intersection(with no metal) spacing
MPM2st	5	Metal2 poly intersection (no metal) to metal spacing
PPM2st	5	Metal2 metal intersection (no poly) to poly spacing

NMOS rules checked by *drc*:

Errortype	Rule	Lambda
dS	diffusion spacing	3.0
iOg	implant-gate overlap	1.5
iSg	implant-gate spacing	2.0
pS	poly spacing	2.0
pOg	poly-gate overlap	2.0
pSd	poly-diff spacing	1.0
cS	cut-cut spacing	2.0
dcSg	diff-cut to gate	2.0
mW	metal width	3.0
iNOg	implants with no gates	

XC	cuts with no D or P	
dW	diffusion width	2.0
ntdW	non-xtr diff width	2.0
iS	implant-implant	1.5
pW	poly width	2.0
gW	gate width	2.0
cW	cut min width	2.0
cL	cut max length	6.0
mOc	metal-cut overlap	1.0
dOc	diff-cut overlap	1.0
pOc	poly-cut overlap	1.0
sBP	buried-poly spacing	2.0
sBD	buried-diffusion spacing	2.0
oBD	buried-diffusion overlap	2.0
oBU	buried-contact surround	1.0
bW	buried-contact width	2.0

SEE ALSO

caesar(cad1), cif2ca(cad1)

A Geometric Design Rule Checker, Dorothea Haken, VLSI Document V053, Carnegie Mellon, 9 June 1980.

FILES

basename.cif
err*.errortype.basename*
drc*basename*.cif
err*basename*.cif

AUTHOR

Dorothea Haken (CMU)

BUGS

The poly-overlap-gate check fails when the overlap is exactly zero (*drc* only).

Spacing checks do not consider mutual connectivity. Sometimes weird things will happen, and the generated spurious errors can be filtered by the bin_filter program, which examines local connectivity. Cuts in diffusion or poly that do not have metal covering are not reported.

Cuts in diffusion or poly that do not have metal covering are not reported (*drc* only).

Diagonal spacing checks do not consider the true diagonal distance.

SUGGESTIONS

Do not have basenames beginning with a number. Otherwise, this leads to serious errors in that *cdrc* assumes that to be the lambda value.

Try to have as short a basename as possible. This is because some flavors of UNIX restrict the length of filenames. Some of the intermediate files that are generated have quite long names.

The default lambda is 50 centimicrons for the *cdrc* routines. This scaling is done to overcome the inability of the routines to check for non-integer lambda violations.

It is advisable to run {*c*}*drc* in the background (batch mode), directing the output to a file, so you can look at the file later if needed.

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 for displaying VLSI circuits

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

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

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

-w *xmin xmax ymin ymax*

(window) This option 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) This 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 *layerlist*

(layer) Normally all layers are plotted. This option specifies which layers NOT to plot. The *layerlist* 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.

-N **(Printronix)** Send output to the Printronix.

-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 -w, -l, and -a 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**) This option 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 *patternfile*

(**Pattern**) This option lets you specify your own layers and stipple patterns. *Patternfile* 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 *fontfilename*

(**Font**) This option indicates which font you want for your text. The *fontfilename* 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*

SEE ALSO

cadrc(cad5)

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

AUTHOR

Dan Fitzpatrick (UCB)

MODIFICATIONS

(UW/NW VLSI Consortium, University of Washington)

BUGS

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

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

The –X option does not work for non-manhattan circuits.

NAME

decNor – Generates CMOS dynamic NOR form decoder layouts.

SYNOPSIS

decNor [*options*] **Inputs** [*OutFile*]

DESCRIPTION

DecNor is a program for generating CMOS dynamic NOR form decoder layouts in the "caesar" format. **DecNor** constructs caesar composition cells from caesar leaf cells and/or other composition cells. All caesar cells reside in the ./ca directory. Leaf cells have names of the form decNor_*.ca while composition cells have names of the form "OutFile"*.ca. Leaf cells must be copied from $UW_VLSI_TOOLS/lib/generators/decNor into ./ca before running **decNor**. The completed layout resides in "OutFile".ca. **Inputs** are the number of inputs to the decoder. "OutFile" can not begin with the string "decNor". The default for "OutFile" is the string "decGen".

As **decNor** is a cfi-based program it creates files of the form *.bd in ./ca.

The following table describes decNor's *options* although an abreviated listing can be obtained by invoking **decNor** with no arguments. Options prepended by "-" are active while those with "*" have not been implemented.

-f Stripped down layout for floor planning. Cells which occupy a large part of the decoder are represented in dummy layers allowing faster layout generation.

-t Layout of worst case path for timing estimates. Cells which are not part of the slowest electrical path are represented in dummy layers allowing faster generation, extraction and simulation.

-s A schematic of the decoder. Cells are represented as symbols (wires and transistors) drawn in black ink (labels) on a yellow background (P+ mask).

*p P-type decode transistors. Since N-type transistors have a lower on resistance they are the default decode transistor type.

-l Labels are added to inputs and outputs. Since labels increase the generation time they are not added as the default. When included they are prepended with "OutFile".

-b *banks*
The array of decode transistors will be repeated " *banks* " times. This feature can be used to distribute decoder outputs to a number of places with minimal additional area. Default is one.

*o *outs*
Stretch decoder to give " *outs* " lambda output spacing. This option simplifies connection by abutment.

*i *ins*
Grow decode xsistors to give " *ins* " lambda input spacing. This option allows the decoder to operate faster.

*v *ver*
Grow input inverters to fill vertical size of " *ver* " lambda. This option allows the decoder to operate faster.

*h *hor*
Grow evaluate/charge xsistors to fill horizontal size of " *hor* " lambda. This option allows the decoder to operate faster.

FILES

/ca/OutFile*.ca
/ca/OutFile*.bd
/ca/decNor_*.ca

SEE ALSO

caesar(CAD1), cfi(5.vlsi)

AUTHORS

David J. Morgan

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

gen_control – generate a control file for RNL

SYNOPSIS:

gen_control
(no arguments)

DESCRIPTION

gen_control is a program designed to quickly specify a control file for RNL simulation. *gen_control* provides for the proper insertion of quotes and use of parenthesis.

Typical file extensions to the basenames are assumed.

When starting up the *gen_control* program, the user will be prompted for the necessary information to be provided.
Assumed standard libraries are:
nwstd.l & nwsim.l

Prompts:

1. Basename:
The control file will be written in: *basename.l*
In the *.l* or control file assumed extensions are:
for the log file: *basename.rlog*
for the network: *basename*
for the plotfile: *basename.beh*

2. Comment:
A one line comment, which could be a short comment about the simulated circuit can be entered.

3. Simulation step increment value:
Enter the value of the simulation step in 0.1 ns units. The appropriate lisp command is automatically generated.

4. Definition of normal vectors:
To define a vector enter its name.
Then there will be a prompt for its type (bit, bin, oct, hex,dec)
Followed by a prompt for its elements.
A <CR> means skip this entry.

5. Definition of single indexed vectors:
Enter *basename* and after prompts: type, start index and number of elements.

6. Definition of a set of double indexed vectors:
Enter *basename* and after prompts: type, indexsize1 and indexsize2.

7. Definition of report for end of simulation step:
One of two types can be specified:
Just a <CR> specifies the normal def-report contents;
<any character><CR> specifies an optional type in which multiple vectors with double indexed nodes can be specified.
Next there will be a prompt for a comment to be included in every report (this portion only is

optional).
Then there will be prompting until a <CR> is entered for

nodenames (just enter the names) or
a vector name (first enter 'vec' and then the name).

In case of the optional report format, the multiple vector specification format is obtained by reponding with *'veci'*. Additional prompts will follow for basename and size.

8. Selection of output mode: logic analyzer style output:
Enter any character for selecting logic analyzer style output and a <CR> for standard output.
A report stating the order of columns in the output of RNL will be automatically generated.

9. Selection of output mode: glitch detection reporting:
Enter any character for selecting glitch detection and a <CR> for standard reporting of transients.

10. Definition of nodes with transient or glitch reporting:
Individual node names, vectors with single indexed node names and vectors with double indexed node names can be specified. Respond appropriately for names vectorsizes.

11. Definition of logic trigger conditions:
There are prompts for defining trigger conditions on individual node names, single vectors in invec type format, and single vectors in bitinvec type format.

12. Definition of additional RNL simulation set-up commands:
Enter the desired RNL commands. Terminate with an additional <CR>.

13. Definition of a timing pattern file:
Respond with <CR> if there is no such file (unlikely) or any other character if such is the case.
The filename assumed is: **basename.time**

14. Definition of wrap-up RNL commands:
Enter the desired simulation wrap-up commands (often just 'exit').
There is no syntax checking in *gen_control*. *gen_control* will put the quotes and parentheses at the right places. Any errors can be easily corrected using a standard text editor on the output file: *basename.1*.
This file can be inspected for correctness. Errors may be reported by RNL when running the simulation.

FILES

The output file is an ascii file and can be inspected. The files containing the library functions, network etc. must be at the correct place.
uwstd.1, uwsim.1, basename.1, basename, basename.rlog, basename.beh, basename.time

SEE ALSO

gen_time manual instructions

AUTHOR

Henriecus Koeman, John Fluke Mfg. Co., Inc.

DIAGNOSICS

none

BUGS

Please let the author know.

NAME

gen_time – generate a stimulus pattern for rnl.

SYNOPSIS

gen_time input_file output_file

DESCRIPTION

gen_time is a program designed to quickly specify input signal patterns which can be read by the lisp command interpreter of RNL. gen_time accepts a simple syntax without quotes and parentheses and accepts a simple means for defining states or commands for specific moments in time. The output of gen_time is typically read by the main control file, which contains the set-up information for the simulation. This control file can easily be obtained using the gen_control program. One of the commands should "load" the outputfile of gen_time.

Syntax summary:

time_range <start_time> <stop_time>
(must be the first command)
node_name <period> <state1> <time1> <state2> <time2> ...
invec <vector_name> <period> <value1> <time1> ....
bitinvec <vector_name> <period> <bitvalues1> <time1> ....
(note no spaces between individual bitvalues as in the
equivalent rnl command!)
command <period> <rnl_command1> <time1> .....
(no alternate syntax allowed in rnl_commands here)
report <period> <time1> <time2> .....
(report 1 0 generates a report after every time
step)
(must be the last command in the input_file)
mask <period> <enable_time> <disable_time>
(applies only to command line immediately following)
maskinv <period> <disable_time> <enable_time>
(applies only to command line immediately following)

FILES

The output file is an ascii file and can be inspected for programmed activity as a function of the time increments.

FURTHER EXPLANATIONS

The rules for the input file are discussed in more detail in the following, in particular those for the more complex waveforms.

Rule #1: Comments.
All lines starting with a semicolon are considered comments and are ignored.

Rule #2: Simulation interval definition must come first.
The first command in the .stim file must be the specification of the simulation interval; syntax and example:

time_range <start_time> <stop_time>
time_range 0 50

Note: Every value of time is in number of simulation step increments 'incr'. The global variable 'incr' is assigned a value with (setq incr <number>) where the number is

the size of the simulation step in 0.1 ns; this is done in the (".l") RNL control file.

Rule #3: The report definition must come last.
The last command in the .stim file must be the specification of how often RNL should print a report (using the def-report specification); syntax and examples:

report <period> <time1> <time2> ....

report 2 1	(report every 2 simulation steps at the end of interval 1; which occur at t=2, 4, 6, etc.)
report 10 3 6 9	(report every 10 simulation steps at the end of intervals 3,6 and 9: t=4, t=7, t=10, t=14, t=17, t=20 etc)
report 1 0	(report at the end of every simulation step)

Rule #4: How input signals are specified.
Signals are defined in one of the following ways:

nodename	(states must be l, h, u or x)
invec vectorname	(states must be a numerical type: decimal, octal (leading "0"), hexadecimal (0x....) or binary (0b...))
bitinvec vectorname	(states can be any combination of 1,0,u and x; no spaces between the elements)

followed by:
the period, and a number of combinations:
<state> <time>

If the period is "0" the specification relates to a one time event (the period is really infinity!).

Syntax and example for a simple waveform definition for simple node:
node_name <period> <state1> <time1> <state2> <time2> .....
node-name1 10 h 0 l 2 u 5 x 8
period is 10 simulation steps, signal h at t=0, l at t=2, u at t=5 and x at t=8; signalchanges repeat themselves at t=10, 12, 15, 18, 20, 22, etc..

Syntax and examples for a numerical vector definition (no undefined states can be specified in this case!):

invec vectorname <period> <state1> <time1> .....

invec name 10 0xa 0 0b1111 2 07 5 3 8

period is 10 simulation steps, vector is the hexadecimal "a" at t=0, binary 1111 at t=2, octal "7" at t=5 and a decimal "3" at t=8. Again, the pattern is repeated 10 simulation steps later.

invec name 0 0xa 0 15 5 017 9

The pattern is a single event: name is hexadecimal "a" at t=0, a decimal "15" at t=5; an octal "17" at t=9. This pattern does not repeat itself!

Syntax and examples for a bitvector definition:
bitinvec <vector_name> <period> <state1> .....
bitinvec vectorname 20 0000 0 1111 5 uuuu 10 xxxx 15
bitinvec vectorname 10 0x1x 0 u00x 5

Rule #5: Use of regular RNL commands allowed only with standard lisp syntax.
RNL commands can also be inserted in the same manner as node and vector stimulus; only the standard rnl syntax (with parentheses is allowed):

syntax:
command <period> <(rnl_command1)> <time1> .....

Rule #6: Masking of input signals and commands.
Except for the time_range command ALL gen_time commands are subject to mask commands, with will blank out the impact of the next command line immediately following the mask command line. After processing this next command line the mask is reset to a default which is a full enable. There are two mask commands:

'mask' and 'maskinv'

'mask' and 'maskinv' themselves are defined as having a period (a one time mask has a period of '0') and only 1 enable and only 1 disable time.

syntax:
mask <period> <enable_time> <disable_time>
maskinv <period> <disable_time> <enable_time>

Example:
mask 0 10 20
node2 5 h 0 l 5 u 10 x 15
will blank out any activity from node2 before
time increment 10 and after time increment 20.
maskinv 0 10 20
node3 5 h 0 l 5 u 10 x 15
will allow only node3 statements to be
effective before time increment 10 and
after time increment 20.

The commands scheduled for the time coinciding with the enable time of the mask will be effective, while the commands schedule for the time coinciding with the disable time will be disregarded.

Example of a typical stimulus file:

```
; Timing file for basic CRC Counter
; Simulation time:
time_range 0 36
; Run the clock at all times:
cl 2 l 0 h 1
; Reset:
r 0 h 0 l 1
; The following sequence is designed to exercise all nodes!
in 0 l 0 h 2 l 12 h 20 l 26 h 28 l 32 h 34
; We will start reporting the unchanged nodes just before
; the last ff changes state, which is at time increment 32:
mask 0 32 36
command 1 (printf "nodes unch:%S\n" (unchanged-since 100)) 0
; We report the state after every simulation step:
report 1 0
```

USING PATTERNS DEFINED USING GEN_TIME.

The output file from gen_time with the shell command:

gen_time basename.stim basename.time

Within the regular RNL control file (basename.l) one should include:

(load "basename.time")

AUTHOR

Henriecus Koeman, John Fluke Mfg. Co., Inc.

DIAGNOSICS

In case of an error in the inputfile gen_time will most likely print the first line number and the line itself where the error was detected and then terminate prematurely.

BUGS

Please let the author know.

NAME

lyra – Performs hierarchical layout rule check on caesar design.

SYNOPSIS

lyra [-vs] [-o output] [-p path] [-r ruleset] [-t technology] rootCaesarFile,
or
lyra -o [-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.

cmos-pw3

MOSIS 3 micron bulk cmos process, (see below for details). This is the default ruleset for technology cmos-pw.

cmos-M1v1

MOSIS 3 micron bulk cmos process, (see below for details).

isocmos

GTE 5 micron isocmos process.

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.

-z **(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 -z 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.

DIAGNOSTICS

CMOS-PW3 MOSIS 3 MICRON CMOS DESIGN RULES, V1.0

"C s" contact-contact separation: 3u
"C w" contact width: 3u
"C2 f" metal2 extension around via: 2.5u
metal extension around via: 2.5u
"C2 s" via-via separation: 3u
"C2 w" via width: 3u
"C/C2 s" via-cut separation: 3u
"D s" active area-active area separation: 4u
"D w" active area width: 4u
"Dn+ w" N+ active area width: 4u
"Dp+ w" P+ active area width: 4u
"Dw w" P+ active area (not gate) width: 3u
N+ active area (not gate) width: 3u
"D/C2 s" if active area is not under via, via-active area separation: 3u
"D/C2 x" if active area is under a via, active area extension around via: 3u
"D/p+ s" N+ active area to P+ spacing: 2u
"M s" metal-metal separation: 4u
"M w" metal width: 3u
"MP/PMM2 x" step missing for metal2 step coverage
"M2 s" metal2-metal2 separation: 5u
"M2 w" metal2 width: 5u
"M2/P st" metal2/metal/poly width: 1u
"M/PMM2 x" metal step width for metal2: 4u
"M/P/M2 st" poly-metal separation when under metal2 with no overlap: 5u
"P s" poly-poly separation: 3u
"PMC x" extra .5 micron in direction of metal in poly-metal contacts
"Pw w" poly (not gate) width: 3u
"P/C2 s" if poly is not under via, via-poly separation: 3u
"P/C2 x" if poly is under a via, poly extension: 3u
"P/D s" poly-active area separation: 2u
"P/M/M2 st" poly-metal separation when under metal2 with no overlap: 5u
"P/PMM2 x" poly step width for metal2: 3u
"T w" Gate area width: 3u
"T/C s" contact to gate separation: 3u
"T/n+ s" P+ extension around gate outside p-well: 3.5u
"T/p+ s" gate inside p-well to P+ (of ohmic contact) separation: 3.5u
"VIA f" via has obtuse corner
"Wp s" p-well to p-well separation: 9u
"Wp w" p-well width: 3u
"Wp/n+Wn s" N+ active area (ohmic contact) to p-well

separation: 7u
"c f" metal and (poly or active area) required under cuts
metal extension around cut: 2u
active area extension around cut: 2u
poly extension around cut: 2u
"pW/n+D x" p-well extension around active area: 4u
"pW/p+D s" separation of p-well from P+ active area: 8u
"p+ s" p+ to p+ separation: 3u
"p+/D x" P+ extension around P+ active area: 2u
"sc f" split ohmic contact must be 4 microns into P+ active area and 4 microns into N+ active area
"tr f" malformed poly or active area abuttment: 3u extension
"tr p" polarity: P+ implanted transistor in p-well
polarity: N+ implanted transistor outside p-well

CMOS-PW3 MOSIS 3 MICRON CMOS DESIGN RULES, V1.1

Same as 1.0 with the following exceptions:

modified rules:
"C2 f" metal2 extension around via: 2u
metal extension around via: 2u
"M2/P st" metal2/metal/poly width: 3u
"M/PMM2 x" metal step width for metal2: 3u
"M/P/M2 st" poly-metal separation when under metal2 with no overlap: 3u
"P/C2 s" if poly is not under via, via-poly separation: 4u
"P/C2 x" if poly is under a via, poly extension: 4u
"P/M/M2 st" poly-metal separation when under metal2 with no overlap: 3u
"P/PMM2 x" poly step width for metal2: 3u

new rule:
"D W" Active Area transistor abuttment width: 4u

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

mexnodes – integrate intermediate nodes extracted by mextra with the original caesar design.

SYNOPSIS

mexnodes [*options*] *basename*

DESCRIPTION

Mexnodes is a shell script that uses *cif2ca* and *caesar* to generate a Caesar-format file. This file allows the user to view the intermediate nodes named by *mextra* on the original design. *Mexnodes* can be helpful when a simulation tool reports errors at a node not named by the user, as such errors are sometimes hard to locate. The output file created by *mexnodes* is named **mx***basename*.**ca**. This file can be then viewed using *caesar* in order to find a given node.

The *options* are as follows:

-t *technology*

Technology is one of nmos, isocmos, or cmos-pw. Default is nmos.

-l *lambda*

Lambda specifies the centimicrons to lambda correspondence of the design. Default is 200 centimicrons per lambda.

FILES

basename.ca
mx*basename*.ca
basename.nodes
basename.cif

SEE ALSO

caesar(CAD1), cif2ca(CAD1), mextra(CAD1)

AUTHOR

Terry J. Ligocki

BUGS

NAME

mextra – Manhattan circuit extractor for VLSI simulation

SYNOPSIS

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

DESCRIPTION

Mextra will read the file *basename*.**cif** and create a circuit description. From this circuit description various electrical checks can be done on your circuit. The circuit description is directly compatible with *esim*, *powest*, and *erc*. There are translation programs to convert *mextra* output to acceptable *spice* input (see *sim2spice*, *pspice* and *spcpp*).

Mextra creates four new files, *basename*.**log**, *basename*.**al**, *basename*.**sim** and *basename*.**nodes**. After *mextra* finishes it is a good idea to read the **.log** file. This contains general information about the extraction. It has a count of the number of transistors and the number of nodes, and it contains messages about possible errors. The **.al** file is a list of aliases which can be used by *esim*. The **.nodes** file is a list of node names and their CIF locations listed in CIF format. It can be read by *cifplot* to make a plot showing the circuit with the named nodes superimposed. The form of this cifplot command is:

cifplot *basename*.**nodes** *basename*.**cif**

The **.sim** file is the circuit description for use with simulation programs and electrical rule checkers. The format of the **.sim** file is described in the man page **simfile**(5).

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 *esim* so that it will understand these aliases.

Attributes

In addition 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, in the same way that a node name is attached. 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 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.

Transistors

For each transistor found by the exractor a line is added to the **.sim** file. The form of the line is:

type gate source drain length width x y
g=*attributes* **s**=*attributes* **d**=*attributes*

Type can be one of three characters, '**e**' for enhancement, '**d**' for depletion, or '**u**' for unusual implant. (Unusual implant refers to transistors which are only partially in an implanted area. It will be necessary to write a filter to replace these transistors with the appropriate model in terms of enhacement and depletion transistors.) *Gate*, *source*, and *drain* are the gate, source, and drain nodes of the transistors. *Length* and *width* are the channel length and width in CIF units. *X* and *y* are the x and y coordinates of the bottom left corner of the transistor. *Attributes* is a comma seperated list of attributes. If no attribute is present for the gate, source, or drain, the **g**=, **s**=, or **d**= fields may be omitted.

The extractor guesses the length and width of a transistor by knowing the area, perimeter, and length of diffusion terminals. For rectangular transistors and butting transistors the reported length and width is accurate. For transistors with corners or for unusually shaped transistors the length and width is not as accurate.

It is possible to design a transistor with three or more diffusion terminals. The extractor considers these as *funny transistors*. They are entered in the **.sim** file in the form:

f*type gate node1 node2 ... nodeN xloc*

The 'f' is followed by the type : 'e', 'd' or 'u'. *Node1 ... nodeN* are the diffusion terminal nodes. As with any circuit with 'u' transistors, any circuit with 'f' transistors must be run through a filter replacing each of the funny transistors with the appropriate model in terms of enhancement and depletion transistors.

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 a node and substrate is in femto-farads. Capacitance values below a certain threshold are not reported. The default threshold is 50 femto-farads.

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
metal2	0.015	0.0
poly	0.05	0.0
diff(n)	0.10	0.1
diff(p)	0.10	0.1
poly/diff	0.40	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 -e 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:

L *node metal2Area metal2Perim metalArea metalPerim polyArea polyPerim diffArea diffPerim diffpArea diffpPerim*

Node is the node name. *x y* is the position of a point on the node. Currently this is always '0 0'. *metal2Area* through *diffpPerim* are the area and perimeter of the metal2, metal, poly, diffusion(n), and diffusion(p) layers in user defined units. (In addition the -e option causes transistors with only one terminal to be recorded in the .sim file as a transistor with source connected to drain.)

If the network is being extracted from the .cif file we suggest the node capacitance not be computed by mextra. Rather the -e option should be used. This puts the burden of computing node capcitance on the programs presim *and* sim2spice2. We feel this is advantageous because presim *and* sim2spice2 are filter programs linked directly to the type of simulation that is to be done. This will hopefully reduce some of the confusion associated with calibration.

Changing Default Values

As part of its start up procedure *mextra* reads two files: /usr/vlsibin/.cadrc and then a search for the first .cadrc from the current directory (.) to the the user's home directory is made. *Mextra* reads these files to set up constants to be changed without recompiling. The keywords for *mextra* are contained within the mextra environment of the .cadrc file. Declaration of

environments in the **.cadrc** file are described in .cadrc(5).

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

units *scale*

where *scale* is in centi-microns. The user may type in the chosen value for the scale directly.

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

mextra -u *scale basename*

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

areatocap *layer value*
perimetertocap *layer value*

value is atto-farads per square micron for area, and atto-farads per micron for perimeter. *layer* may be "poly", "diff", "metal", "metal2", or "poly/diff".

To set the capacitor values to those given in Mead and Conway the following lines would appear in the **.cadrc** file:

```
areatocap poly 40
areatocap diff 100
areatocap metal 30
areatocap poly/diff 400
perimetertocap poly 0
perimetertocap diff 0
perimetertocap diff 0
perimetertocap metal 0
perimetertocap poly/diff 0
```

The threshold for reporting capacitance may be 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

~/.cadrc
basename.cif
basename.al
basename.log
basename.nodes
basename.sim

SEE ALSO

powest(1.vlsi), pspice(1.vlsi), spcpp(1.vlsi), sim2spice(1.vlsi), spice(1.vlsi), drc(1.vlsi), erc(1.vlsi)
caesar(cad1),
cadrc(cad5), simfile(1.vlsi).

AUTHOR

Dan Fitzpatrick (UCB)

MODIFICATIONS

(UW/NW VLSI Consortium, University of Washington)

BUGS

Accepts manhattan simple CIF only, use **cifplot** –O to convert complicated CIF. For unusually shaped transistors the UW/NW modified *mextra* should be used, otherwise values will be quite inaccurate. The modified *mextra* will either yield accurate values or a "reasonable" guess, depending on the complexity of the unusual transistor. The modified *mextra* will tell you when the output values are only best estimates. The length/width ratio for unusually shaped transistors may be inaccurate. This is true for snake transistors. Attributes for funny transistors are not recorded. Node attributes are ignored unless the –o switch is present.

NAME

mtp – Multiple Time-series Plot for simulator output

SYNOPSIS

mtp *behavior-file directive-file plot-file*

DESCRIPTION

Mtp plots the output of *rnl* and *spice* simulations on the Printronix line printer. *Behavior-file* is the *rnl* or *spice* output file, *directive-file* is a "specification file" for the plot, and *plot-file* is an output file to contain the plot suitable for printing on the Printronix line printer.

The use of *mtp* involves the following steps:

1. Generate a behavior file.

   If you are using *rnl*, the directive

   openplot *"behavior-file"*

   will cause the changes to all traced nodes to be written to *behavior-file* in addition to being written to the terminal. Quotes are necessary if the file name has any punctuation in it.

   The RNL directive

   closeplot

   will terminate the behavior file. If the entire *rnl* session is to be recorded *closeplot* is not required, as the file will be terminated when *rnl* exits.

   If you are using *spice*, a behavior file may be specified as the third positional parameter of the *spice* command. Behavior records will be put on this file for all nodes specified on the Spice *PLOT* directive.

2. Generate a plot file from the behavior file using *mtp*.

   The plot is sent to the Printronix printer using the Unix command

   lpr -l *plot-file*

The contents of *behavior-file* are interpreted with the help of *directive-file*. For the basic purpose of plotting the output of *rnl* or *spice*, only a few directives need be supplied:

1. **start** *time*
   Tells *mtp* when to start plotting. If not supplied *time* defaults to 0. Data is skipped on the behavior file until an event is found whose time is greater than or equal to the start time.

2. **stop** *time*
   Tells *mtp* when to stop plotting. A stop value must be specified. If the stop time is greater than the time of the last event on the behavior file, the plot will be concluded with the last event.

3. **scale** *time*
   Tells *mtp* the number of time units per inch. The default value is 1000.0. Because the time unit used by *rnl* for behavior file output is 1.0 nanosecond, this value will produce plots of *rnl* output having a scale of 1.0 microsecond per inch.

4. **logical** *signal*
   This is used primarily for plotting *rnl* output. To select signals A, B and C for plotting in logical format the directives would be

   logical A
   logical B
   logical C

5. **analog** *signal heights*

Analog format is required when dealing with *spice* output because *spice* produces floating point values rather than logic levels. The height in inches of each trace must be specified. To select node voltages for nodes 1, 2 and 3 for plotting in analog format the necesary directives might be

```
analog V(1) 0.5
analog V(2) 0.5
analog V(3) 0.5
```

The order of selection directives in the file determines the order of the traces on the plot. The first signal selected is plotted closest to the time axis. A maximum of 20 signals may be selected on a given plot.

Spaces are used to separate the fields of a directive line. Blank lines or lines starting with # are ignored. Directives are case insensitive except for signal names.

EXAMPLE

The following example uses *mtp* to plot the behavior of a 10 bit counter, cntr10.net, shown here in netlist format:

```
; net file for 10-bit counter

; half adder made from gates
(macro half_adder (a b s c)
  (local h1 h2 h3)
  (nand (h1 2 16) a b)
  (nand (h2 2 16) a h1)
  (nand (h3 2 16) b h1)
  (nand (s 2 16) h2 h3)
  (invert c h1)
)

; one cell of a counter
(macro cell (in out Cin Cout)
  (local c1 c2 c3)
  (invert c1 in)
  (trans phi1 c1 c2)
  (invert c3 c2)
  (half_adder c3 Cin out Cout)
  (trans phi2 out in)
)

; declare global node names
(node count c in out phi1 phi2)

; carry-in to first significant bit controls counting action
(connect count c.0)

; generate the counter
(repeat i 1 10
  (capacitance out.i 1.234)
  (cell in.i out.i c.(1- i) c.i)
)
```

The rnl control file, cntr10.l, is as follows:

```
; RNL initialization file for 10 bit ripple-carry counter

  (load "uwstd.l")
  (load "uwsim.l")

  (read-network "cntr10")

; (setq report-form nil)  This turns off the report generator

  (setq incr 1000)

; bind symbols to node names

  (chflag '(phi1 phi2 out.10 out.9 out.8 out.7 out.6
                out.5  out.4 out.3 out.2 out.1))

  (defun init (dummy)

    (l '(count in.1 in.2 in.3 in.4 in.5
          in.6 in.7 in.8 in.9 in.10))

    (l '(phi2))
    (h '(phi1))

    (step incr)
    (l '(phi1))
    (step incr)

    (x '(in.1 in.2 in.3 in.4 in.5
        in.6 in.7 in.8 in.9 in.10))

    (h '(phi2))
    (step incr)

    (l '(phi2))
    (step incr)

    (h '(count))

    (wr-report)

    'done
  )


  (defvec '(bit bout out.10 out.9 out.8 out.7 out.6
                out.5  out.4 out.3 out.2 out.1))

  (defvec '(dec dout out.10 out.9 out.8 out.7 out.6
                out.5  out.4 out.3 out.2 out.1))
```

```
(def-report '("10 bit counter current state" newline " "
        count (vec bout) (vec dout)))
```

Generate the behavior for the counter using *rnl*

```
netlist cntr10.net cntr10.sim
presim cntr10.sim cntr10
rnl cntr10.l

init                          # initialize the counter

openplot "cntr10.evl"         # open the behavior file

                              # (.evl stands for event list)

c 30                          # run 30 clocks

exit                          # exit rnl
```

Generate the plot.

```
mtp cntr10.evl cntr10.mtp cntr10.plt

lpr cntr10.plt
```

The file *cntr10.mtp* could contain the following:

```
start    0.0
stop     20000.0
scale    1000.0
logical  phi1
logical  phi2
logical  out.1
logical  out.2
logical  out.3
logical  out.4
logical  out.5
```

The **start** and **scale** directives are not necessary but are included for the purpose of illustration. Although not required, these directives typically preceed the signal selection directives in the file.

When *mtp* runs it lists the contents of the directive file on the terminal and reports progress with the following messages:

```
Previous output cntr10.plt removed
Select and preprocess input data
Sort preprocessed events
Generate the plot
Rasterize for the Printronix
mtp complete, plot file is cntr10.plt
```

The "Rasterize for the Printronix" message marks the beginning of the longest step in the process which typically takes about a minute under moderate system loads.

Mtp creates scratch files named *fort.1, fort.2, fort.3, fort.4,* and *fort.7*. If any of these files are present when *mtp* is invoked it will exit with an error message. This can happen if *mtp* is aborted before having time to clean up the scratch files. If this happens the scratch files can be cleaned up with the Unix command

rm fort.[12347]

SEE ALSO

rnl(1.vlsi) spice(1.vlsi),

User's Guide to RNL **VLSI Design Tools Reference Manual**, UW/NW VLSI Consortium, University of Washington, (Christopher Terman, MIT Laboratory for Computer Science).

SPICE User's Guide, **VLSI Design Tools Reference Manual**, UW/NW VLSI Consortium, University of Washington, (A. Vladimirescu *et al.*, 15 Oct. 1980)

AUTHOR

William Beckett (UW)

NAME

mult – generate a cmos multiplier layout (version 1.0).

SYNOPSIS

mult [*options*] *caesarname*

DESCRIPTION

Mult is a module generation program for static cmos multiplier circuits. The layout is produced in "caesar" format. ***Mult*** requires a number of caesar cells with names of the form mult*.ca to exist in directory /ca. These should be copied from $UW_VLSI_TOOLS/lib/generators/mult prior to running *mult*. The generated layout is output in directory /ca in caesar cells with names of the form "caesarname"*.ca. ***Mult*** is a cfl-based program and therefore also produces *.bd files. "Caesarname" may not begin with the string "mult".

The *options* are as follows:

-g Makes the left side horizontal bus ground. This is the default.

-m *mbits*
Sets the number of bits in the multiplicand operand. ***Mbits*** must be in the range 3 to 32. The default is 3.

-n *nbits*
Sets the number of bits in the multiplier operand. ***Nbits*** must be in the range 3 to 32. The default is 3.

-p *P_string*
labels the propduct output bits with labels "P_string0", "P_string1", "P_string2", etc. with "P_string0" attached to the lsb. These labels appear on the right side and the bottom side of the layout. The default is "p".

-s Makes the number representation signed (two's complement). This is the default.

-u Makes the number representation unsigned.

-v Makes the left side horizontal bus Vdd.

-x *X_string*
labels the multiplicand input bits with labels "X_string0", "X_string1", "X_string2", etc. with "X_string0" attached to the lsb. These labels appear on the top side of the layout. The default is "x".

-y *Y_string*
labels the multiplier input bits with labels "Y_string0", "Y_string1", "Y_string2", etc. with "Y_string0" attached to the lsb. These labels appear on the left side of the layout. The default is "y".

FILES

/ca/*caesarname*.ca
/ca/*caesarname*.bd
/ca/*mult*.ca

SEE ALSO

caesar(CAD1), cfl(5.vlsi)

AUTHORS

Wayne E. Winder

NAME

netlist – a simple network description language for VLSI circuits

SYNOPSIS

netlist *infile* [*outfile*] [**-o**] [**-t***tech*] [**-u***units*] [**-s** *n*] [**-d** *n,m*] [**-e** *n,m*] [**-l** *n,m*] [**-l** *n,m*] [**-p** *n,m*]

DESCRIPTION

Netlist requires an input file with any/all extensions on the command line. An optional output file can be specified. Additional options are described below;

-o Uses old input format. Size specifications are taken to be length/width rather than width/length.

-t*tech* Uses *tech* in the technology portion of the units/tech line at the beginning of the simulation file produced (Default is nmos).

-u*units* Sets the number of centi-microns per lambda to *units* (Default is 250). **Warning:** The "units" set by this option appear in the comment line of the *.sim* file. This value is not used by PRESIM and does not influence an RNL simulation.

-s *n* Uses number *n* as initializer for internal node names; useful when you want to merge the results of separate *netlist* runs.

-d *n, n* Sets the default width to *n* and length to *m* for depletion devices. The defaults are *n*=8 and *m*=2.

-e *n, n* Similar to **-d** except for enhancement devices. The defaults are *n*=2 and *m*=2.

-i *n, n* Similar to **-d** except for intrinsic devices. The defaults are *n*=2 and *m*=2.

-l *n, n* Similar to **-d** except for low-power devices. The defaults are *n*=2 and *m*=2.

-p *n, n* Similar to **-d** except for p-channel devices. The defaults are *n*=2 and *m*=2.

In addition, if node alias records (= node1 node2 ...) are declared using "connect" (See *netlist* reference documents) they appear in a file with the name "basename.al". The basename is the input file name minus its last extension.

Netlist is a macro-based language for describing networks of sized transistors. Names in *netlist* refer to nodes, which presumably get interconnected by the user through transistors. Macros for describing transistors can be found in the *NETLIST User's Guide*. In addition to transistor macros *netlist* provides macros that allow the user to set node capacitance, specific node delays (in tenths of nanoseconds), and transistor threshold voltages. The user may also define his own macros.

The **load** command uses the environment variable *RNLPATH* (default *.:$UW_VLSI_TOOLS/lib/rnl*). See the *NETLIST User's Guide* for details.

SEE ALSO

presim(1.vlsi), *rnl*(1.vlsi),

NETLIST User's Guide, **VLSI Design Tools Reference Manual**, UW/NW VLSI Consortium, University of Washington,

AUTHOR

Christopher Terman (MIT)

NAME

pads – generate a cmos padframe layout (version 1.0)

SYNOPSIS

pads *caesarname* < *frame_spec*

DESCRIPTION

pads is a module generation program for a MOSIS 3 micron cmos padframe layout. This generator uses leaf cells derived from the MIT pads received from MOSIS. The leaf cells and the layout that is produced are in "caesar" format. *Pads* looks for caesar leaf cells with names of the form pad*.ca in the directory ./ca. These should be copied from $UW_VLSI_TOOLS/lib/generators/pads prior to running *pads*. *Pads* also reads in a frame_spec file from the home directory (not ./ca). The frame_spec file definition is provided in the text that follows. The generated layout cells (composition cells) appear in directory ./ca with names caesarname*.ca. *Pads* is a cfi-based program and therefore also produces *.bd files. "Caesarname" may not begin with the string "pad".

There are no *options*.

FRAME_SPEC

The frame specification is a text file made up of one frame specifier followed by several pad specifiers. These records are terminated with ';' and may cross line boundaries. Individual fields within records should not cross line boundaries. The syntax is 'c-like'; comments may be placed anywhere with the /* ... */ convention.

The frame specifier is made up of a type specifier followed by an optional connection layer specifier. The type specifier is one of C28_46x34, C40_46x68, C40_69x68, C64_69x68, C64_79x92, or C84_79x92 (the first number indicates the number of pins on the frame, the second and third numbers give the x and y dimensions of the entire frame in hundreds of microns). The connection layer specifier indicates what material connects the individual pad circuitry to the interior of the chip (across the quard ring). This specifier may be METAL2 or POLY. Default is *POLY*.

The pad specifiers are used to determine the type of circuitry to place on specific pad sites. Pad specifiers are made up of pin number, pad type, and optional label and connection specifiers.

The pin number is an integer between 1 and the number of pins for the frame specified (see above). For the 28 pin frame, pin number 1 is in the middle of the right side of the frame. For the 40 and 64 pin frames, pin number 1 is immediately above the middle of the right side of the frame. For the 84 pin frame, pin number 1 is the rightmost pin on the top of the frame. Pin numbering procedes counterclockwise in all cases.

The pad type is one of pad1vdd (power), pad1gnd (ground), pad1in (input), pad1out (output), pad1ttl (ttl output), pad1ts (tri state output), pad1bin (buffered input), pad1bit (buffered ttl input) or pad1sp (frame spacer - never required).

The optional label specifiers are of the form 'BP = label', 'L1 = label', 'L2 = label' and 'L3 = label'. BP, L1, L2 and L3 indicate where on the pad circuitry to place the label; on the bonding pad, on the leftmost connection on the bottom of the pad circuitry (when viewed with bonding pad on top), second from left and third from left, respectively. 'Label' is any string beginning with a letter and containing only non-special characters. Special characters include '=', ';' and '/'. Special characters can be included in strings by placing double quotes around the string and preceding the special character with the backslash character. For details of what connection connects to what portion of the pad circuitry, view the appropriate circuit from pad1*.ca using caesar. The connections should be annotated with local labels to avoid ambiguity. Not all connections appear on all pads.

The optional connection specifier indicates which connection to the interior is to receive a contact, after crossing the guard ring. This specifier is of the form 'CN = layer', where N is 1, 2 or 3 and is identified as above. 'Layer' is one of METAL, POLY, or METAL2. Default is *METAL*. If the layer is the same as the input connection material (specified in the first record), no contact is placed. If different, a contact is placed. POLY may not be routed to METAL2 and vice-versa.

RESTRICTIONS

Pins may not be assigned more than once. Only those pins required need be assigned.

In ceratin corners of certain frames, tristate pad connections do not cross the quard ring.

In the 28, 40, and 64 pin frames, pin 1 should be vdd or blank. In the 84 pin frame, pin 10 should be vdd or blank.

Each frame must include at least one VDD pad and one Ground pad. These pads may only connect to the interior with METAL.

FILES

/ca/caesarname.ca*
/ca/caesarname.bd*
/ca/pad.ca*
$UW_VLSI_TOOLS/src/examples/pads/input
(for a frame_spec example)

SEE ALSO

cfi(5.vlsi)

AUTHORS

Wayne E. Winder

NAME

peg – finite state machine compiler

SYNOPSIS

peg [–s] [–t] [file]

DESCRIPTION

Peg (*PLA Equation Generator*) 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

pla2net - generate netlist macro from truth table of pla

SYNOPSIS

pla2net *basename*

DESCRIPTION

pla2net generates a netlist macro using the truth table definition for a pla as an input. This truth table may have been obtained using PEG and EQNTOTT. *pla2net* expects that a file named '**basename.tt**' is in the current directory; if this is not the case an error message will be generated. The output of *pla2net* will be stored in a file named '*basename*.net'.

The macro defined in the '*basename*.net' file looks as follows:

(macro basename (output input) where:

- the basename is identical to the basename in the commmand line of pla2net;
- output is an outputvector, numbered from left-to-right as in the truth table and a lay-out generated with tpla starting with output.1;
- input is an inputvector, number from left-to-right as in the truth table and a lay-out generated with tpla starting with input.1.

Note: When designing pla's for sequential state machines with PEG, the innermost inputs and outputs of the pla will be the least significant bit. The state register inputs and outputs must be wired accordingly (mirror and shift numbering of input vector in the netlist description for the top level sequential state machine, which includes the feedback register, is necessary).

INPUT FILE

basename.tt

OUTPUT FILE

basename.net

SEE ALSO

Manual entries for PEG, EQNTOTT

AUTHOR

Henriecus Koeman, John Fluke Mfg. Co., Inc.

BUGS

The current version only supports cmos technologies. The source code can easily be modified for other technologies.

NAME

presim – a netlist preprocessor for the rnl VLSI circuit simulator

SYNOPSIS

presim *infile outfile* [*configfile*] [*-g*] [*-cfile,min*] [*-tfile,min*] [*presist,voltage*]

DESCRIPTION

Presim converts the **.sim** file into a binary file to be used by *rnl*.

The parameters and options are as follows:

infile A net list file that must include any/all extensions;

outfile An output filename must be specified on the command line;

configfile (optional) A file to set lambda and RC parameters for nodes and transistors in the netlist (see the *presim* user's guide for descriptions of the parameters and syntax).

-g Suppresses the sum-of-products formation. This may be desired if you think sum-of-products is formed wrong otherwise the advantages of the transistor and node reduction make this option unattractive.

-cfile, min Writes a list of node names and capacitances to the specified *file*. Only capacitances larger than *min* will be included.

-tfile, min Writes a list of transistors and RC values to the specified *file* -- there are two entries for each transistor. The R's come from the size of the transistor, C's from the source/drain capacitance. Only RC values larger than *min* will be included.

-presist, *voltage*
Provides a worse-case estimate of the circuit power consumption by assuming that all the pullups (DEP or LOWP devices with drain=*Vdd*) are all on simultaneously. *Voltage* specifies the supply voltage,

Presim also attempts to open the file *basename*.**al**, where *basename* is defined as the input file name minus its last extension. It is non-fatal for this file to be absent.

SEE ALSO

PRESIM User's Guide, **VLSI Design Tools Reference Manual**, UW/NW VLSI Consortium, University of Washington, (Christopher Terman, MIT Laboratory for Computer Science).

AUTHOR

Christopher Terman (MIT)

BUGS

Propagation of X state information for cmos circuits in *rnl* is unreliable if the gate reduction in *presim* is performed. If this information is required, suppress gate reduction with the -g option in *presim*.

NAME

presto – combinational logic minimization program

SYNOPSIS

presto

DESCRIPTION

Presto is an efficient combinational logic minimization program. This program not only reduces the number of product terms, increases the number of don't care inputs, but also reduces the number of the output connections. Therefore, this program is very useful to pla designers.

Input is taken from standard input. Output goes to standard output.

An example of typical input is as follows:

```
.i 4
.o 2
.l
.p 4
10x1 11
000x 1x
1111 01
0101 10
.e
```

The integer after ".i" is the number of input variables. The integer after ".o" is the number of output variables. The integer after ".p" is the number of input product terms. ".l" is optional for input listing. There is another option ".d" for intermediate results.

In the input part, 1 means logic level 1, 0 means logic level 0, x(or -) means don't care. In the output part, 1 means that the term is connected to the output, 0 means that this term is not connected to the output, and x(or -) means that the output doesn't care whether this term is connected or not. ".e" means the end of the input file. When there is a format error in the input file, the program will give the message: "INPUT FORMAT ERROR!" and abort the job.

AUTHOR

Sheng Fang

NAME

pspice – prepare an input file for the Spice circuit simulator

SYNOPSIS

pspice [**-rm**] [**-nos2s**] [**-d** *defsfile*] [**-m** *modelfile*] [**-e** *expfile*] *basename*

DESCRIPTION

Pspice is a shell script for preparing Spice input from information from several sources. *Pspice* runs *sim2spice* to convert from a *basename*.**sim** format circuit description to a Spice-compatible description and modifies the *sim2spice* node label translation table to be acceptable Spice comments. It then runs *spcpp* to translate a pseudo-Spice formatted file that contains symbolic node labels to a Spice-acceptable file. Finally, *pspice* concatenates the circuit description file, the translation table, a file of untranslated Spice input, and the translated Spice input into a single file named *basename*.**spcin**. This file is usually an acceptable Spice input file. The optional parameters can be used to cause parts of this process to be skipped.

The options and parameters are:

-nos2s Suppresses the execution of the *sim2spice* step.

-rm Indicates that the files created in intermediate steps are to be deleted.

-d *defsfile* Specifies a file to be used as a *sim2spice* definitions file.

-m *modelfile* Specifies a file that contains Spice input that is to be included (untranslated) in the final output. It is intended that *modelfile* name a file containing Spice .MODEL cards as well as other Spice commands that are independent of the particular circuit being modeled.

-e *expfile* Specifies a file that contains pseudo-input for Spice. *Spcpp* will interpret strings in *expfile* that are bracketed by '<' and '>' as node names to be translated into *spice* node numbers using the translation table (*basename*.**names**) created by *sim2spice*. Lines containing bracketed tokens are converted into Spice comments. It is intended that *expfile* contain Spice commands that describe the experiment to be simulated on the circuit. The ability to use mnemonic node names makes the preparation of Spice input much easier and it means that the description of the experiment need only be specified once, even if the circuit is modified and reextracted. If *expfile* is not specified then spcpp is not executed.

basename Specifies the base name for the files describing the circuit. If *sim2spice* is run then a file named *basename*.**sim** must exist. If *sim2spice* is not run then the files *basename*.**names** and *basename*.**spice** must exist.

FILES

basename.sim	circuit description input to *sim2spice*
defsfile	optional *sim2spice* defs input
basename.names	modified *sim2spice* translation table output. This is read by *spcpp* (*)
basename.spice	*sim2spice* output Spice format circuit element definitions (*)
modelfile	optional Spice .MODEL commands to be included in *basename*.**spcin**
expfile	input to *spcpp* containing pseudo-*spice* commands describing the experiment to be simulated
basename.spcx	translated output from *spcpp* (*)
basename.spcin	The Spice input deck created by concatenating *basename*.**spice**, *basename*.**names**, *modelfile*, and *basename*.**spcx**

Note: Files marked (*) are deleted by the **-rm** option.

SEE ALSO

sim2spice(1.vlsi), spcpp(1.vlsi)
spice(1.vlsi)

mextra(1.vlsi), cifplot(CAD1)

AUTHOR

Robert Fowler (UW/NW VLSI Consortium, University of Washington)

DIAGNOSTICS

The error messages are intended to be self explanitory. Note that *sim2spice* and *spcpp* produce their own error messages.

BUGS

The command line is long enough to tempt a user to call *pspice* from yet another shell script. A better way to do this is to set up an alias for *pspice* with the commonly used options already set.

NAME

rnl – timing and logic simulator for VLSI circuits

SYNOPSIS

rnl [*cmdfile*]

DESCRIPTION

Rnl (NetLisp) is a timing logic simulator for digital NMOS circuits with a lisp-like command interpreter. It has also been used with many CMOS circuits with some success. The *Rnl User's Guide* discusses some of the limitations found in simulating CMOS circuits. To use *rnl*, one needs a **.sim** file for the circuit to be simulated. This can be derived from the mask file (e.g., CIF) or developed using *netlist*, a program that processes textual schematics.

One must first convert the **.sim** file to a network file suitable for use by *rnl*. To do this run *presim*:

presim *filename*.**sim** *netfile* [*config_params*]

which converts the file *filename*.**sim** into *netfile*, a binary file for *rnl*. (see *Presim User's Guide* for information on the various configuration parameters.

The optional *cmdfile* is the file *rnl* initially reads for user input. Usually one prepares a command file that loads one or more library files containing RNL function definitions and reads in the network from *netfile*. As simulation proceeds, user defined functions developed for testing the circuit can be added to the command file. At a minimum the command file should contain the commands

(load "uwstd.l")
(load "uwsim.l")
(read-network "*netfile*")

When using the **load** command both *netlist* and *rnl* search the current directory and then any directories specified in the environment variable *RNLPATH*. The value of *RNLPATH* defaults to *$UW_VLSI_TOOLS/lib/rnl*. **Read-network** does not use *RNLPATH*. *Netfile* must be produced by *presim*. When the end-of-file is reached in the command file, input is taken from stdin. Commands and formats to be used are given in the *RNL User's Guide*.

The top level of *rnl* is a simple loop:

(1) read command from current input;
(2) evaluate command, performing specified actions;
(3) print the result and loop back to (1).

The following is a list of the objects that *rnl* knows about

numbers -- signed integers. (16 bits on PDP11s, 24 bits on VAXen, 28 bits on PDP10s).
-- floating point.

strings sequences of characters enclosed in quotes ("). Useful as constants for file names, print statements, etc. Special characters can be introduced into the strings by using the backslash escapes:

'\n'	newline
'\r'	return
'\t'	tab
'\ooo'	ascii code "ooo" where ooo are octal digits

symbols variable names. Any sequence of characters that isn't a number, string, or some special character -- starting symbols with a letter, followed by more letters, numbers, and punctuation is usually a safe bet.

nodes an electrical node.

lists a sequence of objects enclosed in parentheses. Standard LISP syntax applies, including dot notation. The empty list "()" is also called "nil".

subrs primitive, or built-in, functions (like +).

The functions are listed by application area. The areas are:

- arithmetic functions
- predicates
- list functions
- I/O functions
- miscellaneous functions
- special forms
- network/simulation functions
- functions defined in "uwsim.l"

SEE ALSO

netlist(1.vlsi), *presim*(1.vlsi), *simfile*(5.vlsi)

RNL User's Guide, **VLSI Design Tools Reference Manual,** UW/NW VLSI Consortium, University of Washington, (Christopher Terman, MIT Laboratory for Computer Science).

AUTHOR

Christopher Terman (MIT)

BUGS

User defined macros with the same name as a node in the net list puts *rnl* into an infinite loop.

Propagation of X state information for cmos circuits is unreliable if the gate reduction in *presim* is performed. If this information is required, suppress gate reduction with the -g option in *presim*.

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 mextra format to Spice (circuit simulator) format

SYNOPSIS

sim2spice [**–d** *defs*] *basename*.**sim**

DESCRIPTION

Sim2spice reads the *basename*.**sim**, *basename*.**nodes** and *basename*.**al** files created by *mextra* and creates a Spice readable circuit description, *basename*.**spice**. Spice requires node numbers and *sim2spice* generates a translation table *basename*.**names** which shows the *mextra* nodelabel corresponding to a given node number.

The user can specify his/her own translation table by using the *-d* option, where *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.

Sim2spice also reads 'N' lines generated by mextra with the -o switch. In order to compute capacitances from this it must have a set of conversion factors between length/area and capacitance. These are specified in the sim2spice section of '.cadrc' file in exactly the same format as in the mextra section of the '.cadrc' file (see mextra).

The program has been extended so that a comment line beginning with "|=" is interpreted as an MIT .**sim** style node equivalence line.

To create a complete Spice input file it is necessary to append applicable Spice model descriptions as well as the user's Spice simulation commands to the *basename*.**spice** file.

It is recommended in most cases that the user run *pspice* rather than *sim2spice*. *Pspice* incorporates the features of *sim2spice* but will in addition allow the user to build all of the Spice input file in one step. *Pspice* also incorporates the features of *spcpp*.

FILES

basename.sim
basename.nodes
basename.al
basename.spice
basename.names

SEE ALSO

mextra(1.vlsi), spice(1.vlsi), pspice(1.vlsi), spcpp(1.vlsi)

AUTHOR

Dan Fitzpatrick (UCB)

MODIFICATIONS

Neil Soiffer (UCB) -- cMOS fixes.

Rob Fowler (UW/NW VLSI Consortium, University of Washington) -- node equivalence handling and misc. bug fixes.

BUGS

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

Warning: for nMOS circuits the node names "ENMOS" and "DNMOS" are preempted by *sim2spice* as synonyms for "BULK".

The node equivalence handling is not completely general. New nodes can be added to equivalence classes, but classes cannot be merged. This is detected and an error message is produced.

NAME

simscope – view time-series of simulator output.

SYNOPSIS

simscope

DESCRIPTION

simscope is designed to display signal output produced by RNL or SPICE on a Tek 4105 or a GP-19 graphics terminal. To make hardcopies, you need a Tek 4695 printer (or compatible hardcopy device) in conjunction with the Tek 4105 graphics terminal. Any program that might periodically interfere with the display, notably sysline, should be switched off.

General Rules for Using *simscope*:

1. Names to be entered in response to *simscope*'s requests may contain alpha as well as numerical characters.

2. Numbers to be entered in response to *simscope*'s requests may be fixed-point or floating-point numbers (the latter format is also referred to as scientific notation). Examples of fixed-point numbers are 123, 3.55, +45000, etc. Examples of numbers in scientific notation are 3.5e4, 0.333e-9, 0.1e-9, +244.5e05, etc. Numbers may not be negative (negative time scales, times, and positions do not make sense to simscope).

3. While entering a name or a number, characters typed erroneously may be deleted with either the RUBOUT key (CONTROL-H is equivalent) or the BACKSPACE key (DELETE on some keyboards).

4. All numbers displayed by *simscope* are in scientific notation. The mantissa consists of one digit, a decimal point, another six digits, the "e" indicating the exponent, the sign for the exponent, and two digits for the exponent.

5. After reading a behavior file, *simscope* uses the same time units as those used in the behavior file. These are nanoseconds (ns) in RNL-generated files and seconds (s) in SPICE-generated files.

6. Indeterminate RNL signals (state "X") are displayed with a logic level of 0.5.

How to Start and Exit *simscope*

Work with *simscope* is most convenient if you change to the directory that contains the behavior file you want to view. Being in that directory, simply enter *simscope*. *simscope*comes up with a greeting display. The window begin and end times are set to 0 (B = 0.000000+00 and E = 0.000000+00) and, consequently, the time scale is 0 (T = 0.000000+00), too. The mode (see below) is set to fixed-time-scale mode. The Y-scale, in units per division, is set to 1 (Y = 1.000000+00).

Below these indicators the menu is displayed, followed by the request that you hit a key indicating the menu function you wish to select. Valid keys are: f, n, b, e, t, y, c, d, s, r, and q, all of which may be entered as capitals (with the SHIFT key). Each letter represents the first

letter of the corresponding menu function (see below). After you press any of these keys, the corresponding function is immediately activated - no RETURN is necessary.

To exit *simscope*, press the "q" key (Quit).

simscope Functions

File

This function serves two purposes. First, it provides the following information about the file presently loaded:

name of present behaviour file

file begin time (the time of the first signal entry in the file)

file end time (the time of the last signal entry in the file)

for each signal in the file:

first change (the time of the first entry of the signal in the file)

last change (the time of the last entry of the signal in the file)

y-position (the vertical position of the signal in the display; a number between 1 and 99, 1 is the lower end of the window, 99 is the upper end of the window).

name (the signal's name)

The second purpose of the File function is to facilitate the loading of a different file. If you press "y" (yes) in reply to the function prompt, File will ask you for the name of the new behaviour file you want to load. Any other key will terminate the File function, display all signals, and return you to the menu. If you enter a name, the corresponding file is read. Reading the behaviour file may take awhile, during which time the cursor may flash at various positions on the screen (hence the message: "Reading file. Please wait. Don't worry if the cursor flashes a bit.").

Nodes

You will be asked for a node name (default is the node last entered). *simscope* then asks whether you want to display the node's signal or delete the node's signal from the display. + or just RETURN means display, - means delete from display (the y-position of the signal is set to zero). Next simscope asks for the position (vertically) in the window. Enter 99 for the very top of the window, 1 for the bottom of the window, any number between 1 and 99 for an intermediate position. (One division on the y-scale is equivalent to 5 positional units.)

Reposition a signal by entering its name with Nodes, then enter the desired new position.

Assuming that you normally want to change more than one signal in a row, file information (as in the File function) is displayed after you enter a y-position, providing you with a summary of the information on all nodes if the behaviour file.

Use the Display function to display the signals.

Begin

Sets the window begin time ("B = " at left side of the window).

In fixed-time-scale mode, a change of the window begin time moves the window (whose time width remains unchanged) across the file.

In fixed-window-end mode, a change of the window begin time expands or contracts the window in a "rubber-band-like" fashion.

End

Sets the window end time ("E = " at right side of the window).

The time scale will be readjusted automatically to be consistent with the new window end time.

The window begin time is not affected.

Setting the window end time will switch over to fixed-window-end mode. In this mode changes to the window begin time will not affect the window end time, but will readjust the time scale. You can use End for switching to fixed-window-end mode (without actually changing the window end time) by confirming the present (default) value of the end time. To do that press "e", then just hit RETURN.

T-scale

Sets the time scale of the window ("T = " below the window).

The window end time will be readjusted automatically to be consistent with the new time scale.

The window begin time is not affected.

Setting the time scale will switch over to fixed-time-scale mode. In this mode, changes to the window begin time will not affect the time scale, but will readjust the window end time. You can use T-scale for switching to fixed-time-scale mode (without actually changing the time scale) by confirming the present (default) value of the scale. To do that press "t", then just hit RETURN.

Y-scale

Sets the vertical scale in units per division ("Y = " below "B = "). The default is 1, which is a good value for RNL. The vertical extent of SPICE signals is usually larger, however (for example 5 Volts, i.e. 5 units). Increasing the Y-scale allows you to fit more of the larger signals on the screen without overlapping.

Copy

To make hardcopies, you need a Tek 4105 and a Tek 4695 printer (or compatible device). (If you activate the Copy function on a GP-19 terminal, you will get the message: "Use a Tek 4105 terminal. (You can also use MTP). Hit any key to continue.")

Display

All signals with a y-position greater than 1 are displayed. Use this function to display the signals after you have made changes for any node (deletion, positioning or repositioning), or if you want to refresh the display for any reason.

Save

All parameters determining the particular display state are saved for later retrieval. "Save" first asks you for a name of the file in which you want to keep the present state. It then saves in this file the present behaviour file name, window begin time, time scale, window end time, mode (1 for fixed-time-scale mode, 0 for fixed-window-end mode), and the names of all displayed signals with their positions on the y-axis.

Saved states can be restored easily with the Retrieve function. You may conveniently consider the names of "Save" files as markers (possibly with short names like 1, 2, 3, ..., or a1, register3, Load10, etc.), which can be "jumped to" with the Retrieve function.

Retrieve

Retrieve is the function used to restore a previously saved display state. You are asked for the name of the file containing the state to be restored. If the state you want to restore belongs to a behaviour file different from the one on which you are working presently, then the new behaviour file is read (this may take some time).

Quit

This function terminates *simscope* and returns you to the UNIX shell.

The use of *simscope* to display RNL or SPICE results involves the following steps:

1. Generate a behavior file.

   (a) If you are using *RNL*, the directive

   openplot *"behavior-file"*

   will cause the changes to all traced nodes to be written to *behavior-file* in addition to being written to the terminal. Quotes are necessary if the file name has any punctuation in it.

The RNL directive

```
closeplot
```

will terminate the behavior file. If the entire *RNL* session is to be recorded *closeplot* is not required, as the file will be terminated when *RNL* exits.

(b) If you are using *SPICE*, a behavior file may be specified as the third positional parameter of the *SPICE* command. Behavior records will be put on this file for all nodes specified on the SPICE *PLOT* directive.

2. Use the F (File) function of *simscope* to load the behavior file and get information on the signals stored in it.

   Use *simscope*'s other menu functions to display any signal in the file on any position (vertically) on the screen, change the time scale (window size) and window position. After you have analyzed and positioned your signals, make a hard-copy, if desired.

EXAMPLE (Preparation of a Behaviour File with RNL)

The following example uses *simscope* to display the behavior of a 10 bit counter, cntr10.net, shown here in netlist format:

```
; net file for 10-bit counter

; half adder made from gates
(macro half_adder (a b s c)
  (local h1 h2 h3)
  (nand (h1 2 16) a b)
  (nand (h2 2 16) a h1)
  (nand (h3 2 16) b h1)
  (nand (s 2 16) h2 h3)
  (invert c h1)
)

; one cell of a counter
(macro cell (in out Cin Cout)
  (local c1 c2 c3)
  (invert c1 in)
  (trans phi1 c1 c2)
  (invert c3 c2)
  (half_adder c3 Cin out Cout)
  (trans phi2 out in)
)

; declare global node names
(node count c in out phi1 phi2)

; carry-in to first significant bit controls counting action
(connect count c.0)

; generate the counter
(repeat i 1 10
  (capacitance out.i 1.234)
```

```
  (cell in.i out.i c.(1- i) c.i)
)
```

The RNL control file, cntr10.l, is as follows:

```
; RNL initialization file for 10 bit ripple-carry counter

  (load "uwstd.l")
  (load "uwsim.l")

  (read-network "cntr10")

; (setq report-form nil)  This turns off the report generator

  (setq incr 1000)

; bind symbols to node names

  (chflag '(phi1 phi2 out.10 out.9 out.8 out.7 out.6
            out.5 out.4 out.3 out.2 out.1))

  (defun init (dummy)

    (l '(count in.1 in.2 in.3 in.4 in.5
         in.6 in.7 in.8 in.9 in.10))

    (l '(phi2))
    (h '(phi1))

    (step incr)
    (l '(phi1))
    (step incr)

    (x '(in.1 in.2 in.3 in.4 in.5
         in.6 in.7 in.8 in.9 in.10))

    (h '(phi2))
    (step incr)

    (l '(phi2))
    (step incr)

    (h '(count))

    (wr-report)

    'done
  )


  (defvec '(bit bout out.10 out.9 out.8 out.7 out.6
            out.5 out.4 out.3 out.2 out.1))
```

```
(defvec '(dec dout out.10 out.9 out.8 out.7 out.6
          out.5  out.4 out.3 out.2 out.1))

(def-report '("10 bit counter current state" newline " "
        count (vec bout) (vec dout)))
```

Generate the behavior for the counter using *RNL*

```
netlist cntr10.net cntr10.sim
presim cntr10.sim cntr10
RNL cntr10.l

init                          # initialize the counter

openplot "cntr10.beh"         # open the behavior file

                              # (.beh stands for network behavior file)

c 30                          # run 30 clocks

exit                          # exit RNL
```

SEE ALSO

RNL(1.vlsi) SPICE(1.vlsi), mtp(1.vlsi)

SIMSCOPE User's Guide Release 2.0 Available from the UW/NW VLSI Consortium, Sieg Hall, FR-35, University of Washington, Seattle, WA 98195

"User's Guide to RNL" "VLSI Design Tools Reference Manual", UW/NW VLSI Consortium, University of Washington, (Christopher Terman, MIT Laboratory for Computer Science).

"SPICE User's Guide", "VLSI Design Tools Reference Manual," UW/NW VLSI Consortium, University of Washington, (A. Vladimirescu *et al.*, 15 Oct. 1980)

RESTRICTIONS

1. In graphics mode, which is obviously required for simscope, the GP-19 can display upper case characters only. *simscope* Rstill recognizes, and processes correctly, lower case characters. You have to know which characters in your file and which node names are upper case, and which are lower case, and enter them accordingly. Otherwise, *simscope* may tell you that it does not know the name you entered. The Tek 4105 terminal does not have this problem.
2. The File function does not recognize the ~ (tilde) as part of a path name.
3. RNL and SPICE write only changes of signal levels to the behavior file. Therefore, a signal's value before the first file entry is not known. *simscope*'s strategy to deal with this situation is to display this value as indeterminate (0.5, X in RNL).
4. The number of signals in a behavior file is limited to 20 (17).

5. The length of behavior files is restricted to 20,000 signal changes (i.e. to 20,000 lines). This could be extended easily, if need be.

BUGS

1. In case the number of signals in a behavior file is greater than 20 or the behavior file contains more than 20,000 signal changes (i.e. to 20,000 lines), *simscope* crashes.
2. The first line of a behaviour file is discarded, because in the case of RNL behaviour files this line contains irrelevant information different from the information of all other lines. Therefore, the very first signal change of a SPICE behaviour file is lost. This is not noticable in most cases, however.
3. Some versions of SPICE produce behaviour files that contain floating point numbers formatted in a non-standard manner. Encountering of a non-standard format in the behaviour file causes *simscope* to crash. A typical case is that a number like "0.000e-7" is given as "0. e-7". *simscope* recognizes the first format only. The behaviour file can be mended easily by using an editor to globally replace ". e" by ".000e".

These bugs will be removed with the next release of *simscope*.

AUTHOR

Rudolf W. Nottrott (UW/NW VLSI CONSORTIUM)

NAME

spcpp – Spice (circuit simulator) input pre-processor

SYNOPSIS

spcpp [**–c**] [**–s** *n*] [**–d** *lr*] [**–t** *tname*] [**–o** *oname*] *iname*

DESCRIPTION

Spcpp is a program that translates bracketed text tokens in an input file into other text strings. It is intended to allow users of *spice* to prepare their simulation input using mnemonic node names rather than the numeric node numbers required by Spice commands. The program has two major modes of operation. If the user does not specify a file that contains a translation table, then *spcpp* builds a translation table itself numbering the tokens from zero as it encounters them. Alternatively, the user can specify the name of a file containing a translation table to be used. In particular, the **.names** file created by *sim2spice* is usable as a translation table file.

The options and parameters are:

–c Indicates that the first non-whitespace word of each line of the translation table file should be skipped over. This is useful if your translation table has an asterisk ('*') in column 1 of each line to allow it to be read by *spice* as comments.

–s *n* Indicates that *n* lines at the beginning of the translation table file should be skipped over. If no number is specified then only the first line of the file is skipped.

–d *lr* Redefines the token delimiters to be '*l*' and '*r*' respectively. The default delimiters are '<' and '>'.

–t *tname* Specifies a file that contains a translation table (default is to build a translation table as described above). Each line of this file should have at least two non-whitespace words on it. If the **–c** option is specified then the first word on each line is ignored. The next word is interpreted as a string to be translated and following one is interpreted as the target string into which it is translated. Any subsequent words on the line are ignored. For Spice input preparation the target string should be a numeral. The **–s** option allows the file to be prefaced by one or more lines that *spcpp* will ignore.

–o *oname* Specifies a file into which the output is to be written. If this option is not used then the output is written to *iroot*.**spcx** where *iroot* is obtained by stripping away any tags from *iname*.

iname Specifies the name of the file to process.

A bracketed token is defined to be a left delimiter character, zero or more spaces, a word (the token) not containing either right or left delimiters, zero or more spaces, and a right delimiter character. Unmatched delimiter characters are not allowed in any context. Bracketed tokens are not allowed to span lines. Tokens and the strings that they translate into are limited to be at most 40 characters each.

Any line that contains no bracketed tokens is simply copied from the input to the output. If a line does contain a bracketed token then the input line is written into the output a Spice comment line. An output line follows immediately. If the line is valid, then the output line has the untranslated parts immediately below the corresponding parts of the commented input line with the target strings substituted for the bracketed tokens. If an error is detected, then the output line has a caret ('^') immediately below the point at which the first error is detected. An error message line then follows. Since the scanning of the line is abandoned there may be subsequent undetected errors in the remaining part of the line.

Example:

If the following lines are contained in the translation table file:

```
Vdd 1
Input 55
Output 107
foo 23
bar 45
```

then *spcpp* will, upon seeing the lines:

```
.plot trans v(<Input>) v(<Output>), i(<Vdd>)
+ v(<foo>), v(<bar>)
```

will output the lines:

```
* .plot trans v(<Input>) v(<Output>) v(<Vdd>)
  .plot trans v(55) v(107) v(1)
* + v(<foo>), v(<bar>)
  + v(23), v(45)
```

Note that *spcpp* correctly handles Spice continuation cards.

Note also that the substitution process is not recursive. That is, once a token has been translated, the translated string is not rescanned.

The usefulness of *spcpp* for simulating a circuit extracted from a layout depends upon the user being able to ensure that his mnemonic node labels will be retained through the extraction process. The *mextra* and *sim2spice* manual entries will help with this.

Pspice is a shell script that runs *sim2spice* and *spcpp* and concatenates several files is useful for preparing Spice inputs from **.sim** files.

FILES

iname
iroot.spcx
oname
tname

SEE ALSO

mextra(1.vlsi), pspice(1.vlsi), sim2spice(1.vlsi), simtools(1.vlsi), spice(1.vlsi),

SPICE User's Guide, **VLSI Design Tools Reference Manual**, UW/NW VLSI Consortium, University of Washington, (*SPICE Version 2G6 User's Guide*, A. Vladimirescu *et al.*, 15 October 1980)

AUTHOR

Robert Fowler (UW/NW VLSI Consortium, University of Washington)

DIAGNOSTICS

The error messages are intended to be self explanatory. If *spcpp* encounters a syntax error on a line then it suspends processing on that line and writes it as a Spice comment to the output file. It then writes a line containing a caret ('^') under the character at which scanning failed and finally, a line containing an error message. It then goes on to process the remaining lines of the file. If errors have been encountered then at the end of the output file *spcpp* writes messages to the effect that errors have been encountered and exits with status 1. The error

messages written to the output file begin with dollar signs. In addition, some number of messages are directed towards the standard error output.

BUGS

The target strings are not checked to see whether they are valid numerals or not. This can be regarded as either a bug or a feature.

The target string must fit into the space from the left to right token delimiter inclusive. This is normally not a problem since most node numbers will be small integers and the available space will be at least three characters. This was done so that the input lines and the translated outputs would line up vertically.

NAME

spice – circuit simulator

SYNOPSIS

spice *infile outfile* [*mtpfile*]

DESCRIPTION

Spice reads a circuit description from *infile*. Output is written to *outfile*. and error messages to standard error. An optional output file, *mtpfile*, can be used by *mtp* to obtain a multiple time series plot on a Printronix.

Spice calls *spice2g6*, a general-purpose circuit simulation program for nonlinear DC, nonlinear transient, and linear AC analyses. Circuits may contain resistors, capacitors, inductors, mutual inductors, independent voltage and current sources, four types of dependent sources, transmission lines, and the four most common semiconductor devices: diodes, BJTs, JFETs, and MOSFETs.

Spice2g6 has built-in models for the semiconductor devices, and the user need specify only the pertinent model parameter values. The model for the BJT is based on the integral charge model of Gummel and Poon; however, if the Gummel-Poon parameters are not specified, the model reduces to the simpler Ebers-Moll model. In either case, charge storage effects, ohmic resistances, and a current-dependent output conductance may be included. The diode model can be used for either junction diodes or Schottky barrier diodes. The JFET model is based on the FET model of Shichman and Hodges. Three MOSFET models are implemented; MOS1 is described by a square-law I-V characteristic, MOS2 is an analytical model while MOS3 is a semi-empirical model. Both MOS2 and MOS3 include second-order effects such as channel length modulation, subthreshold conduction, scattering limited velocity saturation, small size effects and charge-controlled capacitances.

To build a Spice input file for your circuit from *mextra* output run *sim2spice* or *pspice*.

SEE ALSO

mextra(1.vlsi)
mtp(1.vlsi), sim2spice(1.vlsi), pspice(1.vlsi), spcpp(1.vlsi)

SPICE User's Guide, **VLSI Design Tools Reference Manual**, UW/NW VLSI Consortium, University of Washington, (*SPICE Version 2G6 User's Guide*, A. Vladimirescu *et al.*, 15 October 1980).

Program Reference for Spice2, E. Cohen, ERL Memo. ERL-M592, Electronics Research Laboratory, University of California, Berkeley, June 1976.

SPICE2: A Computer Program to Simulate Semiconductor Circuits, L.W. Nagel, ERL Memo. ERL-M520, Electronics Research Laboratory, University of California, Berkeley, May 1975.

The Simulation of MOS Integrated Circuit Using SPICE2 A. Vladimirescu and Sally Liu, UCB/ERL M80/7, University of California, Berkeley, February 1980.

AUTHOR

(UCB)

BUGS

MOSFET Model, Level=2 does not work, due to a charge conservation problem (it grows).

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.

Isocmos

Complies with GTE 5 micron, isocmos process (lambda = 2.5 microns). Inputs and outputs on same side of PLA. Fabricated and tested.

CS3cis Complies with MOSIS 3 micron bulk CMOS process (lambda = 1.0 microns). Berkeley design, simulated but not fabricated. Inputs and outputs on same side of the PLA.

CS3tran

Same as **CS3cis** except inputs and outputs on opposite sides of the PLA.

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.

-S *num* Stretch power and ground lines by *num* lambda.

-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.

If the template CS3cis or CS3tran is used with an odd number of minterms and the -G option is used, there will be design rule violations; extra pieces of P+ implant along the bottom of the OR plane, which will have to be manually removed.

The templates Tcis and Ttrans imply a technology (fnmos) not supplied on the Berkeley tape. These templates will not be useful unless the associated technology files are obtained.

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

NAME

vic – view an integrated circuit layout (version 2.1).

SYNOPSIS

vic [*options*] *symbolname*

DESCRIPTION

Vic is an interactive graphics display program for integrated circuits that is technology independent and has a built–in hardcopy feature. It understands layouts in Caesar data base format. It currently displays only on the GP19 and Tek4105 graphics terminals, but it can produce a hardcopy on a number of devices.

The *options* are as follows:

–t *technology*

Supported values of *technology* are *nmos* and **cmos-pw**. Default is **cmos-pw**.

–h *plotter*

Supported values for *plotter* are **HP7221CT, HP7221AB, HP7580, HP7580B** and **Tek4662_31**. Default is HP7580.

–g *graphics*

Supported values for *graphics* are **GP19** and **Tek4105**. Default is GP19.

–f *format*

The only Choice for *format* of symbol to be read is **ca** (Caesar files).

COMMANDS

For all the commands, only the portion enclosed in parentheses need be typed and a list of the possible parameters for each command (if any) and current values are shown after the command in the menu.

(lay)ers <list>

sets the layers to be plotted. The list consists of layer names seperated by spaces, or the entire list may be preceded by a "+". In the latter case, the given layers are added to the plot ALREADY on the screen (it should be pointed out that a space must follow immediately after the "+", followed by the additional layers). A null list sets all layers to be plotted. Abbreviations are allowed. The first layer with the abbreviation as its leading part will be selected. (Thus, metal can be abbreviated m, me, met or meta, whereas metal2 will require the entire name. Warning: layer names such as metal2 and cut2 must, therefore, follow metal and cut, respectively in the technology files.) Default is all layers.

(n)esting levels <number>

set the number of levels in the symbol's hierarchy to be plotted. Any symbol at a level greater than this will show up only as a bounding box with its symbol name in the lower left corner. The current symbol is at level 1, its children are at level 2, and so on. Default is 1.

(w)indow

window in on the plot. Use the graphics cursor to move to the desired lower left corner of the window and hit the space bar. Then move to the upper right corner and do the same.

(hard)copy

produce a hardcopy of exactly what is shown on the terminal screen on a pen plotter. A grid may be placed over the hardcopy by specifying anything greater than zero when the program prompts for grid size. For this option to work, the user's terminal must communicate with the host through the plotter, in order that the plotter may intercept the plotting commands. For the Tek4105, the grid must be displayed prior to

hard copying.

(lab)els <value>
turn node labels on/off. Default is on.

(p)lot plot on the graphics terminal with the current options in effect.

(v)iew view on the graphics terminal the current symbol fully instantiated with all layers and node labels.

(g)raphics
return to the graphics screen (Tek4105 only).

(gri)d put a grid on the graphics screen (Tek4105 only).

(he)lp show the menu.

(e)xplain
explain each command.

(q)uit quit from the program.

(s)ymbol <name>
select the symbol to be plotted. The only symbols that can be specified are those in the sub-hierarchy of the top level symbol on the command line. Note that this is not a facility to reinitialize the vic with a new symbol. Executing this command with no name causes the list of symbols to be displayed. Default is the highest level symbol.

<control> C
causes current operation in progress to cease. On the Tek4105, to terminate a hard-copy in progress, depress the <shift> cancel key on the keyboard and type a carriage return.

DIAGNOSTICS

If an error occurs, a message is written to standard error and the program exits with a non-zero status.

FILES

technology.tec2
technology.cmp
symbol.att
symbol.ca

SEE ALSO

caesar(CAD1), tec(5.vlsi)

AUTHORS

Pat Bates
Larry McMurchie
Wayne E. Winder
Bruce A. Yanagida

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. "std" works well for most monitors, "pale" 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 semi-colons (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 "*".

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 "*SI".

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 "*".

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.

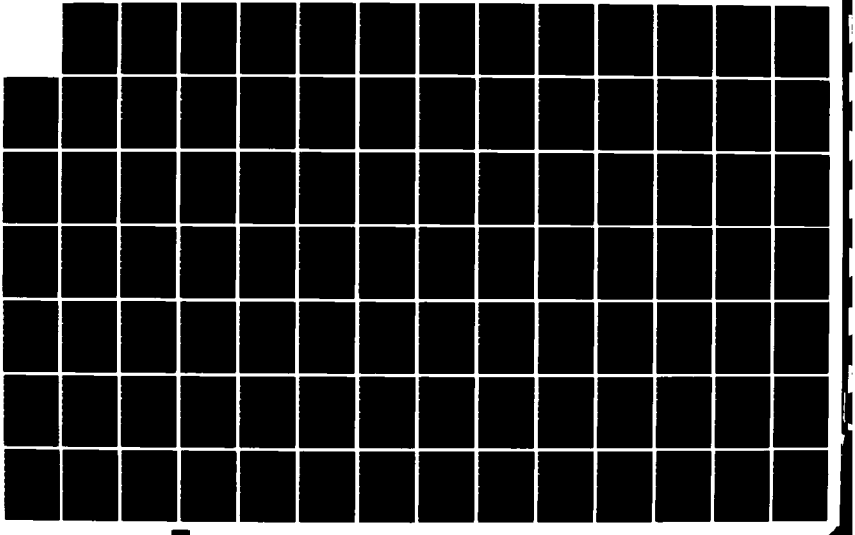
c Contact cut layer.

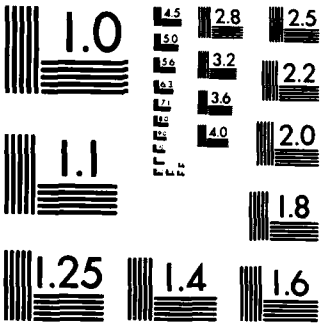
o Overglass hole (gray) layer.

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.





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

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

cdrc, cdrcscript, drc, drcscript - CMOS-BULK 3 micron and NMOS VLSI design rule checkers

SYNOPSIS

cdrc [-ku] *basename*.cif
cdrcscript *basename*.cif **drc [-ku]** *basename*.cif [*lambda*]
drcscript *basename*.cif

DESCRIPTION

Cdrc analyzes a CMOS CIF file for geometric rule violations using MOSIS cmos-bulk 3 micron process rules. *Drc* analyzes an NMOS CIF file for geometric rule violations using MOSIS (buried contact) rules. Both *cdrc* and *drc* are limited to rectilinear, orthogonal geometry. Wires are taken apart into rectangles, and round flashes are approximated by squares. Polygons and non-manhattan rectangles are simply ignored.

The options are as follows:

-k Keep around all intermediate files.

-u Keep around files of unfiltered error messages.

For large files, *cdrc* or *drc* should be run in batch mode, as a 7000 transistor chip takes over 2 11/780 cpu hours.

When *cdrc* or *drc* find violations, they create CIF files of rectangles marking the geometric edges involved. These markers are placed on the error layer (CZ) for cdrc and on the glass layer for drc. Separate files are created for each class of error, named err.*errortype*.*basename*.

To abort *cdrc* or *drc* hit the *BREAK* key and wait while it outputs some error messages until it eventually quits.

{*C*}*drcscript* will merge {*c*}*drc* output files, labels indicating error type, and the original CIF file into a single file, dre*basename*.cif. If this file is processed by *cif2ca* the results may be viewed with *caesar*. Errors show up as light blue boxes in the error layer for *cdrc* or as orange boxes in the glass layer for *drc*. Each pair of boxes involved in an error will have an associated *errortype* label which will be located at the midpoint between the centers of the two boxes.

MOSIS CMOS/BULK 3 micron process rules checked by *cdrc*:

Errortype	Microns	Rule
wWp	3	P-Well width
sWp	9	P-Well to P-Well spacing assuming all p-wells are connected to vss
dW	4	Diffusion size
dS	4	P+ diffusion to P+ diffusion spacing
	4	N+ diffusion to N+ diffusion spacing
	4	N+ diffusion to P+ diffusion spacing outside P-well
	4	N+ diffusion to P+ diffusion spacing inside P-well
pWp+DS	8	P+ diffusion in N-substrate to P-well edge spacing
Wpn+WnS	7	N+ diffusion in N-substrate to P-well edge spacing
pWn+DS	4	N+ diffusion in P-well to P-well edge spacing
pW	3	Poly width
pS	3	Poly to poly spacing
pSd	2	Field poly to diffusion spacing
pOg	3	Poly gate extension over field
gpSd	3	Gate poly to diffusion spacing
p+Od	2	P+ mask overlap of diffusion
	2	N+ mask to P+ diffusion spacing

Tn+S	3.5	P+ mask overlap of poly in diffusion
p+S	3	P+ mask to P+ mask spacing in diffusion
	3	N+ mask to N+ mask spacing in diffusion
Errortype	Microns	Rule
Dp+s	2	P+ mask to N+ diffusion spacing
	2	N+ mask overlap of diffusion
Tp+s	3.5	N+ mask overlap of poly in diffusion : P+ mask to poly spacing in diffusion inside P-Well
bcut		Cut must have metal and (poly or diffusion) underneath
wC	3	Contact width
cL	8	Maximum contact length
cS	3	Contact to contact spacing
pOc	2	Poly overlap of contact
PMCx	2.5	Poly overlap of contact in direction of metal
cSpc	3	Contact to poly channel spacing
mOc	2	Metal overlap of contact
dOc	2	Diffusion overlap of contact
cp+s	3	Contact to P+ and N+ mask spacing
scfp+	4	Shorting contact extension into P+ diffusion
scfn+	4	Shorting contact extension into N+ diffusion
mW	3	Metal width
mS	4	Metal to metal spacing
m2w	5	Metal2 width
m2S	5	Metal2 to metal2 spacing
c2w	3	Via width
c2S	3	Via to via spacing
m2Oc2	2.5	Metal2 extension over via
mOc2	2.5	Metal extension over via
CC2s	3	Via-cut separation
dOv	3	Diffusion overlap of via
dSv	3	Diffusion to via spacing when they dont overlap
pOv	3	Poly overlap of via
pSv	3	Poly to via spacing when they dont overlap
M2Pst	1	Metal2, metal and poly intersection 1 width
MPMM2x	4	Metal extension over the above intersection
PPMM2x	3	Poly extension over the above intersection
PM2st	5	Metal2 metal intersection(no poly) to metal2 poly intersection(with no metal) spacing
MPM2st	5	Metal2 poly intersection (no metal) to metal spacing
PPM2st	5	Metal2 metal intersection (no poly) to poly spacing

NMOS rules checked by *drc*:

Errortype	Rule	Lambda
dS	diffusion spacing	3.0
iOg	implant-gate overlap	1.5
iSg	implant-gate spacing	2.0
pS	poly spacing	2.0
pOg	poly-gate overlap	2.0
pSd	poly-diff spacing	1.0
cS	cut-cut spacing	2.0
dcSg	diff-cut to gate	2.0
mW	metal width	3.0
iNOg	implants with no gates	

XC	cuts with no D or P	
dW	diffusion width	2.0
ntdW	non-xtr diff width	2.0
iS	implant-implant	1.5
pW	poly width	2.0
gW	gate width	2.0
cW	cut min width	2.0
cL	cut max length	6.0
mOc	metal-cut overlap	1.0
dOc	diff-cut overlap	1.0
pOc	poly-cut overlap	1.0
sBP	buried-poly spacing	2.0
sBD	buried-diffusion spacing	2.0
oBD	buried-diffusion overlap	2.0
oBU	buried-contact surround	1.0
bW	buried-contact width	2.0

SEE ALSO

caesar(cad1), cif2ca(cad1)

A Geometric Design Rule Checker, Dorothea Haken, VLSI Document V053, Carnegie Mellon, 9 June 1980.

FILES

basename.cif
err*errortype*.*basename*
drc*basename*.cif
err*basename*.cif

AUTHOR

Dorothea Haken (CMU)

BUGS

The poly-overlap-gate check fails when the overlap is exactly zero (*drc* only).

Spacing checks do not consider mutual connectivity. Sometimes weird things will happen, and the generated spurious errors can be filtered by the bin_filter program, which examines local connectivity. Cuts in diffusion or poly that do not have metal covering are not reported.

Cuts in diffusion or poly that do not have metal covering are not reported (*drc* only).

Diagonal spacing checks do not consider the true diagonal distance.

SUGGESTIONS

Do not have basenames beginning with a number. Otherwise, this leads to serious errors in that *cdrc* assumes that to be the lambda value.

Try to have as short a basename as possible. This is because some flavors of UNIX restrict the length of filenames. Some of the intermediate files that are generated have quite long names.

The default lambda is 50 centimicrons for the *cdrc* routines. This scaling is done to overcome the inability of the routines to check for non-integer lambda violations.

It is advisable to run {*c*}*drc* in the background (batch mode), directing the output to a file, so you can look at the file later if needed.

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 for displaying VLSI circuits

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

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

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

–w ***xmin xmax ymin ymax***

(window) This option 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) This 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 ***layerlist***

(layer) Normally all layers are plotted. This option specifies which layers NOT to plot. The ***layerlist*** **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.

-N **(Printronix)** Send output to the Printronix.

-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 -w, -l, and -a 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)** This option 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 *patternfile*

(Pattern) This option lets you specify your own layers and stipple patterns. *Patternfile* 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 *fontfilename*

(Font) This option indicates which font you want for your text. The *fontfilename* 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*

SEE ALSO

cadrc(cad5)

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

AUTHOR

Dan Fitzpatrick (UCB)

MODIFICATIONS

(UW/NW VLSI Consortium, University of Washington)

BUGS

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

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

The -X option does not work for non-manhattan circuits.

NAME

decNor – Generates CMOS dynamic NOR form decoder layouts.

SYNOPSIS

decNor [*options*] **Inputs** [*OutFile*]

DESCRIPTION

DecNor is a program for generating CMOS dynamic NOR form decoder layouts in the "caesar" format. **DecNor** constructs caesar composition cells from caesar leaf cells and/or other composition cells. All caesar cells reside in the ./ca directory. Leaf cells have names of the form decNor_*.ca while composition cells have names of the form "OutFile"*.ca. Leaf cells must be copied from $UW_VLSI_TOOLS/lib/generators/decNor into ./ca before running **decNor**. The completed layout resides in "OutFile".ca. **Inputs** are the number of inputs to the decoder. "OutFile" can not begin with the string "decNor". The default for "OutFile" is the string "decGen".

As **decNor** is a cfl-based program it creates files of the form *.bd in ./ca.

The following table describes decNor's *options* although an abreviated listing can be obtained by invoking **decNor** with no arguments. Options prepended by "-" are active while those with "*" have not been implemented.

-f Stripped down layout for floor planning. Cells which occupy a large part of the decoder are represented in dummy layers allowing faster layout generation.

-t Layout of worst case path for timing estimates. Cells which are not part of the slowest electrical path are represented in dummy layers allowing faster generation, extraction and simulation.

-s A schematic of the decoder. Cells are represented as symbols (wires and transistors) drawn in black ink (labels) on a yellow background (P+ mask).

*p P-type decode transistors. Since N-type transistors have a lower on resistance they are the default decode transistor type.

-l Labels are added to inputs and outputs. Since labels increase the generation time they are not added as the default. When included they are prepended with "OutFile".

-b *banks*
The array of decode transistors will be repeated " *banks* " times. This feature can be used to distribute decoder outputs to a number of places with minimal additional area. Default is one.

*o *outs*
Stretch decoder to give " *outs* " lambda output spacing. This option simplifies connection by abutment.

*i *ins*
Grow decode xsistors to give " *ins* " lambda input spacing. This option allows the decoder to operate faster.

*v *ver*
Grow input inverters to fill vertical size of " *ver* " lambda. This option allows the decoder to operate faster.

*h *hor*
Grow evaluate/charge xsistors to fill horizontal size of " *hor* " lambda. This option allows the decoder to operate faster.

FILES

/ca/OutFile*.ca
/ca/OutFile*.bd
/ca/decNor_*.ca

SEE ALSO

caesar(CAD1), cfi(5.vlsi)

AUTHORS

David J. Morgan

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

gen_control – generate a control file for RNL

SYNOPSIS:

gen_control
(no arguments)

DESCRIPTION

gen_control is a program designed to quickly specify a control file for RNL simulation. *gen_control* provides for the proper insertion of quotes and use of parenthesis.

Typical file extensions to the basenames are assumed.

When starting up the *gen_control* program, the user will be prompted for the necessary information to be provided.
Assumed standard libraries are:
awstd.l & awsim.l

Prompts:

1. Basename:
The control file will be written in: *basename.l*
In the *.l* or control file assumed extensions are:
for the log file: *basename.rlog*
for the network: *basename*
for the plotfile: *basename.beh*

2. Comment:
A one line comment, which could be a short comment about the simulated circuit can be entered.

3. Simulation step increment value:
Enter the value of the simulation step in 0.1 ns units. The appropriate lisp command is automatically generated.

4. Definition of normal vectors:
To define a vector enter its name.
Then there will be a prompt for its type (bit, bin, oct, hex,dec)
Followed by a prompt for its elements.
A <CR> means skip this entry.

5. Definition of single indexed vectors:
Enter *basename* and after prompts: type, start index and number of elements.

6. Definition of a set of double indexed vectors:
Enter *basename* and after prompts: type, indexsize1 and indexsize2.

7. Definition of report for end of simulation step:
One of two types can be specified:
Just a <CR> specifies the normal def-report contents;
<any character><CR> specifies an optional type in which multiple vectors with double indexed nodes can be specified.
Next there will be a prompt for a comment to be included in every report (this portion only is

optional).
Then there will be prompting until a <CR> is entered for

nodenames (just enter the names) or
a vector name (first enter 'vec' and then the name).

In case of the optional report format, the multiple vector specification format is obtained by reponding with *'veci'*. Additional prompts will follow for basename and size.

8. Selection of output mode: logic analyzer style output:
Enter any character for selecting logic analyzer style output and a <CR> for standard output.
A report stating the order of columns in the output of RNL will be automatically generated.

9. Selection of output mode: glitch detection reporting:
Enter any character for selecting glitch detection and a <CR> for standard reporting of transients.

10. Definition of nodes with transient or glitch reporting:
Individual node names, vectors with single indexed node names and vectors with double indexed node names can be specified. Respond appropriately for names vectorsizes.

11. Definition of logic trigger conditions:
There are prompts for defining trigger conditions on individual node names, single vectors in invec type format, and single vectors in bitinvec type format.

12. Definition of additional RNL simulation set-up commands:
Enter the desired RNL commands. Terminate with an additional <CR>.

13. Definition of a timing pattern file:
Respond with <CR> if there is no such file (unlikely) or any other character if such is the case.
The filename assumed is: **basename.time**

14. Definition of wrap-up RNL commands:
Enter the desired simulation wrap-up commands (often just 'exit').
There is no syntax checking in *gen_control*. *gen_control* will put the quotes and parentheses at the right places. Any errors can be easily corrected using a standard text editor on the output file: *basename.1*.
This file can be inspected for correctness. Errors may be reported by RNL when running the simulation.

FILES

The output file is an ascii file and can be inspected. The files containing the library functions, network etc. must be at the correct place.
uwstd.1, uwsim.1, basename.1, basename, basename.rlog, basename.beh, basename.time

SEE ALSO

gen_time manual instructions

AUTHOR

Henriecus Koeman, John Fluke Mfg. Co., Inc.

DIAGNOSICS

none

BUGS

Please let the author know.

NAME

gen_time – generate a stimulus pattern for rnl.

SYNOPSIS

gen_time input_file output_file

DESCRIPTION

gen_time is a program designed to quickly specify input signal patterns which can be read by the lisp command interpreter of RNL. gen_time accepts a simple syntax without quotes and parentheses and accepts a simple means for defining states or commands for specific moments in time. The output of gen_time is typically read by the main control file, which contains the set-up information for the simulation. This control file can easily be obtained using the gen_control program. One of the commands should "load" the outputfile of gen_time.

Syntax summary:
time_range <start_time> <stop_time>
(must be the first command)
node_name <period> <state1> <time1> <state2> <time2> ...
invec <vector_name> <period> <value1> <time1> ....
bitinvec <vector_name> <period> <bitvalues1> <time1> ....
(note no spaces between individual bitvalues as in the equivalent rnl command!)
command <period> <rnl_command1> <time1> .....
(no alternate syntax allowed in rnl_commands here)
report <period> <time1> <time2> .....
(report 1 0 generates a report after every time step)
(must be the last command in the input_file)
mask <period> <enable_time> <disable_time>
(applies only to command line immediately following)
maskinv <period> <disable_time> <enable_time>
(applies only to command line immediately following)

FILES

The output file is an ascii file and can be inspected for programmed activity as a function of the time increments.

FURTHER EXPLANATIONS

The rules for the input file are discussed in more detail in the following, in particular those for the more complex waveforms.

Rule #1: Comments.
All lines starting with a semicolon are considered comments and are ignored.

Rule #2: Simulation interval definition must come first.
The first command in the .stim file must be the specification of the simulation interval; syntax and example:

time_range <start_time> <stop_time>
time_range 0 50

Note: Every value of time is in number of simulation step increments 'incr'. The global variable 'incr' is assigned a value with (setq incr <number>) where the number is

the size of the simulation step in 0.1 ns; this is done in the (".l") RNL control file.

Rule #3: The report definition must come last.
The last command in the .stim file must be the specification of how often RNL should print a report (using the def-report specification); syntax and examples:

report <period> <time1> <time2> ....

report 2 1	(report every 2 simulation steps at the end of interval 1; which occur at t=2, 4, 6, etc.)
report 10 3 6 9	(report every 10 simulation steps at the end of intervals 3,6 and 9: t=4, t=7, t=10, t=14, t=17, t=20 etc)
report 1 0	(report at the end of every simulation step)

Rule #4: How input signals are specified.
Signals are defined in one of the following ways:

nodename	(states must be l, h, u or x)
invec vectorname	(states must be a numerical type: decimal, octal (leading "0"), hexadecimal (0x....) or binary (0b...))
bitinvec vectorname	(states can be any combination of 1,0,u and x; no spaces between the elements)

followed by:
the period, and a number of combinations:
<state> <time>

If the period is "0" the specification relates to a one time event (the period is really infinity!).

Syntax and example for a simple waveform definition for simple node:
node_name <period> <state1> <time1> <state2> <time2> .....
node-name1 10 h 0 l 2 u 5 x 8
period is 10 simulation steps, signal h at t=0, l at t=2, u at t=5 and x at t=8; signalchanges repeat themselves at t=10, 12, 15, 18, 20, 22, etc..

Syntax and examples for a numerical vector definition (no undefined states can be specified in this case!):

invec vectorname <period> <state1> <time1> .....

invec name 10 0xa 0 0b1111 2 07 5 3 8

period is 10 simulation steps, vector is the hexadecimal "a" at t=0, binary 1111 at t=2, octal "7" at t=5 and a decimal "3" at t=8. Again, the pattern is repeated 10 simulation steps later.

invec name 0 0xa 0 15 5 017 9

The pattern is a single event: name is hexadecimal "a" at t=0, a decimal "15" at t=5; an octal "17" at t=9. This pattern does not repeat itself!

Syntax and examples for a bitvector definition:
bitinvec <vector_name> <period> <state1> .....
bitinvec vectorname 20 0000 0 1111 5 uuuu 10 xxxx 15
bitinvec vectorname 10 0x1x 0 u00x 5

Rule #5: Use of regular RNL commands allowed only with standard lisp syntax.
RNL commands can also be inserted in the same manner as node and vector stimulus; only the standard rnl syntax (with parentheses is allowed):

syntax:
command <period> <(rnl_command1)> <time1> .....

Rule #6: Masking of input signals and commands.
Except for the time_range command ALL gen_time commands are subject to mask commands, with will blank out the impact of the next command line immediately following the mask command line. After processing this next command line the mask is reset to a default which is a full enable. There are two mask commands:

'mask' and 'maskinv'

'mask' and 'maskinv' themselves are defined as having a period (a one time mask has a period of '0') and only 1 enable and only 1 disable time.

syntax:
mask <period> <enable_time> <disable_time>
maskinv <period> <disable_time> <enable_time>

Example:
mask 0 10 20
node2 5 h 0 l 5 u 10 x 15
will blank out any activity from node2 before
time increment 10 and after time increment 20.
maskinv 0 10 20
node3 5 h 0 l 5 u 10 x 15
will allow only node3 statements to be
effective before time increment 10 and
after time increment 20.

The commands scheduled for the time coinciding with the enable time of the mask will be effective, while the commands schedule for the time coinciding with the disable time will be disregarded.

Example of a typical stimulus file:

```
; Timing file for basic CRC Counter
; Simulation time:
time_range 0 36
; Run the clock at all times:
cl 2 l 0 h 1
; Reset:
r 0 h 0 l 1
; The following sequence is designed to exercise all nodes!
in 0 l 0 h 2 l 12 h 20 l 26 h 28 l 32 h 34
; We will start reporting the unchanged nodes just before
; the last ff changes state, which is at time increment 32:
mask 0 32 36
command 1 (printf "nodes unch:%S\n" (unchanged-since 100)) 0
; We report the state after every simulation step:
report 1 0
```

USING PATTERNS DEFINED USING GEN_TIME.

The output file from gen_time with the shell command:

gen_time basename.stim basename.time

Within the regular RNL control file (basename.l) one should include:

(load "basename.time")

AUTHOR

Henriecus Koeman, John Fluke Mfg. Co., Inc.

DIAGNOSICS

In case of an error in the inputfile gen_time will most likely print the first line number and the line itself where the error was detected and then terminate prematurely.

BUGS

Please let the author know.

NAME

lyra – Performs hierarchical layout rule check on caesar design.

SYNOPSIS

lyra [-vs] [-o output] [-p path] [-r ruleset] [-t technology] rootCaesarFile,
or
lyra -o [-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.

cmos-pw3
MOSIS 3 micron bulk cmos process, (see below for details). This is the default ruleset for technology cmos-pw.

cmos-M1v1
MOSIS 3 micron bulk cmos process, (see below for details).

isocmos
GTE 5 micron isocmos process.

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 *Klc*. 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 rulcc). 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.

-z **(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 **-z** 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.

DIAGNOSTICS

CMOS-PW3 MOSIS 3 MICRON CMOS DESIGN RULES, V1.0

"C s" contact-contact separation: 3u
"C w" contact width: 3u
"C2 f" metal2 extension around via: 2.5u
metal extension around via: 2.5u
"C2 s" via-via separation: 3u
"C2 w" via width: 3u
"C/C2 s" via-cut separation: 3u
"D s" active area-active area separation: 4u
"D w" active area width: 4u
"Dn+ w" N+ active area width: 4u
"Dp+ w" P+ active area width: 4u
"Dw w" P+ active area (not gate) width: 3u
N+ active area (not gate) width: 3u
"D/C2 s" if active area is not under via, via-active area separation: 3u
"D/C2 x" if active area is under a via, active area extension around via: 3u
"D/p+ s" N+ active area to P+ spacing: 2u
"M s" metal-metal separation: 4u
"M w" metal width: 3u
"MP/PMM2 x" step missing for metal2 step coverage
"M2 s" metal2-metal2 separation: 5u
"M2 w" metal2 width: 5u
"M2/P st" metal2/metal/poly width: 1u
"M/PMM2 x" metal step width for metal2: 4u
"M/P/M2 st" poly-metal separation when under metal2 with no overlap: 5u
"P s" poly-poly separation: 3u
"PMC x" extra .5 micron in direction of metal in poly-metal contacts
"Pw w" poly (not gate) width: 3u
"P/C2 s" if poly is not under via, via-poly separation: 3u
"P/C2 x" if poly is under a via, poly extension: 3u
"P/D s" poly-active area separation: 2u
"P/M/M2 st" poly-metal separation when under metal2 with no overlap: 5u
"P/PMM2 x" poly step width for metal2: 3u
"T w" Gate area width: 3u
"T/C s" contact to gate separation: 3u
"T/n+ s" P+ extension around gate outside p-well: 3.5u
"T/p+ s" gate inside p-well to P+ (of ohmic contact) separation: 3.5u
"VIA f" via has obtuse corner
"Wp s" p-well to p-well separation: 9u
"Wp w" p-well width: 3u
"Wp/n+Wn s" N+ active area (ohmic contact) to p-well

separation: 7u
"c f" metal and (poly or active area) required under cuts
metal extension around cut: 2u
active area extension around cut: 2u
poly extension around cut: 2u
"pW/n+D x" p-well extension around active area: 4u
"pW/p+D s" separation of p-well from P+ active area: 8u
"p+ s" p+ to p+ separation: 3u
"p+/D x" P+ extension around P+ active area: 2u
"sc f" split ohmic contact must be 4 microns into P+ active area and 4 microns into N+ active area
"tr f" malformed poly or active area abuttment: 3u extension
"tr p" polarity: P+ implanted transistor in p-well
polarity: N+ implanted transistor outside p-well

CMOS-PW3 MOSIS 3 MICRON CMOS DESIGN RULES, V1.1

Same as 1.0 with the following exceptions:

modified rules:
"C2 f" metal2 extension around via: 2u
metal extension around via: 2u
"M2/P st" metal2/metal/poly width: 3u
"M/PMM2 x" metal step width for metal2: 3u
"M/P/M2 st" poly-metal separation when under metal2 with no overlap: 3u
"P/C2 s" if poly is not under via, via-poly separation: 4u
"P/C2 x" if poly is under a via, poly extension: 4u
"P/M/M2 st" poly-metal separation when under metal2 with no overlap: 3u
"P/PMM2 x" poly step width for metal2: 3u

new rule:
"D W" Active Area transistor abuttment width: 4u

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

mexnodes – integrate intermediate nodes extracted by mextra with the original caesar design.

SYNOPSIS

mexnodes [*options*] *basename*

DESCRIPTION

Mexnodes is a shell script that uses *cif2ca* and *caesar* to generate a Caesar-format file. This file allows the user to view the intermediate nodes named by *mextra* on the original design. *Mexnodes* can be helpful when a simulation tool reports errors at a node not named by the user, as such errors are sometimes hard to locate. The output file created by *mexnodes* is named **mx***basename*.**ca**. This file can be then viewed using *caesar* in order to find a given node.

The *options* are as follows:

–t *technology*

Technology is one of nmos, isocmos, or cmos-pw. Default is nmos.

–l *lambda*

Lambda specifies the centimicrons to lambda correspondence of the design. Default is 200 centimicrons per lambda.

FILES

basename.ca
mx*basename*.ca
basename.nodes
basename.cif

SEE ALSO

caesar(CAD1), cif2ca(CAD1), mextra(CAD1)

AUTHOR

Terry J. Ligocki

BUGS

NAME

mextra – Manhattan circuit extractor for VLSI simulation

SYNOPSIS

mextra [**–g**] [**–s** *scale*] [**–o**] *basename*

DESCRIPTION

Mextra will read the file *basename*.**cif** and create a circuit description. From this circuit description various electrical checks can be done on your circuit. The circuit description is directly compatible with *esim*, *powest*, and *erc*. There are translation programs to convert *mextra* output to acceptable *spice* input (see *sim2spice*, *pspice* and *spcpp*).

Mextra creates four new files, *basename*.**log**, *basename*.**al**, *basename*.**sim** and *basename*.**nodes**. After *mextra* finishes it is a good idea to read the **.log** file. This contains general information about the extraction. It has a count of the number of transistors and the number of nodes, and it contains messages about possible errors. The **.al** file is a list of aliases which can be used by *esim*. The **.nodes** file is a list of node names and their CIF locations listed in CIF format. It can be read by *cifplot* to make a plot showing the circuit with the named nodes superimposed. The form of this cifplot command is:

cifplot *basename*.**nodes** *basename*.**cif**

The **.sim** file is the circuit description for use with simulation programs and electrical rule checkers. The format of the **.sim** file is described in the man page **simfile**(5).

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 *esim* so that it will understand these aliases.

Attributes

In addition 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, in the same way that a node name is attached. 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 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.

Transistors

For each transistor found by the exractor a line is added to the .sim file. The form of the line is:

type gate source drain length width x y
g=*attributes* s=*attributes* d=*attributes*

Type can be one of three characters, 'e' for enhancement, 'd' for depletion, or 'u' for unusual implant. (Unusual implant refers to transistors which are only partially in an implanted area. It will be necessary to write a filter to replace these transistors with the appropriate model in terms of enhacement and depletion transistors.) *Gate*, *source*, and *drain* are the gate, source, and drain nodes of the transistors. *Length* and *width* are the channel length and width in CIF units. *X* and *y* are the x and y coordinates of the bottom left corner of the transistor. *Attributes* is a comma seperated list of attributes. If no attribute is present for the gate, source, or drain, the g=, s=, or d= fields may be omitted.

The extractor guesses the length and width of a transistor by knowing the area, perimeter, and length of diffusion terminals. For rectangular transistors and butting transistors the reported length and width is accurate. For transistors with corners or for unusually shaped transistors the length and width is not as accurate.

It is possible to design a transistor with three or more diffusion terminals. The extractor considers these as *funny transistors*. They are entered in the .sim file in the form:

f*type gate node1 node2 ... nodeN xloc*

The 'f' is followed by the type : 'e', 'd' or 'u'. *Node1 ... nodeN* are the diffusion terminal nodes. As with any circuit with 'u' transistors, any circuit with 'f' transistors must be run through a filter replacing each of the funny transistors with the appropriate model in terms of enhancement and depletion transistors.

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 a node and substrate is in femto-farads. Capacitance values below a certain threshold are not reported. The default threshold is 50 femto-farads.

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
metal2	0.015	0.0
poly	0.05	0.0
diff(n)	0.10	0.1
diff(p)	0.10	0.1
poly/diff	0.40	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 **-e** 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:

L *node metal2Area metal2Perim metalArea metalPerim polyArea polyPerim diffArea diffPerim diffpArea diffpPerim*

Node is the node name. *x y* is the position of a point on the node. Currently this is always '0 0'. *metal2Area* through *diffpPerim* are the area and perimeter of the metal2, metal, poly, diffusion(n), and diffusion(p) layers in user defined units. (In addition the **-e** option causes transistors with only one terminal to be recorded in the .sim file as a transistor with source connected to drain.)

If the network is being extracted from the .cif file we suggest the node capacitance not be computed by mextra. Rather the **-e** option should be used. This puts the burden of computing node capcitance on the programs presim *and* sim2spice2. We feel this is advantageous because presim *and* sim2spice2 are filter programs linked directly to the type of simulation that is to be done. This will hopefully reduce some of the confusion associated with calibration.

Changing Default Values

As part of its start up procedure *mextra* reads two files: /usr/vlsibin/.cadrc and then a search for the first .cadrc from the current directory (.) to the the user's home directory is made. *Mextra* reads these files to set up constants to be changed without recompiling. The keywords for *mextra* are contained within the mextra environment of the .cadrc file. Declaration of

environments in the .cadrc file are described in .cadrc(5).

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

units *scale*

where *scale* is in centi-microns. The user may type in the chosen value for the scale directly.

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

mextra -s *scale basename*

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

areatocap *layer value*
perimetertocap *layer value*

value is atto-farads per square micron for area, and atto-farads per micron for perimeter. *layer* may be "poly", "diff", "metal", "metal2", or "poly/diff".

To set the capacitor values to those given in Mead and Conway the following lines would appear in the .cadrc file:

```
areatocap poly 40
areatocap diff 100
areatocap metal 30
areatocap poly/diff 400
perimetertocap poly 0
perimetertocap diff 0
perimetertocap diff 0
perimetertocap metal 0
perimetertocap poly/diff 0
```

The threshold for reporting capacitance may be 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

~/.cadrc
basename.cif
basename.al
basename.log
basename.nodes
basename.sim

SEE ALSO

powest(1.vlsi), pspice(1.vlsi), spcpp(1.vlsi), sim2spice(1.vlsi), spice(1.vlsi), drc(1.vlsi), erc(1.vlsi)
caesar(cad1),
cadrc(cad5), simfile(1.vlsi).

AUTHOR

Dan Fitzpatrick (UCB)

MODIFICATIONS

(UW/NW VLSI Consortium, University of Washington)

BUGS

Accepts manhattan simple CIF only, use **cifplot** –O to convert complicated CIF. For unusually shaped transistors the UW/NW modified *mextra* should be used, otherwise values will be quite inaccurate. The modified *mextra* will either yield accurate values or a "reasonable" guess, depending on the complexity of the unusual transistor. The modified *mextra* will tell you when the output values are only best estimates. The length/width ratio for unusually shaped transistors may be inaccurate. This is true for snake transistors. Attributes for funny transistors are not recorded. Node attributes are ignored unless the –e switch is present.

NAME

mtp – Multiple Time-series Plot for simulator output

SYNOPSIS

mtp ***behavior-file directive-file plot-file***

DESCRIPTION

Mtp plots the output of ***rnl*** and ***spice*** simulations on the Printronix line printer. ***Behavior-file*** is the ***rnl*** or ***spice*** output file, ***directive-file*** is a "specification file" for the plot, and ***plot-file*** is an output file to contain the plot suitable for printing on the Printronix line printer.

The use of ***mtp*** involves the following steps:

1. Generate a behavior file.

   If you are using ***rnl***, the directive

   openplot ***"behavior-file"***

   will cause the changes to all traced nodes to be written to ***behavior-file*** in addition to being written to the terminal. Quotes are necessary if the file name has any punctuation in it.

   The RNL directive

   closeplot

   will terminate the behavior file. If the entire ***rnl*** session is to be recorded ***closeplot*** is not required, as the file will be terminated when ***rnl*** exits.

   If you are using ***spice***, a behavior file may be specified as the third positional parameter of the ***spice*** command. Behavior records will be put on this file for all nodes specified on the Spice ***PLOT*** directive.

2. Generate a plot file from the behavior file using ***mtp***.

   The plot is sent to the Printronix printer using the Unix command

   lpr -l ***plot-file***

The contents of ***behavior-file*** are interpreted with the help of ***directive-file***. For the basic purpose of plotting the output of ***rnl*** or ***spice***, only a few directives need be supplied:

1. **start** ***time***
   Tells ***mtp*** when to start plotting. If not supplied ***time*** defaults to 0. Data is skipped on the behavior file until an event is found whose time is greater than or equal to the start time.

2. **stop** ***time***
   Tells ***mtp*** when to stop plotting. A stop value must be specified. If the stop time is greater than the time of the last event on the behavior file, the plot will be concluded with the last event.

3. **scale** ***time***
   Tells ***mtp*** the number of time units per inch. The default value is 1000.0. Because the time unit used by ***rnl*** for behavior file output is 1.0 nanosecond, this value will produce plots of ***rnl*** output having a scale of 1.0 microsecond per inch.

4. **logical** ***signal***
   This is used primarily for plotting ***rnl*** output. To select signals A, B and C for plotting in logical format the directives would be

   logical A
   logical B
   logical C

5. **analog** *signal heights*

Analog format is required when dealing with *spice* output because *spice* produces floating point values rather than logic levels. The height in inches of each trace must be specified. To select node voltages for nodes 1, 2 and 3 for plotting in analog format the necesary directives might be

```
analog V(1) 0.5
analog V(2) 0.5
analog V(3) 0.5
```

The order of selection directives in the file determines the order of the traces on the plot. The first signal selected is plotted closest to the time axis. A maximum of 20 signals may be selected on a given plot.

Spaces are used to separate the fields of a directive line. Blank lines or lines starting with # are ignored. Directives are case insensitive except for signal names.

EXAMPLE

The following example uses *mtp* to plot the behavior of a 10 bit counter, cntr10.net, shown here in netlist format:

```
; net file for 10-bit counter

; half adder made from gates
(macro half_adder (a b s c)
  (local h1 h2 h3)
  (nand (h1 2 16) a b)
  (nand (h2 2 16) a h1)
  (nand (h3 2 16) b h1)
  (nand (s 2 16) h2 h3)
  (invert c h1)
)

; one cell of a counter
(macro cell (in out Cin Cout)
  (local c1 c2 c3)
  (invert c1 in)
  (trans phi1 c1 c2)
  (invert c3 c2)
  (half_adder c3 Cin out Cout)
  (trans phi2 out in)
)

; declare global node names
(node count c in out phi1 phi2)

; carry-in to first significant bit controls counting action
(connect count c.0)

; generate the counter
(repeat i 1 10
  (capacitance out.i 1.234)
  (cell in.i out.i c.(1- i) c.i)
)
```

The rnl control file, cntr10.l, is as follows:

```
; RNL initialization file for 10 bit ripple-carry counter

  (load "uwstd.l")
  (load "uwsim.l")

  (read-network "cntr10")

; (setq report-form nil)  This turns off the report generator

  (setq incr 1000)

; bind symbols to node names

  (chflag '(phi1 phi2 out.10 out.9 out.8 out.7 out.6
                out.5  out.4 out.3 out.2 out.1))

  (defun init (dummy)

    (l '(count in.1 in.2 in.3 in.4 in.5
          in.6 in.7 in.8 in.9 in.10))

    (l '(phi2))
    (h '(phi1))

    (step incr)
    (l '(phi1))
    (step incr)

    (x '(in.1 in.2 in.3 in.4 in.5
        in.6 in.7 in.8 in.9 in.10))

    (h '(phi2))
    (step incr)

    (l '(phi2))
    (step incr)

    (h '(count))

    (wr-report)

    'done
  )


  (defvec '(bit bout out.10 out.9 out.8 out.7 out.6
                out.5  out.4 out.3 out.2 out.1))

  (defvec '(dec dout out.10 out.9 out.8 out.7 out.6
                out.5  out.4 out.3 out.2 out.1))
```

```
(def-report '("10 bit counter current state" newline " "
        count (vec bout) (vec dout)))
```

Generate the behavior for the counter using *rnl*

```
netlist cntr10.net cntr10.sim
presim cntr10.sim cntr10
rnl cntr10.l

init                            # initialize the counter

openplot "cntr10.evl"           # open the behavior file

                                # (.evl stands for event list)

c 30                            # run 30 clocks

exit                            # exit rnl
```

Generate the plot.

```
mtp cntr10.evl cntr10.mtp cntr10.plt

lpr cntr10.plt
```

The file *cntr10.mtp* could contain the following:

```
start    0.0
stop     20000.0
scale    1000.0
logical  phi1
logical  phi2
logical  out.1
logical  out.2
logical  out.3
logical  out.4
logical  out.5
```

The **start** and **scale** directives are not necessary but are included for the purpose of illustration. Although not required, these directives typically preceed the signal selection directives in the file.

When *mtp* runs it lists the contents of the directive file on the terminal and reports progress with the following messages:

```
Previous output cntr10.plt removed
Select and preprocess input data
Sort preprocessed events
Generate the plot
Rasterize for the Printronix
mtp complete, plot file is cntr10.plt
```

The "Rasterize for the Printronix" message marks the beginning of the longest step in the process which typically takes about a minute under moderate system loads.

Mtp creates scratch files named *fort.1*, *fort.2*, *fort.3*, *fort.4*, and *fort.7*. If any of these files are present when *mtp* is invoked it will exit with an error message. This can happen if *mtp* is aborted before having time to clean up the scratch files. If this happens the scratch files can be cleaned up with the Unix command

rm fort.[12347]

SEE ALSO

rnl(1.vlsi) spice(1.vlsi),

User's Guide to RNL **VLSI Design Tools Reference Manual,** UW/NW VLSI Consortium, University of Washington, (Christopher Terman, MIT Laboratory for Computer Science).

SPICE User's Guide, **VLSI Design Tools Reference Manual,** UW/NW VLSI Consortium, University of Washington, (A. Vladimirescu *et al.*, 15 Oct. 1980)

AUTHOR

William Beckett (UW)

NAME

mult – generate a cmos multiplier layout (version 1.0).

SYNOPSIS

mult [*options*] *caesarname*

DESCRIPTION

Mult is a module generation program for static cmos multiplier circuits. The layout is produced in "caesar" format. ***Mult*** requires a number of caesar cells with names of the form mult*.ca to exist in directory ./ca. These should be copied from $UW_VLSI_TOOLS/lib/generators/mult prior to running ***mult***. The generated layout is output in directory ./ca in caesar cells with names of the form "caesarname"*.ca. ***Mult*** is a cfl-based program and therefore also produces *.bd files. "Caesarname" may not begin with the string "mult".

The *options* are as follows:

-g Makes the left side horizontal bus ground. This is the default.

-m *mbits*
Sets the number of bits in the multiplicand operand. ***Mbits*** must be in the range 3 to 32. The default is 3.

-n *nbits*
Sets the number of bits in the multiplier operand. ***Nbits*** must be in the range 3 to 32. The default is 3.

-p *P_string*
labels the propduct output bits with labels "P_string0", "P_string1", "P_string2", etc. with "P_string0" attached to the lsb. These labels appear on the right side and the bottom side of the layout. The default is "p".

-s Makes the number representation signed (two's complement). This is the default.

-u Makes the number representation unsigned.

-v Makes the left side horizontal bus Vdd.

-x *X_string*
labels the multiplicand input bits with labels "X_string0", "X_string1", "X_string2", etc. with "X_string0" attached to the lsb. These labels appear on the top side of the layout. The default is "x".

-y *Y_string*
labels the multiplier input bits with labels "Y_string0", "Y_string1", "Y_string2", etc. with "Y_string0" attached to the lsb. These labels appear on the left side of the layout. The default is "y".

FILES

./ca/*caesarname**.ca
./ca/*caesarname**.bd
./ca/*mult**.ca

SEE ALSO

caesar(CAD1), cfl(5.vlsi)

AUTHORS

Wayne E. Winder

NAME

netlist – a simple network description language for VLSI circuits

SYNOPSIS

netlist *infile* [*outfile*] [**-o**] [*-tech*] [**-u***units*] [**-s** *n*] [**-d** *n,m*] [**-e** *n,m*] [**-i** *n,m*] [**-l** *n,m*] [**-p** *n,m*]

DESCRIPTION

Netlist requires an input file with any/all extensions on the command line. An optional output file can be specified. Additional options are described below;

-o Uses old input format. Size specifications are taken to be length/width rather than width/length.

-tech Uses *tech* in the technology portion of the units/tech line at the beginning of the simulation file produced (Default is nmos).

-u*units* Sets the number of centi-microns per lambda to *units* (Default is 250). **Warning:** The "units" set by this option appear in the comment line of the *.sim* file. This value is not used by PRESIM and does not influence an RNL simulation.

-s *n* Uses number *n* as initializer for internal node names; useful when you want to merge the results of separate *netlist* runs.

-d *n, n* Sets the default width to *n* and length to *m* for depletion devices. The defaults are *n*=8 and *m*=2.

-e *n, n* Similar to **-d** except for enhancement devices. The defaults are *n*=2 and *m*=2.

-i *n, n* Similar to **-d** except for intrinsic devices. The defaults are *n*=2 and *m*=2.

-l *n, n* Similar to **-d** except for low-power devices. The defaults are *n*=2 and *m*=2.

-p *n, n* Similar to **-d** except for p-channel devices. The defaults are *n*=2 and *m*=2.

In addition, if node alias records (= node1 node2 ...) are declared using "connect" (See *netlist* reference documents) they appear in a file with the name "basename.al". The basename is the input file name minus its last extension.

Netlist is a macro-based language for describing networks of sized transistors. Names in *netlist* refer to nodes, which presumably get interconnected by the user through transistors. Macros for describing transistors can be found in the *NETLIST User's Guide*. In addition to transistor macros *netlist* provides macros that allow the user to set node capacitance, specific node delays (in tenths of nanoseconds), and transistor threshold voltages. The user may also define his own macros.

The **load** command uses the environment variable *RNLPATH* (default *.:$UW_VLSI_TOOLS/lib/rnl*). See the *NETLIST User's Guide* for details.

SEE ALSO

presim(1.vlsi), *rnl*(1.vlsi),

NETLIST User's Guide, **VLSI Design Tools Reference Manual**, UW/NW VLSI Consortium, University of Washington,

AUTHOR

Christopher Terman (MIT)

NAME

pads – generate a cmos padframe layout (version 1.0)

SYNOPSIS

pads *caesarname* < *frame_spec*

DESCRIPTION

pads is a module generation program for a MOSIS 3 micron cmos padframe layout. This generator uses leaf cells derived from the MIT pads received from MOSIS. The leaf cells and the layout that is produced are in "caesar" format. *Pads* looks for caesar leaf cells with names of the form pad*.ca in the directory ./ca. These should be copied from $UW_VLSI_TOOLS/lib/generators/pads prior to running *pads*. *Pads* also reads in a frame_spec file from the home directory (not ./ca). The frame_spec file definition is provided in the text that follows. The generated layout cells (composition cells) appear in directory ./ca with names caesarname*.ca. *Pads* is a cfi-based program and therefore also produces *.bd files. "Caesarname" may not begin with the string "pad".

There are no *options*.

FRAME_SPEC

The frame specification is a text file made up of one frame specifier followed by several pad specifiers. These records are terminated with ';' and may cross line boundaries. Individual fields within records should not cross line boundaries. The syntax is 'c-like'; comments may be placed anywhere with the /* ... */ convention.

The frame specifier is made up of a type specifier followed by an optional connection layer specifier. The type specifier is one of C28_46x34, C40_46x68, C40_69x68, C64_69x68, C64_79x92, or C84_79x92 (the first number indicates the number of pins on the frame, the second and third numbers give the x and y dimensions of the entire frame in hundreds of microns). The connection layer specifier indicates what material connects the individual pad circuitry to the interior of the chip (across the quard ring). This specifier may be METAL2 or POLY. Default is *POLY*.

The pad specifiers are used to determine the type of circuitry to place on specific pad sites. Pad specifiers are made up of pin number, pad type, and optional label and connection specifiers.

The pin number is an integer between 1 and the number of pins for the frame specified (see above). For the 28 pin frame, pin number 1 is in the middle of the right side of the frame. For the 40 and 64 pin frames, pin number 1 is immediately above the middle of the right side of the frame. For the 84 pin frame, pin number 1 is the rightmost pin on the top of the frame. Pin numbering procedes counterclockwise in all cases.

The pad type is one of padlvdd (power), padlgnd (ground), padlin (input), padlout (output), padlttl (ttl output), padlts (tri state output), padlbin (buffered input), padlbit (buffered ttl input) or padlsp (frame spacer - never required).

The optional label specifiers are of the form 'BP = label', 'L1 = label', 'L2 = label' and 'L3 = label'. BP, L1, L2 and L3 indicate where on the pad circuitry to place the label; on the bonding pad, on the leftmost connection on the bottom of the pad circuitry (when viewed with bonding pad on top), second from left and third from left, respectively. 'Label' is any string beginning with a letter and containing only non-special characters. Special characters include '=', ';' and '/'. Special characters can be included in strings by placing double quotes around the string and preceding the special character with the backslash character. For details of what connection connects to what portion of the pad circuitry, view the appropriate circuit from padl*.ca using caesar. The connections should be annotated with local labels to avoid ambiguity. Not all connections appear on all pads.

The optional connection specifier indicates which connection to the interior is to receive a contact, after crossing the guard ring. This specifier is of the form 'CN = layer', where N is 1, 2 or 3 and is identified as above. 'Layer' is one of METAL, POLY, or METAL2. Default is *METAL*. If the layer is the same as the input connection material (specified in the first record), no contact is placed. If different, a contact is placed. POLY may not be routed to METAL2 and vice-versa.

RESTRICTIONS

Pins may not be assigned more than once. Only those pins required need be assigned.

In ceratin corners of certain frames, tristate pad connections do not cross the quard ring.

In the 28, 40, and 64 pin frames, pin 1 should be vdd or blank. In the 84 pin frame, pin 10 should be vdd or blank.

Each frame must include at least one VDD pad and one Ground pad. These pads may only connect to the interior with METAL.

FILES

./ca/*caesarname*.ca
./ca/*caesarname*.bd
./ca/*pad*.ca
$UW_VLSI_TOOLS/src/examples/pads/input
(for a frame_spec example)

SEE ALSO

cfl(5.vlsi)

AUTHORS

Wayne E. Winder

NAME

peg – finite state machine compiler

SYNOPSIS

peg [–s] [–t] [file]

DESCRIPTION

Peg (*PLA Equation Generator*) 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

pla2net - generate netlist macro from truth table of pla

SYNOPSIS

pla2net *basename*

DESCRIPTION

pla2net generates a netlist macro using the truth table definition for a pla as an input. This truth table may have been obtained using PEG and EQNTOTT. *pla2net* expects that a file named '**basename.tt**' is in the current directory; if this is not the case an error message will be generated. The output of *pla2net* will be stored in a file named '*basename*.net'.

The macro defined in the '*basename*.net' file looks as follows:

(macro basename (output input) where:

- the basename is identical to the basename in the commmand line of pla2net;
- output is an outputvector, numbered from left-to-right as in the truth table and a lay-out generated with tpla starting with output.1;
- input is an inputvector, number from left-to-right as in the truth table and a lay-out generated with tpla starting with input.1.

Note: When designing pla's for sequential state machines with PEG, the innermost inputs and outputs of the pla will be the least significant bit. The state register inputs and outputs must be wired accordingly (mirror and shift numbering of input vector in the netlist description for the top level sequential state machine, which includes the feedback register, is necessary).

INPUT FILE

basename.tt

OUTPUT FILE

basename.net

SEE ALSO

Manual entries for PEG, EQNTOTT.

AUTHOR

Henriecus Koeman, John Fluke Mfg. Co., Inc.

BUGS

The current version only supports cmos technologies. The source code can easily be modified for other technologies.

NAME

presim – a netlist preprocessor for the rnl VLSI circuit simulator

SYNOPSIS

presim *infile outfile* [*configfile*] [*-g*] [*–cfile,min*] [*–tfile,min*] [*presist,voltage*]

DESCRIPTION

Presim converts the **.sim** file into a binary file to be used by ***rnl***.

The parameters and options are as follows:

infile A net list file that must include any/all extensions;

outfile An output filename must be specified on the command line;

configfile (optional) A file to set lambda and RC parameters for nodes and transistors in the netlist (see the *presim* user's guide for descriptions of the parameters and syntax).

-g Suppresses the sum-of-products formation. This may be desired if you think sum-of-products is formed wrong otherwise the advantages of the transistor and node reduction make this option unattractive.

–cfile, min Writes a list of node names and capacitances to the specified *file*. Only capacitances larger than *min* will be included.

–tfile, min Writes a list of transistors and RC values to the specified *file* -- there are two entries for each transistor. The R's come from the size of the transistor, C's from the source/drain capacitance. Only RC values larger than *min* will be included.

–presist, *voltage*

Provides a worse-case estimate of the circuit power consumption by assuming that all the pullups (DEP or LOWP devices with drain=*Vdd*) are all on simultaneously. *Voltage* specifies the supply voltage,

Presim also attempts to open the file *basename*.**al**, where *basename* is defined as the input file name minus its last extension. It is non-fatal for this file to be absent.

SEE ALSO

PRESIM User's Guide, **VLSI Design Tools Reference Manual**, UW/NW VLSI Consortium, University of Washington, (Christopher Terman, MIT Laboratory for Computer Science).

AUTHOR

Christopher Terman (MIT)

BUGS

Propagation of X state information for cmos circuits in ***rnl*** is unreliable if the gate reduction in ***presim*** is performed. If this information is required, suppress gate reduction with the -g option in ***presim***.

NAME

presto – combinational logic minimization program

SYNOPSIS

presto

DESCRIPTION

Presto is an efficient combinational logic minimization program. This program not only reduces the number of product terms, increases the number of don't care inputs, but also reduces the number of the output connections. Therefore, this program is very useful to pla designers.

Input is taken from standard input. Output goes to standard output.

An example of typical input is as follows:

```
.i 4
.o 2
.l
.p 4
10x1 11
000x 1x
1111 01
0101 10
.e
```

The integer after ".i" is the number of input variables. The integer after ".o" is the number of output variables. The integer after ".p" is the number of input product terms. ".l" is optional for input listing. There is another option ".d" for intermediate results.

In the input part, 1 means logic level 1, 0 means logic level 0, x(or -) means don't care. In the output part, 1 means that the term is connected to the output, 0 means that this term is not connected to the output, and x(or -) means that the output doesn't care whether this term is connected or not. ".e" means the end of the input file. When there is a format error in the input file, the program will give the message: 'INPUT FORMAT ERROR!' and abort the job.

AUTHOR

Sheng Fang

NAME

pspice – prepare an input file for the Spice circuit simulator

SYNOPSIS

pspice [**-rm**] [**-nos2s**] [**-d** *defsfile*] [**-m** *modelfile*] [**-e** *expfile*] *basename*

DESCRIPTION

Pspice is a shell script for preparing Spice input from information from several sources. ***Pspice*** runs ***sim2spice*** to convert from a ***basename*.sim** format circuit description to a Spice-compatible description and modifies the ***sim2spice*** node label translation table to be acceptable Spice comments. It then runs ***spcpp*** to translate a pseudo-Spice formatted file that contains symbolic node labels to a Spice-acceptable file. Finally, ***pspice*** concatenates the circuit description file, the translation table, a file of untranslated Spice input, and the translated Spice input into a single file named ***basename*.spcin**. This file is usually an acceptable Spice input file. The optional parameters can be used to cause parts of this process to be skipped.

The options and parameters are:

-nos2s Suppresses the execution of the ***sim2spice*** step.

-rm Indicates that the files created in intermediate steps are to be deleted.

-d ***defsfile*** Specifies a file to be used as a ***sim2spice*** definitions file.

-m ***modelfile*** Specifies a file that contains Spice input that is to be included (untranslated) in the final output. It is intended that ***modelfile*** name a file containing Spice .MODEL cards as well as other Spice commands that are independent of the particular circuit being modeled.

-e ***expfile*** Specifies a file that contains pseudo-input for Spice. ***Spcpp*** will interpret strings in ***expfile*** that are bracketed by '<' and '>' as node names to be translated into ***spice*** node numbers using the translation table (***basename*.names**) created by ***sim2spice***. Lines containing bracketed tokens are converted into Spice comments. It is intended that ***expfile*** contain Spice commands that describe the experiment to be simulated on the circuit. The ability to use mnemonic node names makes the preparation of Spice input much easier and it means that the description of the experiment need only be specified once, even if the circuit is modified and reextracted. If ***expfile*** is not specified then spcpp is not executed.

basename Specifies the base name for the files describing the circuit. If ***sim2spice*** is run then a file named ***basename*.sim** must exist. If ***sim2spice*** is not run then the files ***basename*.names** and ***basename*.spice** must exist.

FILES

***basename*.sim**	circuit description input to ***sim2spice***
defsfile	optional ***sim2spice*** defs input
***basename*.names**	modified ***sim2spice*** translation table output. This is read by ***spcpp*** (*)
***basename*.spice**	***sim2spice*** output Spice format circuit element definitions (*)
modelfile	optional Spice .MODEL commands to be included in ***basename*.spcin**
expfile	input to ***spcpp*** containing pseudo-***spice*** commands describing the experiment to be simulated
***basename*.spcx**	translated output from ***spcpp*** (*)
***basename*.spcin**	The Spice input deck created by concatenating ***basename*.spice**, ***basename*.names**, ***modelfile***, and ***basename*.spcx**

Note: Files marked (*) are deleted by the -rm option.

SEE ALSO

sim2spice(1.vlsi), spcpp(1.vlsi)
spice(1.vlsi)

mextra(1.vlsi), cifplot(CAD1)

AUTHOR

Robert Fowler (UW/NW VLSI Consortium, University of Washington)

DIAGNOSTICS

The error messages are intended to be self explanitory. Note that *sim2spice* and *spcpp* produce their own error messages.

BUGS

The command line is long enough to tempt a user to call *pspice* from yet another shell script. A better way to do this is to set up an alias for *pspice* with the commonly used options already set.

NAME

rnl – timing and logic simulator for VLSI circuits

SYNOPSIS

rnl [*cmdfile*]

DESCRIPTION

Rnl (NetLisp) is a timing logic simulator for digital NMOS circuits with a lisp-like command interpreter. It has also been used with many CMOS circuits with some success. The *Rnl User's Guide* discusses some of the limitations found in simulating CMOS circuits. To use *rnl*, one needs a **.sim** file for the circuit to be simulated. This can be derived from the mask file (e.g., CIF) or developed using *netlist*, a program that processes textual schematics.

One must first convert the **.sim** file to a network file suitable for use by *rnl*. To do this run *presim*:

presim *filename*.**sim** *netfile* [*config_params*]

which converts the file *filename*.**sim** into *netfile*, a binary file for *rnl*. (see *Presim User's Guide* for information on the various configuration parameters.

The optional *cmdfile* is the file *rnl* initially reads for user input. Usually one prepares a command file that loads one or more library files containing RNL function definitions and reads in the network from *netfile*. As simulation proceeds, user defined functions developed for testing the circuit can be added to the command file. At a minimum the command file should contain the commands

(load "uwstd.l")
(load "uwsim.l")
(read-network "*netfile*")

When using the **load** command both *netlist* and *rnl* search the current directory and then any directories specified in the environment variable *RNLPATH*. The value of *RNLPATH* defaults to *$UW_VLSI_TOOLS/lib/rnl*. **Read-network** does not use *RNLPATH*. *Netfile* must be produced by *presim*. When the end-of-file is reached in the command file, input is taken from stdin. Commands and formats to be used are given in the *RNL User's Guide*.

The top level of *rnl* is a simple loop:

(1) read command from current input;
(2) evaluate command, performing specified actions;
(3) print the result and loop back to (1).

The following is a list of the objects that *rnl* knows about

numbers -- signed integers. (16 bits on PDP11s, 24 bits on VAXen, 28 bits on PDP10s).
-- floating point.

strings sequences of characters enclosed in quotes ("). Useful as constants for file names, print statements, etc. Special characters can be introduced into the strings by using the backslash escapes:

'\n'	newline
'\r'	return
'\t'	tab
'\ooo'	ascii code "ooo" where ooo are octal digits

symbols variable names. Any sequence of characters that isn't a number, string, or some special character -- starting symbols with a letter, followed by more letters, numbers, and punctuation is usually a safe bet.

nodes an electrical node.

lists a sequence of objects enclosed in parentheses. Standard LISP syntax applies, including dot notation. The empty list "()" is also called "nil".

subrs primitive, or built-in, functions (like +).

The functions are listed by application area. The areas are:

- arithmetic functions
- predicates
- list functions
- I/O functions
- miscellaneous functions
- special forms
- network/simulation functions
- functions defined in "uwsim.l"

SEE ALSO

netlist(1.vlsi), *presim*(1.vlsi), *simfile*(5.vlsi)

RNL User's Guide, **VLSI Design Tools Reference Manual**, UW/NW VLSI Consortium, University of Washington, (Christopher Terman, MIT Laboratory for Computer Science).

AUTHOR

Christopher Terman (MIT)

BUGS

User defined macros with the same name as a node in the net list puts *rnl* into an infinite loop.

Propagation of X state information for cmos circuits is unreliable if the gate reduction in *presim* is performed. If this information is required, suppress gate reduction with the -g option in *presim*.

NAME

rulec – Compile design rules for Lyra

SYNOPSIS

rulec [-le] 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.

-e (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 mextra format to Spice (circuit simulator) format

SYNOPSIS

sim2spice [**–d** *defs*] *basename*.sim

DESCRIPTION

Sim2spice reads the *basename*.sim, *basename*.nodes and *basename*.al files created by *mextra* and creates a Spice readable circuit description, *basename*.spice. Spice requires node numbers and *sim2spice* generates a translation table *basename*.names which shows the *mextra* nodelabel corresponding to a given node number.

The user can specify his/her own translation table by using the *-d* option, where *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.

Sim2spice also reads 'N' lines generated by mextra with the -o switch. In order to compute capacitances from this it must have a set of conversion factors between length/area and capacitance. These are specified in the sim2spice section of '.cadrc' file in exactly the same format as in the mextra section of the '.cadrc' file (see mextra).

The program has been extended so that a comment line beginning with "|=" is interpreted as an MIT .sim style node equivalence line.

To create a complete Spice input file it is necessary to append applicable Spice model descriptions as well as the user's Spice simulation commands to the *basename*.spice file.

It is recommended in most cases that the user run *pspice* rather than *sim2spice*. *Pspice* incorporates the features of *sim2spice* but will in addition allow the user to build all of the Spice input file in one step. *Pspice* also incorporates the features of *spcpp*.

FILES

basename.sim
basename.nodes
basename.al
basename.spice
basename.names

SEE ALSO

mextra(1.vlsi), spice(1.vlsi), pspice(1.vlsi), spcpp(1.vlsi)

AUTHOR

Dan Fitzpatrick (UCB)

MODIFICATIONS

Neil Soiffer (UCB) -- cMOS fixes.

Rob Fowler (UW/NW VLSI Consortium, University of Washington) -- node equivalence handling and misc. bug fixes.

BUGS

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

Warning: for nMOS circuits the node names "ENMOS" and "DNMOS" are preempted by *sim2spice* as synonyms for "BULK".

The node equivalence handling is not completely general. New nodes can be added to equivalence classes, but classes cannot be merged. This is detected and an error message is produced.

NAME

simscope – view time-series of simulator output.

SYNOPSIS

simscope

DESCRIPTION

simscope is designed to display signal output produced by RNL or SPICE on a Tek 4105 or a GP-19 graphics terminal. To make hardcopies, you need a Tek 4695 printer (or compatible hardcopy device) in conjunction with the Tek 4105 graphics terminal. Any program that might periodically interfere with the display, notably sysline, should be switched off.

General Rules for Using *simscope*:

1. Names to be entered in response to *simscope*'s requests may contain alpha as well as numerical characters.

2. Numbers to be entered in response to *simscope*'s requests may be fixed-point or floating-point numbers (the latter format is also referred to as scientific notation). Examples of fixed-point numbers are 123, 3.55, +45000, etc. Examples of numbers in scientific notation are 3.5e4, 0.333e-9, 0.1e-9, +244.5e05, etc. Numbers may not be negative (negative time scales, times, and positions do not make sense to simscope).

3. While entering a name or a number, characters typed erroneously may be deleted with either the RUBOUT key (CONTROL-H is equivalent) or the BACKSPACE key (DELETE on some keyboards).

4. All numbers displayed by *simscope* are in scientific notation. The mantissa consists of one digit, a decimal point, another six digits, the "e" indicating the exponent, the sign for the exponent, and two digits for the exponent.

5. After reading a behavior file, *simscope* uses the same time units as those used in the behavior file. These are nanoseconds (ns) in RNL-generated files and seconds (s) in SPICE-generated files.

6. Indeterminate RNL signals (state "X") are displayed with a logic level of 0.5.

How to Start and Exit *simscope*

Work with *simscope* is most convenient if you change to the directory that contains the behavior file you want to view. Being in that directory, simply enter *simscope*. *simscope*comes up with a greeting display. The window begin and end times are set to 0 (B = 0.000000+00 and E = 0.000000+00) and, consequently, the time scale is 0 (T = 0.000000+00), too. The mode (see below) is set to fixed-time-scale mode. The Y-scale, in units per division, is set to 1 (Y = 1.000000+00).

Below these indicators the menu is displayed, followed by the request that you hit a key indicating the menu function you wish to select. Valid keys are: f, n, b, e, t, y, c, d, s, r, and q, all of which may be entered as capitals (with the SHIFT key). Each letter represents the first

letter of the corresponding menu function (see below). After you press any of these keys, the corresponding function is immediately activated - no RETURN is necessary.

To exit *simscope*, press the "q" key (Quit).

simscope Functions

File

This function serves two purposes. First, it provides the following information about the file presently loaded:

name of present behaviour file

file begin time (the time of the first signal entry in the file)

file end time (the time of the last signal entry in the file)

for each signal in the file:

first change (the time of the first entry of the signal in the file)

last change (the time of the last entry of the signal in the file)

y-position (the vertical position of the signal in the display; a number between 1 and 99, 1 is the lower end of the window, 99 is the upper end of the window).

name (the signal's name)

The second purpose of the File function is to facilitate the loading of a different file. If you press "y" (yes) in reply to the function prompt, File will ask you for the name of the new behaviour file you want to load. Any other key will terminate the File function, display all signals, and return you to the menu. If you enter a name, the corresponding file is read. Reading the behaviour file may take awhile, during which time the cursor may flash at various positions on the screen (hence the message: "Reading file. Please wait. Don't worry if the cursor flashes a bit.").

Nodes

You will be asked for a node name (default is the node last entered). *simscope* then asks whether you want to display the node's signal or delete the node's signal from the display. + or just RETURN means display, - means delete from display (the y-position of the signal is set to zero). Next simscope asks for the position (vertically) in the window. Enter 99 for the very top of the window, 1 for the bottom of the window, any number between 1 and 99 for an intermediate position. (One division on the y-scale is equivalent to 5 positional units.)

Reposition a signal by entering its name with Nodes, then enter the desired new position.

Assuming that you normally want to change more than one signal in a row, file information (as in the File function) is displayed after you enter a y-position, providing you with a summary of the information on all nodes if the behaviour file.

Use the Display function to display the signals.

Begin

Sets the window begin time ("B = " at left side of the window).

In fixed-time-scale mode, a change of the window begin time moves the window (whose time width remains unchanged) across the file.

In fixed-window-end mode, a change of the window begin time expands or contracts the window in a "rubber-band-like" fashion.

End

Sets the window end time ("E = " at right side of the window).

The time scale will be readjusted automatically to be consistent with the new window end time.

The window begin time is not affected.

Setting the window end time will switch over to fixed-window-end mode. In this mode changes to the window begin time will not affect the window end time, but will readjust the time scale. You can use End for switching to fixed-window-end mode (without actually changing the window end time) by confirming the present (default) value of the end time. To do that press "e", then just hit RETURN.

T-scale

Sets the time scale of the window ("T = " below the window).

The window end time will be readjusted automatically to be consistent with the new time scale.

The window begin time is not affected.

Setting the time scale will switch over to fixed-time-scale mode. In this mode, changes to the window begin time will not affect the time scale, but will readjust the window end time. You can use T-scale for switching to fixed-time-scale mode (without actually changing the time scale) by confirming the present (default) value of the scale. To do that press "t", then just hit RETURN.

Y-scale

Sets the vertical scale in units per division ("Y = " below "B = "). The default is 1, which is a good value for RNL. The vertical extent of SPICE signals is usually larger, however (for example 5 Volts, i.e. 5 units). Increasing the Y-scale allows you to fit more of the larger signals on the screen without overlapping.

Copy

To make hardcopies, you need a Tek 4105 and a Tek 4695 printer (or compatible device). (If you activate the Copy function on a GP-19 terminal, you will get the message: "Use a Tek 4105 terminal. (You can also use MTP). Hit any key to continue.")

Display

All signals with a y-position greater than 1 are displayed. Use this function to display the signals after you have made changes for any node (deletion, positioning or repositioning), or if you want to refresh the display for any reason.

Save

All parameters determining the particular display state are saved for later retrieval. "Save" first asks you for a name of the file in which you want to keep the present state. It then saves in this file the present behaviour file name, window begin time, time scale, window end time, mode (1 for fixed-time-scale mode, 0 for fixed-window-end mode), and the names of all displayed signals with their positions on the y-axis.

Saved states can be restored easily with the Retrieve function. You may conveniently consider the names of "Save" files as markers (possibly with short names like 1, 2, 3, ..., or a1, register3, Load10, etc.), which can be "jumped to" with the Retrieve function.

Retrieve

Retrieve is the function used to restore a previously saved display state. You are asked for the name of the file containing the state to be restored. If the state you want to restore belongs to a behaviour file different from the one on which you are working presently, then the new behaviour file is read (this may take some time).

Quit

This function terminates *simscope* and returns you to the UNIX shell.

The use of *simscope* to display RNL or SPICE results involves the following steps:

1. Generate a behavior file.

   (a) If you are using ***RNL***, the directive

   openplot *"behavior-file"*

   will cause the changes to all traced nodes to be written to *behavior-file* in addition to being written to the terminal. Quotes are necessary if the file name has any punctuation in it.

The RNL directive

```
closeplot
```

will terminate the behavior file. If the entire *RNL* session is to be recorded *closeplot* is not required, as the file will be terminated when *RNL* exits.

(b) If you are using *SPICE*, a behavior file may be specified as the third positional parameter of the *SPICE* command. Behavior records will be put on this file for all nodes specified on the SPICE *PLOT* directive.

2. Use the F (File) function of *simscope* to load the behavior file and get information on the signals stored in it.

   Use *simscope*'s other menu functions to display any signal in the file on any position (vertically) on the screen, change the time scale (window size) and window position. After you have analyzed and positioned your signals, make a hard-copy, if desired.

EXAMPLE (Preparation of a Behaviour File with RNL)

The following example uses *simscope* to display the behavior of a 10 bit counter, cntr10.net, shown here in netlist format:

```
; net file for 10-bit counter

; half adder made from gates
(macro half_adder (a b s c)
  (local h1 h2 h3)
  (nand (h1 2 16) a b)
  (nand (h2 2 16) a h1)
  (nand (h3 2 16) b h1)
  (nand (s 2 16) h2 h3)
  (invert c h1)
)

; one cell of a counter
(macro cell (in out Cin Cout)
  (local c1 c2 c3)
  (invert c1 in)
  (trans phi1 c1 c2)
  (invert c3 c2)
  (half_adder c3 Cin out Cout)
  (trans phi2 out in)
)

; declare global node names
(node count c in out phi1 phi2)

; carry-in to first significant bit controls counting action
(connect count c.0)

; generate the counter
(repeat i 1 10
  (capacitance out.i 1.234)
```

```
    (cell in.i out.i c.(1- i) c.i)
  )
```

The RNL control file, cntr10.l, is as follows:

```
  ; RNL initialization file for 10 bit ripple-carry counter

    (load "uwstd.l")
    (load "uwsim.l")

    (read-network "cntr10")

  ; (setq report-form nil)  This turns off the report generator

    (setq incr 1000)

  ; bind symbols to node names

    (chflag '(phi1 phi2 out.10 out.9 out.8 out.7 out.6
                  out.5  out.4 out.3 out.2 out.1))

    (defun init (dummy)

      (l '(count in.1 in.2 in.3 in.4 in.5
              in.6 in.7 in.8 in.9 in.10))

      (l '(phi2))
      (h '(phi1))

      (step incr)
      (l '(phi1))
      (step incr)

      (x '(in.1 in.2 in.3 in.4 in.5
          in.6 in.7 in.8 in.9 in.10))

      (h '(phi2))
      (step incr)

      (l '(phi2))
      (step incr)

      (h '(count))

      (wr-report)

      'done
    )


    (defvec '(bit bout out.10 out.9 out.8 out.7 out.6
                  out.5  out.4 out.3 out.2 out.1))
```

```
(defvec '(dec dout out.10 out.9 out.8 out.7 out.6
          out.5  out.4 out.3 out.2 out.1))

(def-report '("10 bit counter current state" newline " "
        count (vec bout) (vec dout)))
```

Generate the behavior for the counter using *RNL*

```
netlist cntr10.net cntr10.sim
presim cntr10.sim cntr10
RNL cntr10.1

init                          # initialize the counter

openplot "cntr10.beh"         # open the behavior file

                              # (.beh stands for network behavior file)

c 30                          # run 30 clocks

exit                          # exit RNL
```

SEE ALSO

RNL(1.vlsi) SPICE(1.vlsi), mtp(1.vlsi)

SIMSCOPE User's Guide Release 2.0 Available from the UW/NW VLSI Consortium, Sieg Hall, FR-35, University of Washington, Seattle, WA 98195

"User's Guide to RNL" "VLSI Design Tools Reference Manual", UW/NW VLSI Consortium, University of Washington, (Christopher Terman, MIT Laboratory for Computer Science).

"SPICE User's Guide", "VLSI Design Tools Reference Manual," UW/NW VLSI Consortium, University of Washington, (A. Vladimirescu *et al.*, 15 Oct. 1980)

RESTRICTIONS

1. In graphics mode, which is obviously required for simscope, the GP-19 can display upper case characters only. *simscope* Rstill recognizes, and processes correctly, lower case characters. You have to know which characters in your file and which node names are upper case, and which are lower case, and enter them accordingly. Otherwise, *simscope* may tell you that it does not know the name you entered. The Tek 4105 terminal does not have this problem.
2. The File function does not recognize the ~ (tilde) as part of a path name.
3. RNL and SPICE write only changes of signal levels to the behavior file. Therefore, a signal's value before the first file entry is not known. *simscope*'s strategy to deal with this situation is to display this value as indeterminate (0.5, X in RNL).
4. The number of signals in a behavior file is limited to 20 (17).

5. The length of behavior files is restricted to 20,000 signal changes (i.e. to 20,000 lines). This could be extended easily, if need be.

BUGS

1. In case the number of signals in a behavior file is greater than 20 or the behavior file contains more than 20,000 signal changes (i.e. to 20,000 lines), *simscope* crashes.
2. The first line of a behaviour file is discarded, because in the case of RNL behaviour files this line contains irrelevant information different from the information of all other lines. Therefore, the very first signal change of a SPICE behaviour file is lost. This is not noticable in most cases, however.
3. Some versions of SPICE produce behaviour files that contain floating point numbers formatted in a non-standard manner. Encountering of a non-standard format in the behaviour file causes *simscope* to crash. A typical case is that a number like "0.000e-7" is given as "0. e-7". *simscope* recognizes the first format only. The behaviour file can be mended easily by using an editor to globally replace ". e" by ".000e".

These bugs will be removed with the next release of *simscope*.

AUTHOR

Rudolf W. Nottrott (UW/NW VLSI CONSORTIUM)

NAME

spcpp – Spice (circuit simulator) input pre-processor

SYNOPSIS

spcpp [**–c**] [**–s** *n*] [**–d** *lr*] [**–t** *tname*] [**–o** *oname*] *iname*

DESCRIPTION

Spcpp is a program that translates bracketed text tokens in an input file into other text strings. It is intended to allow users of *spice* to prepare their simulation input using mnemonic node names rather than the numeric node numbers required by Spice commands. The program has two major modes of operation. If the user does not specify a file that contains a translation table, then *spcpp* builds a translation table itself numbering the tokens from zero as it encounters them. Alternatively, the user can specify the name of a file containing a translation table to be used. In particular, the **.names** file created by *sim2spice* is usable as a translation table file.

The options and parameters are:

–c Indicates that the first non-whitespace word of each line of the translation table file should be skipped over. This is useful if your translation table has an asterisk ('*') in column 1 of each line to allow it to be read by *spice* as comments.

–s *n* Indicates that *n* lines at the beginning of the translation table file should be skipped over. If no number is specified then only the first line of the file is skipped.

–d *lr* Redefines the token delimiters to be '*l*' and '*r*' respectively. The default delimiters are '<' and '>'.

–t *tname* Specifies a file that contains a translation table (default is to build a translation table as described above). Each line of this file should have at least two non-whitespace words on it. If the **–c** option is specified then the first word on each line is ignored. The next word is interpreted as a string to be translated and following one is interpreted as the target string into which it is translated. Any subsequent words on the line are ignored. For Spice input preparation the target string should be a numeral. The **–s** option allows the file to be prefaced by one or more lines that *spcpp* will ignore.

–o *oname* Specifies a file into which the output is to be written. If this option is not used then the output is written to *iroot*.**spcx** where *iroot* is obtained by stripping away any tags from *iname*.

iname Specifies the name of the file to process.

A bracketed token is defined to be a left delimiter character, zero or more spaces, a word (the token) not containing either right or left delimiters, zero or more spaces, and a right delimiter character. Unmatched delimiter characters are not allowed in any context. Bracketed tokens are not allowed to span lines. Tokens and the strings that they translate into are limited to be at most 40 characters each.

Any line that contains no bracketed tokens is simply copied from the input to the output. If a line does contain a bracketed token then the input line is written into the output a Spice comment line. An output line follows immediately. If the line is valid, then the output line has the untranslated parts immediately below the corresponding parts of the commented input line with the target strings substituted for the bracketed tokens. If an error is detected, then the output line has a caret ("^") immediately below the point at which the first error is detected. An error message line then follows. Since the scanning of the line is abandoned there may be subsequent undetected errors in the remaining part of the line.

Example:

If the following lines are contained in the translation table file:

```
Vdd 1
Input 55
Output 107
foo 23
bar 45
```

then *spcpp* will, upon seeing the lines:

```
.plot trans v(<Input>) v(<Output>), i(<Vdd>)
+ v(<foo>), v(<bar>)
```

will output the lines:

```
* .plot trans v(<Input>) v(<Output>) v(<Vdd>)
  .plot trans v(55) v(107) v(1)
* + v(<foo>), v(<bar>)
  + v(23), v(45)
```

Note that *spcpp* correctly handles Spice continuation cards.

Note also that the substitution process is not recursive. That is, once a token has been translated, the translated string is not rescanned.

The usefulness of *spcpp* for simulating a circuit extracted from a layout depends upon the user being able to ensure that his mnemonic node labels will be retained through the extraction process. The *mextra* and *sim2spice* manual entries will help with this.

Pspice is a shell script that runs *sim2spice* and *spcpp* and concatenates several files is useful for preparing Spice inputs from .sim files.

FILES

iname
iroot.spcx
oname
tname

SEE ALSO

mextra(1.vlsi), pspice(1.vlsi), sim2spice(1.vlsi), simtools(1.vlsi), spice(1.vlsi),

SPICE User's Guide, **VLSI Design Tools Reference Manual**, UW/NW VLSI Consortium, University of Washington, (*SPICE Version 2G6 User's Guide*, A. Vladimirescu *et al.*, 15 October 1980)

AUTHOR

Robert Fowler (UW/NW VLSI Consortium, University of Washington)

DIAGNOSTICS

The error messages are intended to be self explanatory. If *spcpp* encounters a syntax error on a line then it suspends processing on that line and writes it as a Spice comment to the output file. It then writes a line containing a caret ("^") under the character at which scanning failed and finally, a line containing an error message. It then goes on to process the remaining lines of the file. If errors have been encountered then at the end of the output file *spcpp* writes messages to the effect that errors have been encountered and exits with status 1. The error

messages written to the output file begin with dollar signs. In addition, some number of messages are directed towards the standard error output.

BUGS

The target strings are not checked to see whether they are valid numerals or not. This can be regarded as either a bug or a feature.

The target string must fit into the space from the left to right token delimiter inclusive. This is normally not a problem since most node numbers will be small integers and the available space will be at least three characters. This was done so that the input lines and the translated outputs would line up vertically.

NAME

spice – circuit simulator

SYNOPSIS

spice *infile outfile* [*mtpfile*]

DESCRIPTION

Spice reads a circuit description from *infile*. Output is written to *outfile*. and error messages to standard error. An optional output file, *mtpfile*, can be used by *mtp* to obtain a multiple time series plot on a Printronix.

Spice calls *spice2g6*, a general-purpose circuit simulation program for nonlinear DC, nonlinear transient, and linear AC analyses. Circuits may contain resistors, capacitors, inductors, mutual inductors, independent voltage and current sources, four types of dependent sources, transmission lines, and the four most common semiconductor devices: diodes, BJTs, JFETs, and MOSFETs.

Spice2g6 has built-in models for the semiconductor devices, and the user need specify only the pertinent model parameter values. The model for the BJT is based on the integral charge model of Gummel and Poon; however, if the Gummel-Poon parameters are not specified, the model reduces to the simpler Ebers-Moll model. In either case, charge storage effects, ohmic resistances, and a current-dependent output conductance may be included. The diode model can be used for either junction diodes or Schottky barrier diodes. The JFET model is based on the FET model of Shichman and Hodges. Three MOSFET models are implemented; MOS1 is described by a square-law I-V characteristic, MOS2 is an analytical model while MOS3 is a semi-empirical model. Both MOS2 and MOS3 include second-order effects such as channel length modulation, subthreshold conduction, scattering limited velocity saturation, small size effects and charge-controlled capacitances.

To build a Spice input file for your circuit from *mextra* output run *sim2spice* or *pspice*.

SEE ALSO

mextra(1.vlsi)
mtp(1.vlsi), sim2spice(1.vlsi), pspice(1.vlsi), spcpp(1.vlsi)

SPICE User's Guide, **VLSI Design Tools Reference Manual**, UW/NW VLSI Consortium, University of Washington, (*SPICE Version 2G6 User's Guide*, A. Vladimirescu *et al.*, 15 October 1980).

Program Reference for Spice2, E. Cohen, ERL Memo. ERL-M592, Electronics Research Laboratory, University of California, Berkeley, June 1976.

SPICE2: A Computer Program to Simulate Semiconductor Circuits, L.W. Nagel, ERL Memo. ERL-M520, Electronics Research Laboratory, University of California, Berkeley, May 1975.

The Simulation of MOS Integrated Circuit Using SPICE2 A. Vladimirescu and Sally Liu, UCB/ERL M80/7, University of California, Berkeley, February 1980.

AUTHOR

(UCB)

BUGS

MOSFET Model, Level=2 does not work, due to a charge conservation problem (it grows).

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.

isocmos
Complies with GTE 5 micron, isocmos process (lambda = 2.5 microns). Inputs and outputs on same side of PLA. Fabricated and tested.

CS3cis Complies with MOSIS 3 micron bulk CMOS process (lambda = 1.0 microns). Berkeley design, simulated but not fabricated. Inputs and outputs on same side of the PLA.

CS3tran
Same as **CS3cis** except inputs and outputs on opposite sides of the PLA.

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.

-S *num* Stretch power and ground lines by *num* lambda.

-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.

If the template CS3cis or CS3tran is used with an odd number of minterms and the -G option is used, there will be design rule violations; extra pieces of P+ implant along the bottom of the OR plane, which will have to be manually removed.

The templates Tcis and Ttrans imply a technology (fnmos) not supplied on the Berkeley tape. These templates will not be useful unless the associated technology files are obtained.

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

NAME

vic – view an integrated circuit layout (version 2.1).

SYNOPSIS

vic [*options*] *symbolname*

DESCRIPTION

Vic is an interactive graphics display program for integrated circuits that is technology independent and has a built–in hardcopy feature. It understands layouts in Caesar data base format. It currently displays only on the GP19 and Tek4105 graphics terminals, but it can produce a hardcopy on a number of devices.

The *options* are as follows:

–t *technology*

Supported values of *technology* are *nmos* and **cmos-pw**. Default is **cmos-pw**.

–h *plotter*

Supported values for *plotter* are **HP7221CT, HP7221AB, HP7580, HP7580B** and **Tek4662_31**. Default is HP7580.

–g *graphics*

Supported values for *graphics* are **GP19** and **Tek4105**. Default is GP19.

–f *format*

The only Choice for *format* of symbol to be read is **ca** (Caesar files).

COMMANDS

For all the commands, only the portion enclosed in parentheses need be typed and a list of the possible parameters for each command (if any) and current values are shown after the command in the menu.

(lay)ers < list>

sets the layers to be plotted. The list consists of layer names seperated by spaces, or the entire list may be preceded by a "+". In the latter case, the given layers are added to the plot ALREADY on the screen (it should be pointed out that a space must follow immediately after the "+", followed by the additional layers). A null list sets all layers to be plotted. Abbreviations are allowed. The first layer with the abbreviation as its leading part will be selected. (Thus, metal can be abbreviated m, me, met or meta, whereas metal2 will require the entire name. Warning: layer names such as metal2 and cut2 must, therefore, follow metal and cut, respectively in the technology files.) Default is all layers.

(n)esting levels < number>

set the number of levels in the symbol's hierarchy to be plotted. Any symbol at a level greater than this will show up only as a bounding box with its symbol name in the lower left corner. The current symbol is at level 1, its children are at level 2, and so on. Default is 1.

(w)indow

window in on the plot. Use the graphics cursor to move to the desired lower left corner of the window and hit the space bar. Then move to the upper right corner and do the same.

(hard)copy

produce a hardcopy of exactly what is shown on the terminal screen on a pen plotter. A grid may be placed over the hardcopy by specifying anything greater than zero when the program prompts for grid size. For this option to work, the user's terminal must communicate with the host through the plotter, in order that the plotter may intercept the plotting commands. For the Tek4105, the grid must be displayed prior to

hard copying.

(lab)els < value>
turn node labels on/off. Default is on.

(p)lot plot on the graphics terminal with the current options in effect.

(v)iew view on the graphics terminal the current symbol fully instantiated with all layers and node labels.

(g)raphics
return to the graphics screen (Tek4105 only).

(gri)d put a grid on the graphics screen (Tek4105 only).

(he)lp show the menu.

(e)xplain
explain each command.

(q)uit quit from the program.

(s)ymbol < name>
select the symbol to be plotted. The only symbols that can be specified are those in the sub-hierarchy of the top level symbol on the command line. Note that this is not a facility to reinitialize the vic with a new symbol. Executing this command with no name causes the list of symbols to be displayed. Default is the highest level symbol.

< control> C
causes current operation in progress to cease. On the Tek4105, to terminate a hard-copy in progress, depress the < shift> cancel key on the keyboard and type a carriage return.

DIAGNOSTICS

If an error occurs, a message is written to standard error and the program exits with a non-zero status.

FILES

technology.tec2
technology.cmp
symbol.att
symbol.ca

SEE ALSO

caesar(CAD1), tec(5.vlsi)

AUTHORS

Pat Bates
Larry McMurchie
Wayne E. Winder
Bruce A. Yanagida

Coordinate Free LAP Reference Manual
Version 1.0

Contents

1 Introduction 2
- 1.1 Overview . . . 2

2 Entities 3
- 2.1 Primitives . . . 3
- 2.2 Symbols . . . 3
- 2.3 Symbolic Points . . . 3
- 2.4 Borders . . . 5

3 Operators 6
- 3.1 Alignment operators . . . 6
- 3.2 Center to center alignment . . . 6
- 3.3 Center line alignment . . . 7
- 3.4 Common edge alignment . . . 7
- 3.5 Border alignment . . . 8
- 3.6 Point alignment . . . 8
- 3.7 Origin to origin alignment . . . 9
- 3.8 Linear transformations . . . 9
- 3.9 Array constructors . . . 10
- 3.10 Tiling operators . . . 10
- 3.11 Library access . . . 11
- 3.12 Symbol union . . . 12
- 3.13 Initialization and termination . . . 13
- 3.14 Garbage collection . . . 13
- 3.15 Setting CFL Attributes . . . 13
- 3.16 Floating argument conversion . . . 14

4 Routers 15
- 4.1 Planar routers . . . 16
- 4.2 Non-planar routers . . . 17
- 4.3 Routing to Library Symbols . . . 18

5 Macros 20

6 Wire Facility 21

7 Running CFL 22
- 7.1 Diagnostic facilities . . . 22
- 7.2 Reminders . . . 23

8 Appendix 24

1 Introduction

This document provides a general description of Coordinate Free LAP (CFL). This first section summarizes the system's capabilities. The next section, Entities, describes the objects which are manipulated by CFL. The third section, Operators, describes the CFL operators by functional group. Chapter 4 describes the routers. Chapters 5 and 6 describe CFL macros and the wire facility. Chapter 7 describes the access to the CFL library and other procedural issues related to development of CFL applications. The Appendix provides an alphabetic listing of all of the CFL routines, with their calling sequences, grouped into categories which correspond to the chapters of this manual.

1.1 Overview

CFL is a library of subroutines written in C intended to facilitate the construction of VLSI circuit layouts. The system is organized algebraically in that there is a data type called SYMBOL, a set of primitive operands of this type, and a set of operators which generate new SYMBOLs by forming combinations of existing SYMBOLs.

The system has a very small set of geometric primitives which may combined to make objects. There is a somewhat larger set of non-primitive objects called macros, which may be used to generate frequently used structures such as contacts. Routing facilities are provided which generate a variety of planar and non-planar wiring patterns used to connect functional blocks. Additionally, there is a coordinate dependent facility called wire for generating arbitrary configurations of material.

The system has been designed to operate in conjunction with Caesar in that it is capable of both accessing symbols generated using Caesar and writing symbols which may be accessed by Caesar. Although CFL has sufficient functionality to allow definitions to be developed for all artwork including the lower level cells in a design, it is intended to be used more in the mode of chip assembly. Hence the typical application would involve using Ceasar to generate lower level cells or tiles and then using CFL facilities to assemble these leaf cells into higher level modules.

To insure that a wide variety of assembly situations can be accomodated, CFL includes approximately 70 variants of operators for juxtaposing, transforming, and replicating heirarchies of symbols. The syntax of these operators is quite compact since generated symbols are simply stored in program variables of type SYMBOL *.

All of the calculations which support the operators of CFL are performed from descriptions of the borders of the symbols. The information in the border descriptions includes the bounding box and lists of coordinates of the points where each kind of material in the symbol makes contact with the bounding box. If a border description is available for a particular symbol, CFL will not require access to any of the rest of the geometry of symbol.

The system will automatically generate border descriptions from the geometry whenever the need arises but it will also automatically save them on disk when library symbols are written out. In this way, modules which have a large number of rectangles may be accessed from the library without the need of reading all of the files associated with their sub-modules. This capability allows CFL to assemble large blocks of circuitry extermely quickly.

CFL provides automatic heirarchy compression when symbols are written to disk so that only those symbols which represent meaningful functional groups need be saved.

2 Entities

The operations provided by CFL are defined with respect to a number of basic entities. These entities include primitive geometric objects and compound objects, called symbols; the boundaries of these symbols, called borders; and individual symbolic points along these boundaries.

2.1 Primitives

CFL has the following two primitive symbols -

```
box(layer,dx,dy)        - box

label(name,dx,dy,pos)   - rectangular label
```

These are the same two primitives used by Caesar. All coordinates are dimensionless. **box** creates a box on the specified layer with dimensions **dx** and **dy**. **label** creates a Caesar label. The label has associated with it a rectangle with dimensions **dx** and **dy** and a name. This rectangle and name form the graphic for the label displayed by Caesar. **pos** is used to specify the position of the name of the label relative to its center. **label** does not place any material on a mask layer and, from the point of view of CFL operators, labels have no extent. That is, the bounding box of a label is always |(0,0),(0,0| independent of the size of its associated graphic rectangle.

2.2 Symbols

A CFL symbol is either a primitive object or an object formed by combining primitive objects and other symbols using the CFL operators. Each symbol is a collection of geometry (boxes), calls to other symbols (calls) and labels. CFL represents symbols internally as data structures having lists of boxes, calls, and labels and all references to symbols within a CFL application program are made through pointers to these structures. The pointers are declared with the declarator SYMBOL*.

For example,

```
SYMBOL *box1,*box2,*cross1,*pair1;

box1 = box("metal", 3,10);      /* vertical bar   */
box2 = box("metal",10, 3);      /* horizontal bar */

cross1 = cc(box1,box2);         /* metal cross    */
pair1  = cx(cross1,cross1);     /* two adjacent crosses */
```

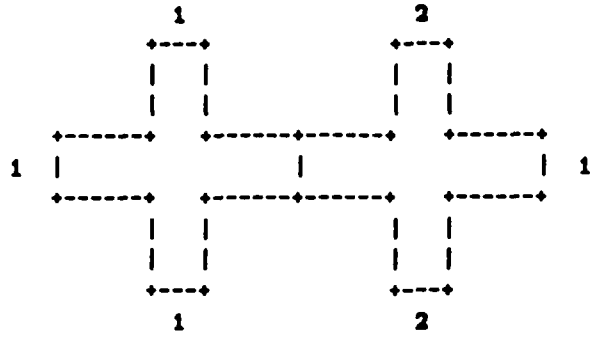
In this example, **cc** is the center to center alignment operator of CFL. It creates a new object by juxtaposing the center of the vertical bar and the center of the horizontal bar. The operator **cx** constructs a horizontal pair of crosses, aligned by their horizontal center lines, with the right edge of the first cross touching the left edge of the second cross. (All CFL operators are declared SYMBOL * by the include file 'cfl.h' which must be included in CFL application programs.)

2.3 Symbolic Points

For each symbol, CFL maintains a list of coordinates which mark the centers of all intersections of mask layers and the bounding box. These sets of coordinates, called crossings, are maintained seperately for each

mask layer and for each of the four sides of the bounding box. Each crossing may be referred to by specifying its symbol, side of the bounding box, layer and ordinal along the side.

For example, the symbol, pair1, generated above looks some thing like this -



The crossings are given by the following four-tuples -

```
(pair1, "top",   "metal", 1)
(pair1, "top",   "metal", 2)

(pair1, "bot",   "metal", 1)
(pair1, "bot",   "metal", 2)

(pair1, "left",  "metal", 1)

(pair1, "right", "metal", 1)
```

The string literals 'top', 'bot', 'left', and 'right' are used by CFL to indicate the sides of bounding boxes. Layernames like 'metal' are, of course, technology dependent. For each technology, CFL uses the long format Caesar layer names. All crossing ordinals start at 1 and increase along the coordinate corresponding to the bounding box edge in question. Top and bottom ordinals increase from left to right. Left and right ordinals increase bottom to top.

Several of the routing operators in CFL have symbolic points as arguments. These arguments are declared to be of type PT * and are generated by the symbolic point descriptor constructor, pt. For example, to construct symbolic points which refer to the leftmost and rightmost metal crossings in pair1 above, the following program statments are used -

```
PT *p1,*p2;

p1 = pt(pair1, "left",  "metal", 1);
p2 = pt(pair1, "right", "metal", 1);
```

2.4 Borders

CFL borders are used as arguments to some of the routers. A border is similar to a symbolic point in that it is referenced through a descriptor declared **BORDER *** and constructed using a constructor, in this case, **bd**. In the simplest case, a border contains all the crossings associated with a given symbol, side and layer. The same convention described above is used for numbering the crossings. Hence,

```
BORDER *b1,*b2;

b1 = bd(pair1, "top", "metal");
b2 = bd(pair1, "bot", "metal");
```

constructs two borders; b1, containing all the metal crossings on the top of pair1, and b2, containing all the metal crossings on the bottom of pair1.

In addition to the basic border constructor which, by default, includes all crossings in its resulting border description, CFL provides operators **bdin** and **bdex** for including and excluding specific crossing ordinals from border descriptions. In general then, the border description facilities are capable of directing the routers to consider any subset of crossings along the side of a particular symbol. For example, the following statements consturct a description of the top of pair1 which includes only the second crossing:

```
b1 = bd(pair1, "top", "metal");
b1 = bdex(b1,1);
```

In the special instance that the ordinal argument is zero, **bdin** will include all crossings in its resulting border and **bdex** will exclude all corssings from its resulting border.

Currently CFL borders are limitted to a maximum of 512 crossings. To obtain the number of crossings currently in a border b:

```
n = nbc(b);
```

3 Operators

CLF has six classes of operators -

```
Alignment operators
Linear transformations
Array constructors
Tiling operators
Library access operators
Miscellaneous operators
```

3.1 Alignment operators

The alignment operators combine a pair of symbols by placing them in one of several relationships with respect to each other. The coordinate free nature of CFL stems largely from the fact that the alignment operators typically specify the position of one symbol relative to another rather than the position of either of them relative to a more global set of coordinates. CFL has six categories of alignment implemented as the following thirteen alignment operators -

```
1. Center to center              cc
2. Center line to center line    cx,cy
3. Edge to edge                  ll,rr,tt,bb
4. Border to border              bx,by
5. Point to point or center      pax,pay,cp
6. Origin to origin              oo
```

Each of these operators has two arguments **s1** and **s2** which are symbol pointers, declared SYMBOL *. The operators form a new symbol containing **s1** and **s2** positioned acccording to the indicated alignment criterion. The position of **s2** relative to **s1** in this new symbol is called the (0,0) position. All of the alignment operators have three additional variations which allow the specification of offsets from this (0,0) position in the x, y, or both directions. The variations are formed by suffixing the operator name with **dx**, **dy**, or **dxy**. For example, the cx operator has the following four forms:

```
cx(s1,s2)          - pair in x, center aligned
cxdx(s1,s2,dx)     - pair in x, center aligned, x offset
cxdy(s1,s2,dy)     - pair in x, center aligned, y offset
cxdxy(s1,s2,dx,dy) - pair in x, center aligned, xy offset
```

3.2 Center to center alignment

The cc operator forms a symbol which consists of **s1** overlayed with **s2**. The symbols are aligned so that the centers of their respective bounding boxes are coincident. This position is the (0,0) position with respect to any offsets supplied by the **dx**, **dy**, and **dxy** variants.

```
cc(s1,s2)          - align center to center
ccdx(s1,s2,dx)     - align center to center, x offset
ccdy(s1,s2,dy)     - align center to center, y offset
ccdxy(s1,s2,dx,dy) - align center to center  xy offset
```

3.3 Center line alignment

The **cx** operator forms the horizontal pair of symbols (**s1**,**s2**). The right side of the bounding box of **s1** is adjacent to the left side of the bounding box of **s2** and the cells are aligned by their horizontal center lines. This position is taken to be the (0,0) position with respect to the offsets which may be supplied using the **dx**, **dy**, and **dxy** variants.

The **cy** operator forms the vertical pair of sumbols (**s1**,**s2**). The top side of the bounding box of **s1** is adjacent to the bottom side of the bounding box of **s2** and the cells are aligned by their vertical center lines. This position is taken to be the (0,0) position with respect to the offsets which may be supplied using the **dx**, **dy**, and **dxy** variants.

```
cx(s1,s2)          -  pair in x, center aligned
cxdx(s1,s2,dx)     -  pair in x, center aligned, x offset
cxdy(s1,s2,dy)     -  pair in x, center aligned, y offset
cxdxy(s1,s2,dx,dy) -  pair in x, center aligned, xy offset

cy(s1,s2)          -  pair in y, center aligned
cydx(s1,s2,dx)     -  pair in y, center aligned, x offset
cydy(s1,s2,dy)     -  pair in y, center aligned, y offset
cydxy(s1,s2,dx,dy) -  pair in y, center aligned, xy offset
```

3.4 Common edge alignment

The operators in this group align a pair of symbols (**s1**,**s2**) so that a specified edge of their bounding boxes lies on the same line. In the (0,0) position, the bounding boxes of the symbols are adjacent. There are four operators, each with offset variants. **s2** is placed to the right of **s1** in the case of the **bb** and **tt** operators. **s2** is placed above **s1** in the case of the **ll** and **rr** operators.

```
bb(s1,s2)          -  align bottom to bottom
bbdx(s1,s2,dx)     -  align bottom to bottom, x  offset
bbdy(s1,s2,dy)     -  align bottom to bottom, y  offset
bbdxy(s1,s2,dx,dy) -  align bottom to bottom  xy offset

ll(s1,s2)          -  align left to left
lldx(s1,s2,dx)     -  align left to left, x  offset
lldy(s1,s2,dy)     -  align left to left, y  offset
lldxy(s1,s2,dx,dy) -  align left to left, xy offset

rr(s1,s2)          -  align right to right
rrdx(s1,s2,dx)     -  align right to right, x  offset
rrdy(s1,s2,dy)     -  align right to right, y  offset
rrdxy(s1,s2,dx,dy) -  align right to right, xy offset

tt(s1,s2)          -  align top to top
ttdx(s1,s2,dx)     -  align top to top, x  offset
ttdy(s1,s2,dy)     -  align top to top, y  offset
ttdxy(s1,s2,dx,dy) -  align top to top, xy offset
```

3.5 Border alignment

The **bx** operator forms the horizontal pair of symbols (**s1**,**s2**). The right side of the bounding box of **s1** is adjacent to the left side of the bounding box of **s2** and the symbols are aligned so that corresponding patterns of material at the common border match up. This position is taken to be the (0,0) position with respect to the offsets which may be supplied using the **dx**, **dy**, and **dxy** variants.

The **by** operator forms the vertical pair of symbols (**s1**,**s2**). The top side of the bounding box of **s1** is adjacent to the bottom side of the bounding box of **s2** and the symbols are aligned so that corresponding patterns of material at the common border match up. This position is taken to be the (0,0) position with respect to the offsets which may be supplied using the **dx**, **dy**, and **dxy** variants.

The algorithm that performs the alignment for **bx** and **by** will use center line alignment if it can not identify the same pattern of material common to the adjacent borders of the symbols and issue a warning message.

```
bx(s1,s2)           -  pair in x, borders aligned
bxdx(s1,s2,dx)      -  pair in x, borders aligned, x offset
bxdy(s1,s2,dy)      -  pair in x, borders aligned, y offset
bxdxy(s1,s2,dx,dy)  -  pair in x, borders aligned, xy offset

by(s1,s2)           -  pair in y, borders aligned
bydx(s1,s2,dx)      -  pair in y, borders aligned, x offset
bydy(s1,s2,dy)      -  pair in y, borders aligned, y offset
bydxy(s1,s2,dx,dy)  -  pair in y, borders aligned, xy offset
```

3.6 Point alignment

The point to point alignment operators are similar to the border alignment operators except that the symbols are aligned so that specific points on the respective borders are adjacent. This allows cases to be handled that do not meet all the conditions necessary for running the automatic alignment algorithm of the border alignment operators.

The **pax** operator forms the horizontal pair of symbols (**s1**,**s2**). In the (0,0) position, the right side of the bounding box of **s1** is adjacent to the left side of the bounding box of **s2** and the symbols are aligned so that the symbolic point **n1** on the right side of **s1** is adjacent to the symbolic point **n2** on the left side of **s2**. Both points are taken to be on the same layer.

The **pay** operator forms the vertical pair of symbols (**s1**,**s2**). In the (0,0) position, the top side of the bounding box of **s1** is adjacent to the bottom side of the bounding box of **s2** and the symbols are aligned so that the symbolic point **n1** on the top of **s1** is adjacent to the symbolic point **n2** on the bottom of **s2**. Both points are taken to be on the same layer.

```
pax(s1,n1,s2,n2,layer)          -  point align in x
paxdx(s1,n1,s2,n2,layer,dx)     -  point align in x, x offset
paxdy(s1,n1,s2,n2,layer,dy)     -  point align in x, y offset
paxdxy(s1,n1,s2,n2,layer,dx,dy) -  point align in x, xy offset

pay(s1,n1,s2,n2,layer)          -  point align in y
paydx(s1,n1,s2,n2,layer,dx)     -  point align in y, x offset
paydy(s1,n1,s2,n2,layer,dy)     -  point align in y, y offset
paydxy(s1,n1,s2,n2,layer,dx,dy) -  point align in y, xy offset
```

The center to point alignment operator is intended primarily for placing labels along the borders of symbols. Typically, the inputs and outputs of a symbol will appear as a series of crossings along the edges of its bounding box and it will be desirable to label these inputs and outputs. Labels of this type are used often for the purpose of simulating an extracted circuit, for example.

The center to point alignment operators have a symbol and a point argument. The result is a pair of symbols in which, in the (0,0) position, the center of the symbol argument is concident with the specified point. In the case of the other alignment operators, offsets refer to the positioning of the second symbol relative to the first. In the case of cp, the offsets refer to the positioning of the symbol relative to the point.

```
cp(s1,p1)                 - center to point
cpdx(s1,p1,dx)            - center to point,  x offset
cpdy(s1,p1,dy)            - center to point,  y offset
cpdxy(s1,p1,dx,dy)        - center to point, xy offset
```

A typical application of cp to label a number of inputs of a symbol t1 would look like the following:

```
t1 = cp(label("in.1",0,0,4),pt(t1,"left","metal",1);
t1 = cp(label("in.2",0,0,4),pt(t1,"left","metal",2);
t1 = cp(label("in.3",0,0,4),pt(t1,"left","metal",3);
t1 = cp(label("in.4",0,0,4),pt(t1,"left","metal",4);
```

3.7 Origin to origin alignment

The oo operator forms a symbol which consists of s1 overlayed with s2. The cells are aligned so that the origins of their respective geometry are coincident. This position is the (0,0) position with respect to any offsets supplied by the dx, dy, and dxy variants. Unlike the other alignment operators, the operation of oo is coordinate dependent. It is a special purpose operator intended to be used primarily in conjunction with the routers to locate generated wiring within its containing cell.

```
oo(s1,s2)             - align origin to origin
oodx(s1,s2,dx)        - align origin to origin, x  offset
oody(s1,s2,dy)        - align origin to origin, y  offset
oodxy(s1,s2,dx,dy)    - align origin to origin, xy offset
```

3.8 Linear transformations

There are three linear transformations -

```
mx(s)             - mirror in x
my(s)             - mirror in y
rot(s,n)          - rotate
```

The argument to rot is in degrees and must be an integer multiple of 90. Note that mx and my mirror in the indicated coordinate as opposed to around the indicated axis. That is, mx operates by replacing x with -x for all x coordinates in its argument.

3.9 Array constructors

There are three array constructors **nx**, **ny**, and **nxy**, which can be used to generate horizontal, vertical, or rectangular arrays of a given symbol. As in the case of the alignment operators, offset variants of these operators are also defined. The interpretation of the offsets is, however, slightly different. **dx** and **dy**, when supplied, are taken to be the spacings between the bounding boxes of successive array elements. The (0,0) position is when the bounding boxes are adjacent. The variants of the array operators which would produce a non-rectangular structure are not defined, for example, **nxdy**.

```
nx(s,n)                 - repeat in x
nxdx(s,n,dx)            - repeat in x, x offset

nxy(s,nx,ny)            - repeat in x and y
nxydx(s,nx,ny,dx)       - repeat in x and y, x  offset
nxydy(s,nx,ny,dy)       - repeat in x and y, y  offset
nxydxy(s,nx,ny,dx,dy)   - repeat in x and y, xy offset

ny(s,n)                 - repeat in y
nydy(s,n,dy)            - repeat in y, y offset
```

There are three additional array constructors **repx**, **repy** and **repxy** which construct arrays of particular spacial periods. The arguments to these routines are also given as **dx** and **dy** but they specify the periods rather than offsets. These operators do not have variants for providing additional offsets.

```
repx(s,n,dx)            - repeat in x with period dx
repxy(s,nx,ny,dx,dy)    - repeat in x and y, with periods dx dy
repy(s,n,dy)            - repeat in y with period dy
```

3.10 Tiling operators

Tiling is similar to an array operation except that each element of the generated array can be a different symbol. There are three tiling operators **vx**, **vy**, and **vxy**, which can be used to generate horizontal, vertical, or rectangular tilings. These opertors are similar to the array operators except that the first argument is an array of symbol pointers rather than a single symbol pointer. The tiling operators, then, operate on vectors of symbols so their mnemonic starts with **v**. There are no offset variants for the tiling operators since the offset for each tile could potentially be different. All symbols in a row are arranged so that their bounding boxes are in contact and aligned by their center lines.

```
vx(s,n)                 - vector in x
vxy(s,nx,ny)            - vector in x and y
vy(s,n)                 - vector in y
```

vxy generates a plane of the symbols **s** aranged so that their bounding boxes are in contact. All symbols in the same row of the plane are assumed to have the same height. **s** is treated as though it is dimensioned **s[ny][nx]** in the calling program.

3.11 Library access

There are two operations which access library sybmols -

```
gs(name)          - get library symbol
gsp(name)         - get library symbol with prefix
ps(name,s)        - put symbol in the symbol table
psp(name,s)       - put symbol in the symbol table with prefix
```

gs will read a library symbol in Caesar format, place the symbol in the data base and return a pointer to it. If the symbol is already in the data base, **gs** simply returns the pointer, that is, it will read the symbol only once.

ps compresses the heirarchy below its argument and marks that argument as a library symbol. **cflstop** will cause all library symbols to be written to the disk in the Caesar format.

The heirarchy compressor removes from the heirarchy all cells which are not marked as library cells, that is, cells which were not read in with **gs** or cells which have not been marked as permanent by a call to **ps**. Therefore, **ps** can be used to not only to save symbols but also to control the actual structure of the heirarchy.

Currently, CFL does not support any search path for accessing library symbols. All symbols are read from the sub-directory ./ca and all output symbols are written into ./ca. Ceasar or CFL symbols from global cell libraries must be copied into ./ca before they can be accessed.

gsp is identical to **gs** except that the CFL attribute 'prefix' is prepended to the name, before the symbol is referenced. **psp** is identical to **ps** except that the CFL attribute 'prefix' is prepended to the name when the symbol is made permanent. The file on which the symbol is written will also have a name which includes the prefix.

gsp and **psp** are intended for use with generators, such as a ROM or PLA generator, in which several instances of the generated circuit may be parts of a single larger circuit. Since errors will generally result if the sub-cells of the various generated instances are not kept distinct, CFL provides a global symbol name prefix as a way of simplifying the construction of names which include an instance indication as a prefix.

The following call sets the prefix attribute to the value ROM1 -

```
cflsetc("prefix","ROM1");
```

Following this call, the call

```
s1 = psp("s1",s1);
```

will make **s1** a permanent symbol with the name ROM1s1. ROM1s1.ca will be the name of the file containing this symbol on exit.

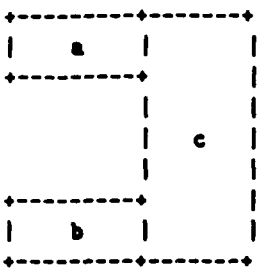
The remaining sections describe operators and routines do not fit in any of the above categories.

3.12 Symbol union

```
su(s1,s2)   -  symbol union
```

The **su** operator performs symbol union. This operation merges two symbols and includes only one instance of common structure. This operation is useful in cases where it is necessary to align more than one symbol to the same border of a given symbol.

For example, suppose it is desired to align the symbols **a**, **b**, and **c** as follows:



If **a** and **c** are combined with **tt**, the bounding box of the result includes both **a** and **c**, so **b** can not be easily combined with this result. The symetric situation occurrs if a pair is made of **b** and **c** using **bb**. The difficulty is overcome with **su** in the following manner:

```
r = su(tt(a,c),bb(b,c))
```

A pair is made of **a** and **c** aligned top to top and a pair is made of **b** and **c** aligned bottom to bottom. The result, **r**, is the union of the **ac** pair and the **bc** pair and includes only a single instance of **c**.

It should be stressed that the implementation of **su** is less general than its name implies. It is intended for application in a commonly occurring although restricted circumstance in which **s1** and **s2** consist only of calls and two of these calls, one in **s1** and one in **s2** refer to the same symbol.

The limitiations of **su** may be best presented by outlining its algorithm. **su** overlays **s1** and **s2** so that common sub-cells appear to coincide. Then, only one instance of the common sub-cells is included in the result symbol. The method begins by searching **s1** and **s2** for a common sub-cell to use as an alignment key for super-position. The sub-cell must be called by both **s1** and **s2** and its transform must be involve only translation in both cases. The search finds the first symbol on the call list of **s1** which is also called by **s2** and which meets these criteria. (To qualify for the union operation **s1** and **s2** must contain only calls - no boxes and no labels.)

Next a pair of transforms is generated, **t1** for the **s1** instance and **t2** for the **s2** instance, which will translate the alignment key to the origins of its respective containers. Since the transforms in the calls are strictly translations, the inverses can be constructed by simply negating the displacement elements of the translation matrices.

Finally, the union is generated by copying the call list of **s1** to the call list of the result applying **t1** to all calls. During this operation, a set of the symbols called is constructed to be used when selecting sub-cells of **s2** to include in the union. Next, the call list of **s2** is copied to the call list of the result excluding all cells which are already there since they were called by **s1**. **t2** is applied to any cell included from **s2**.

Recall that **t1** translates the key cell to the origin of **s1** and also that **t2** translates the key cell to the origin of **s2**. Since both of these origins have the coordinates (0,0) the alignment keys would exactly coincide in the result. Generalizing this, **su** assumes that any instance of a cell called by **s2** which is also called by **s1**, were it to be included in the result, would exactly overlay some instance already present in the result by virtue of a call in **s1**. As a consequence of this assumption, **su** will not work correctly in cases in which there are sub-cells called by both **s1** and **s2** which are not intended to be coincident in the result.

3.13 Initialization and termination

```
cflstart(technology)      - initialize cfl
cflstop()                 - terminate  cfl
```

CFL can be used with any of the various technologies currently supported by the design system. It obtains its table of layer names from the **.tec2** technology files in the PLAP path. For MOSIS bulk CMOS the technology name is 'cmos-pw' and for NMOS the name is simply 'nmos'.

cflstart initializes the package. It must be called before invoking any other CFL functions. **cflstop** causes all permanent symbols to be written to disk and should be called just prior to exiting a CFL application.

3.14 Garbage collection

```
cflcollect()              - release temporary symbols
```

CFL does not provide automatic garbage collection - symbols, once created, are retained indefinitely. In most instances this will not cause a problem since the number of symbols generated by typical CFL applications will not be very large. For some applications however looping constructs can generate a substantial number of 'intermediate' symbols which will cause a memory overflow. (In practice this condition occurrs around 50K symbols.)

When **cflcollect** is called by the application, all space associated with temporary symbols is released. Temporary symbols are those which were not read in from the library or which have not been saved with a call to **ps**. After calling **cflcollect** an application must not attempt to reference the values of any of its SYMBOL * variables which were pointing to temporary symbols before the call. The pointers are not modified by **cflcollect** but the storage they point to will have been returned to the system.

A typical application requiring **cflcollect** is one in which a looping construct must be used a number of times to build up a set of larger structures. After each structure is generated it is saved by calling **ps**. **cflcollect** is then called to purge the temporaries from memory and the next structure is generated and saved.

3.15 Setting CFL Attributes

```
cflset(a,v)               - set CFL integer attribute
cflsetc(a,v)              - set CFL character attribute
```

These routines provide a mechanism for applications to modify some of the parameters which control the operation of CFL. Currently only two of these parameters are implemented - the integer attribute 'grain' and the character string attribute 'prefix'. 'grain' determines the smallest resolvable unit of distance and 'prefix' sets the prefix to be used with the **gsp** and **psp** operators. The default grain is 2 which means that

all CFL dimensions are multiplied by 2 when they are stored internally and written to output files. A setting of 2 makes CFL's .ca files compatible with Caesar's .ca files. (grain should always be a multiple of 2.) The default prefix is empty.

3.16 Floating argument conversion

All CFL routines which have dimensions or displacements as arguments expect that these arguments will be given as integers. However, internally, CFL scales these integers so that it is possible to represent fractional values, say 1.5. Floating point numbers may be used as dimension or displacement arguments to all routines by first applying the conversion routine -

```
f(r)            - convert floating argument
```

For example,

```
s3 = cxdx(s1,s2,f(1.5));
```

4 Routers

CFL does not currently provide high level routing facilities such as a general channel router or switchbox router. Rather, the CFL routers consist of a set of wiring pattern generators each of which is specialized to a particular routing situation. These routers, which are designed to be used in combination with each other and the other CFL operators, support a set of elementary routing operations from which more sophisticated patterns may be constructed.

There are two types of routing facilities available in CFL, planar routers and non-planar routers.

The planar routers are -

```
pp   - point to point router
pr   - general planar router
ext  - border extender
fill - Caesar fill operation
```

and the non-planar routers are -

```
plx  - horizontal point to line router
ply  - vertical   point to line router

elb  - general elbow
tee  - tee
```

Generally speaking the planar routing facilities of CFL are technology independent whereas the non-planar routing facilities are tecnhnology dependent since contacts must be specified.

Since CFL is coordinate free, the routers operate from border descriptions and from symbolic point designations. The generation of symbolic point and border descriptors is described in Chapter 2.

Most CFL operators produce a new symbol by combining existing symbols. The arguments to these operators have no particular spacial relationship to each other before the operation takes place. The routers, on the other hand, rather than combining symbols, must form connections between them. This process requires that the symbols to be connected have a previously established fixed spatial relationship.

Symbols acquire a fixed spatial relationship as soon as they become constituents of some higher level symbol. CFL refers to a higher level symbol, **s0**, which contains symbols **s1** and **s2** as a container of **s1** and **s2**. Within any container, the relative positions of **s1** and **s2** are fixed.

The routers, like all other CFL operators, are SYMBOL * valued functions. When a router is invoked it produces a pattern of wiring as its result. This pattern of wiring is not 'written' into place directly by the routing operation, rather it is a seperate symbol in its own right. Therefore to connect two symbols using the routers, two steps are necessary:

1. Use one of the routers to generate the pattern of wiring necessary to form the desired connections.

2. Use the origin to origin alignment operator to locate the generated wiring pattern in the container so that the intended connections are made.

In all cases, the wiring is generated in the coordinate system of the container and often the two steps above may be combined using a statement of the following form -

```
result = oo(container,router(container,...));
```

The rationale for having the generation and placement of wiring patterns performed as separate steps rather than as an atomic operation is that in many cases routing problems require the generation of complex patterns in which wiring generated by one call to a router must itself be connected to the wiring generated by another call to a router. Separating the generation allows the generated wiring to become a separate symbol which may be then operated on using any CFL operator.

4.1 Planar routers

```
pp(s0,p1,p2,w)    -  point to point router
```

The point to point router generates a single wire of width **w** for connecting symbolic point **p1** with symbolic point **p2**. The layer of **p1** must match the layer of **p2**. **p1** and **p2** must be uniquely locatable within the containing symbol, **s0**.

```
pr(s0,b1,b2,w)    -  planar router
```

The planar router generates a planar wiring pattern for connecting the points specified by the border descriptor **b1** to the points specified by the border descriptor **b2**. The borders must be on the same layer and contain the same number of points. Recall that the functions **bdln** and **bdex** may be used to obtain border descriptors for any subset of the crossings along a border of a symbol. The symbols specified by the descriptors **b1** and **b2** must be uniquely locatable within the containing symbol **s0**. All wires will have a width of **w**.

To simplify the diagnostic process, pr will construct wiring patterns whether or not there is sufficient space for the number of wires requested. It will, however, issue a warning message if any of the generated wires are closer than the CFL parameter minsep. Also, it will insure that runs which are parallel to edges of the bounding boxes of the symbols specified by **b1** and **b2** are no closer than the CFL parameter **arcoff**. **minsep** and **arcoff** have the default value of 4 which is sufficient for most situations. They may be modified at any time with the following call -

```
preset(minsep,arcoff);
```

```
ext(b,d,w)        -  border extender
```

ext generates a pattern of wiring for extending all points in the border **b** perpendicularly for a distance of **d** using wires of width **w**. In the special instance that **w** is zero, **ext** will use the widths of the crossings in the border. Typically these wiaths will not all be the same.

```
fill(s,side,d)    -  Caesar fill
```

fill is similar to **ext** except that all layers crossing the indicated side are extended a distance **d**. The extensions have the same widths as the crossings. As in the case of the border descriptor constructors, the side argument is one of the text strings 'top', 'bot', 'left', 'right'.

For example, suppose it is desired to generate a symbol **s2** which consists of 5 instances of a symbol **s1** placed a distance 10 apart and connected by extending the material on the right side of **s1**. The following code will generate this configuration.

```
s2 = cx(nx(oo(s1,fill(s1,"right",10)),4),s1);
```

The **fill** operator generates a pattern which consists of the layers of the right side of **s1** extended for a distance of 10. **oo** concatenates this fill pattern to the original symbol **s1**. The resulting filled symbol is then repeated 4 times by **nx**. Finally, **cx** is used to place the right most instance of **s1** on the row.

4.2 Non-planar routers

```
plx(s0,p1,p2,w,ct)  -  horizontal point to line router
ply(s0,p1,p2,w,ct)  -  vertical   point to line router
```

plx generates a single wire for connecting the symbolic point **p1** to the vertical line running through the symbolic point **p2**. The connection is made horizontally. The layer of **p1** is taken to be the layer of the wire. **p1** and **p2** must be uniquely locatable in the container symbol, **s0**. The width of the generated segment is **w**. If not NULL, the contact, **ct**, is placed with its origin at the intersection of the vertical line and the generated wire.

ply generates a single wire for connecting the symbolic point **p1** to the horizontal line running through the symbolic point **p2**. The connection is made vertically. The layer of **p1** is taken to be the layer of the wire. **p1** and **p2** must be uniquely locatable in the container symbol, **s0**. The width of the generated segment is **w**. If not NULL, the contact, **ct**, is placed with its origin at the intersection of the horizontal line and the generated wire.

```
elb(s0,b1,b2,w1,w2,rev,ct)   -  non-planar elbow
```

elb generates a wiring pattern for connecting the points in border **b1** with the points in border **b2**. Wires from **b1** will have width **w1**, wires from **b2** will have width **w2**. The pattern generated must form an elbow but it is not necessary that **b1** and **b2** be on the same layer. The contact, **ct**, will be placed with its origin at the intersections of the wires from **b1** and the wires from **b2** if these wires are not on the same layer.

The **rev** parameter may be either TRUE (1), or FALSE (0). If FALSE the connections from **b1** to **b2** will be in the normal order of the borders, that is, low order points in **b1** will connect to low order points in **b2**. If rev is TRUE, the connections will be reversed, that is, low order points in **b1** will connect to high order points in **b2**.

Through combinations of selecting subsets of the borders with **bdln** and **bdex** and utilizing the normal and reverse options, a succession of **elb** invocations may be used to form a set of elbows between **b1** and **b2** which implement any desired ordering of the connections.

tee(s0,b1,b2,w,rev,ct) - tee

tee generates a wiring pattern for connecting the indicated border of the tee connected symbol, **b1**, to the wiring in the transverse routing symbol specified in the border, **b2**. The wiring in the routing symbol is assumed to run perpendicular to the wiring generated for connecting the tee connected symbol. The routing symbol, presumably generated by a prior call to a router, is also assumed to consist strictly of parallel lines, no elbows. All generated wires will have width **w**. The contact, **ct**, will be placed with its origin at the intersection of the generated wires and the wires existing in the routing symbol. The **rev** parameter is set to TRUE if the connection order is to be reversed.

Through combinations of selecting subsets of the borders with **bdin** and **bdex** and utilizing the normal and reverse options, a succession of tee invocations may be used to form a set of tee patterns between **b1** and **b2** which implement any desired ordering of the connections.

All of the non-planar routers have a contact argument. The provsion for positioning this contact in generated routing is coordinate dependent in that the contacts are always positioned so that their origins, coordinate (0,0), coincides with the intersections of wires on different layers. If the contacts are symetric and generated with the CFL **box** primitive, as is the case with the NMOS macros **gb** and **rb**, the origins will be in the geometric centers because the **box** primitive is designed to make boxes which are symetric about the origin whenever possible. If other, asymetric, forms of contacts are needed they may be generated according to the above criterion using the CFL **wire** facility described in Chapter 6.

4.3 Routing to Library Symbols

Use of the routers generally requires that three pointers into a symbol heirarchy be supplied - the container and the two symbols to be connected. When symbols are retrieved from the library using **gs** only one pointer is provided. A typical problem of this form is to retrieve from the library both a complete circuit and a pad frame and then to connect the circuit to the pads. The CFL procedure **locate** may be used to obtain a pointer to any uniquely named sub-symbol within a symbol heirarchy. All symbols saved with **ps** are named symbols.

For example, the following program places an experimental CMOS register called **reg_plus** in a pad frame and connects the pads. The program illustrates a number of the routing facilities of CFL.

The pad frame for this chip was generated using the CMOS pad frame generator described in man pads.

The program begins by reading in the register and pad frame using **gs**. Note that, provided they are available, only the border descriptions are read - not the geometry files.

Since there are connections on all four sides of the register, it is necessary to extend the points where the wires connect somewhat to prevent wiring generated for adjacent sides from colliding. The amount of this extension was determined empirically using Caesar to view the results of the trials.

The pad frame has subcells for its left, right, bottom and top sides called 'reg_pads_l', 'reg_pads_r', 'reg_pads_b', and 'reg_pads_t'. Pointers to these subcells are obtained using **locate** and border descriptors are constructed for their inner-most sides.

Next, border descriptors are constructed for the sides of the register. Note that the SYMBOL **reg** now contains both the original register and the extensions. **ce** is used to place the register in the center of the pad frame forming the SYMBOL **reg_chip**.

The planar router parameters **minset** and **arcoff** are set to 5 using **prset** since the routing is being done on the 'metal2' layer. Then **pr** is used to generate the connections for the four sides.

Finally, **reg_chip**, which now includes the pad frame, the register, and the routing is saved using **ps**. The program in this example, which generates connections to most of the pins on an 84 pin pad frame, executes in about 2 seconds. Also, as long as any modifications to the design of the register do not affect the number and order of its external connections, this same program may be used without modification to assemble the chip.

```
#include <stdio.h>
#include "cfl.h"

main()
{
        SYMBOL *pads, *reg, *reg_chip;

        BORDER *bpl, *bpr, *bpt, *bpb;
        BORDER *brl, *brr, *brt, *brb;

        cflstart("cmos-pw");

/*  Obtain pad frame and register                                   */

        pads = gs("reg_pads");
        reg  = gs("reg_plus");

/*  Extend terminals of the register                                */

        reg = oo(reg,ext(bd(reg, "left",  "metal2"), 1400, 8));
        reg = oo(reg,ext(bd(reg, "right", "metal2"), 1400, 8));
        reg = oo(reg,ext(bd(reg, "bot",   "metal2"), 1400, 8));
        reg = oo(reg,ext(bd(reg, "top",   "metal2"), 1400, 5));

/*  Construct border descriptions for the pad frame                 */

        bpl  = bd(locate(pads,"reg_pads_l"), "right", "metal2");
        bpr  = bd(locate(pads,"reg_pads_r"), "left",  "metal2");
        bpb  = bd(locate(pads,"reg_pads_b"), "top",   "metal2");
        bpt  = bd(locate(pads,"reg_pads_t"), "bot",   "metal2");

/*  Construct border descriptions for the register                  */

        brl = bd(reg, "left",  "metal2");
        brr = bd(reg, "right", "metal2");
        brb = bd(reg, "bot",   "metal2");
        brt = bd(reg, "top",   "metal2");

/*  Place the register in the center of the pad frame               */

        reg_chip = cc(pads,reg);

/*  Connect the register to the pads                                */

        preset(5,5);
        reg_chip = oo(reg_chip, pr(reg_chip, bpl, brl,8));
        reg_chip = oo(reg_chip, pr(reg_chip, bpr, brr,8));
        reg_chip = oo(reg_chip, pr(reg_chip, bpb, brb,8));
        reg_chip = oo(reg_chip, pr(reg_chip, bpt, brt,5));

        ps("reg_chip",reg_chip);

        cflstop();
}
```

5 Macros

CFL has two groups of macros - technology independent macros and technology dependent macros. Currently, all of the technology dependent macros are NMOS. The technology independent macros are -

```
alpha(s,layer,w)          - character string, width w

cross(layer1,dx1,dy1,
      layer2,dx2,dy2)     - two boxes, centers aligned

letter(c,layer,w)         - alphanumeric letter, width w
                            letter
lne(layer,w,dx,dy)        - el, north east
lnw(layer,w,dx,dy)        - el, north west
lse(layer,w,dx,dy)        - el, south east
lsw(layer,w,dx,dy)        - el, south west
```

alpha generates a string of characters which are 5w wide, 8w high with 2w spacing in between. The same rules apply to letter. The character set that is available is

```
A - Z
0 - 9
- . , : ; ? ! / [ ] + =
```

Currently, space (or blank) is not available and neither are lower case letters.

The NMOS macros are -

```
be()              - butting contact, east
bn()              - butting contact, north
bs()              - butting contact, south
bw()              - butting contact, west

gb()              - diffusion - metal contact
rb()              - poly - metal contact

padelne(dx,dy)    - pad frame elbow, north east corner
padelnw(dx,dy)    - pad frame elbow, north west corner
padelse(dx,dy)    - pad frame elbow, south east corner
padelsw(dx,dy)    - pad frame elbow, south west corner
padext(dx)        - pad frame extension

pullup(l,dir)     - pullup, length and direction
```

There are three NMOS pads (input, output and tri-state) that are designed to fit on the pad frame. These pads are not CFL macros but are available as library cells.

The pullup has a minimum gate width.

6 Wire Facility

To provide for the parametric generation of particularly complex leaf cells, or cells with specific coordinate requirements like router contacts, CFL includes the wire facility which allows the use of symbol relative coordinates. Note that the use of this facility can intorduce significant coordinate dependency into a design so it should not in general be used in cases where the relative operators are able to serve. The procedures associated with the wire facility are the following -

```
wire(layer,width)  -  Initialize a wire
at(x0,y0)          -  Move to the point (x0,y0)
dx(dx0)            -  Draw to the point (x+dx0,y)
dy(dy0)            -  Draw to the point (x,y+dy0)
iso(s)             -  Include symbol origin
wl(layer)          -  Reset the wire layer
ww(width)          -  Reset the wire width
x(x0)              -  Draw to the point (x0,y)
y(y0)              -  Draw to the point (x,y0)
```

wire is of type SYMBOL *. All of the procedures apply to the wire generated by the last call to **wire**. Note that the symbol generated by **wire** may contain an arbitrary number of physical 'wires' which need not be connected. The only thing they have in common is their coordinate system.

The procedure **iso** has a symbol as its argument. **iso** includes that symbol positioned so that its origin coincides with the current wire position. Note that the current wire position, or more precisely, the position within the coordinate system of the current wire, is initialized with the **at** procedure and maintained by all move and draw procedures.

7 Running CFL

Currently CFL is installed in $UW_VLSI_TOOLS. It consists of two files, $UW_VLSI_TOOLS/include/cfl.h and $UW_VLSI_TOOLS/lib/libcfl.a. **cfl.h** must be included in any program that intends to use CFL facilities. When compiled this program must be linked with **libcfl.a**. Application programs may achieve access to CFL by including the line

```
#include "cfl.h"
```

in soucre modules which reference CFL facilities. These modules may then be compiled and linked using the following command line:

```
cc module.c $UW_VLSI_TOOLS/lib/libcfl.a  -I$UW_VLSI_TOOLS/include
```

When executed, a CFL program will attempt to reference the sub-directory ./ca of the current directory for both reading and writing symbols. It will not create this directory if it does not exist, so prior to using CFL, this sub-directory must be created.

7.1 Diagnostic facilities

CFL contains a number of internal checks for parameter range and consistency, successful completion of the routers, access to data files, etc. A failure of any of these checks will abort the application program with an error message describing the condition that caused the error and the name of the routine in which the error occurred. As this latter routine will probably be a CFL internal routine not directly called by the application, the CFL error handler will also list the last 8 application level routines called before the error occurred. These routines are listed in the reverse of the order in which they were called.

This information is hopefully sufficient to indicate the last point of successful execution in the application as well as the cause of the error. In cases where more information is needed the the call

```
prtsymbol(s);
```

may be used to output the internal structure corresponding to the symbol **s** on the standard output.

The most frequent types of errors tend to be mismatches between what is thought to be on a border of a symbol and what is actually there. Discrepencies arise easily since a single piece of material which extends beyond the intended border causes all other crossings to be dropped by expanding the bounding box. In the case of persistent errors it has usually been necessary to modify the problem program to output intermediate symbols using **ps** and then to view these symbols using Caesar.

The most difficult to debug errors are those which cause something in the CFL library to abort without calling the error handler. This can happen if a routine is called with the wrong number of arguments, if an attempt is made to use a temporary SYMBOL pointer invalidated by a call to cflcollect, or if the array argument passed to one of the vector operators is not really as long as the call indicates. To help manage errors of this latter category, it is helpfull if larger systems are decomposed into small independently verifyable sections.

7.2 Reminders

1. All **dx** and **dy** coordinates are dimensionless integer values.
2. All layer arguments are alphanumeric layer names for the technology being used.
3. All primitives, macros, and operators are declared SYMBOL * in **cfl.h** and return pointers to the symbols they create.
4. The CFL library contains a large number of operators which are automatically declared by the **cfl.h** file. As most of these names are short, potential naming conflicts are likely between the CFL operators and the application's variables. To help avoid this problem, none of the names reserved by CFL contains a digit. Hence variables may be safely named box1, box2, and so on.
5. A quick way to get a complete up to date listing of all CFL facilities alphabetically arranged by category with a one line description for each is **cat**

8 Appendix

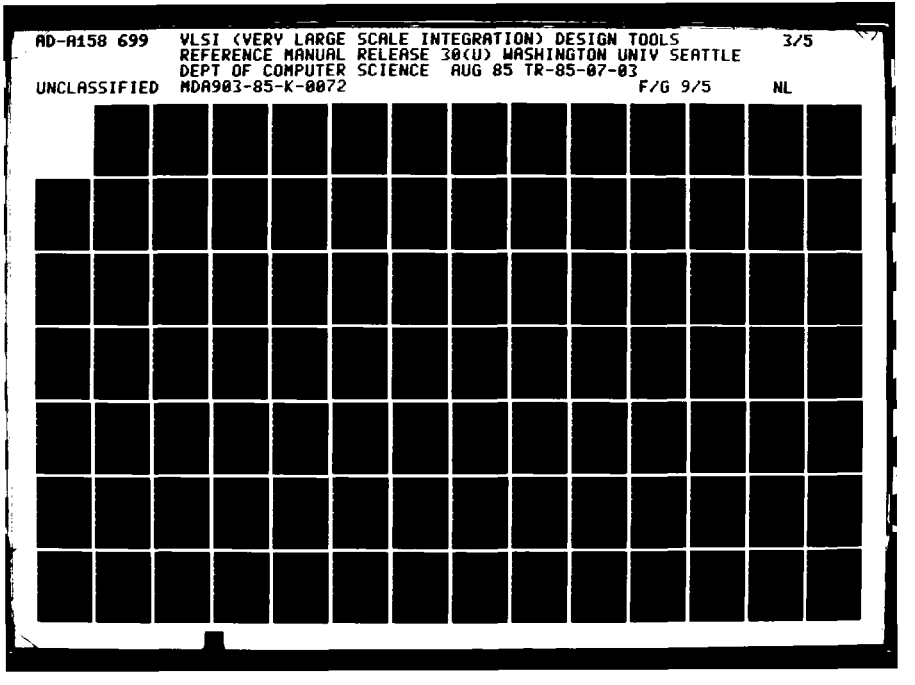
This Appendix lists all of the CFL functions described in this manual. The functions are arranged alphabetically and grouped in accordance with the chapters of the manual.

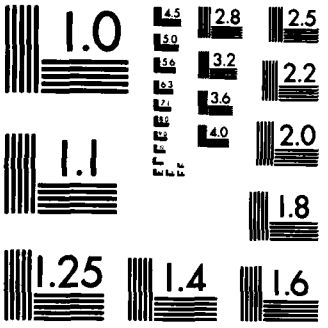
```
/*  Data constructors                                              */

BORDER *bd(s,side,layername)   /* symbolic border descriptor        */
SYMBOL *s;
char   *side;
char   *layername;

BORDER *bdex(bi,i)             /* exclude i from border descriptor  */
BORDER *bi;
int     i;

BORDER *bdin(bi,i)             /* include i in border descriptor    */
BORDER *bi;
int     i;

int f(r)                       /* convert floating argument         */
float r;

PT *pt(s,side,layername,n)     /* symbolic point                    */
SYMBOL *s;
char   *side;
char   *layername;
int     n;


/*  Operators                                                      */

SYMBOL *bb(s1,s2)              /* align bottom to bottom            */
SYMBOL *s1,*s2;

SYMBOL *bbdx(s1,s2,dx)         /* align bottom to bottom, x  offset */
SYMBOL *s1,*s2;
int     dx;

SYMBOL *bbdxy(s1,s2,dx,dy)     /* align bottom to bottom, xy offset */
SYMBOL *s1,*s2;
int     dx,dy;

SYMBOL *bbdy(s1,s2,dy)         /* align bottom to bottom, y  offset */
SYMBOL *s1,*s2;
int     dy;

SYMBOL *bx(s1,s2)              /* pair in x, borders aligned        */
SYMBOL *s1,*s2;
```

```
SYMBOL *bxdx(s1,s2,dx)          /* pair in x, borders aligned, x  offset */
SYMBOL *s1,*s2;
int     dx;

SYMBOL *bxdxy(s1,s2,dx,dy)      /* pair in x, borders aligned, xy offset */
SYMBOL *s1,*s2;
int     dx,dy;

SYMBOL *bxdy(s1,s2,dy)          /* pair in x, borders aligned, y  offset */
SYMBOL *s1,*s2;
int     dy;

SYMBOL *by(s1,s2)               /* pair in y, borders aligned            */
SYMBOL *s1,*s2;

SYMBOL *bydx(s1,s2,dx)          /* pair in y, borders aligned, x  offset */
SYMBOL *s1,*s2;
int     dx;

SYMBOL *bydxy(s1,s2,dx,dy)      /* pair in y, borders aligned, xy offset */
SYMBOL *s1,*s2;
int     dx,dy;

SYMBOL *bydy(s1,s2,dy)          /* pair in y, borders aligned, y  offset */
SYMBOL *s1,*s2;
int     dy;

SYMBOL *cc(s1,s2)               /* align center to center                */
SYMBOL *s1,*s2;

SYMBOL *ccdx(s1,s2,dx)          /* align center to center, x  offset     */
SYMBOL *s1,*s2;
int     dx;

SYMBOL *ccdxy(s1,s2,dx,dy)      /* align center to center, xy offset     */
SYMBOL *s1,*s2;
int     dx,dy;

SYMBOL *ccdy(s1,s2,dy)          /* align center to center, y  offset     */
SYMBOL *s1,*s2;
int     dy;

SYMBOL *cp(s1,p1)               /* align center to point                 */
SYMBOL *s1;
PT     *p1;

SYMBOL *cpdx(s1,p1,dx)          /* align center to point, x  offset      */
SYMBOL *s1;
PT     *p1;
int     dx;

SYMBOL *cpdxy(s1,p1,dx,dy)      /* align center to point, xy offset      */
SYMBOL *s1;
```

```
PT      *pi;
int      dx,dy;

SYMBOL *cpdy(si,pi,dy)          /* align center to point, y  offset      */
SYMBOL *si;
PT      *pi;
int      dy;

SYMBOL *cx(si,s2)               /* pair in x, center aligned             */
SYMBOL *si,*s2;

SYMBOL *cxdx(si,s2,dx)          /* pair in x, center aligned, x  offset  */
SYMBOL *si,*s2;
int      dx;

SYMBOL *cxdxy(si,s2,dx,dy)      /* pair in x, center aligned, xy offset  */
SYMBOL *si,*s2;
int      dx,dy;

SYMBOL *cxdy(si,s2,dy)          /* pair in x, center aligned, y  offset  */
SYMBOL *si,*s2;
int      dy;

SYMBOL *cy(si,s2)               /* pair in y, center aligned             */
SYMBOL *si,*s2;

SYMBOL *cydx(si,s2,dx)          /* pair in y, center aligned, x  offset  */
SYMBOL *si,*s2;
int      dx;

SYMBOL *cydxy(si,s2,dx,dy)      /* pair in y, center aligned, xy offset  */
SYMBOL *si,*s2;
int      dx,dy;

SYMBOL *cydy(si,s2,dy)          /* pair in y, center aligned, y  offset  */
SYMBOL *si,*s2;
int      dy;

SYMBOL *ll(si,s2)               /* align left to left                    */
SYMBOL *si,*s2;

SYMBOL *lldx(si,s2,dx)          /* align left to left, x  offset         */
SYMBOL *si,*s2;
int      dx;

SYMBOL *lldxy(si,s2,dx,dy)      /* align left to left, xy offset         */
SYMBOL *si,*s2;
int      dx,dy;

SYMBOL *lldy(si,s2,dy)          /* align left to left, y  offset         */
SYMBOL *si,*s2;
int      dy;

SYMBOL *mx(s)                   /* mirror in x                           */
```

```
SYMBOL *s;

SYMBOL *my(s)                   /* mirror in y                          */
SYMBOL *s;

SYMBOL *nx(s,n)                 /* repeat in x                          */
SYMBOL *s;
int     n;

SYMBOL *nxdx(s,n,dx)            /* repeat in x, x spacing               */
SYMBOL *s;
int     n;
int     dx;

SYMBOL *nxy(s,nx,ny)            /* repeat in x and y                    */
SYMBOL *s;
int     nx,ny;

SYMBOL *nxydxy(s,nx,ny,dx,dy)   /* repeat in x and y, xy spacing        */
SYMBOL *s;
int     nx,ny;
int     dx,dy;

SYMBOL *ny(s,n)                 /* repeat in y                          */
SYMBOL *s;
int     n;

SYMBOL *nydy(s,n,dy)            /* repeat in y, y spacing               */
SYMBOL *s;
int     n;
int     dy;

SYMBOL *oo(s1,s2)               /* align origin to origin               */
SYMBOL *s1,*s2;

SYMBOL *oodx(s1,s2,dx)          /* align origin to origin, x  offset    */
SYMBOL *s1,*s2;
int     dx;

SYMBOL *oodxy(s1,s2,dx,dy)      /* align origin to origin, xy offset    */
SYMBOL *s1,*s2;
int     dx,dy;

SYMBOL *oody(s1,s2,dy)          /* align origin to origin, y  offset    */
SYMBOL *s1,*s2;
int     dy;

SYMBOL *pax(s1,n1,s2,n2,layername)      /* point align in x             */
SYMBOL *s1,*s2;
int     n1,n2;
char    *layername;

SYMBOL *paxdx(s1,n1,s2,n2,layername,dx) /* point align in x, x  offset */
SYMBOL *s1,*s2;
```


AD-A158 699 VLSI (VERY LARGE SCALE INTEGRATION) DESIGN TOOLS 3/5
REFERENCE MANUAL RELEASE 30(U) WASHINGTON UNIV SEATTLE
DEPT OF COMPUTER SCIENCE AUG 85 TR-85-07-03
UNCLASSIFIED MDA903-85-K-0072 F/G 9/5 NL



```
int      n1,n2;
char    *layername;
int      dx;

SYMBOL *paxdxy(s1,n1,s2,n2,layername,dx,dy)   /* point align in x, xy offset */
SYMBOL *s1,*s2;
int      n1,n2;
char    *layername;
int      dx,dy;

SYMBOL *paxdy(s1,n1,s2,n2,layername,dy)  /* point align in x, y  offset */
SYMBOL *s1,*s2;
int      n1,n2;
char    *layername;
int      dy;

SYMBOL *pay(s1,n1,s2,n2,layername)       /* point align in y            */
SYMBOL *s1,*s2;
int      n1,n2;
char    *layername;

SYMBOL *paydx(s1,n1,s2,n2,layername,dx)  /* point align in y, x  offset */
SYMBOL *s1,*s2;
int      n1,n2;
char    *layername;
int      dx;

SYMBOL *paydxy(s1,n1,s2,n2,layername,dx,dy)   /* point align in y, xy offset */
SYMBOL *s1,*s2;
int      n1,n2;
char    *layername;
int      dx,dy;

SYMBOL *paydy(s1,n1,s2,n2,layername,dy)  /* point align in y, y  offset */
SYMBOL *s1,*s2;
int      n1,n2;
char    *layername;
int      dy;

SYMBOL *repx(s,n,dx)           /* repeat in x, spacial period dx        */
SYMBOL *s;
int      n;
int      dx;

SYMBOL *repxy(s,nx,ny,dx,dy)   /* repeat in x and y,  periods dx dy     */
SYMBOL *s;
int      nx,ny;
int      dx,dy;

SYMBOL *repy(s,n,dy)           /* repeat in y, spacial period dy        */
SYMBOL *s;
int      n;
int      dy;
```

```
SYMBOL *rot(s,n)                /* rotate                              */
SYMBOL *s;
int n;

SYMBOL *rr(s1,s2)               /* align right to right                */
SYMBOL *s1,*s2;

SYMBOL *rrdx(s1,s2,dx)          /* align right to right, x  offset     */
SYMBOL *s1,*s2;
int     dx;

SYMBOL *rrdxy(s1,s2,dx,dy)      /* align right to right, xy offset     */
SYMBOL *s1,*s2;
int     dx,dy;

SYMBOL *rrdy(s1,s2,dy)          /* align right to right, y  offset     */
SYMBOL *s1,*s2;
int     dy;

SYMBOL *su(s1,s2)               /* symbol union                        */
SYMBOL *s1,*s2;

SYMBOL *tt(s1,s2)               /* align top to top                    */
SYMBOL *s1,*s2;

SYMBOL *ttdx(s1,s2,dx)          /* align top to top, x  offset         */
SYMBOL *s1,*s2;
int     dx;

SYMBOL *ttdxy(s1,s2,dx,dy)      /* align top to top, xy offset         */
SYMBOL *s1,*s2;
int     dx,dy;

SYMBOL *ttdy(s1,s2,dy)          /* align top to top, y  offset         */
SYMBOL *s1,*s2;
int     dy;

SYMBOL *vx(s,n)                 /* vector in x                         */
SYMBOL *s[];
int n;

SYMBOL *vxy(s,nx,ny)            /* vector in x and y                   */
SYMBOL *s[];
int nx,ny;

SYMBOL *vy(s,n)                 /* vector in y                         */
SYMBOL *s[];
int n;


/*  CFL control and library access                                     */


int cflcollect()                /* collect temporary symbols           */
```

```
int cflsetc(a,v)                /* set string parameter                 */
char *a;
char *v;

int cflsetv(a,v)                /* set integer parameter                */
char *a;
int   v;

int cflstart(tech)              /* initialize cfl                       */
char *tech;

int cflstop()                   /* terminate  cfl                       */

SYMBOL *gs(cell)                /* get library symbol                   */
char *cell;

SYMBOL *gsp(cell)               /* get library symbol with prefix       */
char *cell;

SYMBOL *ps(y,s)     /* put symbol in the symbol table                   */
char *y;
SYMBOL *s;

SYMBOL *psp(y,s)    /* put symbol in the symbol table with prefix       */
char   *y;
SYMBOL *s;


/*  Routers                                                             */


SYMBOL *elb(s0,b1,b2,w1,w2,ct,rev)      /* compound elbow               */
SYMBOL *s0;         /* container cell                                   */
BORDER *b1;         /* border of tee connected cell                     */
BORDER *b2;         /* border of transverse routing cell                */
int     w1,         /* width of wiring from b1                          */
        w2;         /* width of wiring from b2                          */
SYMBOL *ct;         /* contact symbol                                   */
int     rev;        /* TRUE if elbow is to reverse connection order     */

SYMBOL *ext(b,d,w)                      /*  border extender             */
BORDER *b;
int     d;
int     w;

SYMBOL *fill(s,side,d)                  /* Caesar fill operator         */
SYMBOL *s;
char   *side;
int     d;

SYMBOL *locate(s1,s2)                   /*  Locate s2 within s1.        */
SYMBOL *s1;
char   *s2;
```

```
int nbc(b)      /*  Obtain the number of crossings on the border b      */
BORDER *b;

SYMBOL *plx(s0,p1,p2,w,ct)              /* point to line in x          */
SYMBOL *s0;         /* container cell                                  */
PT     *p1;         /* symbolic point 1                                */
PT     *p2;         /* symbolic point 2                                */
int     w;          /* width                                           */
SYMBOL *ct;         /* contact                                         */

SYMBOL *ply(s0,p1,p2,w,ct)              /* point to line in y          */
SYMBOL *s0;         /* container cell                                  */
PT     *p1;         /* symbolic point 1                                */
PT     *p2;         /* symbolic point 2                                */
int     w;          /* width                                           */
SYMBOL *ct;         /* contact                                         */

SYMBOL *pp(s0,p1,p2,w)                  /*  point to point router      */
SYMBOL *s0;         /* container cell                                  */
PT     *p1;         /* symbolic point 1                                */
PT     *p2;         /* symbolic point 2                                */
int     w;          /* width                                           */

SYMBOL *pr(s0,b1,b2,w)                  /*  planar router              */
SYMBOL *s0;         /* container cell                                  */
BORDER *b1;         /* border of sub-cell 1                            */
BORDER *b2;         /* border of sub-cell 2                            */
int     w;          /* width                                           */

PROC prset(minsep,arcoff)       /*  set parameters for the planar router */
int   minsep,                   /* planar router, minimum separation    */
      arcoff;                   /* planar router, offset of the first arc*/

SYMBOL *tee(s0,b1,b2,w,ct,rev)          /* tee                         */
SYMBOL *s0;         /* container cell                                  */
BORDER *b1;         /* border of tee connected cel                     */
BORDER *b2;         /* border of transverse routin                     */
int     w;          /* width                                           */
SYMBOL *ct;         /* contact symbol                                  */
int     rev;        /* reverse flag                                    */
                    /*   TRUE  - reversed tee                          */
                    /*   FALSE - forward  tee                          */


/*  Macros                                                             */


SYMBOL *be()                    /* butting contact, east               */

SYMBOL *bn()                    /* butting contact, north              */

SYMBOL *box(layer,dx,dy)        /* box                                 */
char *layer;
```

```
int dx,dy;

SYMBOL *bs()                    /* butting contact, south          */

SYMBOL *bw()                    /* butting contact, west           */

SYMBOL *cross(layer1,dx1,dy1,layer2,dx2,dy2)    /* cross           */
char *layer1,*layer2;
int   dx1,dy1;
int   dx2,dy2;

SYMBOL *gb()                    /* metal diffusion contact         */

SYMBOL *label(name,dx,dy,pos)   /* label                           */
char *name;
int   dx,dy;
int   pos;

SYMBOL *lne(layer,w,dx,dy)      /* el, north east                  */
char *layer;
int   w;
int   dx,dy;

SYMBOL *lnw(layer,w,dx,dy)      /* el, north west                  */
char *layer;
int   w;
int   dx,dy;

SYMBOL *lse(layer,w,dx,dy)      /* el, south east                  */
char *layer;
int   w;
int   dx,dy;

SYMBOL *lsw(layer,w,dx,dy)      /* el, south west                  */
char *layer;
int   w;
int   dx,dy;

SYMBOL *padelne(dx,dy)          /* pad elbow - north east corner   */
int dx,dy;

SYMBOL *padelnw(dx,dy)          /* pad elbow - north west corner   */
int dx,dy;

SYMBOL *padelse(dx,dy)          /* pad elbow - south east corner   */
int dx,dy;

SYMBOL *padelsw(dx,dy)          /* pad elbow - south west corner   */
int dx,dy;

SYMBOL *padext(dx)              /* pad frame extension             */
int dx;

SYMBOL *pullup(l,s)             /* pullup                          */
```

```
int l;
int n;

SYMBOL *rb()                    /* metal polysilicon contact          */

/*  Wire facility                                                     */


SYMBOL *alpha(s,layer,w)        /* string of alpha characters         */
char *s;
char *layer;
int w;

int at(x0,y0)                   /* set the wire position to (x0,y0)   */
int x0,y0;

int dx(dx0)                     /* draw to the point (x+dx0,y)        */
int dx0;

int dy(dy0)                     /* draw to the point (x,y+dy0)        */
int dy0;

int iso(s)                      /* include symbol origin              */
SYMBOL *s;

SYMBOL *letter(c,layer,w)       /* letter                             */
char c;
char *layer;
int  w;

int wl(layer)                   /* reset wire layer                   */
char   *layer;

SYMBOL *wire(layer,width)       /* initialize wire                    */
char   *layer;
int     width;

int ww(width)                   /* reset wire width                   */
int     width;

int x(x0)                       /* draw to the point (x0,y)           */
int x0;

int y(y0)                       /* draw to the point (x,y0)           */
int y0;
```

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 realigned 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 "*1". 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 "*1".

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 s 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 s, 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 s: 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 **e**. 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

:elf -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 *self -xbl*; to generate CIF files suitable for manufacturing, use *self*; to get CIF files for circuit extraction and/or simulation, use *self -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 *self* 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 **sedltcell** 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 I 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/I layer is transparent with respect to the mask layers: a separate color exists for each combination of mask layer and G/S/I. 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.

Standard Cell Library Guide

UW/NW VLSI Consortium
University of Washington
Seattle, WA 98195

1. Introduction

Various pre-generated cells are made available to the layout designer for use in custom designs laid out using efl or caesar. The various cells are to be used with the particular technology under which they are listed. The designer should avoid using the same names of these symbols when designing his own symbols.

1.1. nMOS Cell Library

These cells were provided courtesy of MOSIS, and as such correspond to MOSIS specifications for fabrication. They are current as of April 1984. For more information on the pads and padframes, refer to the MOSIS document *nmos.doc* included in the cells' source directory.

1.1.1. Padframes

Pin 1 of these padframes is connected to the substrate, and therefore should not be used for a signal line.

28p23x34 - 28 pin frame with dimensions of 2300 by 3400 microns ;
28p46x34 - 28 pin frame with dimensions of 4600 by 3400 microns ;
40p46x34 - 40 pin frame with dimensions of 4600 by 3400 microns ;
40p46x68 - 40 pin frame with dimensions of 4600 by 6800 microns ;
40p69x68 - 40 pin frame with dimensions of 6900 by 6800 microns ;
64p46x68 - 64 pin frame with dimensions of 4600 by 6800 microns ;
64p69x68 - 64 pin frame with dimensions of 6900 by 6800 microns ;
64p79x92 - 64 pin frame with dimensions of 7900 by 9200 microns ;
84p69x68 - 84 pin frame with dimensions of 6900 by 6800 microns ;
84p79x92 - 84 pin frame with dimensions of 7900 by 9200 microns ;

1.1.2. Pads

The names of some of these cells have been shortened due to system limits on the length of filenames. The MOSIS names are listed in parentheses with the renamed version ahead of them in boldface.

PadVdd - VDD pad ;
PadGround - GND pad ;
PadIn - Protected signal input pad ;
PadOut - Output pad ;
PadClkO (PadClockedOut) - Clocked output pad ;
Pad3State (PadTriState) - Tri-state input/output pad ;
PadClkBar (PadClockBar) - Two-phase non-overlapping clock-bar pad ;

1.2. CMOSPW Cell Library

These cells are intended for fabrication under the MOSIS 3 micron CMOS process "CBP2", and correspond to MOSIS specifications. They do not use second poly or metal, and were laid out using lambda = 1 micron. The pads and padframes are provided courtesy of Paul Bassett of the Massachusetts Institute of Technology. For a description of the differences between the three pad groups covered below, as well as a more complete description of the individual pad drivers and padframes, see Appendix A on the 3-micron bulk cmos pads and padframes.

1.2.1. Padframes

Pin 1 of these padframes is connected to the substrate and should not be used for a signal line.

28p46x34 - 28 pin padframe with dimensions of 4600 by 3400 microns;
40p46x68 - 40 pin padframe with dimensions of 4600 by 6800 microns;
40p69x68 - 40 pin padframe with dimensions of 6900 by 6800 microns;
64p69x68 - 64 pin padframe with dimensions of 6900 by 6800 microns;
64p79x92 - 64 pin padframe with dimensions of 7900 by 9200 microns;
84p79x92 - 84 pin padframe with dimensions of 7900 by 9200 microns;

1.2.2. Group 1 Pads

These pads are the largest of the three groups of CMOSPW pads provided, measuring 300 by 640 microns.

pad1out - Output pad;
pad1out-ttl - TTL output pad;
pad1ts - Tristate pad;
pad1in - Input pad;
pad1bin - Buffered input pad;
pad1bin-ttl - Buffered TTL input pad;
pad1gnd - Ground pad;
pad1vdd - Vdd pad;
pad1space - Spacer pad for padframes;
pad1 - Complete group of the pad1 cells;

1.2.3. Group 2 Pads

This group has pads measuring 200 by 430 microns, and does not include any output pads.

pad2in - Input pad;
pad2bin - Buffered input pad;
pad2bin-ttl - Buffered TTL input pad;
pad2gnd - Ground pad;
pad2vdd - Vdd pad;
pad2space - Spacer pad for padframes;
pad2 - Complete group of the pad2 cells;

1.2.4. Group 3 Pads

Group 3 contains an only unbuffered input pad, with a Vdd and Ground pad, all being 200 by 306 micron size.

pad3in - Input pad;
pad3gnd - Ground pad;
pad3vdd - Vdd pad;
pad3space - Spacer pad for padframes;
pad3 - Complete group of the pad3 cells;

1.2.5. Miscellaneous Cells

Refer to Appendix B for descriptions of symbols as well as testing information.

_ - Short guard ring (from MIT);
__ - Long guard ring (from MIT);
inv - Basic inverter;
nand2 - Two input nand gate;
nand3 - Three input nand gate;
nand4 - Four input nand gate;
nor2 - Two input nor gate;
nor3 - Three input nor gate;
nor4 - Four input nor gate;
xor2 - Two input exclusive-or gate;
clkinv - Clocked inverter;
sel2inv - Two input inverted selector;
dlatch - D Type latch;
dlatchr - D Type latch with reset;

Appendix A

CMOSPW PADS AND PADFRAMES

These pads are intended for fabrication under the MOSIS 3 micron CMOS process "CBPM2". They do not use second poly. They were laid out using lambda = 1 micron. They are provided courtesy of Paul Bassett of the Massachusetts Institute of Technology.

1. The Pad Types

The pads are divided into 3 groups. All of the pads in each group are compatible with the other members of its group but the groups are not compatible with each other due to power bus mismatches. Some of the pertinent properties of each of the groups are briefly discussed below followed by a brief discussion of each of the pad types. Members of different groups that have the same function are differentiated by a number in the name e.g. **pad1in** vs **pad2in**; since Group 1 has a complete set of pads, each type of pad will be discussed briefly only for that group. In particular, note that the pad2 and pad3 groups do not have the via and second layer metal layers over the pads required for the process "CBPM2".

2. Group One

This group is the only complete group and the pads in this group are also the largest pads. All of the pads are 300 x 640 microns. This size is dictated mostly by the size of driver transistors and the input buffer stages on the output pads; however, the pads have been expanded somewhat to fit the pad-to-pad spacings of the MOSIS standard padframes more closely. Two consequences of this are that the pads could be made slightly smaller if this is desired or the spacings of the driver transistors could be increased slightly more to provide some additional latchup protection. The input protection on all of the input pads consists of a well resistor, approximately 20 x 20 microns, followed p+-to-substrate and n+-to-well diodes providing additional clipping above Vdd and below Gnd, both diodes are approximately 15 x 15 microns. These pads have two Vdd buses and one Ground bus. The top Vdd bus is 60 microns wide and the bottom Vdd bus is 20 microns wide and has a 20 micron wide strip of n+ diffusion under it providing a guard ring to separate the pads from the internal circuitry, this guard ring is only broken where the inputs and outputs to the pads cross it. The Ground bus which runs through the middle of the pads is 74 microns wide. The input and output signal lines extend past the lower Vdd bus by 6 microns to allow connecting to them without design rule violations or modifications to the pads. Therefore, even though all of the pads have their lower left-hand corner at 0,0, the lower left-hand corner of the lower Vdd buses are at 0,6.

pad1out – output pad. While this is intended for driving principally capacitive loads such as other MOS devices, it can sink current for TTL. The signal is presented at point "DATA". It presents a small, though not minimal, load on this point. Experiments show that it can source or sink about 30ma, and has a delay of about 20ns into a very light load and 25ns into 50pf.

pad1ottl – TTL output pad. This pad is similar to the regular output pad except that it has an n-type pullup and the input buffer has been changed to drive the pullup and pulldown separately. This pad is experimental in that it has not ever been fabricated ; in spice it simulates correctly, it pulls HIGH to between 2.5 and 3 volts. How high it will pull in real operation is the major point in question. If it does pull high enough for TTL compatibility, it should be faster than the regular output pad.

pad1ts – bidirectional tristate pad. If point "OUT-ENAB" is set low, the pin is left to float, and whatever signal comes in from the outside appears at point "IN" (which is not buffered). If point "OUT-ENAB" is set high, the signal on point "OUT" is placed on the pin (and is also available on "IN"). This presents a fairly small, though not minimal, load on "OUT", but a moderately heavy load (sorry) on "OUT-ENAB".

pad1in – unbuffered input pad. This has the "lightning arrestor" resistor and protective diodes, but no logic. The signal appears at point "DATA".

pad1bin – buffered input. It presents both the true data at the point labeled "DATA", and the inverted data at "-DATA". Both are driven by fairly strong buffers.

pad1bit – TTL input pad. This has input amplifiers designed to have a threshhold near 1.5 volts for sensing the output of TTL chips. It presents both the true data at the point labeled "DATA", and the inverted data at "-DATA", though the latter's threshhold is not offset as far as it should be. The output from "DATA" is fairly strong but the "-DATA" output is weak.

pad1gnd, pad1vdd – Vdd and Gnd pads. The appropriate voltages come out on 100 micron wide metal lines. The ground bus is broken in the Vdd pad and the lower Vdd bus is broken on the Gnd but the guard ring does continue under the ground line, without any contacts (obviously).

pad1sp – spacer for pad frames. This cell is mainly meant for making it easy to fill in spaces between pads; all it contains is the three power and ground buses.

_ – short guard ring. This cell is a 40 micron long piece of the 20 micron wide guard ring.

__ – long guard ring. This cell is a 400 micron long piece of the guard ring.

pad1 – the complete group. A cell containing an instance of every cell in the group.

3. Group Two

The pads in the second group are much smaller than those in the first group, 200 x 430 microns. This size is dictated by the size of the buffers on the buffered input pads. There are no output pads in this family. Each of the pads again has two Vdd buses, which are both 20 microns wide, and one Ground bus, which is 28 microns wide. The input protection is the same as for the Group 1 input pads and there is also a 20 micron wide guard ring under the lower Vdd bus.

4. Group Three

This group actually consists of just an unbuffered input pad so the size of the other pads, Vdd etc., is dictated by this pad and is 200 x 306 microns. This pad has a 30 wide upper Vdd bus, a 10 micron wide Ground bus and a 20 micron wide lower Vdd bus with the guard ring under it like the other pads. The input protection on this pad is the same as on the others.

5. MOSIS Standard Pad Frames

Pad frames have been developed for some of the MOSIS standard pad frames. The frames contain the indicated number of pads, all of which are initially unbuffered input pads. The pads are arranged to meet the MOSIS specifications and where necessary, the **pad1sp** instance has been used to fill in the buses and the guard ring in between the pads. Also, each of the pad frames has instantiated in the middle of it a set of the available pads. Since the output pads are fairly large, not all of the padframe spacing would allow using all of the allotted pins, only those frames that allow a full complement of pads have been implemented; the available frames are:

FRAME NAME	DIE SIZE (MICRONS)	INTERIOR PROJECT SIZE	PINS/PACKAGE	PIN ROW SPACING
28p46x34	4600 x 3400	3320 x 2120	28 DIP	0.6"
40p46x68	4600 x 6800	3320 x 5520	40 DIP	0.6"
40p69x68	6900 x 6800	5620 x 5520	40 DIP	0.6"
64p69x68	6900 x 6800	5620 x 5520	64 DIP	0.9"
64p79x92	7900 x 9200	6620 x 7920	64 DIP	0.9"
84p79x92	7900 x 9200	6620 x 7920	84 PGA	-

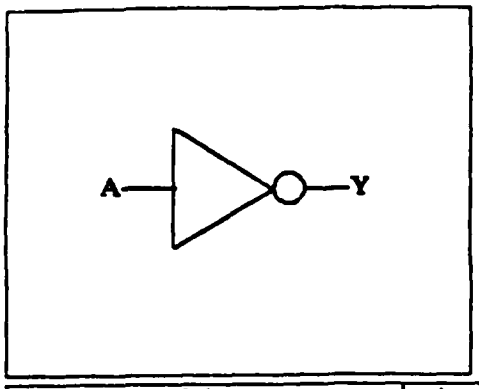
Appendix B

CMOSPW CELL DESCRIPTIONS

Herein is contained a description of most of the standard cells available in the CMOSPW cell library. At the end of this appendix is located a writeup on simulation conditions.

CMOSPW LIBRARY CELL INV

INVERTER



Truth Table

A	Y
0	1
1	0

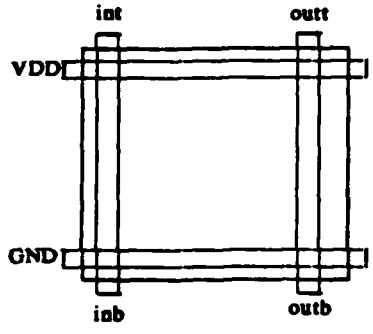
$Y = \bar{A}$

Nearest Functional Equivalent:

CMOS 4049, 4069
TTL 7404

Nodes	A	Y								
Input Load	1	-								

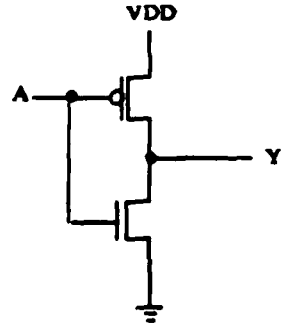
Block Diagram of I/O pins **Cell Width: 33 λ**



AC Characteristics V_{DD} = 5V
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	t_{pHL}	2.0	4.5
Propagation delay (low to high)	t_{pLH}	3.0	6.5
Output fall time	t_{THL}	3.0	8.5
Output rise time	t_{TLH}	4.5	14.0

Circuit Schematic Diagram



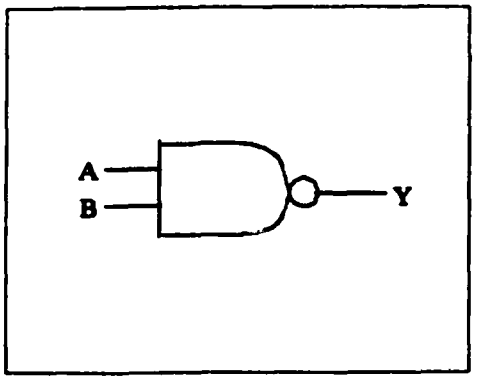
NOTES:
1) rail separation: 41 λ
2) bounding box ($x \times y$): 33λ ×56λ

Table values from SPICE simulation

CAESAR file: $UW_VLSI_TOOLS/src/cellib/cmospw/inv.ca
DB files: $UW_VLSI_TOOLS/src/cellib/cmospw/{inv.att,inv.sym}

CMOSPW LIBRARY CELL NAND2

2-INPUT NAND



Truth Table

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

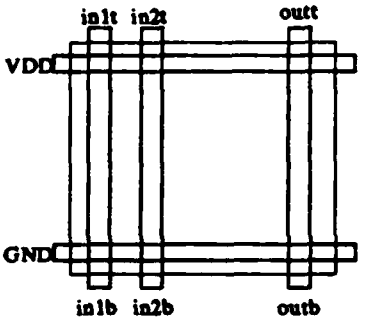
$Y = \overline{AB}$

Nearest Functional Equivalent:

CMOS 4011
TTL 7400

Nodes	A	B	Y							
Input Load	1	1	-							

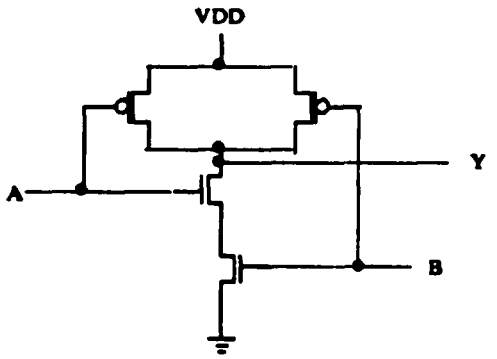
Block Diagram of I/O pins **Cell Width: 37 λ**



AC Characteristics V_{DD} = 5V
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	t_{pHL}	1.0	4.0
Propagation delay (low to high)	t_{pLH}	2.5	7.0
Output fall time	t_{THL}	3.0	9.5
Output rise time	t_{TLH}	4.5	14.5

Circuit Schematic Diagram



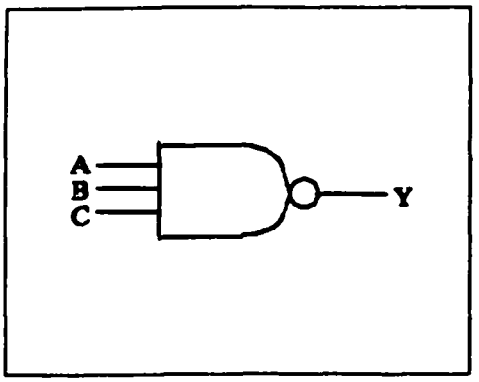
NOTES:
1) rail separation: 48 λ
2) bounding box ($x \times y$): 37λ ×63λ

Table values from SPICE simulation

CAESAR file: $UW_VLSI_TOOLS/src/cellib/cmospw/nand2.ca
DB files: $UW_VLSI_TOOLS/src/cellib/cmospw/{nand2.att, nand2.sym}

CMOSPW LIBRARY CELL NAND3

3-INPUT NAND



Truth Table

A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

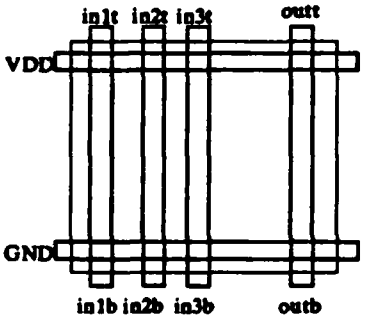
$Y = \overline{ABC}$

Nearest Functional Equivalent:

CMOS 4023
TTL 7410

Nodes	A	B	C	Y						
Input Load	1	1	1	-						

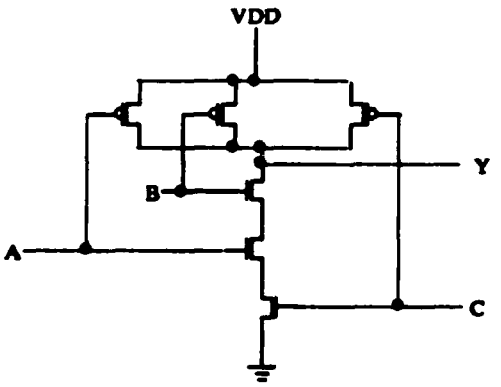
Block Diagram of I/O pins **Cell Width: 53 λ**



AC Characteristics V_{DD} = 5V
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	t_{pHL}	2.0	6.5
Propagation delay (low to high)	t_{pLH}	2.5	7.5
Output fall time	t_{THL}	3.0	12.5
Output rise time	t_{TLH}	5.0	16.0

Circuit Schematic Diagram



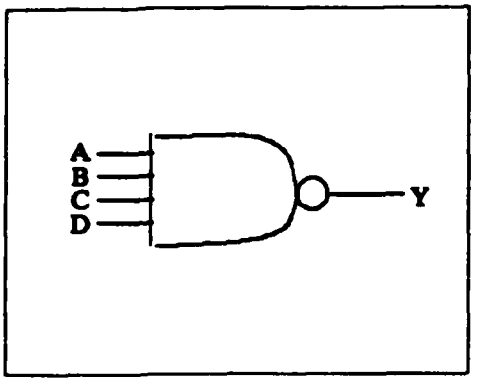
NOTES:
1) rail separation: 47 λ
2) bounding box ($x \times y$): 53λ ×62λ

Table values from SPICE simulation

CAESAR file: $UW_VLSI_TOOLS/src/cellib/cmospw/nand3.ca
DB files: $UW_VLSI_TOOLS/src/cellib/cmospw/{nand3.att, nand3.sym}

CMOSPW LIBRARY CELL NAND4

4-INPUT NAND



Truth Table

A	B	C	D	Y
0	X	X	X	1
X	0	X	X	1
X	X	0	X	1
X	X	X	0	1
1	1	1	1	0

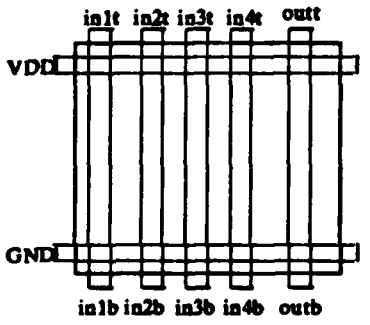
$Y = \overline{ABCD}$

Nearest Functional Equivalent:

CMOS 4012
TTL 7420

Nodes	A	B	C	D	Y					
Input Load	1	1	1	1	-					

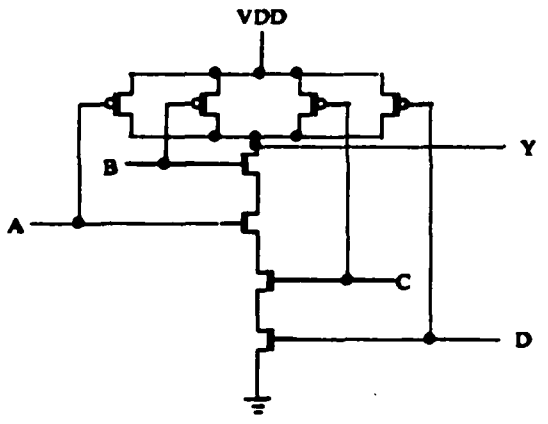
Block Diagram of I/O pins **Cell Width: 58 λ**



AC Characteristics V_{DD} = 5V
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	t_{pHL}	2.0	8.0
Propagation delay (low to high)	t_{pLH}	3.0	6.5
Output fall time	t_{THL}	5.0	18.5
Output rise time	t_{TLH}	5.0	10.0

Circuit Schematic Diagram



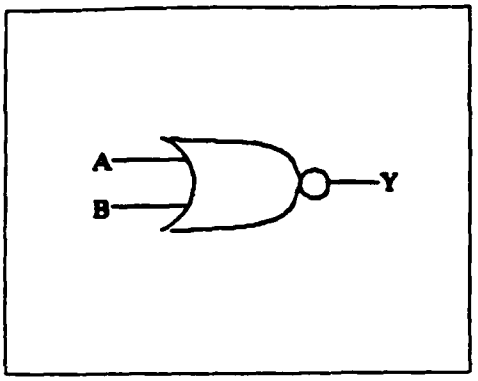
NOTES:
1) rail separation: 53 λ
2) bounding box ($x \times y$): 58λ ×68λ

Table values from SPICE simulation

CAESAR file: $UW_VLSI_TOOLS/src/cellib/cmospw/nand4.ca
DB files: $UW_VLSI_TOOLS/src/cellib/cmospw/{nand4.att, nand4.sym}

CMOSPW LIBRARY CELL NOR2

2-INPUT NOR



Truth Table

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

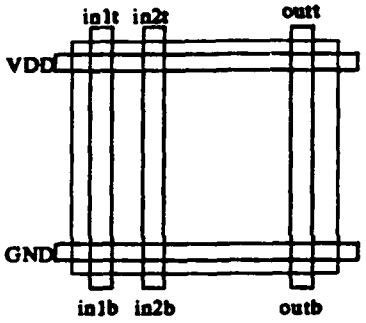
$Y = \overline{A + B}$

Nearest Functional Equivalent:

CMOS 4001
TTL 7402

Nodes	A	B	Y							
Input Load	1	1	-							

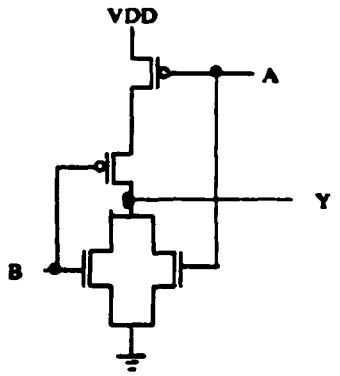
Block Diagram of I/O pins **Cell Width: 36 λ**



AC Characteristics V_{DD} = 5V
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	t_{pHL}	2.5	5.0
Propagation delay (low to high)	t_{pLH}	3.5	10.5
Output fall time	t_{THL}	4.0	10.0
Output rise time	t_{TLH}	8.0	22.5

Circuit Schematic Diagram



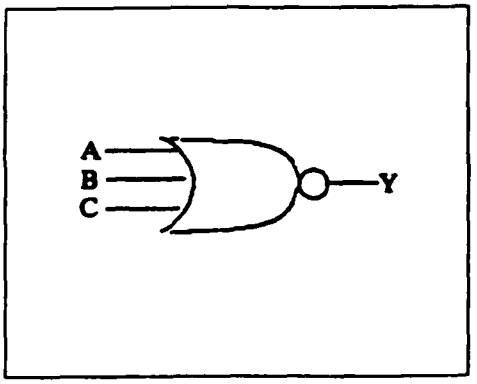
NOTES:
1) rail separation: 47 λ
2) bounding box ($x \times y$): 36λ ×62λ

Table values from SPICE simulation

CAESAR file: $UW_VLSI_TOOLS/src/cellib/cmospw/nor2.ca
DB files: $UW_VLSI_TOOLS/src/cellib/cmospw/{nor2.att, nor2.sym}

CMOSPW LIBRARY CELL NOR3

3-INPUT NOR



Truth Table

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

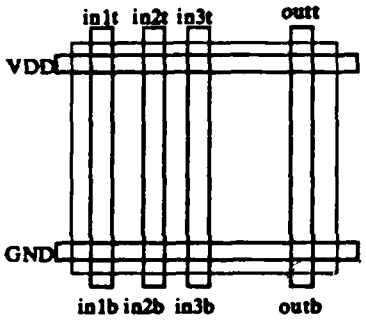
$Y = \overline{A + B + C}$

Nearest Functional Equivalent:

CMOS 4000
TTL 7427

Nodes	A	B	C	Y						
Input Load	1	1	1	-						

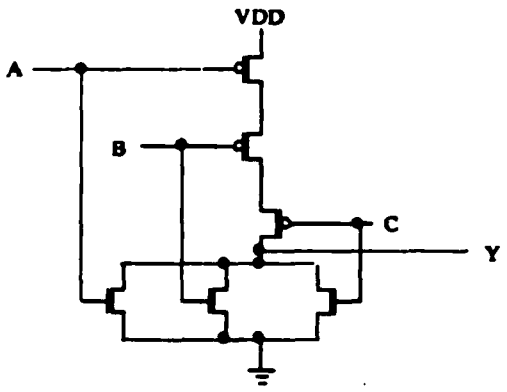
Block Diagram of I/O pins **Cell Width: 50 λ**



AC Characteristics V_{DD} = 5V
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	t_{pHL}	2.5	5.0
Propagation delay (low to high)	t_{pLH}	6.0	16.0
Output fall time	t_{THL}	4.0	12.5
Output rise time	t_{TLH}	13.0	36.5

Circuit Schematic Diagram



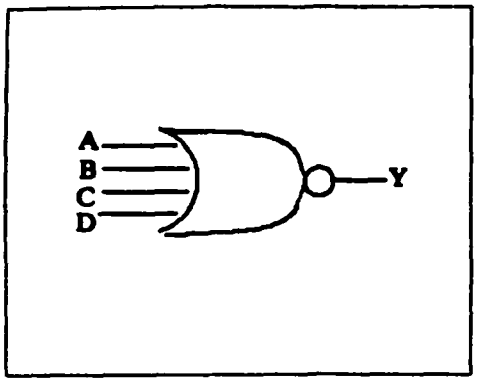
NOTES:
1) rail separation: 47 λ
2) bounding box (x ×y): 50λ ×62λ

Table values from SPICE simulation

CAESAR file: $UW_VLSI_TOOLS/src/cellib/cmospw/nor3.ca
DB files: $UW_VLSI_TOOLS/src/cellib/cmospw/{nor3.att, nor3.sym}

CMOSPW LIBRARY CELL NOR4

4-INPUT NOR



Truth Table

A	B	C	D	Y
0	0	0	0	1
X	X	X	1	0
X	X	1	X	0
X	1	X	X	0
1	X	X	X	0

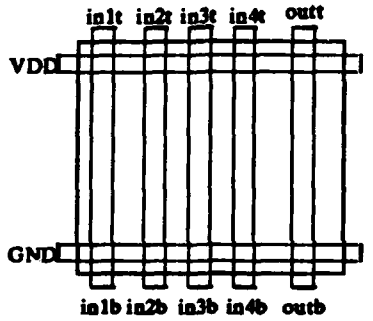
$Y = \overline{A + B + C + D}$

Nearest Functional Equivalent:

CMOS 4002
TTL 7425

Nodes	A	B	C	D	Y					
Input Load	1	1	1	1	-					

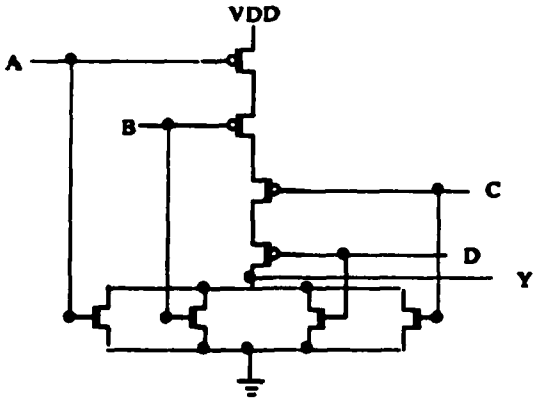
Block Diagram of I/O pins **Cell Width: 57 λ**



AC Characteristics V_{DD} = 5V
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	t_{pHL}	3.0	5.5
Propagation delay (low to high)	t_{pLH}	5.5	24.0
Output fall time	t_{THL}	6.5	11.5
Output rise time	t_{TLH}	19.0	51.0

Circuit Schematic Diagram



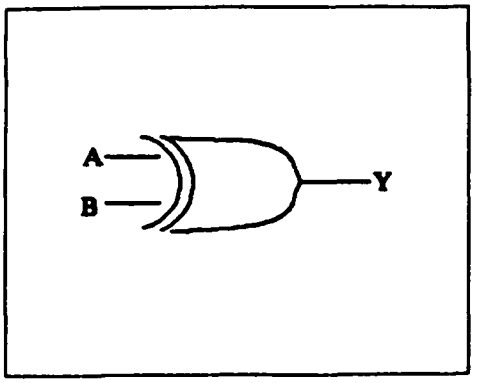
NOTES:
1) rail separation: 58 λ
2) bounding box ($x \times y$): 57λ ×73λ

Table values from SPICE simulation

CAESAR file: $UW_VLSI_TOOLS/src/cellib/cmospw/nor4.ca
DB files: $UW_VLSI_TOOLS/src/cellib/cmospw/{nor4.att, nor4.sym}

CMOSPW LIBRARY CELL XOR2

2-INPUT XOR



Truth Table

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

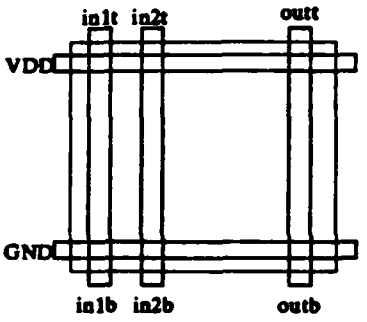
$Y = A \oplus B$

Nearest Functional Equivalent:

CMOS 4030
TTL 7486

Nodes	A	B	Y							
Input Load	2	2	-							

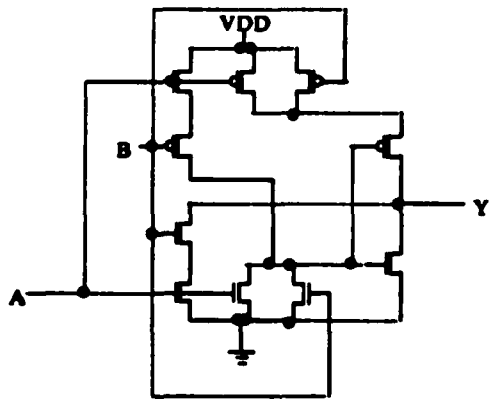
Block Diagram of I/O pins **Cell Width: 59 λ**



AC Characteristics V_{DD} = 5V
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	t_{pHL}	7.5	11.5
Propagation delay (low to high)	t_{pLH}	7.0	19.0
Output fall time	t_{THL}	7.0	12.5
Output rise time	t_{TLH}	12.5	40.5

Circuit Schematic Diagram



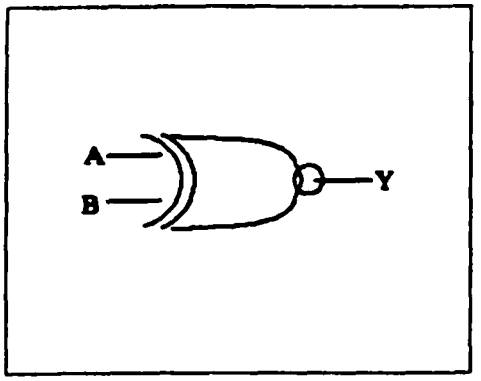
NOTES:
1) rail separation: 69 λ
2) bounding box ($x \times y$): 59λ×84λ

Table values from SPICE simulation

CAESAR file: $UW_VLSI_TOOLS/src/cellib/cmospw/xor2.ca
DB files: $UW_VLSI_TOOLS/src/cellib/cmospw/{xor2.att, xor2.sym}

CMOSPW LIBRARY CELL XNOR2

2-INPUT XNOR



Truth Table

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

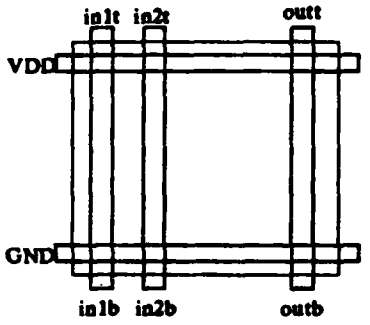
$Y = \overline{A \oplus B}$

Nearest Functional Equivalent:

CMOS
TTL

Nodes	A	B	Y							
Input Load	2	2	-							

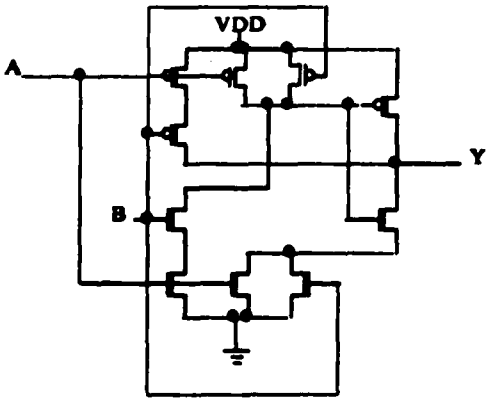
Block Diagram of I/O pins **Cell Width: 59 λ**



AC Characteristics V_{DD} = 5V
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	t_{pHL}	3.0	6.0
Propagation delay (low to high)	t_{pLH}	2.5	13.0
Output fall time	t_{THL}	3.5	12.0
Output rise time	t_{TLH}	11.0	26.

Circuit Schematic Diagram

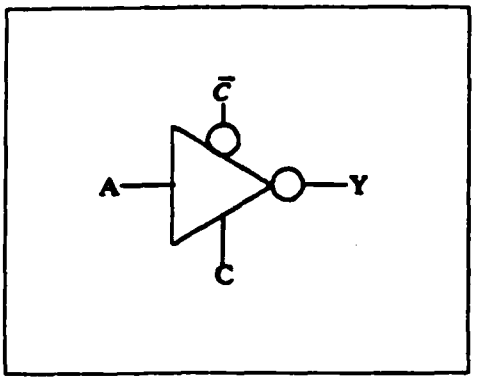


NOTES:
1) rail separation: 71 λ
2) bounding box ($x \times y$): 59λ ×86λ

Table values from SPICE simulation

CAESAR file: $UW_VLSI_TOOLS/src/cellib/cmospw/xnor2.ca
DB files: $UW_VLSI_TOOLS/src/cellib/cmospw/{xnor2.att, xnor2.sym}

CMOSPW LIBRARY CELL CLKINV — CLOCKED INVERTER



Truth Table

C	C-	A	Y
0	1	0	Z
0	1	1	Z
1	0	0	1
1	0	1	0

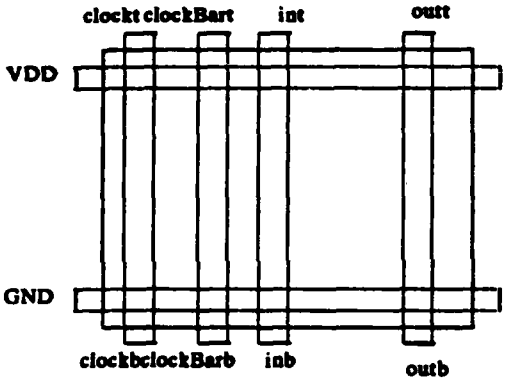
$Y = \bar{A}$

Nearest Functional Equivalent:

CMOS
TTL

Nodes	C	C-	A	Y						
Input Load	.67	.33	1	-						

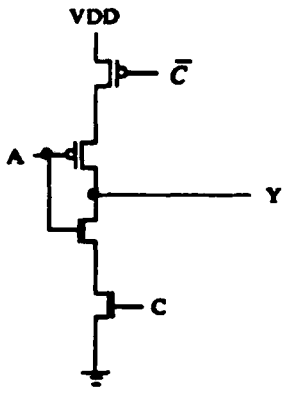
Block Diagram of I/O pins — **Cell Width: 52 λ**



AC Characteristics V_{DD} = 5V
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	t_{pHL}	3.7	7.0
Propagation delay (low to high)	t_{pLH}	5.5	17.5
Output fall time	t_{THL}	5.5	18.5
Output rise time	t_{TLH}	11.0	38.5

Circuit Schematic Diagram

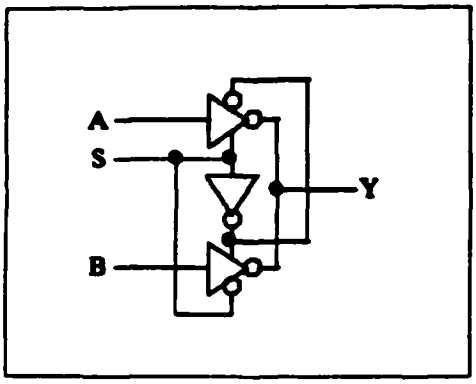


NOTES:
1) rail separation: 44 λ
2) bounding box (x ×y): 52λ ×59λ

Table values from SPICE simulation

CAESAR file: $UW_VLSI_TOOLS/src/cellib/cmospw/clkinv.ca
DB files: $UW_VLSI_TOOLS/src/cellib/cmospw/{clkinv.att, clkinv.sym}

CMOSPW LIBRARY CELL SEL2INV 2-INPUT INVERTED SELECTOR



Truth Table

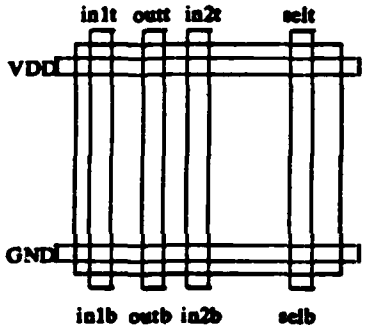
S	A	B	Y
1	1	X	0
1	0	X	1
0	X	1	0
0	X	0	1

Nearest Functional Equivalent:

CMOS
TTL 74157

Nodes	S	A	B	Y						
Input Load	2	1	1	-						

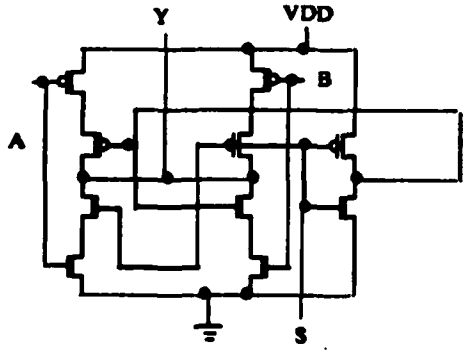
Block Diagram of I/O pins **Cell Width: 74 λ**



AC Characteristics V_{DD} = 5V
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	t_{pHL}	4.5	7.5
Propagation delay (low to high)	t_{pLH}	8.5	21.0
Output fall time	t_{THL}	8.5	23.0
Output rise time	t_{TLH}	20.5	45.0

Circuit Schematic Diagram



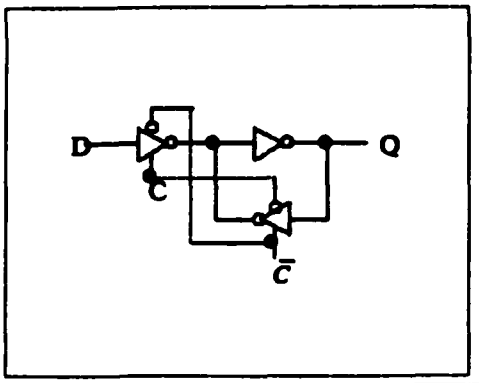
NOTES:
1) rail separation: 55 λ
2) bounding box ($x \times y$): 74λ×70λ

Table values from SPICE simulation

CAESAR file: $UW_VLSI_TOOLS/src/cellib/cmospw/sel2inv.ca
DB files: $UW_VLSI_TOOLS/src/cellib/cmospw/{sel2inv.att, sel2inv.sym}

CMOSPW LIBRARY CELL DLATCH

D-LATCH



Truth Table

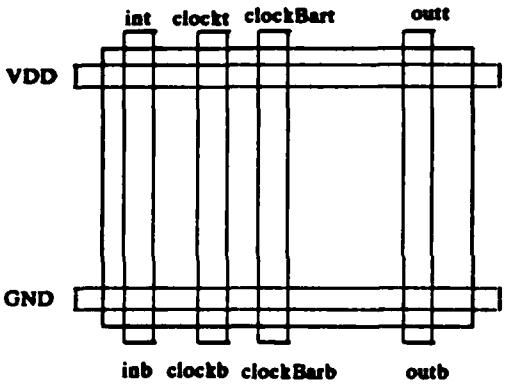
C	C-	D	Q
1	0	1	1
1	0	0	0
0	1	X	Q

Nearest Functional Equivalent:

CMOS
TTL 74LS77

Nodes	C	C-	D	Q						
Input Load	1	1	1	-						

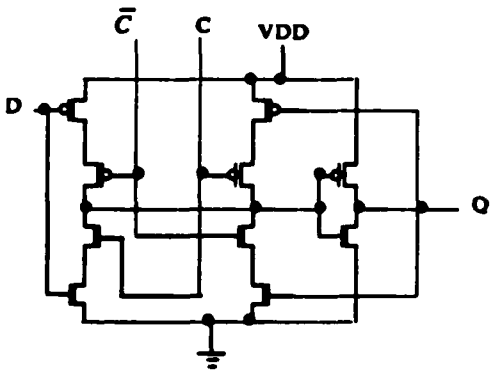
Block Diagram of I/O pins **Cell Width: 77 λ**



AC Characteristics V_{DD} = 5V
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	t_{pHL}	6.5	10.0
Propagation delay (low to high)	t_{pLH}	8.0	21.0
Output fall time	t_{THL}	6.0	10.5
Output rise time	t_{TLH}	7.0	13.5

Circuit Schematic Diagram

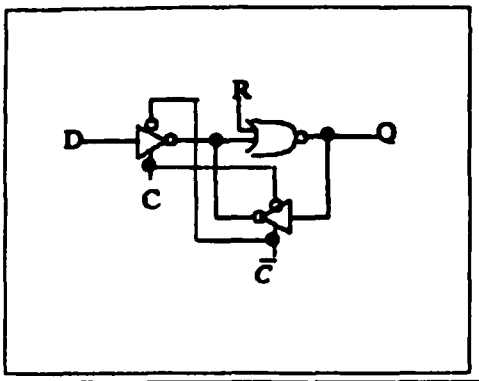


NOTES:
1) rail separation: 58 λ
2) bounding box (x ×y): 77λ ×73λ

Table values from SPICE simulation

CAESAR file: $UW_VLSI_TOOLS/src/cellib/cmospw/dlatch.ca
DB files: $UW_VLSI_TOOLS/src/cellib/cmospw/{dlatch.att, dlatch.sym}

CMOSPW LIBRARY CELL DLATCHR — D-LATCH WITH RESET



Truth Table

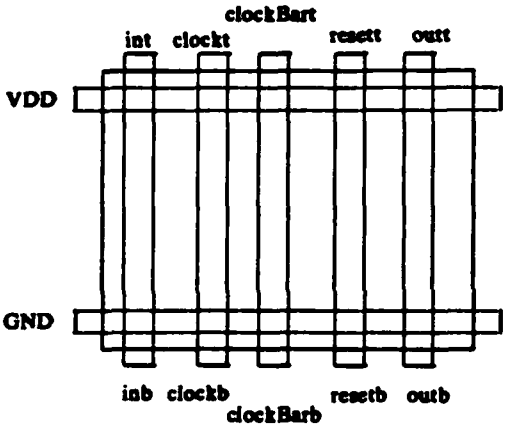
C	C-	R	D	Q
1	0	0	D	D
0	1	0	X	Q
X	X	1	X	0

Nearest Functional Equivalent:

CMOS
TTL

Nodes	C	C-	R	D	Q					
Input Load	1	1	1	1	-					

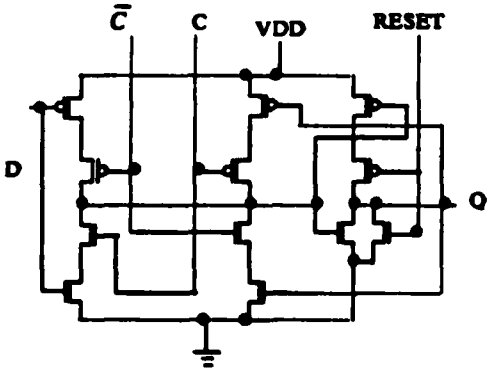
Block Diagram of I/O pins — **Cell Width: 88 λ**



AC Characteristics V_{DD} = 5V
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	t_{pHL}	7.5	11.0
Propagation delay (low to high)	t_{pLH}	8.0	15.0
Output fall time	t_{THL}	7.0	12.0
Output rise time	t_{TLH}	9.5	36.5

Circuit Schematic Diagram



NOTES:
1) rail separation: 58 λ
2) bounding box (x ×y): 88λ ×73λ

3) Reset propagation delay: 2.0 6.0
4) Reset fall time: 3.5 11.5

Table values from SPICE simulation

CAESAR file: $UW_VLSI_TOOLS/src/cellib/cmospw/dlatchr.ca
DB files: $UW_VLSI_TOOLS/src/cellib/cmospw/{dlatchr.att, dlatchr.sym}

Simulation Conditions

All CMOSPW standard cells were simulated by spice (a circuit simulator program). The circuits were generated from the layout artwork by first creating CIF (Caltech Intermediate Form) files from the layout. The circuit extraction program mextra was then used to extract the circuit from the CIF file. Following are some electrical data of the timing simulations.

Temperature parameter was set to 27°C

Control signal transition time was 5 ns

VDD = 5V GND = 0V

Output fall time (t_{THL}) was measured from 90% (4.5V) point to 10% point (0.5V) of the output signal.

Output rise time (t_{TLH}) was measured from 10% (0.5V) point to 90% point (4.5V) of the output signal.

Propagation delay (t_{pHL} or t_{pLH}) was measured from the control signal 50% point (2.5V) to the output signal 50% point (2.5V).

Output disable time from high level (t_{pHZ}) was measured as following. Output signal was pre-charged to high level (5V) at the beginning of the simulation. t_{pHZ} was measured from the control signal 50% point to the output signal 90% point, i.e. 0.5V swing.

Output disable time from low level (t_{pLZ}) was measured as following. Output signal was set to low level (0V) at the beginning of the simulation. t_{pLZ} was measured from the control signal 50% point to the output signal 10% point (0.5V swing).

Enable time to high level (t_{pZH}) was gained by setting the output signal to low level at the beginning and measured from the control signal 50% point to the output 50% point.

Enable time to low level (t_{pZL}) was gained by pre-charging the output signal to high level at the beginning and measured from the control signal 50% point to the output 50% point.

For all cells (except pads), fan out load = 1 means loading the cell with one inverter; fan out load = 10 means loading the cell with 10 inverters in parallel.

For the pads, fan out load of 1 means a loading of 5 pF while fan out load of 10 means a loading of 50 pF. During the measurement, output of each pad was load with 2 resistors: one 16.67K resistor from Vdd to output and one 5K resistor from output to Gnd.

Following is a list of the SPICE model parameter values (see SPICE reference manual for parameter description) that were used:

	PMOS	NMOS
LEVEL	2	2
VTO	-0.9	0.9
CGSO	4.0E-10	5.2E-10
CGDO	4.0E-10	5.2E-10
CGBO	4.0E-10	5.2E-10
RSH	95.0	20.0
CJ	2.0E-4	3.2E-4
CJSW	4.5E-10	9E-10
JS	1.0E-4	1.0E-4
TOX	5.0E-8	5.0E-8
NSUB	5.0E15	2.5E16
TPG	-1	+1
XJ	6.0E-7	8.0E-7
LD	5.0E-7	6.4E-7
UO	200	450
UCRIT	8.0E4	8.0E4
UEXP	0.15	0.15
UTRA	0.3	0.3
VMAX	5.0E4	5.0E4

NETLIST User's Guide

UW/NW VLSI Consortium

Department of Computer Science
University of Washington
Seattle, WA 98195

(This document is based on portions of the document "User's Guide to NET, PRESIM and RNL/NL," by Christopher J. Terman, Laboratory for Computer Science, M.I.T., Cambridge, MA 02139.)

To run NETLIST type

netlist *infile* [*outfile*] [-o] [-s#] [-d#,#] [-e#,#]

infile is the name of the NETLIST input file, if outfile is specified, that file is used for output. The options are:

-o	old format input. size specifications are taken to be length/width rather than width/length.
-ttech	Uses *tech* in the technology portion of the units/tech line at the beginning of the simulation file produced (Default is ???, unknown).
-uunits	Sets the number of centi-microns per lambda to *units* (Default is 250). **Warning:** The "units" set by this option appear in the comment line of the *.sim* file. This value is not used by PRESIM and does not influence an RNL simulation.
-s#	use specified number as initializer for internal node names; useful when you want to merge the results of separate NETLIST runs.
-d#,#	set the default width (first number) and length (second number) for depletion devices. The defaults are 8 and 2.
-e#,#	like -d except for enhancement devices. The defaults are 2 and 2.
-i#,#	like -d except for intrinsic devices. The defaults are 2 and 2.
-l#,#	like -d except for low-power devices. The defaults are 2 and 2.
-p#,#	like -d except for p-channel devices. The defaults are 2 and 2.

A NETLIST file can insert other NETLIST files by using the include com- mand:

include filename.net

The single argument must be a string (i.e., enclosed in quotes).

Available through NETLIST are all the regular built-in functions of RNL (i.e., a subset of standard LISP primitivies) -- see RNL documentation for a description of these subroutines. In addition, NETLIST offers some spe- cial functions useful for building a description of a transistor network. These functions are described below.

NETLIST is a macro-based language for describing networks of sized transistors. Names in NETLIST refer to nodes, which presumably get inter- connected by the user through transistors. A node name has two forms

(n width length)

n is the name of the network node, length and width specify a transistor size. This is used in NETLIST constructs where mention of a name causes creation of a transistor.

n

n is the name of the network node; when transistor sizes are required they are taken from the appropriate defaults

When using a name to refer to a node, it must first be "declared" (this allows typo's to be caught early on). Nodes are declared by using the node statement or the local statement (see below). The node statement looks like:

(node n1 n2 n3 ...)

where n1, n2, etc are the names to be declared. Note: when using structured names (see the repeat statement) only the first component has to be declared.

The interconnect capacitance associated with a node can be specified as follows:

(capacitance n 1.234)
(setq pf-sq-micron-of-diffusion 10e-4)
(capacitance n (13 pf-sq-micron-of-diffusion))*

The first argument is the name of the node, the second the capacitance in pf (must be a number).

A resistance can be specified as follows:

(resistor n1 n2 1500)

The first two arguments (n1 and n2) are the names of the nodes to which the resistor is connected, the last argument is the resistance value in ohms (must be a number).

An electrical node can be given several names by using the connect statement:

(connect n1 n2 n3 ...)

The names n1, n2, etc. will all refer to the same electrical node. This statement is useful for connecting i/o signals to the edge of an array generated by a repeat statement.

The voltage threshold for logic high and low states can be set by the NETLIST command threshold:

(threshold n 0.2 0.8)

would set the logic low threshold for node n to 0.2 (normalized voltage) and the high threshold to 0.8. If no threshold is specified, the node will be given the default thresholds as given in the configuration file for PRESIM (see PRESIM.DOC for details).

The "delay" of a node (the transition times for changes in the node's value) can be specified by user with the delay command:

(delay n plh phl)

where plh and phl are integers specifying the low-to-high transition delay and the high-to-low delay respectively. Delays are specified in RNL time units (1/10th nanosecond). If you do not specify a delay for a node, RNL will calculate one based on the impedence of the driving transistors and the

capacitance of the node; user-specified delays override the usual RNL calculation.

(ratio gate_ratio)

set a global parameter gate_ratio for use in cnand, cnor, cinvert and the transistors connected to the input signal in the clkinv. Default gate_ratio is 2.0.

Node interconnections are accomplished by one of the following NETLIST statements:

(trans g s d [w [l]])
(etrans g s d [w [l]])

enhancement mode transistor with gate g, source s, and drain d. l and w specify length and width of transistor (can be ommited).

(dtrans g s d [w [l]])

like etrans, except depletion mode transistor

(itrans g s d [w [l]])

like etrans, except intrinsic transistor

(ltrans g s d [w [l]])

like etrans, except low-power transistor

(ptrans g s d [w [l]])

like etrans, except p-channel transistor

(tgate out in node nodebar)

wires a CMOS tranmission gate from the signals node and nodebar. The order is alphebetical, the is the n type is gated by node and the p type by nodebar. The size of the p type device is set explicitly on the nodes node and nodebar not by the ratio commands. Additional arguments (more control) may be added but there must be an even (2N) number or it will complain. Nodes in and out are have their usual meanings.

(pullup a)

depletion-mode pullup (to vdd) of a.

(pulldown a n-1 ... n-k)

chain of k transistors from a to gnd, gates of transistor are n-1, ..., n-k.

(invert a b)

two-transistor NMOS inverter with output a and input b.

(cinvert a b)

two-transistor CMOS inverter with output a and input b. The size of the p type transistor is determined by the current value of ratio. See the command ratio for adjusting this value. Default 2.0.

(clkinv out in clk clk-)

CMOS clocked inverter. This function builds a clocked inverter from clk, clk- and in nodes. Clk gates the n type transistor and clk- the p type (just like tgate). The size of the p device gated by in is determined from the current value of gate_ratio (set by the ratio command). The size of the p device gated by nodes clk and clk- are set using standard node syntax and does not use the ratio command.

(nor a n-1 ... n-k)

pulls up a, and creates k transistors from a to gnd controlled by n-1 through n-k.

(cnor a n-1 ... n-k)

produces a CMOS nor gate with output a and inputs n-1 ... n-k. The node a is pulled up with a chain p type devices and is connected to gnd with the n type devices. Both sets are gated by the list of inputs. The size of the p type transistor is determined by the current value of ratio. See the command ratio for adjusting this value. Default 2.0.

(nand a n-1 ... n-k)

equivalent to

(pullup a)
(pulldown a n-1 ... n-k)

(cnand a n-1 ... n-k)

produces a CMOS nand gate with output a and inputs n-1 ... n-k. The node a is connected to Vdd through the p type devices and pulled down by chain of n type devices. Both sets are gated by the list of inputs. The size of the p type transistor is determined by the current value of ratio. See the command ratio for adjusting this value. Default 2.0.

(and-or-invert a (n-1 ... n-k) ... (m-1 ... m-l))

equivalent to

(pullup a)
(pulldown a n-1 ... n-k)
...
(pulldown a m-1 ... m-l)

Iteration construct is repeat statement:

(repeat index low high
[(local l-1 ... l-j)]
...)

where index will be given successive values starting with low and finishing with high. You can use the index in structured names, e.g.:

foo.index foo.(1+ index) foo.(1- index).bar ...

local variables are described under macros.

For ease of circuit entry, the user can build and call parameterized macros. macro definitions have the form

(macro n (p-1 ... p-k)
[(local l-1 ... l-j)]
...)

where n is the name of the new NETLIST function being created, p-1 ... p-k are the formal parameters, l-1 ... l-j are the optional local node names used in the body.

The macro is invoked as follows:

(n a-1 ... a-k)

which causes the body to be interpreted after

1) all occurrences of p-1 in the body have been replaced by a-1, etc.

2) all occurrences of l-1 in the body have been replaced by a new, unique node name. Unique names will be a number (like for anonymous nodes in pulldowns).

3.0 Examples

In the following examples

```
e g s d l w
```

specifies an enhancement-mode transistor with gate g, source s, and drain d with length l and width w.

```
d g s d l w
```

is similiar, except transistor is depletion mode.

Quickie examples:

```
(invert a b)
d a a vdd 8 2
e b a gnd 2 2

(invert a (b 17 5))
d a a vdd 8 2
e b a gnd 5 17

(invert (a 2 2) (b 2 4))
d a a vdd 2 2
e b a gnd 2 4

(nor (a 16 2) (b 2 4) c d)
d a a vdd 2 16
e b a gnd 4 2
e c a gnd 2 2
e d a gnd 2 2

(and-or-invert a (b c d) (e f) (g))
d a a vdd 8 2
e b a 1001 2 2
e c 1001 1002 2 2
e d 1002 gnd 2 2
e e a 1003 2 2
e f 1003 gnd 2 2
e g a gnd 2 2
```

Two dimensional array of foo's:

```
(repeat i 1 8 (repeat j 1 8 foo.i.j))
```

generates

```
foo.1.1  foo.1.2  foo.1.3  ... foo.1.8
foo.2.1  ...              foo.8.8
```

Simple two-inverter dynamic memory cell:

```
(macro bitcell (output output-enb input input-enb refresh)
      (local a b c)
      (trans input-enb input a 2 4)
      (invert b a)
      (invert (c 2 2) (b 2 8))
      (trans refresh a c)
      (trans output-enb c output 2 4)
)

(bitcell bit0 renb bit0 wenb phi2)
```

generates

```
e wenb bit0 1001 4 2
d 1002 1002 vdd 8 2
e 1001 1002 gnd 2 2
d 1003 1003 vdd 2 2
e 1002 1003 gnd 8 2
e phi2 1001 1003 2 2
e renb 1003 bit0 4 2
```

Assume you had an alu bit-slice macro of the following form

```
(alu carry-in operand1 operand2 result carry-out)
```

then the following macro would produce an n-bit alu:

```
(macro ALU (n databus1 databus2 resultbus cin cout)
      (connect cin cout.0)
      (repeat i 1 n
(alu cout.(1- i) databus1.i databus2.i resultbus.i cout.i))
(connect cout cout.n)
)
```

Instead of using the connect statement one could have conditionalized the calculation of the arguments to alu:

```
(macro ALU (n databus1 databus2 resultbus cin cout)
(repeat i 1 n
(alu (cond ((== i 1) cin) (t cout.(1- i)))
      databus1.i
      databus2.i
      resultbus.i
      (cond ((== i n) cout) (t cout.n))) ))
)
```

The file /usr/vlsi/nl/pads.net contains the following macros:

```
(input-pad world)                  ; the input pad
(output-pad world in)              ; the output pad
(tristate-pad world in direction)  ; the tristate pad
(clockbar-pad world ~phi1 ~phi2)   ; the clock pad
```

PRESIM User's Guide

UW/NW VLSI Consortium

Department of Computer Science
University of Washington
Seattle, WA 98195

(This document is based on portions of the document "User's Guide to NET, PRESIM and RNL/NL," by Christopher J. Terman, Laboratory for Computer Science, M.I.T., Cambridge, MA 02139.)

One must first convert the .sim file to a network file suitable for use by RNL or NL -- to do this we run PRESIM:

presim foo.sim foo [config] options...

which converts the file foo.sim into a binary file for RNL/NL called foo.

The -g option:

> Suppresses the sum-of-products formation. This may be desired if you think sum-of-products is formed wrong otherwise the advantages of the transistor and node reduction make this option unattractive.

The -c option:

-cfile,minvalue

writes a list of node names and capacitances to the specified file. Only capacitances larger than minvalue will be included.

The -t option:

-tfile,minvalue

writes a list of transistors and RC values to the specified file -- there are two entries for each transistor. The R's come from the size of the transistor, C's from the source/drain capacitance. Only RC values larger than minvalue will be included.

The -p option:

-presist,voltage

provides a worse-case estimate of the circuit power consumption by assuming that all the pullups (DEP or LOWP devices with drain=VDD) are all on simultaneously. "Voltage" specifies the supply

voltage, for example "-p5" specifies a VDD of 5 volts. The result is printed after PRESIM completes its other processing. When figuring the resistance of a pullup device the "power" characteristic resistance as set in the config file is used.

Presim's results are dependent on a number of parameters that vary with the technology used. A set of variables is built into Presim that allow calculations to proceed when the optional config file is not present, but you should realize that these values do not correspond to any particular process. The config file can be used to override these built-in values. The correct format for configuration files is given with the following example. This config file contains the default parameter values. The "lambda" parameter specified in this file is ignored for .sim files in the UCB format. UCB .sim files have their dimensions specified in centimicrons. The "units:" parameter in MIT format .sim files is ignored by presim, see the Netlist users guide for details.

(Resistor values not explicitly provided in the configuration file are estimated by linear interpolation. The resistor values are stored, sorted first by width, then by length not by the ratio.)

parameter value comments...

Lines beginning with ";" are treated as all comment. The parameter names and their default values are:

```
; configuration file for "standard" MPC process

capm2a    .00000    ; 2nd metal capacitance -- area, pf/sq-micron
capm2p    .00000    ; 2nd metal capacitance -- perimeter, pf/micron
capma     .00003    ; 1st metal capacitance -- area, pf/sq-micron
capmp     .00000    ; 1st metal capacitance -- perimeter, pf/micron
cappa     .00004    ; poly capacitance -- area, pf/sq-micron
cappp     .00000    ; poly capacitance -- perimeter, pf/micron
capda     .00010    ; n-diffusion capacitance -- area, pf/sq-micron
capdp     .00060    ; n-diffusion capacitance -- perimeter, pf/micron
cappda    .00010    ; p-diffusion capacitance -- area, pf/sq-micron
cappdp    .00060    ; p-diffusion capacitance -- perimeter, pf/micron
capga     .00040    ; gate capacitance -- area, pf/sq-micron

lambda    2.5   ; microns/lambda (conversion from .sim file units
                ; to units used in cap parameters)

lowthresh 0.3   ; logic low threshold as a normalized voltage
highthresh 0.8  ; logic high threshold as a normalized voltage

cntpullup 0     ; <>0 means that the capacitor formed by gate of
                ; pullup should be included in capacitance of output
                ; node

diffperim 0     ; <>0 means do not include diffusion perimeters
                ; that border on transistor gates when figuring
                ; sidewall capacitance (*)

subparea 0 ; <>0 means that poly over transistor region will not
           ; be counted as part of the poly-bulk capacitor (*)
```

```
diffext 0   ; diffusion extension for each transistor, i.e., each
            ; transistor is assumed to have a rectangular source
            ; and drain diffusion extending diffext units wide and
            ; transistor-width units high. The effect of the
            ; diffusion extension is to add some capacitance to
            ; the source and drain node of each transistor --
            ; useful when processing the output of NET to improve
            ; the capacitive loading approximations without adding
            ; explicit load capacitors. diffext is specified in
            ; lambda (it will be converted using the lambda factor
            ; above).

resistance channel context width length resist
; this command specifies the equivalent resistance for a transistor
; of type channel with the specified width and length. Transistors
; matching this entry will have the specified resistance; linear
; interpolation is done if the width and/or length is not matched
; exactly.
;  channel is one of "enh", "dep", "intrinsic", "low-power",
;                    "pullup", or "p-chan"
;  context is one of "static", "dynamic-high", "dynamic-low", or "power"
;  width is given in lambda
;  length is given in lambda
;  resist is given in ohms
```

(*) These paramters should be 1 only when processing the output of the node extractor. They cause various corrections to be made to the interconnect component of a node's capacitance -- usually only extracted .sim files have information regarding interconnect capacitance.

PRESIM uses these parameters in calculating the capacitance for each electrical node and the resistance for each transistor channel.

The location of worst case config files for different technologies can be found in the technology manual page (use the UNIX command **man technology 5**).

RNL 4.2 User's Guide

UW/NW VLSI Consortium
Sieg Hall, FR-35,
University of Washington,
Seattle, WA 98195

(This manual documents version 4.2 (UW) RNL. The manual is based on Chris Terman's manual of similar title.)

1. INTRODUCTION

RNL is a timing logic simulator for digital MOS circuits. It is an event driven simulator that uses a simple RC (resistance capacitance) model of the circuit to estimate node transition times and to estimate the effects of charge sharing. The user interface is a simple LISP interpreter. This allows both interactive simulation and the programming of complex simulations. See Chapter 2 of "Simulation Tools for LSI Design" by C. Terman for details of the algorithm. A short introduction to the model is included in the "Theory of Operation" section of this guide.

The version of RNL described herein is version 4.2 as distributed by the UW/NW VLSI Consortium. It differs from previous versions in that it is considerably faster for many simulations. In addition the user interface has been augmented.

To use RNL, one needs .sim file for the circuit to be simulated. This can be extracted from the mask file (e.g., CIF) or developed using NETLIST, a program that processes textual schematics.

The first step is to convert the .sim file to a network file suitable for use by RNL by running PRESIM:

% presim foo.sim foo [confile] [-cfile,min] [-tfile,min] [presist,voltage] < CR>

Presim converts the file "foo.sim" into a binary file for RNL called "foo". The other parameters are optional and are described in detail elsewhere. The conversion process involves the computation of the effective resistances of the transistors as well as the capacitances of the circuit nodes. In order to have a consistent estimation of capacitances we reccommend that if you are using the circuit extractor **mextra** that you use the "-o" option to force the program to output the dimensions of the circuit nodes rather than to estimate their capacitances. (See the PRESIM documents for information on options and sections of this manual on calibration.)

To invoke RNL, either type

% rnl <CR>

or

% rnl cmdfile <CR>

If the "cmdfile" argument is provided then it should be the name of a file that contains a sequence of RNL commands. At the very least this command file should load one or more libraries of standard functions and should read in the binary description of the circuit prepared by PRESIM. For all but the simplest of circuits the command file will also contain commands and the definitions of LISP functions written by you to help in your simulation by performing such tasks as test vector generation and the simulation of the environment in which your circuit is designed to operate.

A minimal command file would contain the commands:

```
(load "uwstd.l")
(load "uwsim.l")
(read-network "foo")
```

where the file *"foo"* was prepared from a ".sim" file by PRESIM.

When the end of the command file is reached, input is taken from the standard input.

2. RNL THEORY OF OPERATION

It is not necessary to have a detailed understanding of the internal computations of RNL in order to begin to use it. It is, however, useful to have a general idea of what is going on. In particular, this section will be helpful when reading the discussion on some of the limitations of RNL's circuit model. The rest of this section is a discussion of RNL's internal computations and is quoted directly from C. Terman's original RNL User's Guide.

The RNL simulator are designed to handle ratioed logic, bidirectionality, and charge sharing/storage. They can be used to determine the functionality and approximate timing behavior of circuits commonly found in digital MOS designs.

RNL uses the following simple recipe for simulating a circuit. Recall that PRESIM has established the capacitance of each node and the size of each transistor. (The network extractor written by C. Baker automatically derives both from the mask files; if the network is derived from a NETLIST description, the user must explicitly specify the interconnect capacitance for nodes where it is important.)

Once input values have been assigned by the user, RNL calculates the effects of the new values by repeating the following operations until no further nodes change values:

(1) when nodes are added to the network (the result of some transistor turning on), compute the "charge sharing" implications of the new node's capacitance and logic state on its electrical neighbors.

(2) for each node that might be affected calculate V_{thev} and R_{thev}, the parameters for the Thevenin equivalent circuit. The new logic state of the node is determined from V_{thev}.

(3) if the node has changed state, calculate the transition time using the node's capacitance.

(4) propagate changes (if any) to other nodes.

Basic to the operation of the simulators is the notion of an event -- an event specifies (i) a node in the network, (ii) a new logic state, and (iii) a time at which the node's value is changed to the new logic state. RNL maintains a list of events, sorted by time, that tells what processing remains to be done. Whenever the user changes an input, an event is added to the list; when the list is empty the network has "settled" and RNL waits for further input.

When started with an initial list, RNL sequentially processes the next event on the list, stopping (1) when the list is empty, (2) when a node the user is tracing changes value, or (3) when the specified amount of simulated time has elapsed. Processing an event entails

(a) removing the event from the event list.

(b) changing the node's state to reflect its new value.

(c) calculating any consequences, i.e., new events, resulting from the node's new value. First all nodes that might be affected by the change are found and marked -- this includes the source and drain nodes of transistors with the current node as a gate, and all nodes connected to these nodes by conducting transistors (the search through the network stops only when an input or a non-conducting transistor is reached). For each marked node two calculations are made: first a "charge sharing" calculation is performed (see 2.1) to model changes of state due to charging/discharging of the node capacitances. Second, a "final value" calculation is done (see section 2.2) to determine the nodes ultimate logical state.

Since nodes are only added to the event list when their values change, portions of the circuit unaffected by the current set of changes to the inputs are not re-evaluated -- the algorithm is event-driven (sometimes called selective trace).

A node can have up to two events pending:

(1) a "charge sharing" event describing an immediate change in the node's state due to the redistribution of charge among the capacitors for nodes on the connection list. This type of event is only generated when a node is added to a subnetwork (i.e., when a transistor turns on).

(2) a "final value" event describing what the final, driven state of the node will be.

The simulation computation computes both types of events for each node and then does the following:

(a) when a new charge sharing event is scheduled, throw away pending events of either flavor. If the new charge sharing event is for the same value that the node currently has, it can be thrown away too, i.e., the node wil end up with no events pending.

(b) when a new final value event is scheduled it will be ignored if

(i) there is a pending final value event for the same value which is scheduled to happen at an earlier time than the new event. If this test fails, any pending final event is discarded, and the remaining conditions checked.

(ii) there is a pending charge sharing event for the same value as the new final value event.

(iii) there is no charge sharing event and the new event is for the same value that the node currently has.

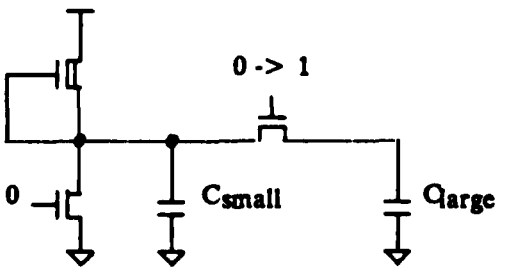
If none of the tests are successful, the new final value event is added to the event list.

These rules are based on the observation that the event that was last calculated reflects the latest network configuration and hence should override events calculated earlier. Charge sharing events throw away final value events since the charge sharing calculation is immediately followed by a new final value calculation.

The next two sections describe the two parts of the simulation computation.

2.1. Charge sharing computation

This portion of the simulation calculation tries to model various capacitive effects that happen when two (or more) previously unconnected nodes become connected. For example:



In this circuit the transfer gate has just turned on, connecting a bus (represented by C_{large}, initially at

logic low) with an inverter whose output is a logic high. If C_{large} and the pass gate are large enough, the inverter output will go low (C_{small} is discharged) initially, but eventually both the inverter output and bus will go to a logic high. In RNL, this sequence of events happens in two steps: a charge sharing calculation that predicts the first transition, and a final value calculation that predicts the first transition, and a final value calculation that predicts the ultimate state.

The charge sharing computation proceeds as follows:

(1) set $C_L = C_X = C_H = 0$;

(2) compute connection list: starting with current node, include all nodes in the network that can be reached via non-off transistors (includes transitors with gates with logic state "X"). For the charge sharing calculation, depletion transistors are considered to be "off" since they (usually) represent a high impedance connection over which charge sharing would happen very slowly.

(3) visiting each node on conection list, calculate summary capacitances (C_L , C_X , C_H: each node contributes to the sum corresponding to the node's current state). Actually steps (2) and (3) can be merged into a single computation.

(4) compute initial state:

$$\text{INITIAL STATE} = \begin{cases} 1 & C_H/(C_L + C_X + C_H) > V_{high} \\ 0 & (C_H + C_X)/(C_L + C_X + C_H) < V_{low} \\ X & \text{otherwise} \end{cases}$$

For each node on the connection list, schedule a transition to the initial state with zero delay -- this event may be ignored under the conditions described at the end of the previous section.

V_{high} is the logic high threshold of the node, V_{low} the logic low threshold; these can be set separately for each node or one can use the default setting (see NETLIST and PRESIM documentation).

Note that although the computation could be made node-by-node, groups of electrically connected nodes are dealt with as a whole since their events are obviously related.

2.2. Final value computation

After the charge sharing calculation is done, RNL revisits each group of affected nodes to compute their final values. As we saw in the example of the previous section, a node's ultimate value may differ from its charge-sharing value.

The final value computation computes two peices of information about each node.

(1) its final logic state. Recall that the transistor network containing the node is being modeled by an equivalent resistor network. To determine the logic state of a node, RNL computes the Thevenin equivalent for the node in question from the modeling resistor network (more on how this is done below) -- the Thevenin equivalent voltage is used to calculate the final logic state of the node.

(2) if the node value is changing, an estimate of the transistion time is needed. If the transition is from high to low, RNL computes the effective resistance to GND for the node (R_{GND}) and then calculates the trasnsition time as R_{GND} *(capacitance not already at GND). A similar calculation is made for low to high transitions. Transitions to X are defined to take the same time as the shorter of the high-to-low and low-to-high transitions.

The following subsections deal with each part of the final value computation.

2.3. Network analysis

This section outlines how the Thevenin equivalent circuit for a given node is calculated from a larger network. We start by describing the simple transistor model, show how the Thevenin equivalent is derived using information about each transistor, and end by showing how the new value of a node is computed.

The transistor model in RNL can be quite simple since it is only used to predict the final logic state of a node and how long each state transition takes. Although the channel resistances of the transistors change as their terminal voltages vary, it might be possible to use "average channel resistances" to characterize the transistors' behavior. RNL does just this -- transistors are modeled as resistors whose resistances are determined by the logic state of the transistor's terminal nodes and the type of transistor:

Rtransistor = (length/width) * type * state where

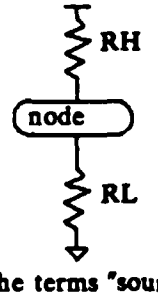
width, length	are the dimensions of the active transistor area.
type	is the average channel resistance per unit area for the particular type of transistor.
state	a scale factor that depends on the logic state of the transistor's terminal nodes.

The following table shows type*state for an enhancement transistor (V_g is the logic state of the gate node).

	V_g	type * state
enhancement	0	∞
	1	enh
	X	[enh, ∞]

where enh is the characteristic channel resistance of an enhancement device. When the state of the gate and/or source nodes of a transistor are X, the resistance of the transistor is also "unknown" and is specified by an interval.

We can now describe how V_{thev} and R_{thev} for a given node can be calculated from a network of nodes and transistors. The network analysis subroutine does a tree walk of the network returning the values of the two resistors, R_H and R_L, that make up the characteristic voltage divider for a node:

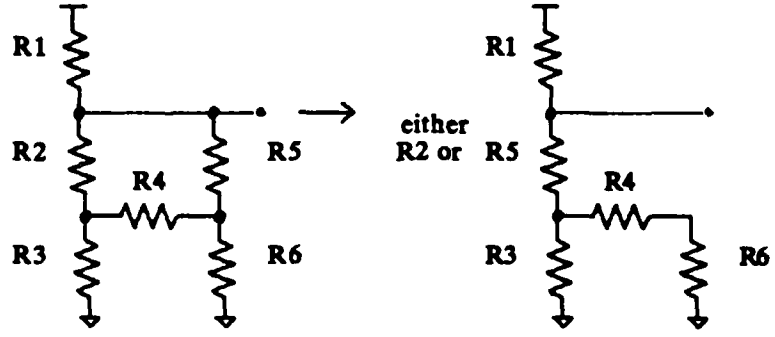


The subroutine is outlined below. The terms "source" and "drain" are used to distinguish between the two terminal nodes of a transistor and do not imply anything about their relative potential.

```
if node is a logic low input {
return with RH = ∞ and RL = 0+
}else if node is a logic high input{
 return with RH = 0+ and RL = ∞
```

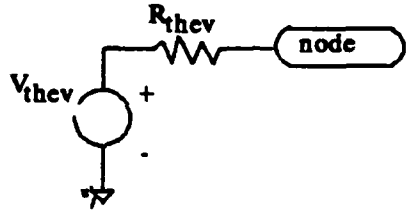
```
}else{
 local_RH: = local_RL : = ∞
 mark current node
 for each "on" transistor L with source connection to current node{
  if drain node is not marked{
   recursively analyze drain node
   derive a voltage divider that approximates the effect of the
    drain node on the current node (approximation uses RH and RL
    for the drain node and the equivalent resistance for t)
   parallel approximating voltage divider with local_RH and local_RL
  }
 }
 return with RH = local_RH and RL = local_RL }
```

Cycles are avoided by marking each node as it is visited: this keeps the tree walk expanding outward from the starting node. If the network does contain cycles, the subroutine only approximates the true resistance to VDD and GND. For example, consider the following gate where the output (the pulled-up node) is the node of interest:



In the circuit on the left the pulldown path contains a cycle; RNL treats the cycle as if the circuit looked as shown on the right. This approximation avoids having to solve a system of equations at simulation time; fortunately, very few networks actually contain such cycles. It is also worth remembering that the resistor network is itself only an approximation -- it is not worth a large investment of computation time to calculate an exact equivalent to the resistor network.

The final state of a node can be characterized by a voltage source with a series resistor, i.e., the Thevenin circuit equivalent for all pieces of the network that influence the value of the given node.



V_{thev} a voltage interval $[V_-, V_+]$ in the range [0,1] specifying the possible voltages the output node may have.

R_{thev} a resistance interval $[R_-, R_+]$ in the range $[0+, \infty]$.

V_{thev} and R_{thev} are, in general, intervals since the equivalent transistor resistances from which they are derived might themselves lie in an interval. Using the values returned by the network analysis subroutine, we have:

$$V_- = \frac{RL_-}{RL_- + RH_+} \quad \text{and} \quad V_+ = \frac{RL_+}{RL_+ + RH_-}$$

$$R_{thev} = RL_+ \,||\, RH_+$$

Because we are interested only in the worst case determination of the voltage level, we need only consider the worst case Thevenin resistance. As will be seen in the next section, the final value is directly related to V_- and V_+.

Different values for Rthev lead to different conclusions about the state of the node:

input ($R_{thev} = 0+$). Node is designated input node (e.g., VDD or GND). The value of input nodes can only be changed by explicit simulator commands -- the assumption is that they supply enough current to be unaffected by connection (even shorts to other inputs) made by transistors. *driven*. ($0+ < R_{thev} < \infty$). Node is part of a voltage divider between two inputs, i.e., it is connected by transistors to other driven or input nodes. As will be seen below, the logic state of a driven node is determined by Vthev. *charged* ($R_{thev} = \infty$). Node is connected, if at all, only to other charged nodes. Charged nodes will maintain their current logic state until either (1) reconnected to some other part of the network, or (2) a user-specified decay interval elapses at which time logic state changes to X.

2.4. Calculating transition delays

Once the final logic state of a node has been determined using the Thevenin equivalents, RNL calculates transition times using one of the followig characteristic resistances calculated for each node:

R_{GND} the effective resistance of all direct paths to GND. A simple serial/parallel calculation is used to determine R_{GND}.

R_{VDD} the effective resistance to R_{VDD}, computed in a similar fashion.

These two resistances can be calculated at the same time as the Thevenin equivalents; the network analysis routines actually return four values: the intervals RL and RH, and the values RGND and RVDD. The calculation proceeds as follows:

(1) set *inputseen* = false, $C_L = C_X = C_H = 0$;

(2) calculate connection list and summary capacitances (C_L, C_X, C_H: each node contributes to the sum corresponding to the node's current state). If an output node is reached during the construction of the connection list, set *inputseen* = *true*.

(3) if *inputseen* is false, schedule a decay transition for each node on the connection list, then exit.

(4) if *inputseen* is true, for each node on the connection list:

(a) if node is an input, continue with next node on list.

(b) calculate V_{thev} (also R_{GND} and R_{VDD} to be used later).

(c) compute the node's final state:

$$\text{FINAL STATE} = \begin{cases} 1 & V > V_{high} \\ 0 & V_+ < V_{low} \\ X & \text{otherwise} \end{cases}$$

and the effective capacitance and resistance

$$C_{eff} = \begin{cases} C_L + C_X & \text{if final state = 1} \\ C_H + C_X & \text{if final state = 0} \\ min(C_L + C_X, C_H + C_X) & \text{otherwise} \end{cases}$$

$$R_{eff} = \begin{cases} R_{VDD} & \text{if final state = 1} \\ R_{GND} & \text{if final state = 0} \\ min(R_{GND}, R_{VDD}) & \text{otherwise} \end{cases}$$

(d) schedule a transition to the final state with a delay of $R_{eff} * C_{eff}$ nanoseconds, or use user-specified delay if present.

Note that the effective capacitance, C_{eff}, depends on the summary capacitances, not just the capacitance of the node in question. This means that none of the connected nodes will reach its final value much before the others.

2.5. Calibrating the model

The charge sharing calculation described in section 2.1 depends only on the capacitance associated with each node. These capacitances are specified by the designer as part of the NETLIST description or in the PRESIM parameter file, both of which are described elsewhere in this document.

The final value computation uses both the node capacitances and resistance information about each transistor. The circuit data base contains the size and type of each transistor -- what the designer must provide in addition is the characteristic resistance for each type of channel (i.e., the resistance of a square transistor of that type). See the description of the set-params subr in section 7.5 for how this information is specified to RNL.

Actually, RNL uses three characteristic resistances:

a *static resistance* used in calculating RH and RL. a *dynamic-low resistance* used in calculating the resistance of paths to GND. a *dynamic-high resistance* used in calculating the resistance of paths to VDD.

A single characteristic resistance won't suffice: RNL uses resistances to determine both the voltage level and transition times -- a resistance value that gives an accurate estimate of the voltage level may not necessarily result in good transition time estimates. Thus, "static" resistances used for voltage level calculations can be specified separately from "dynamic" resistances used for transition time calculations.

There are two sets of dynamic resistances: dynamic-high resistances used when calculating the resistance of paths to VDD, and dynamic-low resistances used when calculating resistance to GND. Ordinarily, these are set to the same value for a particular type of transistor; some useful exceptions:

(1) setting the dynamic-low resistances very high for devices which should not appear in pulldown paths. The very high transition times that result will serve to flag "strange" circuits.

(2) setting the dynamic-low resistance for enhancement devices to be appropriate for pulldowns, while setting their dynamic-high resistance to be correct for source-follower configurations.

(3) since pullups are treated as separate types, the dynamic-high resistance for depletion devices can be set for a source-follower configuration.

Future versions of the RNL will distinguish between different circuit contexts for the same type of device: e.g., enhancement devices would be classed as ordinary, pulldown, source-follower, transfer, etc. Having the ability to set separate dynamic resistances for a transfer device (for example) means that the transition times for high-going and low-going transitions involving the transfer device can be much more accurate.

The static resistances can be estimated from measurements (actual or SPICE'd) of the low threshold for standard logic gates -- there is considerable flexibility since there are many more adjustable parameters than are needed. Dynamic resistances can be estimated by measuring high- and low-going transition times of standard circuit configurations and choosing the characteristic resistances to give an R-C time constant equal to the time the actual waveform takes to cross the desired threshold.

3. RNL CALIBRATION VIA THE PRESIM CONFIG FILE

Provided here is a brief description for setting the parameters in the presim configuration file. This is not the only way to obtain these values but the scheme does provide some consistancy between analysis models like those used in SPICE. Throughout it is assumed that the Presim User's Guide has been consulted and is available for parameter names, defaults etc.

3.1. Capacitance

There are three basic types of capacitance values that can be set by the use of the configuration file.

1) Capacitance from the area of the node interconnect. This case breaks down into 3 subcases; metal area (1st and 2nd layers), polysilicon area and diffusion area (both types in CMOS).
2) Capacitance from the perimeter of the node interconnect. Parameters for all layers are provided by presim.
3) Capacitance from the area of the gate regions of a node.

All capacitance is assumed grounded.

3.1.1. Area Capacitance

In NMOS the diffusion area capacitance can be estimated as directly proportional to the SPICE model parameter C_j with proportionality constant K_{EQ}. For abrupt junctions (a good approximation considering) K_{EQ} is given by,

$$K_{EQ} = 2\frac{\phi^{1/2}}{V_2 - V_1}(\sqrt{\phi + V_2} - \sqrt{\phi + V_1}).$$

$V_1 - V_2$ is the voltage range and can be assumed rail to rail. One must also be careful that the units are correct for presim (presim: pf/micron, SPICE: F/m). For a complete discussion of this approximation see "Analysis and Design of Digital Integrated Circuits," D. A. Hodges and H. G. Jackson, McGraw-Hill Book Co., New York, p. 137.

Similarly in CMOS, one uses the C_j for the two types of diffusion n^+ and p^+. The contribution of metal and polysilicon areas can be assumed to be an order of magnitude smaller that diffusion. As of yet we have no experience with second layers of poly and metal.

3.1.2. Perimeter Capacitance

For the diffusion perimeter contributions one uses the values for CJSW provided in the SPICE models. Warning, get the units (presim: pf/micron, SPICE: F/m) correct!

3.1.3. Gate Capacitance

Gate capacitance can be estimated from ratio of the silicon permittivity and the oxide thickness times the area of the active gate,

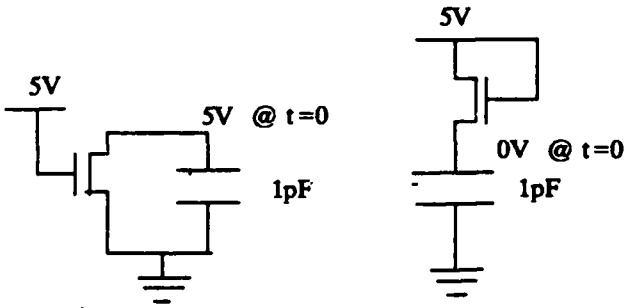
$$C_{gate} = \frac{\epsilon_{Si}}{t_{ox}} LW$$

Again one must be aware to the difference in the units used in presim and SPICE.

3.2. Resistance values

Establishing the resistance values is much more complicated. As the circuit elements (e.g. static logic, plas etc.) have a direct bearing on the representation of transistors by the resistors R_{dynlow} and $R_{dynhigh}$.

In most cases it is sufficient to perform circuit analysis on single transistors charging a fixed capacitive load. Two examples that should be included in any suite are shown below.



The type of transistor used in these experiments would vary from depletion, p type enhancements or n type enhancements. Of course the gate voltages and must be adjusted for these various transistor types.

The resistance value is defined then as,

$$R = \frac{\delta t}{C_{load.}}$$

This is in effect inverting the calculation done by RNL. The R values can be computed for the all types and the typical sizes (length and width) of transistors used in the circuits to be simulated by RNL. This should be streesed, the table maintained by presim is indexed by type and the length and width independently not by the ratio $\frac{L}{W}$.

Interestingly as a practical matter using the above definition for the resistance includes some of the effect of the volatage dependent capacitance. This can be represented by the writing δt as

$$\delta t = R(C_{load} + C(V)_{parasitic}).$$

From this the ratio $\frac{C(V)_{parasitic}}{C_{load}}$ is the contribution to the resistance R from the voltage dependent capacitance.

4. SOME OBSERVATIONS ON THE LIMITATIONS OF RNL'S CIRCUIT MODEL

We begin this section quoting from the original User's Guide by C. Terman.

It should be remembered that the programs are based on a model of what actually happens. As with any model, there are likely to be discrepancies between the predictions of the model and what actually happens. The tools described here try hard to be conservative, i.e. give a pessimistic prediction -- but this can't be guaranteed. Thus, it's wise to acquaint oneself with how the models work and where their shortcomings lie; think of the tools as performing a calculation you could do by hand

(only a lot faster and with greater accuracy and consistency); for your own protection, don't treat the tools as black boxes.

With this warning in mind it will be assumed that the reader has acquainted themselves with the model. The basics are provided in the Theory of Operation section of this user's guide.

As a practical matter RNL provides sufficient parameters for nodes and circuit elements to reproduce the overall behavior obtained from other more elaborate circuit analysis tools. However, it should be remembered that as the designer pushes the tolerances no simulation may reflect the physical device.

4.1. Propagation of X States

The main considerations for X state evaluation are;

1) Initial resistance values (R_{GND} and R_{Vdd}) for charging and discharging the capacitive load are assumed infinite.
2) In evaluating a network stage, transistors that are gated by nodes in state X are assumed to have a resistance represented by an interval and do not terminate the stage evaluation. This interval is included only in the Thevenin state evaluation and not in the resistance values used for estimating the delay times.
3) Recalling that an explicit estimate of transition times to the X state are not made. The transition time is defined to be the minimum of [tplh, tphl].

There are two quite different interpretations of X states. One is to consider it as some intermediate voltage, say 2.5V. This is inconsistent with condition 2) because some contribution to the delay time would be made with this as the gate voltage. Within the RNL model, X is best considered as an undefined voltage. Condition 2) is then a very conservative statement of what these undefined nodes contribute to delay times.

4.1.1. NMOS and CMOS Inverters

The effects of these conditions are highlighted by the propagation of an X state through NMOS and CMOS inverters. In both cases the state calculation reaches the correct answer that the output should also make a transition to X. The transition times for the two cases are now considered separately.

- For a NMOS inverter the transition time to logic H is independent of the inverter input (i.e. it uses a depletion pullup) and it reports this as the transition time $R_{Vdd} * C_{load}$.
- In the CMOS case however, both logic H and logic L transition times depend upon the inverter input. Then from condition 2), in CMOS no contributions are made to lower the transition time from infinity. This leads to a rather unrealistic estimate for the delay time on the output.

On a node by node basis RNL does provide the capability to override the RC time constant. By explicitly setting the rise and fall times for a node, transitions to X are propagated with minimum of the two values. By its nature this solution removes one of the attractive features of RNL, the dynamic evaluation of signal delay times. Moreover, in the cases where there are more inputs to the output node (e.g. a nand) this effect can be difficult to track down. Let a word to the wise be sufficient.

4.2. Node Overdrive

Important considerations for the following discussion;

1) The replacement of transistors with resistors is independent of the logic thresholds declared for the transistor terminals.
2) Node states are determined by comparison of Thevenin resistance ratios with node logic values V_{low} and V_{high}.

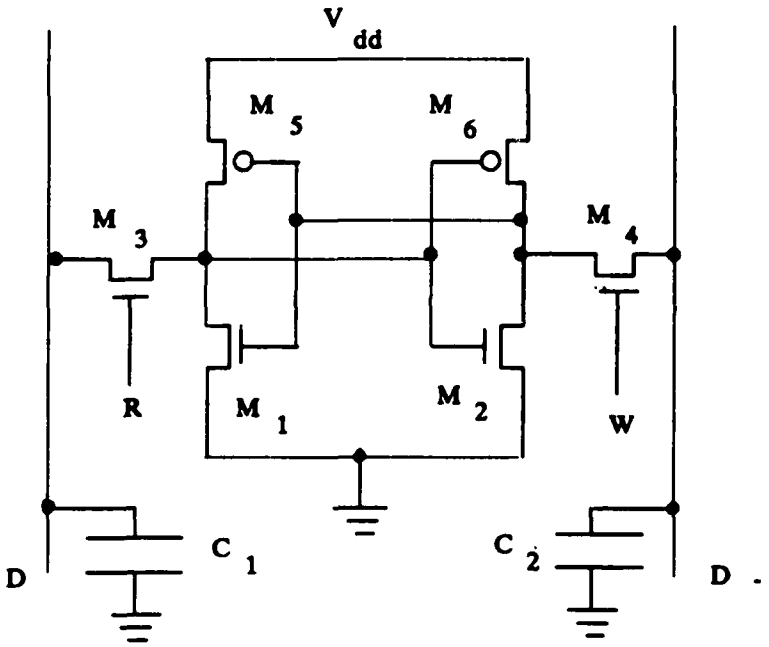
When two (or more) charging elements are driving a single node, accurate modeling of node overdrive guarantees the right element wins. Node overdrive is used more frequently in CMOS design

and should be of particular interest to those designers. Using this property in a digital circuit introduces significant dependence on analog properties of the circuit elements and nodes. In such cases it is suggested that an analysis tool such as SPICE be used to characterize the behavior of the subcircuit. With the SPICE results one can use several equivalent approaches to modeling the subcircuit with RNL. The following approach is trail and error but does not require that the device sizes be changed from those analyzed by SPICE.

From the theory of operation section of this user's guide recall that the final state of a node is determined by comparison of the Thevenin resistance ratios to parameters V_{low} and V_{high}. Furthermore, these parameters can also be set on a node by node basis. Values can then be found that reproduce the SPICE behavior for the node of interest. Depending upon the translation from transistors to resistors, the range V_{low} to V_{high} can be quite narrow. Again success of any of these methods depends on the tolerances in the design and all can be made to fail.

4.2.1. Memory cell

An example of node overdrive that is commonly found in NMOS and CMOS designs is the 6 transistor memory cell. Only the CMOS example is shown here but the technique presented here works equally well for the analogous NMOS design.



We include from Hodges and Jackson ("Analysis and Design of Digital Integrated Circuits," D. A. Hodges and H. G. Jackson, McGraw-Hill Book Co., New York, p. 380.) a discussion on the sizing of the transistors in such a memory cell.

> *To read a 1, D and Dbar are initially biased at about 3V. When the cell is selected, current flows through M_4 and M_2 to ground and through M_5 and M_3 to D. The gate voltage of M_2 does not fall below 3V, so it remains on. However, to avoid altering the state of the cell when reading, the conductance of M_2 must be about three times that of M_4 so that the drain voltage of M_2 does not rise above V_T. The operations of writing and reading a 0 are complementary to those just described.*

Such conservative design style should provide the designer with a working circuit without appeal to detailed analysis. Optimization, however, of the memory cell would be difficult to accomplish with RNL.

Simulating circuits of this type brings into focus an important conceptual difference between RNL and analysis tools. In RNL the logic threshold voltages V_{low} and V_{high} are declared independently from the replacement of transistors with resistors. In the memory cell this independence means that RNL could find transistor sizes that predict correct behavior for any set logic threshold voltages.

Design of the 6 transistor memory is by no means the only case where the independence of the logic thresholds and transistor replacement can provide misleading results. As a rule subcircuits with analog properties should be analyzed using other tools first and then RNL parameterized to fit that behavior.

4.3. Floating Nodes

Important considerations for this discussion are;

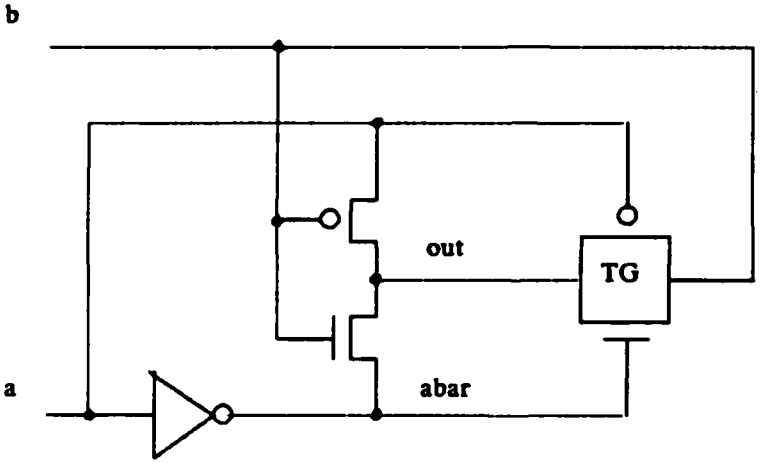
1) The effects of a charge sharing event are evaluated immediately.

2) Only one charge sharing and one final state event can be pending for each node in the circuit.

Floating nodes present another case where the analog properties dominate the prediction of the behavior of the subcircuit. In many cases a floating node can have several inputs making it difficult to find a set of RNL parameters for the nodes involved. The following example demonstrates these difficulties.

4.3.1. Exclusive or

This is a CMOS design of an exclusive or where all transistors can be of minimum size. It requires the transmission gate (TG) to compensate for the poor transmission of a logic low by a p type enhancement and conversely logic high by a n type enhancement.



Initially it was thought that **out** would require the most attention in the parameterization. Naive RNL simulation provided reasonable predictions for the cases where TG contributed to the final state of **out**. The states that utilize the TG are indicated in the state table below. RNL did not predict proper behavior in other cases. For example, when **b** was high and **a** changes from low to high, **out** was predicted to be X. This was not the result of SPICE analysis. It was also found that correct predictions for the other states could only be obtained when nodes **a** and **b** were set at the same time in the simulation. Another alternative was to declare **abar** as an input. This evidence suggests that the other nodes (e.g. **a, abar, b**) should be the focus. Why?

a	b	out	
0	0	1	(TG)
0	1	0	(TG)
1	0	1	
1	1	0	

In the first case the simultaneous transition of **a** and **b** produce the proper final state because the result of charge sharing is the correct final state. Secondly, declaring any node an input (in this case **abar**) provides it with zero impedance for that state. For the duration of it being an input, this excludes any possible state changes including the offending charge sharing event.

The difficulty in simulating this circuit with RNL centers around the fact that the node **out** is controlled by the nodes that are also its inputs, namely **a** and **abar**. Transitions of either node can promote the scheduling of both charge sharing and final state transistions. The effects of the charge sharing event are by design evaluated immediately. Without careful consideration of the V_{low} and V_{high} for the nodes **a** and **abar**, the charge sharing events can result in a final state of X for node **out**. The correct final state for **out** is then discarded by RNL (only one charge sharing and final state change can be pending) and is replaced by a transition to X.

4.3.2. Summary

From the examples presented RNL, used with some care, does have the ability to reproduce many of the results obtained from other analysis tools. Due to the independence of transistor replacement and node logic voltages detailed analysis of subcircuits should be done when analog properties dominate the subcircuits behavior.

4.4. Problems With Circuit Initialization.

The functions **sim-init** and **switch-init** operate by walking through the network and for each node that is in the **X** state scheduling on the event queue an immediate transition of that node to the **0** state. A subsequent advancement of the simulated time will allow these transitions to occur and their effects to propagate. This is especially useful in circuits that contain storage elements that cannot be controlled by the inputs to the circuit. A typical example is a divider circuit composed of a chain of latches and whose only input is a clock line. Because RNL interprets **X** as the "unknown" rather than as the "intermediate" state, a flip-flop which contains **X**'s will remain "unknown" rather than go through a transition from a meta-stable to a well-defined state. (Circuit ananysis simulators such as SPICE have their own problems when it comes to resolving mets-stable states.)

While it is sometimes necessary to use **sim-init** or **switch-init**, there are cases in which these functions will be unreasonably expensive, taking hours or even days to initialize the circuit. This is especially true if **sim-init** is called before any other attempts are made to initialize the circuit. This phenomenon has two realted causes:

Sim-init schedules its (nominally simultaneous) transitions in the order that it finds the nodes in the node hash table. This is in contrast to scheduling nodes closer to inputs first, or other related schemes. This can result in a lot of extra events being scheduled and evaluated. A typical example is when an output of a NOR gate is initialized before its inputs. The effects of the output being in a **0** state are computed and propagated even though the evaluation of the inputs will eventually put the output in a **1** state.

The evaluation of events during initialization can be very, very expensive compared to comparable events during the normal operation of the circuit. The reason for this is again the

interpretation of the X state. When RNL attempts to evaluate the effects of an event on some node A, it examines the states of all the other nodes connected to A. Since the interpretation of X is "unknown", a transistor whose gate is X might be on. If A is the source or drain of that transistor then it might affect the final state of A and therefore the computation must consider this. If the node A happens to be affected by many transistors (consider one line in a bus), and if the control lines gating those transistors are in state X, then each time any of the nodes connected to the bus through one of these "maybe on" transistors goes through a transition all of the nodes will be examined. This is exactly what happens when **sim-init** is invoked before the control lines are initialized. Eventually the control lines may be initialized, but by then the damage has been done.

The moral is that one should be careful when one attempts to initialize a circuit in RNL. Good design practice dictates that for testability purposes it should be easy and efficient to put a circuit into some known state by driving its inputs. One should attempt to use that initialization protocol to initialize simulated circuits also. The cases in which this does not seem to work are those in which there are embedded state machines that must already be in a well-defined state before they can be initialized. If the outputs of these sub-circuits control large parts of the chip then a special initialization protocol for these parts should be considered. Only after one has initialized what can be controlled from inputs and only after one has eliminated the X's on the major control lines of the circuit should one consider using **sim-step** to do the residual initialization.

5. USER INTERFACE

The user interface of RNL is a simple LISP interpreter. This is a brief introduction to that version of LISP.

The interpreter continually executes the following loop:

(1) read a command from current input;

(2) evaluate the command, performing the specified actions;

(3) print the result and loop back to (1).

There are two syntactic forms for specifying commands to this loop. The most general looks like

(function argument argument ... argument)

i.e., a list of names, numbers, etc. separated by white space (spaces, tabs, and newlines) and enclosed in parentheses. The parentheses delimit the command, so that the white space can be used to format the input any way one pleases. The arguments themselves may also be of the form (function arg ... arg). The interpreter first reads the entire command -- up to the closing parenthesis. The first element of the list is interpreted as a function. The arguments are then evaluated in left-to-right order and the results passed to the function. The value returned by the function is printed and the reader invoked once again. For example, given the following input

```
(* 17 (+ 3 2)
(/ 10 2))
```

RNL would respond by typing 425 and then wait for more input. Note that nothing happened after the first newline since the first parenthesis had not yet been closed.

The reader for the command interpreter also accepts commands of the form:

function argument argument ... argument <newline>

This is equivalent to

(function '(argument argument ... argument))

The "'" is shorthand for the **quote** special form. This keeps its argument from being evaluated. **Quote** is explained in more detail below. Many of the simple simulator functions contained in the file "uwsim.l" are written this way in order to eliminate the typing of parentheses when invoking common commands.

Comments can be included by preceding them with a semicolon (";"). All characters following the ";" up through the next newline are ignored.

5.1. Objects and Values

The RNL LISP interpreter allows you to access the following types of objects:

numbers -- signed integers. (16 bits on PDP11s, 24 bits on VAXen, 28 bits on PDP10s).

-- floating point. (the standard single precision format for the machine.

strings sequence of characters enclosed in quotes ("). Useful as constants for file names, print statements, etc. Special characters can be introduced into the strings by using the backslash escapes:

'\n ' newline

'\r 'return

'\t 'tab

'\ooo ' ascii code "ooo" where ooo are octal digits

symbols are like variables in other programming languages. A symbol is referred to by its *print name*: any sequence of characters (not including a period ".") delimited by "white space" that isn't a number or string. Special symbols (including white space and control characters) can be included in symbol names by using the backslash escape convention. Long symbol names with embedded blanks and all other special characters can be created by enclosing the name in a pair of vertical bars.

Example: |long symbol\014 name| defines a

long name with a form feed in the middle. Use long symbols in preference to strings.

nodes are the electrical nodes of your circuit. Although they may have names that resemble symbols, they are a distinct data type. Note that many nodes have print names that are numbers. Symbols and numbers are distinguished from nodes by the context in which they are used. In addition to numbers and symbols, nodes can have structured names of the form "a.b.c. ..." where each of a is a symbol and b, c, etc are symbols or numbers. This allows you to create arrays and hierarchical naming schemes for your nodes. It has the unfortunate side effect of forcing you to use the vertical bar convention to enter symbol names containing periods, i.e. |a.b.c|.

lists are sequences of objects enclosed in parentheses. Standard LISP syntax applies, including dot notation. The empty list "()" is also called "nil".

subrs primitive, or built-in, functions (like +).

One can evaluate an object for a value; numbers, strings, subrs, and nodes are "self-evaluating", i.e., the object and its value are one and the same.

Evaluating a symbol yields the value last assigned to that symbol by the user (see the setq function). Symbols actually have two distinct values: the value used during evaluation and one used only when the symbol is used as a function name. A useful example of this is the symbol "*" which when used as a function denotes multiplication, but which when used as an argument denotes the last value returned by a command to the top level interpreter.

Evaluating a list is like making a function call. The function value (or the ordinary value if there is no function value) of the first element of the list is the function. The values of the remaining list elements are the arguments. For example, evaluating

(+ a 3)

looks up the function value of the "+" symbol (in this case it will be the subr for addition), then calls

the function with the values (recursively computed) of "a" and "3". The value of the list will be the value returned by the function.

Certain lists have special meaning to the system and are called special forms. Two special forms of particular interest are discussed here, the remainder are described in a later section. The quote special form,

(quote arg) or 'arg

allows us to create symbol and list constants. Thus the value of (quote a) is the symbol "a", and the value of '(+ 2 3) is a list of three elements.

User defined functions are represented by the lambda special form:

(lambda (param param ...) exp exp ...)

The symbol "lambda" indicates that this list is actually a user function. It is followed by a list giving the names of the arguments and finally by a sequence of expressions which make up the body of the function. The value returned by the function when called will be the value of the last expression in the body. For example,

((lambda (x) (+ x 3)) 4)

evaluates to 7. We can give this function a name by making the lambda expression the value of some symbol:

(setq plus-3 '(lambda (x) (+ x 3)))

(plus-3 4)

also evaluates to 7. In doing this we set the ordinary value of plus-3 to the lambda expression. A better way of doing this is to use

(defun plus-3 (x) (+ x 3))

which makes (lambda (x) (+ x 3)) the function definition of the symbol "plus-3".

Note that setq changes the "expression value" of a symbol, while defun changes the "function value" -- this distinction is unimportant in most applications, but is useful if you wish to change the definition of a built-in function (beware of the implications before trying to change built-in function definitions though!).

This version of *rnl* is case sensitive. Special nodes with names "Vdd" and "VDD" are aliased to "vdd"; "Gnd" and "GND" are aliases of "gnd".

5.2. About Efficiency

LISP symbols and circuit nodes with the same name are in fact *different* objects. Functions that take nodes as arguments also work with LISP symbols, but the symbol is converted to a node each time the function is called. This entails getting the print name of the the symbol and using it as the key in a hash table lookup of the node. While this is implemented efficiently, it is still much more expensive than using the node directly. Often used arguments should therefore be converted from symbols to nodes using **find-node**. (This conversion can be done only after the network has been loaded because the nodes are not created until then.)

5.3. Useful Symbols

The following symbols are defined by RNL and are accessible to the user. They are useful in writing your simulations.

*	is the value last returned by the top level loop. This is mostly useful when you are poking around interactively.
base	controls the radix for printing integers. If not one of 2, 8, 10, or 16 then base 10 is used. The input radix is controlled by using the Unix conventions extended by using "0B" to signal a binary integer.
current-time	is the current value of simulated time, expressed in tenths of nanoseconds.

end-of-file	returned when EOF is read.
event-list-empty	is set to t when there are no pending electrical events on the queue, nil otherwise.
user-interrupt	is set to t if the user types the "quit" character. (Default is ctrl-\.) A check for **user-interrupt** as a condition for exiting a time consuming loop is often useful.

6. BUILT-IN FUNCTIONS

The file "uwstd.l" contains functions that are usually found in LISP environments. This section documents these functions in addition to the functions that are really built in to RNL. The functions are grouped by application area. The areas are:

arithmetic functions and predicates

list and symbol manipulation functions

i/o functions

special forms

network/simulation functions

Unless otherwise stated all functions evaluate their arguments. In addition to standard functions described in this section, you have the option of loading the file "uwsim.l" which contains a suite of functions that implement a collection of useful and relatively easy to use "front-end" facilities for doing circuit simulations.

The notation used in the descriptions of the functions is intended to make clear the types of the arguments. Arguments are prefixed as follows:

*g_* for any type,

*s_* for a symbol,

*t_* for a string,

*p_* for types with unique print names (symbols, nodes, strings, and integers),

*c_* for symbols and nodes,

*l_* for list arguments,

*n_* for any number,

*i_* for integer,

*f_* for floating point.

Quoting the formal parameter means that the argument **evaluates** to the required type. Special types are mentioned explicitly. For example (**func** *'i_arg*) means that *i_arg* evaluates to an integer and (**func** *'vec)* means that **func** takes an argument that evaluates to a circuit node vector.

6.1. Arithmetic functions

Unless otherwise stated, the arithmetic functions take both floating point and integer arguments, returning a floating point result if any argument was a floating point number. Warning: overflow and underflow are not checked by these functions.

(* *'n_arg1* ... *'n_argn*)

returns the product of its arguments.

(+ *'n_arg1 . . . 'n_argn*)
returns the sum of its arguments.

(- *'n_arg1 . . . 'n_argn*)
returns the first argument minus the sum of the remaining ones. The form (- *n_arg*) returns the negation of its argument.

(/ *'n_arg1 'n_arg2 'n_arg3 . . .*)
returns the quotient of *n_arg1* divided by the product of the rest of the arguments. If all arguments are integers the result is an integer truncated towards zero.

(% *'i_arg1 'i_arg2*) returns the remainder of *i_arg1* divided by *i_arg2*.

(1+ *'i_arg*)
like (+ 'i_arg 1) but restricted to integers.

(1- *'i_arg*)
like (- 'i_arg 1) but restricted to integers.

(< *'n_arg1 'n_arg2*)
returns t if n_arg1 less than n_arg2.

(<= *'n_arg1 'n_arg2*)
returns t if n_arg1 less than or equal to n_arg2.

(== *'n_arg1 'n_arg2*)
returns t if n_arg1 (numerically) equal to n_arg2.

(!= *'n_arg1 'n_arg2*)
returns t if n_arg1 (numerically) not equal to n_arg2.

(> *'n_arg1 'n_arg2*)
returns t if n_arg1 greater than n_arg2.

(>= *'n_arg1 'n_arg2*)
returns t if n_arg1 greater than or equal to n_arg2.

(abs *'n_arg*)
returns the absolute value of *n_arg*.

(fix *'n_arg*)
returns integer part of *n_arg*,
truncated towards zero.

(float *'n_arg*)
returns floating point version of *n_arg*.

(max *'i_arg1 . . . 'i_argn* **)**
returns maximum of its (integer arguments.

(min *'i_arg1 . . . 'i_argn* **)**
returns minimum of its (integer) arguments.

(numberp *'g_arg* **)**
returns t if *g_arg* is a integer, nil otherwise.

6.2. Functions and Predicates for List and Symbol Manipulation

(alphalessp *'p_arg1 'p_arg2* **)**
returns t if *p_arg*' s print name is lexicographically less than *p_arg2*' s.

(append *'l_arg1 'l_arg2* **)**
returns list of *l_arg1* with *l_arg2* appended to it. (This is defined in "uwstd.l".)

(atom *'g_arg* **)**
returns t if *g_arg* is not a list.

(car *'l_arg* **)**
returns first element of list *l_arg*.

(cdr *'l_arg* **)**
returns list of all but first element of *l_arg*.

(c*r *'l_arg* **)**
equivalent to multiple car and cdr (* = aa,ad,da, or dd). (This is defined in "uwstd.l".)

(char-to-num *'p_arg* **)**
returns the ASCII code for the first character of *s-arg*' s print name.

(cons *'g_arg1 'g_arg20* **)**
returns a list *l* such that (**car** *l*) = *g_arg1* and (**cdr** *l*) = *g_arg2*.

(eq *'g_arg1 'g_arg2* **)**
returns t if *g_arg1* and *g_arg2* are the same (identical !!) LISP object.

(equal *'g_arg1 'g_arg2* **)**
returns t if *g_arg1* and *g_arg1* are conformable.

(explode *'p_arg* **)**
returns a list of symbols whose single character names are the characters of *p_arg*' s print name.

(**fset** *'s_arg 'lambda*)
sets *s_arg*' s function definition.

(**fsymeval** *'s_arg*)
returns *s_arg*' s function definition.

(**get** *'s_arg 'g_name*)
returns value of *s_arg*' s *g_name* property. (This is defined in "uwstd.l".)

(**implode** *(p_arg1 p_arg2 . . .)*)
inverse of explode. Arguments with no sensible print name are ignored.

(**length** *'l_arg*)
returns number of elements in list *l_arg*.

(**list** *'g_arg1 g_arg2 . . .*)
makes a list with elements *g_arg1*, etc.

(**make-symbol** *'p_arg1 'p_arg2 . . .*)
returns symbol whose pname is concatenation of pnames of its arguments. This is very useful for converting nodes to symbols. See **implode.**

(**mapcar** *'u_func 'l_arg*)
returns a list whose elements are the result of applying *u_func* to each of the elements in *l_arg*. (This is defined in "uwstd.l".)

(**memq** *'g_arg 'l_arg*)
returns tail of *l_arg* begining with *g_arg*, if *g_arg* is not in *l_arg*, then it returns **nil.** This uses **eq** to test equality.

(**null** *'g_arg*)
returns **t** if *g_arg* is **nil.** **not** is a synonym for **null.**

(**plist** *'s_arg*)
returns *s_arg*' s property list. (This is an added built-in routine. It is not intended to be used directly by users. See "get".)

(**plset** *'s_arg 'l_arg*)
sets *s_arg*' s property list to *l_arg* and returns *l_arg*.

(This is an added built-in routine. It is not intended to be used directly by users. See "putprop".)

(**pname** *'p_arg*)
returns string equal to *p_arg*' s pname.

(**putprop** *'s_arg 'g_val 'g_name*)

sets *s_arg*' s *g_name* property to *g_value* and return *g_val*. (This is defined in "uwstd.l".)

(remprop *'s_arg 'g_name* **)**

returns *s_arg*' s property list from *g_name* on and removes *g_name* property from list. (This is defined in "uwstd.l".)

(rplaca *'l_arg 'g_arg* **)**

replaces car of *l_arg* with *g_arg*.

(rplacd *'l_arg 'g_arg* **)**

replaces cdr of *l_arg* with *g_arg*.

(set *'s_arg 'g_arg* **)**

sets value of *s_arg* to *g_arg* and returns *s_arg*.

(setq *s_arg 'g_arg* **)**

sets value of *s_arg* to *g_arg* and returns *s_arg*.

(stringp *'g_arg* **)**

returns t if *g_arg* is a string.

6.3. I/O functions

The notation [*'fid*] indicates an optional file identifier argument. If it is included the operation is directed to the designated file. If omitted the operation goes to the standard input or output device as appropriate.

The strings used to specify filenames can contain all the standard UNIX file name expansion conventions including the use of environment variables (e.g. " . . / myfile").

The base used for printing integers is controlled by the value of the symbol "base". The default is decimal, base 10.

(close *'fid* **)**

close file specified by *fid*.

(flush *'fid* **)**

force buffered output for file *fid*.

(load *'p_name* **)**

take input from file named by *p_name*.

If the file is not found in the present working directory then the directories specified in the environment variable RNLPATH will be searched. If RNLPATH is not defined then the default directory $UW_VLSI_TOOLS/lib/rnl is searched. UW_VLSI_TOOLS is an environment variable that is used in many of the tools distributed by UW/NW VLSI Consortium. This searching is done only for **load**.

(log-file *'p_name* **)**

closes the currently open log file (if any) and opens a log file named *p_name* and returns a *fid* for the file. (log-file nil) closes the currently open log file. A log file contains a verbatim copy of everything that goes to your terminal during the time that it is open.

(**openi** *'p_name*)

open file with name *p_name* for input, return *fid* for the file opened.

(**openo** *'p_name*)

open file with name *p_name* for output, return *fid* for the file opened.

(**prin1** *'g_arg* [*'fid*])

prints *g_arg* without trailing newline.

(**princ** *'g_arg* [*'fid*])

like prin1 without quotes around strings.

(**print** *'g_arg* [*'fid*])

print *g_arg* followed by newline.

(**printf** [*'fid*] *'string 'g_arg1 'g_arg2 . . .*)

print *g_arg1 ...* under format control specified by *string*. This is similar to the printf in the stdio library for C. Escapes for printing the *g_arg*' s are: %c-> ASCII char, %%-> print '%', and %S-> print LISP object.

```
Example: (setq  a  10)
         (setq  b  '(list of symbols))
         (printf  "a=%d0=%S0  a  b)
   produces
         a=10
         b=(list of symbols)
```

(**read** [*'fid*])

read an S-expression from an appropriate file. Returns the expression read or the symbol **end-of-file** if the end of the file is reached. This does not recognize the shorthand syntax of the standard read-eval-print loop.

(**read-network** *'p_name*)

read a network in file named *p_name*.

This file must be the output of PRESIM. The network described in the file is merged with the networks already loaded. There is no way to undo the loading of a network other than restarting RNL.

(**read-state** *'p_name*)

reset state of circuit to that in file named *p_name*.

(**terpri** [*'fid*])

output newline.

(**write-state** *'p_name*)

current state in file named 'p_name. This can be read by **read-state** routine.

6.4. LISP Control Structures and Special Forms

Although special forms look like function calls, their behavior can be quite different (especially with respect to the evaluation of their "arguments").

Warning: Lambda bound variables (parameters to defun, lambda, or prog) remain bound until the form is exited. Functions called from within these forms will see the new bindings. This is not consistent with other dialects of LISP.

(**and** *g_exp1 g_exp2 ...*)

evaluates *g_exps* left to right until the first **nil** result is found. **And** returns the value of the last argument it evaluates.

(**cond** *clause clause ...*)

Following the **cond** is a sequence of clauses. Each clause is of the form (*pred exp exp ...*). In each clause the first expression is taken to be a predicate and the remainder of the expression the body of the clause. **Cond** evaluates the predicates in turn, stopping at the clause with the first non-nil value for the predicate. The body of that clause is then evaluated and the value of the last expression in the body is returned as the value of the **cond**. If no predicates evaluate to non-nil, nil is returned as the value of the **cond**.

(**defun** *s_name (s_par ...) exp ...*)

define function *s_name* with formal parameters *s_par . . .* This is equivalent to (**fset** *'s_name* '(**lambda** *(s_par ...) exp ...*)).

(**do** *(l_vrb1 ...) l_exit1 exp1 exp2 ...*)

is a generalized iteration expression.

It has three components:

(i) a list of iteration symbol declarations,
(ii) an exit clause,
(iii) and a body that is executed on each iteration.

The list of iteration symbol declarations are used to declare temporary (i.e. "prog" bound) symbols whose scope is the **do** expression. Upon exiting the **do** they revert to their previous status. Each declaration in the list has one of the forms:

(*s_sym*) or (*s_sym 'g_init*) or (*s_sym 'g_init 'g_iter*)

wher *s_sym* is the symbol declared, *'g_init* is is an expression whose value is assigned to *s_sym* at the start of the first iteration,, and *'g_iter* is an expression that is evaluated at the start of each successive iteration to provide successive values for *s_sym*. If *'g_iter* is omitted then the value is not changed automatically. The initialization and iteration expressions are evaluated in left-to-right order. For example, the following declaration list says that i starts at zero and is incremented by two, and that j starts at 3 more than the value of symbol k and is squared on each iteration.

((i 0 (+ i 2)) (j (+ k 3) (* j j)))

The exit clause, *l_exit*, is a list of the form

(pred eexp1 eexp2 . . .)

After the new values have been assigned to the iteration symbols, the predicate *pred* is evaluated. If its value is non-nil, then the *eexp*'s are evaluated in order and the value of the last is returned as the value of the **do**. If the value of *pred* is nil, then the body of the loop, i.e. expressions *exp1, exp2, ...*, is evaluated. This process is repeated until *pred* is non-nil.

(eval *'g_arg* **)**

evaluates *g_arg* and returns the result.

(exit)

exits *rnl* in an orderly fashion, flushing buffers, closing files etc.

(lambda *(s_par1 s_par2 ...) exp1 exp2...* **)**

is the definition of a (nameless) function with formal parameters *s_par1 s_par2 . . ..* **lambda** itself is not a function.

(or *g_exp1 g_exp2 ...* **)**

evaluates the *g_exps* left to right, stops when the first **nil** is returned. Or returns the value of the last argument it evaluates.

(prog *(s_1 s_2 . . .) 'g_exp 'g_exp ...* **)**

saves the old values of *s_n* s, binds the symbols to **nil**, and evaluates each of the *g_exps* in order. The old values are restored when execution of the **prog** is completed. Returns the value of the last of the *g_exps*. Prog is used to allocate local variables within a function.

(quote *g_arg* **)**

returns *g_arg* without evaluating it. The syntax *'g_arg* is equivalent.

(repeat *s_index 'i_lower 'i_upper expr_list* **)**

is a simple iteration similar to a **for** loop in Pascal. *Expr_list* is the body of the loop. *S_index* is the index variable. *i_lower* and *i_upper* are integers representing the initial and the final values of the index. The index is increased by one each time the loop is executed. Returns the final value of the index.

6.5. Network functions

An electrical network consists of a list of nodes transistors, capacitors, and resistors. The functions described in this section allow user-defined functions to deal with the network.

(! *'l_nodes* **)**

prints a list of the transistors whose gates are connected to nodes in *'l_nodes*.

The syntax: ! node1 node2 ... is more common. Returns **nil.**

(? *'l_nodes* **)**

prints a list of transistors whose source or drain are connected to nodes in *l_nodes*. The syntax: ? node1 node2 ... is more common. Returns **nil.**

This command is useful for wandering through the network trying to track down the source of a particular value.

(**find-node** *'p_arg*)

returns electrical node with print name the same as *p_arg*' s. Returns **nil** if there is no such node.

(**match-node** *'p_pattern*)

uses *p_pattern*' s print name as a pattern using '*' as a wild card. Returns a list of symbols whose print names correspond to print names of nodes that match the pattern.

(**node-value** *'p_arg*)

if a node exists with a print name matching *p_arg*' s return its value (one of 0, 1, or X), otherwise **nil.**

(**node-time** *'p_arg*)

returns the latest time in 0.1ns units at which a change occurred in the state of the nodename provided in the argument. If the node is not found or has not changed state, 'nil' is returned.

(**set-delay** *'c_node 'i_tplh 'i_tphl*)

Set the transition times for the specified node; tplh (low-to-high transition time) and tphl (high-to-low transition time) are integers specifying time in 10ths of nanoseconds. If either tplh or tphl are negative, the node's times become unspecified and the transition times will be determined by the usual RC calculation. This command allows one to override the timing calculation. This is useful when the RC calculation gets the wrong answer for one reason or another. Usually this is worth doing only on critical nodes, such as clocks, where an timing error can be significant.

(**set-node** *'c_node 'g_exp*)

set value of node *c_node* to *g_exp*. adds/removes node as an input. If exp evaluates to

0	node is added to low input list
1	node is added to high input list
U,u	node is added to undefined input list
X,x	..see text...

The node will be stuck at this input value until changed by another call to set-node. If exp is X (remember exp is evaluated so you'll probably want to type 'X), node is removed from the input lists. At the next simulation step it will acquire whatever value it would naturally have.

(**set-params** *name value*)

give a value to one of the simulation parameters:

report	flag (default = t). If non-nil, nodes given the value of X because of improper pullup/pulldown ratio or because of charge decay will cause a warning message to be printed.
unitdelay	flag (default = nil). If non-nil, all node transitions happen with unit delay.
decay	fixnum (default = 0). If non-zero, tells the number of time units (10ths of nanoseconds) it takes for charge on a node to decay to X. A value of 0 implies no decay at all.

maxres number (default = 1E10). Capacitors on the far side of transistors bigger than this value don't contribute to summary capacitance used in calculating transition times.

(**set-threshold** *'c_node 'f_vlow 'f_vhigh*)

set *c_node*' s logic thresholds to *f_vlow* (low) and *f_vhigh* (high). vlow and vhigh should be numbers in the range [0,1]. These thresholds are used when converting from a Thevenin equivalent voltage to a logic state -- sometimes it is useful to be able to override the defaults for special nodes which otherwise will turn out X.

(**sim-init**)

This finds all nodes whose values are X and queues a transition to 0 for those nodes. The integer returned is the number of affected nodes. A call to **sim-step** or one of the higher level simulation commands such as "step", "s", or "c" will allow these changes to propagate. Initialization should be done by manipulation of the inputs of the circuit, simulating the real initialization sequence. **Sim-init** can be used to initialize nodes that cannot otherwise be initialized. Using **sim-init** without first simulating the setting of the inputs can be very, very expensive, especially when trying to initialize circuits conected by a bus.

(**sim-step** *'i_stop-time*)

simulation step using RC model. This runs the electrical simulation until *current-time* = *i_stop-time*.

If the simulation runs to completion then **nil** is returned. If a node that has the STOPONCHANGE flag set goes through a transition, then the simulation is stopped at that point and the node that had the transition is returned.

(**stop-on-change** *'c_node 'g_switch*)

If *g_switch* is not **nil** then set *c_node*' s "STOPONCHANGE" flag. If *g_switch* is **nil** then *c_node*' s "STOPONCHANGE" flag is cleared.

(**switch-init**)

like sim-init, except prepares network for initialization by switch-step instead of sim-step.

(**switch-step** *'stop-time*)

simulation step using switch model. This is similar to **sim-step**, but transistors are modelled as switches and transitions have unit delay. This algorithm is somewhat faster than the usual RNL calculation for many circuits, but can give X answers for circuits for which transistor size is important for correct operation (e.g., bit line in a dynamic memory). To ensure correct operation, one should not use **sim-step** until the event list is empty (and vice versa) -- i.e., all events scheduled by a particular algorithm must be handled by the same algorithm. The value of **event-list-empty** can be tested to see if the all events have been handled.

This routine may be useful when debugging the basic functionality of a circuit, or when simulating a circuit which has not been correctly sized (one that gives ratio errors using **sim-step**). Since the switch-level algorithms are much faster when dealing with large groups of interconnected nodes, switch-step may be particularly useful when initializing a network.

(**walk-net** *'function*)

function should be a symbol or a lambda expression that takes a circuit node as its only argument. **walk-net** applies that function to every node in the network.

(trace-node *'c_node 'g_switch*)

If *g_switch* is not nil then start to trace *c_node*, otherwise stop tracing *c_node*. This is useful for trying to track down exactly what is happening to a subciruit at a very low level. The first form turns on tracing for the specified node, the second turns it off. Sample outputs:

```
[1]; event 1: b=H @ 10.0ns
[2];  b => clist: d <input seen>
[3];  d: rgnd=[3.05e+04,6.11e+04], rvdd=[1.00e+10,1.00e+10]
[4];          d-rgnd=1.18e+04, d-rvdd=1.00e+10, lhdelay=0, hldelay=0
[5];     cap: high=0.000000e+00, low=0.000000e+00, x=1.014720e+00
[6];     => value=L @ 1.20e+01 ns
[7];   enquing d [event 1: b] L @ 220 (delta = 120)
```

[1]: node b makes a transition to H at 10.0ns

[2]: a list (clist) of nodes affected by b is reported. In this case one node (d)is found before an input ends the search. Inputs can be forced nodes, vdd or gnd.

[3]: Report the result of the Thevenin calculations for nodes on the clist, rgnd Thevenin resistance to ground, rvdd Thevenin resistance to vdd. Note in this first case rgnd is computed to be an interval. This is the result of an input node having a value of X.

[4]: Report the value of the resistors to gnd (d-gnd) and vdd (d-vdd) used in the delay calculations. Note these values are not necessarily the same as those in [3]. This is the result of using different values for R_{static}, R_{dynlow} and $R_{dynhigh}$.

[5]: Report the value of total capacitance charged high, low and x for current node.

[6,7]: Compute new logic value for node and enque it at current-time + delta (delta = RC). R and C are choosen from the values given in lines [4] and [5].

```
[8]; event 1:  H @ 10.0ns
[9];  c => clist: d <input seen>
[10]; d: rgnd=[1.00e+10,1.00e+10], rvdd=[3.05e+04,3.05e+04]
[11];         d-rgnd=1.00e+10, d-rvdd=5.90e+03, lhdelay=0, hldelay=0
[12];    cap: high=0.000000e+00, low=1.014720e+00, x=0.000000e+00
[13];    => value=H @ 6.00e+00 ns
[14];  enquing d [event 4: c] H @ 531 (delta = 60)
```

This example is substantially the same as the first except that the Thevenin resistance is no longer an interval.

(trace-all-nodes *'g_switch*)

If *g_switch* is not nil then start to trace all nodes, otherwise stop the trace. Sometimes this is the only way to track down oscillating subcircuits.

7. The uwsim.l package.

The *uwsim.l* package is intended to provide a powerful and easy to use front end for *rnl*. In addition, it is can serve as the basis for customized front ends for specific projects. In this document we concentrate on the functions intendend to be directly called by users. The programmer who

intends to extend this or to customize it is invited to peruse the code.

7.1. Syntax and symbols.

The *uwsim.l* package defines two new data-structures: Vectors of circuit nodes are defined by *vecdef*' s. A *vecdef* is a list of the form (*s_mode s_name c_node1 ... c_noden).* *s_mode* is one of {hex, dec, oct, bin, bit} and controls the formatting of output and input for the vector. The first four modes allow numeric input in any base and output in the specified base (and are limited to 23 nodes), the bit mode uses bit vector input and output. If a numeric vector contains undefined nodes it is printed as a bit vector. The first element in the list of nodes is the high-order bit of the numeric vector.

You can request that a report about the state of your circuit be printed at certain times during the simulation (usually at the end of a clock cycle). The format of this report is specified by a report-form which is a list containing *vecdefs* , strings, symbols corresponding to nodes, nodes, and the format control symbols { newline, tab, and page} printing of a report. The car of the list is a string that heads the report.

The following symbols are global variables defined in uwsim.l and should not be used for nodenames. These variables however can be used in programs for the appropriate purpose.

t	The property list of "t" is used to hold tokens describing which packages have been loaded. *Uwsim.l* requires that *uwstd.l* be loaded first.
incr	This variable is the simulation step time interval in 0.1ns units, used by the step and clock functions; the default value is 1000 for 100.0ns.
relative-timing	If this variable is not "nil" then transition times are reported relative to the start of the simulation step. Default is 'not nil' = 't'.
switch-level	If "switch-level" is non-nil then the switch-level model is used, otherwise the RC model. Default: switch-level = 'nil'. Note: the simulator switch level model does not appear to work satisfactorywith cmos and dynamic circuitry. Its use is discouraged with this version of rnl.
lanalyze	The global variable lanalyze, if 'not nil' = 't' prevents printing of all description texts and limits the report output defined in def-report to node and vector states. The first column of the report is the current time in ns. Default is "nil".
glitch-detect	If this variable is 'not nil' = 't' then only multiple changes of the nodes defined in the chflag command are reported. Single transitions are NOT reported. A chflag command defined which nodes are subject to glitch-detection must be included before starting the simulation.The default value of glitch-detect is 'nil'.
triggering	If this variable is 'not nil' = 't' the logic triggering function is enabled. The default is 'nil'.
trigger_flag	This variable is used to prevent additional triggercommand execution when a triggercondition persists. This variable is not for external use.
trigger_index	Contains the number of occurences of the trigger condition. Is initialized at 1. Can be used to influence trigger handling as a function of the number of trigger conditions detected.

trigger_file Contains the filename where rnl will look for the commands to be executed in case a trigger condition occurs. Default is "trig_file".

7.2. Functions Intended for Direct Use by Users.

(defvec *'vecdef* **)**

optimizes the representation of *vecdef* by converting LISP symbols into nodes and stores that representation as the vecdef property of *vecdef*' s name.

(defvecv)

Defines a single indexed vector of type 'type' (bit/bin/oct/hex/dec) name 'basename' and elements number.(start_index+number_elements) down to number.(start_index)

(defvecl)

Defines 'index1' vectors with names 'basename_1' ... 'basename_index' with double indexed node names.

(def-report *'l_arg* **)**

creates an optimized report form. *l_arg* is a list, beginning with a title string, containing the following LISP forms: strings are printed in the report without surrounding quotes.

newline	inserts a newline in the output.
tab	inserts a tab in the output.
page	inserts a form feed in the output.
s_arg	for symbols other than those above causes the state of the corresponding node (if any) to be displayed. An error is reported and a newline is inserted if *s_arg* cannot be converted to a node.
(vec s_arg) or (vec vecdef)	causes the state of a vector to be inserted in the report. If the first form is used, *s_arg* should have been previously been given a vector definition. If the second form is used, the vector definition is created on the fly.
(function g_arg)	causes *g_arg* to be evaluated when this form is encountered in the printing of the report

(def_reportl *l_arg* **)**

same as **def_report** except multiple vectors can be defined as in **defvecl** ; such vectors are defined with (*vecl* basename)

(**h** *'l_nodes*)

makes the nodes in *l_nodes* inputs at logical high (1). (Alternate syntax: h node1 node2 ...)

(**l** *'l_nodes*)

makes the nodes in *l_nodes* inputs at logical low (0). (Alternate syntax: l node1 node2 ...)

(**x** *'l_nodes*)

removes the nodes in *l_nodes* from the input list. They can now be driven high or low by the modelled circuit. (Alternate syntax: x node1 node2 ...)

(**u** *'l_nodes*)

makes the nodes in *l_nodes* to be undefined inputs. (Alternate syntax: u node1 node2 ...)

(**t** *'l_nodes*)

turns on traces for nodes in *l_nodes*. (Alternate syntax: t node1 node2 ...)

(**ut** *'l_nodes*)

turns off traces for nodes in *l_nodes*. (Alternate syntax: ut node1 node2 ...)

(**invec** *'(vec_name g_val1 g_val2 ...)*)

checks the type of *vec_name*. If the vector is one of the numeric types (hex, dec, oct, or bin) then it assigns the numeric value *g_val1* to the vector. Otherwise, it treats the vector as a bit vector and assigns the value *g_val1* to the first node in the vector, *g_val2* to the second, etc.

(**bitinvec** *'(vec-name g_val1 g_val2 ...)*)

is like **invec** but forces the input to be in the bit vector form. This function allows elements of the vector to be set to **x** or **u**.

(**openplot** *'(file_name)*)

opens a plot file to receive notifications of reported transitions. The resulting plot file can be processed by the program *mtp* to produce plots that resemble logic analyzer displays. Note the output is controlled by specifications of *chflag, chflagv* and *chflagi* !

(**closeplot** *'l_arg*)

closes the plot file. The argument is ignored.

(**markplot** *marker*)

inserts a marker with the name *marker* in the plot file.

(**s** *'l_arg*)

runs a simulation step for **incr** simulated time and generates a report at the end. (See **def-report.**)

(**c** *'(i_arg)*)

runs a two-phase non-overlapping clock for *i_arg* cycles. This assumes that the clock nodes are called **phi1** and **phi2.** Each of the 4 periods is *incr* long. At the end of *i_arg* cycles a report is attempted using the user's declared format.

(vecnames *arg*)

arg should be either vector definition or a symbol with a vector definiton. It prints out the names and current values of the vector.

(vecnodes *arg*)

arg should be either vector definition or a symbol with a vector definiton. It returns the list of component nodes of the vector.

(unchanged-since '*n_time*)

returns a list of nodes that have not changed since *n_time*. This is useful for helping to decide whether your simulation has adequate coverage.

(unchanged '*l_arg*)

is shorthand for (unchanged-since 0). The argument is ignored.

(chflag '*l_arg*)

sets the "STOPONCHANGE" flag for the nodes in *l_arg*.

(unchflag '*l_arg*)

clears the "STOPONCHANGE" flag for the nodes in *l_arg*.

(chflagv)

This command will add all nodes in the vector, assumed to have normal extensions of a base_name (like in.1 in.2 in.3 ..... etc) to the list of nodes with the "STOPONCHANGE" flag set. Start_index is the value of the first node (normally "0" or "1") and vector_size is the number of elements in the vector.

(unchflagv)

This command will remove all nodes in the vector from the list of nodes with their "STOPONCHANGE" flagset. Assumed is that the nodenames all have extensions of type: basename.1 basename.2 etc. Complement of the "chlagv" command.

(chflagi)

Sets the "STOPONCHANGE" flag for all nodes withnames basename.i.j where i from 1 to index1 and j from 1 to index2.

(unchflagi)

Disables the "STOPONCHANGE" flag for all nodes with names basename.i.j where i from 1 to index1 and j from 1 to index 2.

(wr-format)

This command writes the nodes and vectors in order as they appear in a logic analyzer style output.

(init '*state*)

State can be either '*l*' or '*h*'. This routine does more reporting than *sim_init*; also signals can be preset high (useful since many dynamic CMOS circuits are pre-charged high and start-up time of normal simulation can be substantially reduced.

Init does the following:

The routine finds all nodes with undefined states 'x' in the network and reports the number of undefined nodes found. Then, as determined by the argument either low or high is applied to these nodes and a simulation step is run.
Thereafer all previously clamped nodes are released again and up to 10 simulation steps are run until the network has settled and the number of simulation steps necessary to settle the network is reported. A warning message is provided if the network has not settled within 10 simulation steps.
Finally, the routine again traverses the network and reports the number of nodes still in the undefined states.

7.3. Functions Intended to be Called by Other Functions.

There are a lot of these functions. You are invited to look at *uwsim1*.

In addition, they provide examples of how many of the elements of this language are utilized.

NETLIST & RNL
Tutorial for Beginners

Rudolf W. Nottrott & Henriecus Koeman

UW/NW VLSI Consortium
Sieg Hall, FR-35,
University of Washington,
Seattle, WA 98195

This document is intended as a guide for people who want to acquire a basic working knowledge of the RNL digital circuit simulator and the NETLIST network description program in the UW/VLSI VAX 11/780 environment. Examples are given for the preparation of logic network description files and the production of the corresponding *.sim* and binary files for input to RNL. Next, instruction on the use of RNL commands to set up a clearly defined network state for a simulation is provided. Performing actual simulations of some of the networks defined previously, frequently needed commands, such as those for setting circuit values, asking about node information, running a simulation step, etc., are explained and applied to the network in both interactive and batch mode. Further references are listed in the appendix.

If you are not familiar with the UNIX operating system, read the introductory document UNIX-quick.

Table of Contents

1. Format Conventions for this Document

2. Overview of the Position of NETLIST and RNL in the UW/VLSI Tool Environment

3. Logic Network Descriptions for NETLIST

3.1 A CMOS Inverter as a Simple Example

3.1.1 The NETLIST Logic Circuit Description of an Inverter

3.1.2 Processing the Description File with NETLIST and PRESIM

3.2 NETLIST Description File for a Ten-Bit Shift Register (Made from Latches and MS-Flip-Flops)

3.2.1 Defining Macros as Building Blocks (Latches and Flip-Flops) - Building Block Library

3.2.2 Making the Register with Macros from the Library - Loops and Indexed Symbols

3.2.3 Processing the Register Description File with NETLIST and PRESIM

3.2.4 Converting a Network Description into a Macro

3.2.5 Sizing of NETLIST Functions with Two or More Transistors (CINVERT, CLKINV, etc.)

4. Circuit Simulation with RNL

4.1 Interactive Command Input and Batch Command Input

4.2 Practicing RNL Simulations - The Shift Register

4.3 Defining Control and Stimulus Fiels the "Easy" Way

4.4 Printing RNL Output Using MTP

4.5 Displaying RNL Output On a Graphics Display Terminal Using Simscope

5. Summary and Outlook

Appendix 1 - Further References

Appendix 2 - Description of the .sim file of the example "inverter" (section 3.1.2)

Appendix 3 - Preparation of a simple "config" File

Appendix 4 - The "alias" file of the example "shift" (section 3.2.3).

Appendix 5 - Summary of Typical Commands Associated with Simulation Tools

Appendix 6 - Makefile Commands

Appendix 7 - A Simple Makefile

1. Format Conventions for this Document:

The UNIX system prompt is given as a small percent sign (%) in bold-face type.

(The RNL command interpreter has no prompting sign, which can sometimes be confusing.)

All information RNL prints out in response to your entries, including error messages, are indicated in **small bold-face type.** If a short description of what RNL returns to you is given instead of the actual RNL response, it is also shown in small bold-face type.

Your entries are indicated in **bold-face type,** normal size.

Certain commands and phrases within the text are also printed in bold-faced letters for emphasis.

Program names within the text are frequently capitalized for emphasis (e.g. PRESIM for presim).

File names are indicated in *italic type* (unless they are part of an entry sequence, in which case they are shown in bold-face as is everything else that is part of an entry).

<CR> stands for the RETURN key.

^ (Caret) stands for the CONTROL key (frequently labeled CTRL). If ^ precedes a character, the CONTROL key has to be pressed and held down while the character key is pressed.

DELETE and <DEL> stand for the DELETE key.

2. Overview of the Position of NETLIST and RNL in the UW/VLSI Tool Environment

The *.sim* file plays a central role in connecting NETLIST and RNL with the rest of the UW/VLSI tools, as well as with each other (see Figure 1). The name *.sim* file derives from the mandatory file name extension *.sim*.

In this tutorial, we will use NETLIST to produce a *.sim* file from a logic network description file, then transform the *.sim* file with the program PRESIM into a binary network description file for input to RNL. We will not deal with the generation of the *.sim* file with other tools, such as MEXTRA, nor will we consider the possibility of using the *.sim* file for input to programs other than RNL.

RNL must be given two sets of information for a simulation run. These are:

(a) a description of the network to be simulated and

(b) the commands to control the simulation run.

The network description specifies the elements of the network (such as transistors, NANDs, etc.) and the way they are interconnected. The RNL command input performs functions such as setting the initial state of the network, providing signals to be applied to the network (input signals, clock, ...), specifying the format of the simulation report, etc.

We will deal with the network description first, and later use the networks defined in this way to illustrate the generation of the RNL command input.

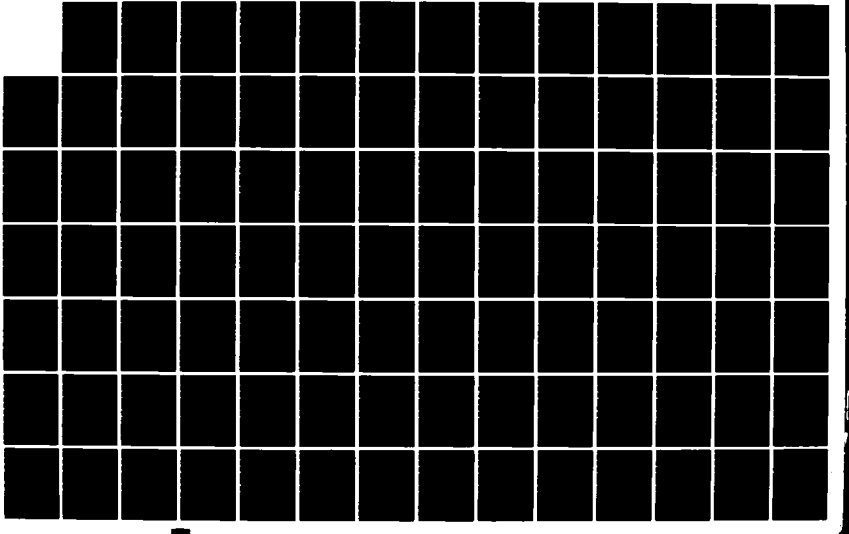
The RNL command input can be entered in two ways: interactively, or via a command file submitted to RNL (batch mode). You may use a command file to initialize the network or to apply complex stimulus signals, and then continue to work with RNL interactively. In interactive mode, you can do simple simulations and develop programs in RNL LISP, which may be run in batch mode subsequently. In most RNL sessions you will probably find yourself alternating between interactive and batch mode.

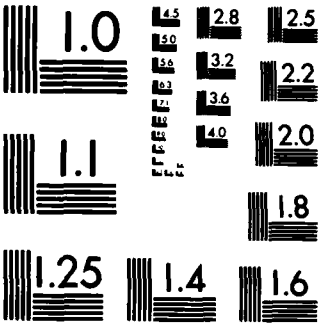
There are conventions for the names of the files to be processed by NETLIST, PRESIM and RNL. These conventions will be observed in this tutorial. All files related to a particular circuit are given the same main name followed by a period (.) and an extension. The binary input file for RNL is an exception to this rule - it has no extension.

The meanings of the extensions are:

.net	network description file, input to NETLIST
.sim	intermediate file, output of NETLIST and input to PRESIM; this file is a true "mediator" - it forms the connection not only between NETLIST and RNL, but also between NETLIST and the other VLSI tools, and RNL and the other VLSI tools.
.l	command file for RNL (batch mode)
.al	alias file produced by NETLIST*

* Normally you do not have much to do with the *.al* file, it is created automatically by NETLIST and used automatically by PRESIM.





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

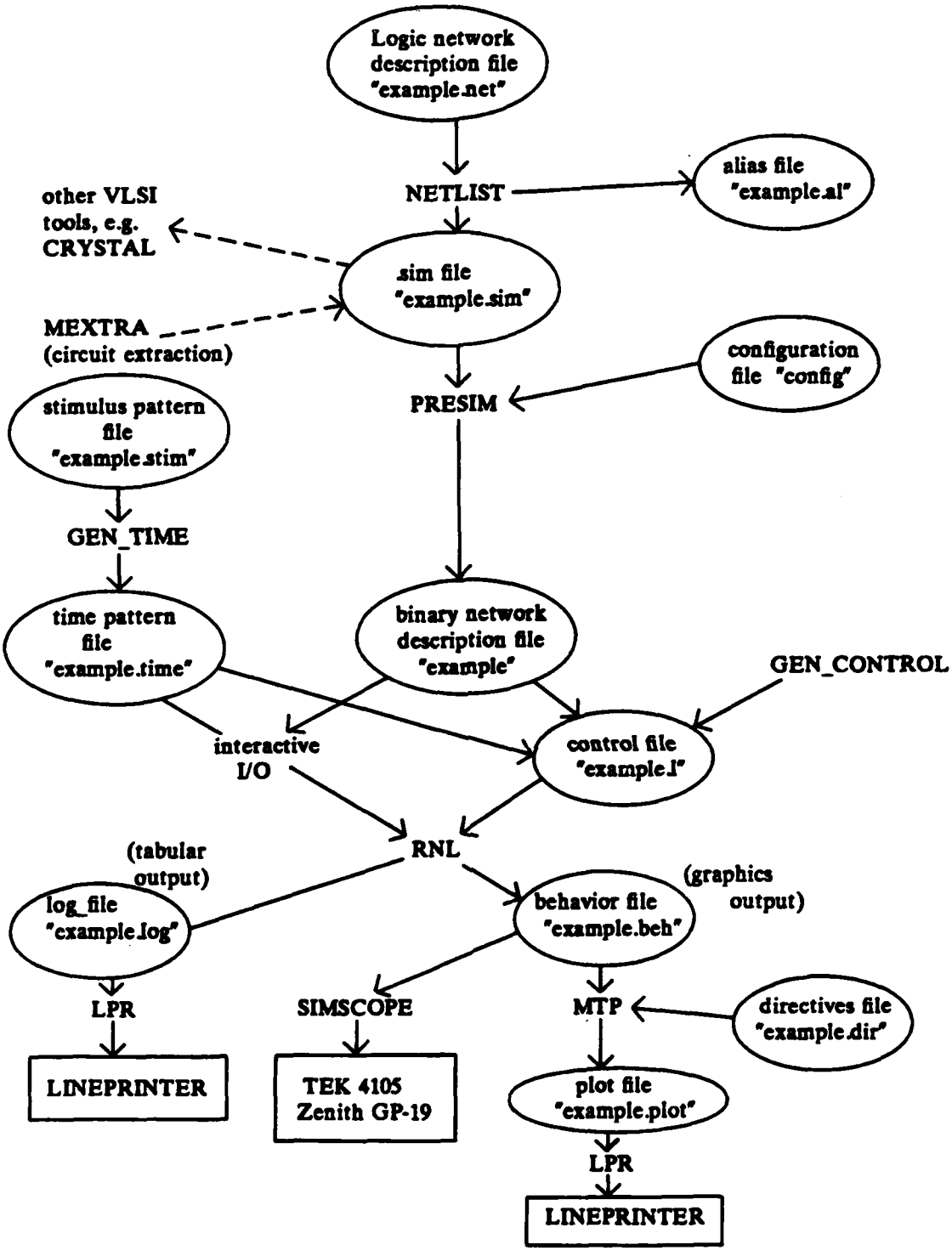


Figure 1
Position of NETLIST and RNL in the UW/NW VLSI Tool Environment.

no extension: binary network description file, output of PRESIM, input to RNL

You need not follow these file name conventions, but it is strongly recommended since it makes the communication between various users much easier.

3. Logic Network Descriptions for NETLIST

3.1. A CMOS Inverter as a Simple Example

3.1.1. The NETLIST Logic Circuit Description of an Inverter

Recall from section 1. that RNL needs to know the network before you can start your simulation*. We will describe here how the network can be specified for NETLIST in a logic network description file using a LISP-like command syntax.

You therefore should familiarize yourself with the basics of LISP, if necessary. To help you with this, this section will provide an example of a logic network description and an implicit description of some important LISP properties, but will omit much of the detail and the intricacies of LISP.

Figure 2. shows the circuit diagram of our first example, a simple CMOS inverter. We will prepare the logic network description for the inverter according to this diagram.

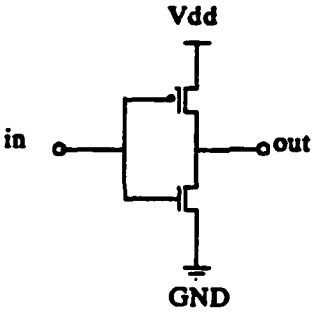


Figure 2 CMOS Inverter

* However, it is possible to add to your network in the course of the simulation.

The general form of a NETLIST (and RNL) command is:

```
(command_name  argument_1  argument_2  argument_3  ...)
```

In most cases the parentheses are required. You will find later in this tutorial examples for cases where the parentheses can be left out. (In particular, RNL has a syntax simplification designed to save you some typing of parentheses).

Here is a listing of the logic network description, followed by the explanation of its commands (write the network description into a file named *inverter.net*.) :

```
; (1) NETWORK DESCRIPTION FOR A CMOS INVERTER
; ------------------------------------------------------------------

; (2) DECLARATION OF THE NODES IN THE NETWORK
(node  in  out)

; (3) P-CHANNEL ENHANCEMENT TRANSISTOR (PULL-UP)
(ptrans  in  out  Vdd  8  8)

; (4) N-CHANNEL ENHANCEMENT TRANSISTOR (PULL-DOWN)
(etrans  in  GND  out  4  8)

; (5) SPECIFYING AN INTERCONNECT CAPACITANCE FOR THE OUTPUT NODE
(capacitance  out  0.03)
```

Hereafter, the numerals enclosed in parentheses will be used to indicate each part of the description file.

(1) Note that a semicolon causes the rest of the line to be treated as a comment, i.e., to be ignored by the NETLIST program. Blank lines are also ignored*.

This first comment serves as a title to the network description file.

(2) You must declare any node you want to name for subsequent reference. (you could think of this declaration as bringing the nodes named by you to the attention of RNL)**. There are a few exceptions to this rule, however. Some nodes, common to most circuits, are known to RNL without declaration. These are the ground and drain voltage potential connections*** symbolized GND and Vdd (the symbols GND and Vdd are not sensitive to upper or lower case, so

* You should make ample use of such comments and blank lines to make your network file as "readable" as possible.

** There are almost always additional nodes named by NETLIST (or other programs producing a *.sim* file). This happens, for example, when NETLIST processes a macro which has internal (local) nodes. Such names go undeclared. You will find many of them in most .sim files.

*** We use the term "drain voltage potential" despite the fact that in many cases the drain of a transistor may be connected, not to Vdd, but to any other node. Notably this is the case with transmission gates.

gnd and vdd are equivalent symbols).

Nodes are declared with the command

(node n1 n2 n3 n4 ...),

where n1, n2, n3, n4, .. are the names of the nodes to be referred to in the network.

(3) The declaration of the nodes has provided the "skeleton" for the network. Next you must "fill in" the remainder of the circuit. For a transistor this is done with a command of the form

(transistor-type gate source drain width length).

Transistor-type represents a mnemonic for various types of transistors, such as

ptrans for p-channel enhancement-mode transistor

etrans for n-channel enhancement-mode transistor

dtrans for n-channel depletion-mode transistor

(see NETLIST User's Guide for more available transistor types and other circuit elements.)

Gate, source, and drain represent the names of the nodes to which the gate, the source, and the drain of the transistor are connected.

The pull-up of the inverter is specified as a p-channel enhancement-mode transistor, and the appropriate nodes are "in", "out", and "Vdd".

The **width** and the **length** of the transistor's gate area in units of lambda may be specified optionally. If omitted, both width and length default to 2 lambda. Our pull-up is given a width of 8 lambda and a length of 8 lambda.

The width and the length of the gate area determine the resistance of the transistor*. In this way you can influence the ratio of the pull-up to pull-down.

(4) The pull-down is specified, analogously to (3), as an n-channel enhancement-mode transistor with a gate width of 4 and a gate length of 8.

(5) The final element to be specified in the inverter is the interconnect capacitance. The command

(capacitance out 0.03)

tells NETLIST that a capacitance of 0.03 pF is to be inserted between the nodes "out" and GND. (One side of a capacitance specified with this command is always to GND). The specification of this capacitance is an estimate of the load capacitance of the inverter.

* RNL determines the resistance by looking up a two-dimensional table in which the dimensions are length and width. In this way the influence of the geometry of the gate area on the effective (empirical) resistance may be taken into account.

3.1.2. Processing the Description File with NETLIST and PRESIM

After the logic network description has been written to the file *inverter.net*, it has to be processed with the NETLIST and PRESIM programs.

Substitute "inverter" for "example" in the file names from Figure 1. (You already prepared the file *inverter.net*.)

Then enter:

```
% netlist inverter.net inverter.sim <CR>
```

This causes NETLIST to process the network description file *inverter.net*, writing its output to the file *inverter.sim*. If you omit the filename *inverter.sim*, NETLIST will display the output on the screen**. (Enter **% man netlist** or see NETLIST User's Guide to get more information about optional parameters for NETLIST).

If everything worked out correctly, the only response you will get is the UNIX prompting sign (%).

You may want to look at the *inverter.sim* file produced by NETLIST. Its content is listed and analyzed for this example in the appendix.

The next step is to process *inverter.sim* with PRESIM. PRESIM transforms the transistors in the *.sim* file into resistors of equivalent size. This is done because RNL uses resistor models for the transistors and estimates transition time delays from the equivalent network formed by the resistors and the circuit capacitances.

There is an optional configuration file which can be used to give to PRESIM technology-dependent parameters, such as the specific resistances of the transistor channels. If you do not use this configuration file in the PRESIM run, default values for the specific channel resistances are assumed. The assigned default values are the same for all the different transistor types, which results in a resistance ratio of 1 if the gate areas for the pull-down and the pull-up transistors are sized equally.

An explanation is given in the appendix on how to prepare a simple configuration file to change the channel resistance of the p-channel transistor to twice the value of that of the n-channel transistor. Read this appendix or simply prepare the very short *config* file (six lines) from the listing there. You can then run PRESIM with the *config* file as a parameter:

```
% presim inverter.sim inverter config <CR>
```

This will cause PRESIM to process *inverter.sim*, putting the output into the file *inverter*. This output is a binary file.

PRESIM will give you some information about what it did:

```
Version 4.2
8 nodes; transistors: enh=1 intrinsic=0 p-chan=1 dep=0 low-power=0 pullup=0 resistor=0
Total transistors eliminated = 2
```

** In fact, it will go to the UNIX standard output, which is normally assigned to the screen. Of course, you may redirect the standard output (e.g. to a file).

First, it tells you its version number, which is 4.2. Then, you are informed that eight nodes were found in the network. If you look at Figure 2, you might count only four nodes, but PRESIM counts some of the nodes more than once.

PRESIM tells you how many transistors of the various types it found in the circuit and how many transistors it "eliminated".

You could now use the *inverter* file as the binary network description for RNL and run a simple simulation. However, the example of the inverter was only intended to be a simple exercise to give you a feeling for the way networks are described using NETLIST. We will consider the network description of a more complex network before proceeding to an actual simulation.

3.2. NETLIST Description File for a Ten-Bit Shift Register Made from Latches and Flip-Flops

3.2.1. Defining Macros as Building Blocks (Latches and Flip-Flops) - Building Block Library

We are going to build the shift register in a modular fashion as illustrated in Figure 3a through 3c. First we make a latch from inverters (3a), then we put together two latches to get a master-slave flip flop (3b), and finally we chain ten flip-flops to build the shift register (3c).

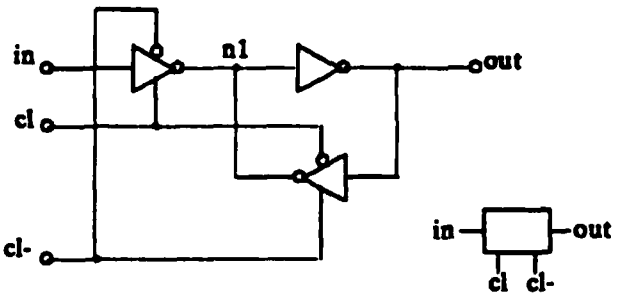


Figure 3a Latch

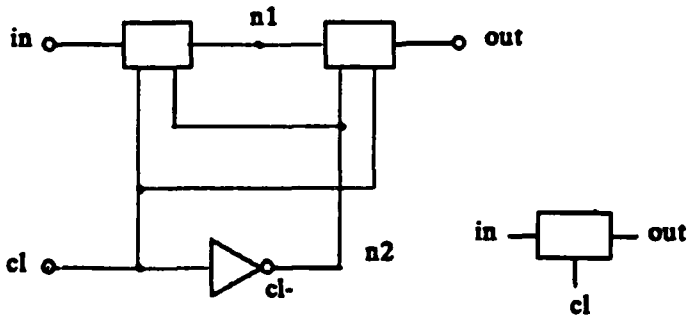


Figure 3b MS Flip Flop (made from Latches)

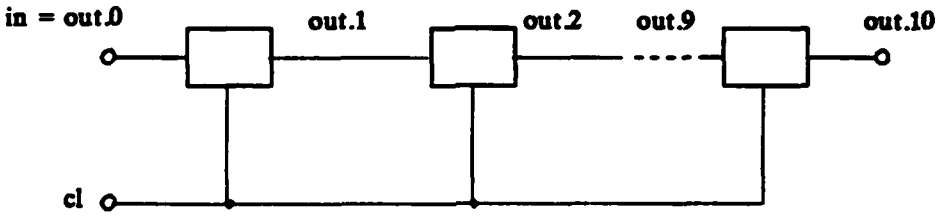


Figure 3c Shift Register (made from ms-ff)

Since it can already be seen that we may need latches and flip-flops in future designs, we will define these building blocks such that we can later call them without having to redefine them. This is done with macro definitions, which can be stored in a library file and easily loaded into future network description files. The library file, which we shall call network library, has the same format as the *net* files. We give it the name *lib.net*.*

Here is the listing of the macro definition of the latch, followed by the explanation of its statements (see also Figure 3.a):

```
; (1) MACRO DEFINITION FOR A CMOS LATCH
; ------------------------------------------------------------

; (2) NAMING THE MACRO AND ITS PARAMETERS
(macro latch (out in cl cl-)
```

* It is not necessary to store the macro definitions in a library file. You may choose to write the macro definitions directly at the beginning of the network description file.

```
; (3) DECLARATION OF THE NODES LOCAL TO THE LATCH
(local n1)

; (4) FIRST CLOCKED CMOS INVERTER
(clkinv n1 in cl cl-)

; (5) UNCLOCKED CMOS INVERTER
(cinvert out n1)

; (6) SECOND CLOCKED CMOS INVERTER
(clkinv n1 out cl- cl)

; (7) CLOSING PARENTHESIS FOR THE MACRO
)
```

(1) This is the title identifying the library entry.

(2) A macro definition has the general format

```
(macro name (param1 param2 param3 ...)
body of the macro
)
```

Note the closing parenthesis after "body of the macro", and make sure that you never forget them in any macro definition. You should invent short and descriptive names, but do not use the name of any of the other NETLIST functions (as listed in the NETLIST User's guide.). The body of the macro is made up of the statements (2) through (6).

We give our macro the name "latch". The name is followed by a list of parameters param1, param2, param3, ... These parameters represent the values to be used when the macro is called later. In the latch macro, they represent the names of the nodes that are used to connect the latch to other circuits.

(3) There is one node to which one need not refer when the latch is used later. This node, "n1", is only of local importance to the latch. Therefore it is declared as a local node in the latch macro. Locally declared node names may be declared and re-used in other macros, since they are considered free symbols outside the macro of their declaration.

(4) As in the previous example of the inverter, the nodes form only a "skeleton" for the network which must be "filled in" with the circuit elements. The circuit elements are two clocked inverters and an unclocked inverter. These elements are available in standard form as commands for the network description. (This means we need not have taken the trouble to describe the inverter with single transistors in section 3.1. However, we did this as an example of a simple circuit, and to demonstrate the use of transistors in a network description.)

A clocked CMOS inverter is specified with a command of the form
(clkinv out in clk clk-).

"clkinv" is a mnemonic for "CMOS clocked inverter". "out", "in", "clk", and "clk-" represent the names of the nodes to which the output, input, clock input, and negated clock input are connected (the clock input is the base of the n-channel transistor, the negated clock input is the base of the p-channel transistor). In (4), a clocked inverter is connected to the respective nodes out, in, cl, and cl-. (The gate sizes and ratio of the clocked inverter "clkinv", and the devices "cnand", "cnor" and "cinvert", can be changed with "width" and "length" values and with the "ratio" command; see section 3.2.5 and NETLIST User's Guide.)

(5) A simple unclocked inverter is inserted. The general command for the specification of this CMOS inverter is (cinvert out in), with out and in representing the names of the nodes to which the output and the input of the inverter are connected.

(6) The second clocked CMOS inverter is specified analogously to (4).

After completing the macro definition, save it in the library file *lib.net*.

In the following definition of the master-slave flip-flop (see also Figure 3b), you can simply call the latch by its macro name.

The macro definition of the master-slave flip-flop is analogous to that of the latch. Its statements, followed by the explanation of their meanings, are listed below:

```
; (1) MACRO DEFINITION FOR A CMOS MASTER-SLAVE FLIP-FLOP
; ------------------------------------------------------------------------------

; (2) NAMING THE MACRO AND ITS PARAMETERS
(macro mff (out in cl)

; (3) DECLARATION OF THE NODES LOCAL TO THE FLIP-FLOP
(local n1 n2)

; (4) FIRST LATCH
(latch n1 in cl n2)

; (5) SECOND LATCH
(latch out n1 n2 cl)

; (6) CMOS INVERTER
(cinvert n2 cl)

; (7) CLOSING PARENTHESIS FOR THE MACRO
)
```

Most of the statements in this macro definition will already be familiar to you. Note that the previously defined macro "latch" is used in the same way as other circuit elements. If you had not defined the latch yourself, you might not even know whether "latch" is a macro or a basic NETLIST function. This property allows you to define successively more complex building blocks and nevertheless use

them with the same ease as the "primitive" functions.

As we did with the macro definition for the latch, we now add the macro definition for the master-slave flip-flop to our library file *lib.net*. It must be inserted after the latch, since it uses "latch" as a circuit element and will therefore call the "latch" macro.

3.2.2. Making the Register with Macros from the Library - Loops and Indexed Symbols

Looking at Figure 3c, it is now easy to make the ten-bit (or any number of bits) shift register by chaining ten of the flip-flops defined in our macro library *lib.net*. We could write down the call for "msff" with the appropriate parameters ten times, but NETLIST has the facility of a loop and indexed symbols, which makes the specification of such repetitive elements as the flip-flops of the shift register very compact. (In accordance with the file name conventions, write the following network description into a file named *shift.net*.):

```
; (1) CIRCUIT DESCRIPTION FOR THE 10-BIT SHIFT REGISTER
; ------------------------------------------------------------------------

; (2) LOADING THE FUNCTIONS FROM THE MACRO LIBRARY
(load "lib.net")

; (3) NODE DECLARATION FOR THE NETWORK
(node in out cl)

; (4) LOOP CALLING THE MASTER-SLAVE FLIP-FLOP 10 TIMES
(repeat i 1 10
   (msff out.i out.(- i 1) cl)
)

; (5) ASSIGNING AN ADDITIONAL NAME TO THE INDEXED NODE OUT.0
(connect in out.0)
```

(1) This is the usual title.

(2) Load the macro library *lib.net*, which has the effect of inserting the macro definitions for latch and msff before the description of the shift register.

(3) Declare the network nodes to which you want to refer. Remember, there are two kinds of node declarations: global, as here and in the example of the inverter (section 3.1); and local, as in our macro definitions.

(4) This part illustrates two new facilities which we have at our disposal when we want to specify the description of network structures that contain the same sub-circuit repetitively.

In most cases of such network structures, as in our shift register, the regular pattern of subcircuits makes it possible to refer to their nodes with a collective name followed by an index.

out.0, out.1, out.2, ..., out.10 (see Figure 3c) are examples for **indexed node names**. "out" is the collective node name for a group of nodes having a similar function or position in the network; ".0"., ".1", ... are the indices uniquely identifying each of the individual nodes.

Note one other application of indexed nodes in the "node" command of (3). Indexed nodes can be declared by simply declaring their collective node name. It is not necessary to list each individual node. Thus, our declaration of "out" in (3) represents out.0, out.1, out.2, ... out.10.

Generally, an index can be represented by any symbol or expression. In the "repeat" loop, it is given as "i" and "(- i 1)", respectively. (- i 1) is the LISP form of subtracting 1 from i, and just one example of how an index can be calculated from an expression.

The symbolic index enables us to use a loop calling our master-slave flip-flop ten times. A loop has the format

```
(repeat   loop-index  start-value  end-value
          body of the loop
)
```

The "body of the loop" can be any sequence of commands. The loop in (4) starts with the index "i" set to 1. "i" is increased by 1 after each call to the flip-flop. In this way the loop specifies flip-flops with connections to

```
out.1,  out.0,  cl
out.2,  out.1,  cl
out.3,  out.2,  cl
  .
  .
  .
out.10,  out.9,  cl
```

After "i" has reached the value 10, the loop is exited.*

(5) The network description using "repeat" and indexed nodes looks very compact, but a few nodes were designated cumbersome names. Rather than using the name "out.0" for the input to the shift register (see Figure 3c), we will call it "in". This can easily be accomplished with the **connect** command, which equates the names "in" and "out.0". Another application for the **connect** command is in connecting two or more nodes electrically.

* Upon leaving the loop, the value of the symbol of the loop index will be restored to the value it had before entering the "repeat" form.

3.2.3. Processing the Shift Register Description File with NETLIST and PRESIM

You should be familiar with the procedure for processing a description file with NETLIST and PRESIM, from section 3.2.1. However, be aware of one additional file created by NETLIST. This file is called an alias file and it contains, for each node, all the different symbolic names that have been assigned to that node. Remember that you specified explicitly with the **(connect in out.0)** statement that the names "in" and "out.0" were to denote the same node.

The alias file created by NETLIST has the same main name as the other files related to the network, which is *shift* for the shift register, followed by the extension *.al*. A short explanation of the *shift.al* file is given in the appendix.

Substitute "shift" for "example" in the file names from Figure 1. (You have already prepared the file *shift.net.)*

Then enter:

```
% netlist shift.net shift.sim <CR>
```

This will cause NETLIST to process the network description file *shift.net* in the same way as explained before for the inverter.

(Feel free to inspect the *shift.sim* file produced by NETLIST, which should not be difficult since you know how interpret *.sim* files from the example *inverter.sim* analyzed in the appendix.)

To process the description file with PRESIM enter:

```
% presim shift.sim shift config <CR>
```

PRESIM gives you the following information:

```
Version 4.2
139 nodes; transistors: enh=110 intrinsic=0 p-chan=110 dep=0 low-power=0 pullup=0 resistor=0
Total transistors eliminated = 220
```

It is very much similar to the case of the inverter except that the shift register circuit is much bigger.

You have now seen, for several examples, how to produce the network description needed by RNL. The next section examines a special aspect of network descriptions (converting network descriptions to macros). You may want to skip this now, and proceed to actual simulations of two of the networks we have described so far.

3.2.4. Converting a Network Description into a Macro

You have seen how to use macros as building blocks in the description of a network. A network description can easily be turned into a macro in the following way:

- Specify as parameters the nodes you want to access when you call the macro. In the example below, (output in cl) are the parameters for the output, input and clock connections. Later, before you call the macro, you must "globally" declare (with the "nodes" command) the actual names for the respective nodes.

- Declare the remaining nodes as local nodes. Be careful not to use names that are to be used as global node names in the "main" network. In the example below, the indexed name "o" is used to represent all the individual indexed nodes o.0, o.1, o.2, ..., o.10 (formerly out.0, out.1, out.2, ..., out.10).

Here is the listing of a possible macro definition for the ten-bit shift register, derived from the shift register's network description:

```
; (1) MACRO 10-BIT SHIFT REGISTER
;-------------------------------------------------
;   THIS MACRO MAY BE CALLED ONLY WHEN THE "LATCH" AND
;   "MSFF" MACROS HAVE BEEN DEFINED OR LOADED
;   PREVIOUSLY.

; (2) MACRO DEFINITION
(macro shiftreg_10 (output in cl)

    ; (3) NODE DECLARATION FOR THE NETWORK
    (local o)

    ; (4) LOOP CALLING THE MASTER-SLAVE FLIP-FLOP 10 TIMES
    (repeat i 1 10
      (msff o.i o.(- i 1) cl)
    )

    ; (5) ASSIGNING AN ADDITIONAL NAME TO NODES o.0
    ;    AND o.10
    (connect in o.0)
    (connect output o.10)

; CLOSING PARENTHESIS FOR THE MACRO.
)
```

After defining the macro, you can add it to the library *lib.net* and use it in the same manner as you have used "latch" and "msff".

3.2.5. Sizing of NETLIST Functions with Two or More Transistors (CINVERT, CLKINV, etc.)

In section 3.2.1 the NETLIST functions "cinvert" and "clkinv" were used to define the macros "latch" and "msff". We did not specify any gate sizes there, so the default values were assumed. You will see here what the default values are and how they can be changed.

In NMOS functions, such as inverters, NANDs, NORs, etc. (represented by the NETLIST functions "invert", "nand", "nor")* each individual transistor can be identified by the node to which its gate is connected. This is either one of the input nodes, or, in the case of the depletion-mode pull-up transistor, the output node (since the gate of a depletion-mode pull-up is connected to its source, which is the output). You can specify the gate size for each transistor by specifying width and length together with the node to which the gate is connected in the following manner:

```
(invert (out width-o length-o) (in width-i length-i))
(nand   (out width-o length-o) (in1 width-1 length-1) (in2 width-2 length-2) ...).
```

Thus, (invert (out 4 6) (in 8 10)) creates an NMOS inverter whose enhancement-mode pull-down has a gate area of 8 by 10 lambda, and whose depletion-mode pull-up has a gate area of 4 by 6 lambda. The case is similar for the "nand" and the "nor", the only difference is that you have more than one input. The default gate sizes are 2 by 2 lambda for an enhancement-mode transitors and 2 (width) by 8 (length) lambda for a depletion-mode transistor.

In CMOS devices the situation is different. Normally an input connects to two gates - one gate of a p-channel transistor and one gate of an n-channel transistor. This makes the sizing specifications somewhat more complicated, since a node does not any longer uniquely identify an individual transistor. NETLIST permits you to specify width and length together with a node as in the NMOS case, e.g.,

```
(cinvert out (in width length))
```

However, these values determine only the gate width and length of the n-channel transistor and the gate length of the p-channel transistor. Defaults are 2 lambda. You can set the width of the p-channel transistor to a multiple of the width of the n-channel transistor with the command

```
(ratio value)
```

which must precede the function it is to affect. For example,

```
(ratio 3)
(cinvert out (in 4 6))
```

sets the "ratio" for "cinvert" (and all following CMOS functions until the next "ratio" command is encountered) to 3. The n-channel transistor (pull-down) of the inverter gets a gate area which is 4 lambda wide and 6 lambda long. The the p-channel transistor (pull-up) gets a gate area which is 3 * 4 = 12 lambda wide and 6 lambda long.

The default "ratio" is 2.

If a node connects to only one gate, width and length of the gate area are set in the same manner as

* We did not use them but concentrated on CMOS functions. If you want to know more about them, see the NETLIST User's Guide.

described above for NMOS. Thus, in the case of a clocked inverter, you can specify the dimensions of the gate area of the two transistors connected to the input node in the same manner as in the case of "cinvert", whereas the dimensions of the two transistors whose gates are connected to "cl" and "cl-" are specified independently:

```
(ratio 3)
(clkinv out (in 4 6) (cl 8 10) (cl- 12 14))
```

The gate dimensions of the two transistors forming the inverter are the same as in the "cinvert" above. The gate area of the p-channel gating transistor connected to "cl-" is 12 lambda wide and 14 lambda long, the gate area of the n-channel gating transistor connected to "cl" is 8 lambda wide and 10 lambda long.

The sizes of the other CMOS functions available in NETLIST are set in a similar manner (see NETLIST User's Guide).

4. Circuit Simulation with RNL

RNL can take its command directly from the user's keyboard or from a file or a mixture of both. There is a certain amount of information that must be supplied all the time: which standard libraries should be used, which network to read, where to store the result of information if that is desired (most of the time) etc. Then the timing or pattern information must be added. In most situations you want to create a file with all standard simulation set-up information (which we will call the "control" or ".l" file) and another file which is called by a command in the control file containing all the timing information of signals to be applied to the circuit under test. It may be of interest to first exercise the circuit in the fully interactive way as shown in sections 4.1 and 4.2. A number of utilities have been designed which will allow unsophisticated users to define set-up files and pattern files without having to worry about lisp syntax at all. These utilities are discussed in section 4.3. It is not necessary to work through sections 4.1 and 4.2 to have sufficient understanding for working on section 4.3.

4.1. Interactive Command Input and Batch Command Input

You have now completed all the necessary preparations to run a simulation of one of the circuits described previously. We will run RNL and start out with our simple inverter.

```
% rnl <CR>
```

RNL comes up with its version number

```
Version 4.2
```

and waits at the beginning of the next line for your command input. (RNL does not have prompting sign.) Every command you enter now is immediately executed and, if necessary, commented by RNL. This is why this mode of operation is called the interactive mode. Correspondingly, this way of entering commands is called interactive command input.*

* Just in case you want to exit RNL, the command to leave RNL in an orderly fashion is (exit) or simply exit.

Before starting on your simulation, you must load the two files *uwstd.l* and *uwsim.l* , containing function definitions for RNL.

```
(load "uwstd.l") <CR>

done

(load "uwsim.l") <CR>

Loading uwsim.l
Done loading uwsim.l
```

(The loading of the *uwstd.l* file is implied).

Next, load the binary network description file *inverter*.

```
(read-network "inverter") <CR>
```

RNL will prompt with information about the network:

```
; 5 nodes; transistors: enh=0 intrinsic=0 p-chan=0 dep=0 low-power=0 pullup=0 resistor=0
```

There is a simple command ("s" - we will use it shortly) to run a simulation step for an amount of time defaulted to 100 ns. To change this, a variable incr can be set. incr * 0.1 ns is the new length of the of the simulation step. For example, an incr of 10 results in a simulation step length of 1 ns. The command to assign a value to a symbolic variable* is (setq symbol value).

Let us carry out the simulation in steps of one nanosecond. Set incr to 10 so that the product of "incr" and the internal step width is 1.

```
(setq incr 10) <CR>

10
```

RNL echoes (returns) the value assigned to incr.

Frequently it is convenient to refer to a group of nodes, rather than to one individual node. You can

* Strictly speaking, "incr" is a LISP symbol. Symbols in LISP are not quite the same as variables in other programming languages. In many cases however, as with "incr", they act just like "conventional" variables. We therefore will frequently designate as variables objects which should strictly be called symbols.

denote a symbolic name for a list of node names with "setq". We will give the name "nodes" to the list of the two nodes "in" and "out".

```
(setq nodes '(in out)) <CR>

(in out)
```

Again, RNL returns the value it assigned to the variable, which is the list (in out).

The final step is to specify details for the reports on the simulation step. There are two standard report forms available in RNL.

The first type of report lists the state of nodes whenever this state changes. We want to obtain such a report for the changes in our "nodes", i.e. "in" and "out". Each node has a "change-flag" telling RNL that such a report is requested. For a list of nodes (n1, n2, ...), this "change-flag" is set with the command (chflag '(n1 n2 ...)). Since we have already defined a list of nodes (in out), named "nodes", we can enter

```
(chflag nodes) <CR>

t
```

RNL has now set the change-flags of "in" and "out" to "true".

The second type of report lists the state of nodes at the end of a simulation step. To obtain such a report on the nodes "in" and "out", use the "def-report" command:

```
(def-report '("STATE AT END OF SIMULATION STEP:" in out)) <CR>

("STATE AT END OF SIMULATION STEP:" in out)
```

The capitalized text in quotes is the title of the report. It is followed by the names of the nodes that *are to be included in the report.*

Now try a simulation. Setting the input of the inverter to high potential* is simply done by entering

```
h in <CR>
```

* We will frequently refer to high potential as High, and to low potential as Low.

RNL's reply **done** means that it carried out your command. (From now on, we will not mention this "done".)

"h" is the mnemonic for High, followed by the name of the node to be set to High. (You can specify more names, separated by a blank).

Run a simulation step by entering the mnemonic **s** :

```
s <CR>
```

In accordance with your specifications in "chflag" and "def-report", RNL will reply:

```
Step begins @ 0 ns.
in=1 @ 0
out=0 @ 0.6
STATE AT END OF SIMULATION STEP:
Current time= 1
in=1 out=0
```

After starting up RNL, the starting time for a simulation is always set to zero, so your first simulation step begins at 0. The reports on changes in the states of the nodes "in" and "out" show that "in" was set to High at time zero, and that "out" changed to Low at 0.6 ns. This is exactly what an inverter should do. The time delay in the change of the output is caused by the time needed to load the gate capacitance of the inverter and the time needed to unload the output node capacitance of 0.03 pF (see 3.1.1). The report at the end of the simulation step tells you that the time is now 1 ns, and repeats the state of the nodes "in and "out", as required in the "def-report" command. (For other commands to run a simulation step see RNL User's Guide).

Now consider another state of the inverter, set the input to Low ("l" is the mnemonic for Low):

```
l in <CR>
```

and do a simulation step:

```
s <CR>
```

```
Step begins @ 1 ns.
in=0 @ 1
out=1 @ 0.6
STATE AT END OF SIMULATION STEP:
Current time= 2
in=0 out=1
```

The report given by RNL is analogous to the one just explained. This time the simulation starts out at 1 ns with the input set to Low. The output changes to High at 0.6 ns (relative to the starting

point of the simulation step), again the delay is caused by the gate and output capacitances. At the end of the simulation step the time is 2ns, "in" is Low and "out" is High.

RNL uses three different logic states to characterize the potential of a node. They are:

High, symbolized H, with the value 1. Logic high is assumed whenever the simulated voltage level of the node is between a high threshold V_{high} and 1.

Low, symbolized L, with the value 0. Logic low is assumed whenever the simulated voltage level of the node is between 0 and a low threshold V_{low}.

Undefined, symbolized X. An undefined state is assumed whenever the simulated voltage level of the node is between a low threshold V_{low} and a high threshold a high threshold V_{high}.

You can set any node to one of the three states with the following commands:

h Set the node to High (shown above).

l Set the node to Low (shown above).

u Set the node to Undefined.

Setting a node in this way has the effect of connecting the node to a voltage source with zero impedance, thus overriding any other value the circuit might try to impose. Nodes with their values fixed in this way are called input nodes (because they are used like an input to a circuit, and RNL internally puts them on an "input list"). They will stay at the assigned logic level until you release them. You release nodes with the command x, followed by the names of the nodes you want to release. After the node has been released, it is free to assume whatever level it wants to assume naturally in the circuit (it will assume this level after an "s" command is executed).

The simple example of the inverter has given you a good idea of how to "operate" RNL in **interactive mode** : You enter a command and receive an immediate reply. Now exit RNL to prepare a "batch" command file:

```
exit  <CR>
%
```

You are back with UNIX.

There is one other mode of RNL operation which we shall call **batch mode.** You can write any sequence of RNL command into a file and specify the name of this file as a parameter when you start RNL. We will call this file "RNL command file", or simply command file. You will almost always want to have such a command file to save yourself the work of keying the "load", "read-network", and other commands that are invariably needed to set up the proper conditions for a simulation.

To see how the command file is used, write the first seven of the above commands into a file *inverter.l:*

```
(load "uwstd.l")
(load "uwsim.l")
(read-network "inverter")
(setq incr 10)
(setq nodes '(in out))
(chflag nodes)
(def-report '("STATE AT END OF SIMULATION STEP:" in out))
```

Now run RNL again, with the command file *inverter.l* as a parameter:

```
% rnl inverter.l <CR>
```

You will get almost the same replies as before when you entered the commands interactively. (The only difference is that returned values, such as the 10 in (setq incr 10), are not shown.)

Now set the input High and run a simulation step, then set the input Low and run another simulation step. Again, you will get the same answers as before.

In order to set up the proper conditions for a simulation, most commonly one starts out with the execution of the commands from a command file and then continues in interactive mode. RNL LISP programs for time-consuming simulations may be developed in interactive mode, written into a command file, and later run in batch mode.

We will do the simulations of the shift register in this mixed way in the next sections.

4.2. Practicing RNL Simulations - The Shift Register

You can modify on the command file prepared in the previous section and use the modified file when we start up RNL for the simulation of the shift register. Write the following modified commands into the file *shift.l* :

```
(load "uwstd.l")
(load "uwsim.l")
(read-network "shift")
(setq incr 100)
(setq nodes '(in out.10 cl))
(chflag nodes)
(def-report '("STATE AT END OF SIMULATION STEP:" in out.10 cl))
```

Now run RNL again with *shift.l* as the start-up command file:

```
% rnl shift.l <CR>
```

RNL's reply is similar to the one discussed in the previous section.

Let us do some initializing and propagate input through the shift register ((1) through (17)).

(1) Initialize the network

```
(sim-init)

52
```

When you start up RNL, the state of the nodes is Undefined, i.e. neither High nor Low. This state, symbolized "X", is not very useful at the outset of a simulation. It is preferable to start with a definite, stable state. The (sim-init) command tries to help you with this. It returns the number of all Undefined nodes and then sets them to Low. In this case, 52 nodes were set to Low. If the number of nodes set to Low was not 0, you should do a simulation step and propagate the new values through the network (we will do that in (2)). If this leads to Undefined node values again, do another (sim-init).

Repeat the sequence of (sim-init) followed by a simulation step until (sim-init) returns 0. If you cannot settle the network in this way after four to five repetitions, RNL might not been able to simulate your design properly (for example, if a an input signal and a feed-back signal derived from this input simultaneously drive a node). In such cases refer to section four of the RNL User's Guide.

(2) Simulation step to propagate the nodes set to Low

```
s

Step begins @ 0 ns.
out.10=1 @ 0.2
out.10=0 @ 0.4
STATE AT END OF SIMULATION STEP:
Current time= 10
in=0 out.10=0 cl=0

done
```

As a result of propagating the Low nodes from the previous (sim-init), "out.10" changes twice during the simulation step. The reports are analogous to the ones explained in the previous section.

(3) Next initialization step.

```
(sim-init)

0
```

This time the number of nodes set to Low by (sim-init) was 0, i.e., the network had been settled to a stable state with the commands in (1) and (2).

(4) Simulation step to propagate the nodes set to Low.

```
s

Step begins @ 10 ns.
STATE AT END OF SIMULATION STEP:
Current time= 20
in=0 out.10=0 cl=0

done
```

As expected, no changes occurred in this step (no nodes were changed in the previous (sim-init)). Reports as usual.

(5) Give a name to a group of nodes.

```
(setq all_nodes '(in out.1 out.2 out.3 out.4 out.5
                  out.6 out.7 out.8 out.9 out.10) )

(in (-struct- out 1) (-struct- out 2) (-struct- out 3) (-struct- out 4 ) (-struct- out 5) (-struct- out 6) (-struct- out 7)
(-struct- out 8) (-struct- out 9) (-struct- out 10))
```

We give the name "all_nodes" to the group af nodes forming the shift path. RNL returns its internal representation of this list of nodes.

(6) Set the change- flag for the group of nodes.

```
(chflag all_nodes)
```

```
(in (-struct- out 1) (-struct- out 2) (-struct- out 3) (-struct- out 4 ) (-struct- out 5) (-struct- out 6) (-struct- out 7)
(-struct- out 8) (-struct- out 9) (-struct- out 10))
```

This command sets the change-flag for each node of all_nodes.

We could have used the command

```
(chflag '(in  out.1  out.2  out.3  out.4  out.5
          out.6  out.7  out.8  out.9  out.10)
)
```

to achieve the same result, but instead used the symbol *all_nodes* to give an example of the usage of a name for a group of nodes.

(7) Set the clock input to Low.

```
l  cl

demo


s

Step begins @ 20 ns.
STATE AT END OF SIMULATION STEP:
Current time= 30
in=0 out.10=0 cl=0

demo
```

The clock is set to Low, followed by a simulation step. Reports as usual.

(8) Set all nodes in the shift path to High.

```
(repeat  i  0  10
    (h  '(out.(eval i)))
)
10


s

Step begins @ 30 ns.
```

```
out.10=1 @ 0
out.9=1 @ 0
out.8=1 @ 0
out.7=1 @ 0
out.6=1 @ 0
out.5=1 @ 0
out.4=1 @ 0
out.3=1 @ 0
out.2=1 @ 0
out.1=1 @ 0
in=1 @ 0
STATE AT END OF SIMULATION STEP:
Current time= 40
in=1 out.10=1 cl=0

done
```

The repeat command is similar to the one in section 3.2.2, where it was used to make a shift register from ten ms-flip-flops. Of course, the body of the loop is different here. It is the command

(h '(out.(eval i))).

There are several peculiarities in the form of this command (which you have already seen in its simpler forms, e.g. h out.1).

RNL LISP has a syntax simplification which permits you to write

command argument_1 argument_2 argument_3 ...

instead of

(command '(argument_1 argument_2 argument_3 ...))

Therefore, **h out.10** is equivalent to **(h '(out.10))**.

However, the simplified syntax may not be used if the command is part of another command, such as inside the "repeat".

The **(eval i)** is needed because RNL does not evaluate a symbol if it is preceded by an apostrophe ('). **(eval i)** returns the value of "i", which varies from 1 to 10.

Since we set the change-flag for "all_nodes", RNL reports all the changes in the corresponding nodes' values, followed by the usual final report at the end of the simulation step.

(9) Apply one clock cycle

```
h  cl
done


s

Stop begins @ 40 ns.
cl=1 @ 0
STATE AT END OF SIMULATION STEP:
Current time= 50
in=1 out.10=1 cl=1

done


l  cl
done


s

Stop begins @ 50 ns.
cl=0 @ 0
STATE AT END OF SIMULATION STEP:
Current time= 60
in=1 out.10=1 cl=0

done
```

We set the clock first to High, then to Low, each time followed by a simulation step. This is equivalent to a clock cycle of the length of two simulation steps.

(10) Release all nodes in the shift path.

```
(repeat  i  0  10
    (x  '(out.(eval i)))
)
10


s
```

```
Step begins @ 60 ns.
STATE AT END OF SIMULATION STEP:
Current time= 70
in=1 out.10=1 cl=0

done
```

Recall that we have set all the nodes in the shift path to High (8), which is equivalent to connecting them to Vdd. By setting them to "x", we enable them to assume whatever state they may naturally go to in the course of the simulation. Thus, the "repeat" loop releases all the nodes in the shift path of the register.

(11) Set input to Low.

```
l in
done

s

Step begins @ 70 ns. in=0 @ 8
STATE AT END OF SIMULATION STEP:
Current time= 8
in=0 out.10=1 cl=0

done
```

With the previous steps, all the cells of the shift register have been set to a state of High. Now we set the input to Low in order to later watch this Low signal shift through the register.

(12) Shift the Low input for one clock cycle

```
h cl
done

s

Step begins @ 80 ns.
```

```
ci=1 @ 0
STATE AT END OF SIMULATION STEP:
Current time= 90
in=0 out.10=1 ci=1
done


! ci
done


s

Step begins @ 90 ns.
ci=0 @ 0
out.1=0 @ 0.5
STATE AT END OF SIMULATION STEP:
Current time= 100
in=0 out.10=1 ci=0
done


! out.1 out.2 out.3

out.1=L (vl=0.30 vh=0.80) (0.038400 pf) affects:
input to functions for the following nodes:
      15
      15
      8
      8

out.2=H (vl=0.30 vh=0.80) (0.038400 pf) affects:
input to functions for the following nodes:
      27
      27
      20
      20

out.3=H (vl=0.30 vh=0.80) (0.038400 pf) affects:
input to functions for the following nodes:
      39
      39
      32
      32
done
```

One clock cycle shifts the Low from the input to the output of the first register cell. At the end of the clock cycle we use the ! (exclamation sign) command to check the values of the first three cells of the register.

! provides the values of the specified nodes, their logic thresholds (normalized to 1), and their capacitances. Next it provides the names of all nodes to which the nodes specified with ! are inputs. In the example, these node names are numerical. The numerical node names were assigned by NETLIST to nodes inside the shift register. They were not declared in *shift.net* but came with the "latch" and "msff" macros. (If you want to know more about them, you have to scrutinize *shift.sim*.)

(? (question mark) is a command similar to ! . It provides information about transistors for which a node is either gate or source, and about the sum-of-products representation of the node. See RNL User's Guide for more information.)

Using ! (and ?), you can "walk" through the network and check node values and connections.

The listing produced by ! shows that the Low input has moved to "out.1", as it should have after one clock cycle.

(13) Shift the input a second clock cycle.

```
h cl
done

s

Stop begins @ 100 ns.
cl=1 @ 0
STATE AT END OF SIMULATION STEP:
Current time= 110
in=0 out.10=1 cl=1

done

l cl
done

s

Stop begins @ 110 ns.
cl=0 @ 0
out.2=0 @ 0.8
STATE AT END OF SIMULATION STEP:
```

```
Current time= 120 in=0 out.10=1 cl=0

demo


! out.1  out.2  out.3

out.1=L (vl=0.30 vh=0.80) (0.038400 pf) affects:
input to functions for the following nodes:
     15
     15
     8
     8

out.2=L (vl=0.30 vh=0.80) (0.038400 pf) affects:
input to functions for the following nodes:
     27
     27
     20
     20

out.3=H (vl=0.30 vh=0.80) (0.038400 pf) affects:
input to functions for the following nodes:
     39
     39
     32
     32

demo
```

Reports analogous to (12). The Low input has now moved to "out.2".

(14) Shift the Low input a third clock cycle.

```
h  cl
demo

s

Step begins @ 120 ns.
cl=1 @ 0
STATE AT END OF SIMULATION STEP:
Current time= 130
```

```
in=0 out.10=1 cl=1

done


! cl
done


s

Step begins @ 130 ns.
cl=0 @ 0
out.3=0 @ 0.8
STATE AT END OF SIMULATION STEP:
Current time= 140
in=0 out.10=1 cl=0

done


! out.1 out.2 out.3

out.1=L (vl=0.30 vh=0.80) (0.038400 pf) affects:
input to functions for the following nodes:
     15
     15
     8
     8

out.2=L (vl=0.30 vh=0.80) (0.038400 pf) affects:
input to functions for the following nodes:
     27
     27
     20
     20

out.3=L (vl=0.30 vh=0.80) (0.038400 pf) affects:
input to functions for the following nodes:
     39
     39
     32
     32

done
```

Reports analogous to (12). The Low input has now moved to "out.3".

(15) Do seven more clock cycles to shift the Low input completely to the end (out.10) of the shift register.

```
(repeat 7 1 7
    (h  '(cl))
    (s  '(x))
    (l  '(cl))
    (s  '(x))
)

Step begins @ 140 ns.  cl=1 @ 0 STATE AT END OF SIMULATION STEP: Current time= 150 in=0 out.10=1
cl=1

Step begins @ 150 ns.  cl=0 @ 0 out.4=0 @ 0.5 STATE AT END OF SIMULATION STEP: Current time= 160
in=0 out.10=1 cl=0

Step begins @ 160 ns.  cl=1 @ 0 STATE AT END OF SIMULATION STEP: Current time= 170 in=0 out.10=1
cl=1

Step begins @ 170 ns.  cl=0 @ 0 out.5=0 @ 0.5 STATE AT END OF SIMULATION STEP: Current time= 180
in=0 out.10=1 cl=0

Step begins @ 180 ns.  cl=1 @ 0 STATE AT END OF SIMULATION STEP: Current time= 190 in=0 out.10=1
cl=1

Step begins @ 190 ns.  cl=0 @ 0 out.6=0 @ 0.5 STATE AT END OF SIMULATION STEP: Current time= 200
in=0 out.10=1 cl=0

Step begins @ 200 ns.  cl=1 @ 0 STATE AT END OF SIMULATION STEP: Current time= 210 in=0 out.10=1
cl=1

Step begins @ 210 ns.  cl=0 @ 0 out.7=0 @ 0.5 STATE AT END OF SIMULATION STEP: Current time= 220
in=0 out.10=1 cl=0

Step begins @ 220 ns.  cl=1 @ 0 STATE AT END OF SIMULATION STEP: Current time= 230 in=0 out.10=1
cl=1

Step begins @ 230 ns.  cl=0 @ 0 out.8=0 @ 0.5 STATE AT END OF SIMULATION STEP: Current time= 240
in=0 out.10=1 cl=0

Step begins @ 240 ns.  cl=1 @ 0 STATE AT END OF SIMULATION STEP: Current time= 250 in=0 out.10=1
cl=1

Step begins @ 250 ns.  cl=0 @ 0 out.9=0 @ 0.5 STATE AT END OF SIMULATION STEP: Current time= 260
in=0 out.10=1 cl=0

Step begins @ 260 ns.  cl=1 @ 0 STATE AT END OF SIMULATION STEP: Current time= 270 in=0 out.10=1
cl=1

Step begins @ 270 ns.  cl=0 @ 0 out.10=0 @ 0.5 STATE AT END OF SIMULATION STEP: Current time=
280 in=0 out.10=0 cl=0


7
```

(The simulation report is printed more compactly here to save space. Otherwise it is analogous to the reports in (12) through (14)). At the end of each clock cycle the Low input appears shifted one more cell toward the output. After the last clock cycle, it has reached "out.10".

(16) Define a clock function with the number of cycles as a parameter. The function is called "cycles".

```
(defun cycles (a)
  (repeat i 1 a
    (h '(cl))
    (s '(x))
    (l '(cl))
    (s '(x))
  )
)

cycles
```

"defun" is one of the most useful functions in RNL LISP. It enables you to define your own functions that can subsequently be used in the same straightforward way as all other RNL functions. "cycles" illustrates this point. "defun" is followed by the name you give the function, which, in turn, is followed by a list of parameters representing the arguments to the function (see also (17)). The body of the function definition is made up of other functions or sequences of functions. In the case of "cycles", the body of the function definition is a "repeat" loop that is to be iterated a times. In the next paragraph you see how easily one can specify a sequence of a clock cycles with this newly defined function. (RNL has a clock function "c", which you could have used instead. We defined our own clock function here in order to illustrate the power of "defun". Another important function for you to explore with the help of the RNL User's Guide is "do".)

(17) Set the input to the shift register to High and propagate it through the shift register for four cycles, using the "cycles" function. Then look at the outputs of latches 3, 4, and 5.

```
h in
done


(cycles 4)

Step begins @ 280 ns. cl=1 @ 0 in=1 @ 0 STATE AT END OF SIMULATION STEP: Current time= 290 in=1
out.10=0 cl=1
```

```
Step begins @ 290 ns.  cl=0 @ 0 out.1=1 @ 1.2 STATE AT END OF SIMULATION STEP: Current time= 300
in=1 out.10=0 cl=0

Step begins @ 300 ns.  cl=1 @ 0 STATE AT END OF SIMULATION STEP: Current time= 310 in=1 out.10=0
cl=1

Step begins @ 310 ns.  cl=0 @ 0 out.2=1 @ 1.2 STATE AT END OF SIMULATION STEP: Current time= 320
in=1 out.10=0 cl=0

Step begins @ 320 ns.  cl=1 @ 0 STATE AT END OF SIMULATION STEP: Current time= 330 in=1 out.10=0
cl=1

Step begins @ 330 ns.  cl=0 @ 0 out.3=1 @ 1.2 STATE AT END OF SIMULATION STEP: Current time= 340
in=1 out.10=0 cl=0

Step begins @ 340 ns.  cl=1 @ 0 STATE AT END OF SIMULATION STEP: Current time= 350 in=1 out.10=0
cl=1

Step begins @ 350 ns.  cl=0 @ 0 out.4=1 @ 1.2 STATE AT END OF SIMULATION STEP: Current time= 360
in=1 out.10=0 cl=0


4



! out.3  out.4  out.5
out.3=H (vl=0.30 vh=0.80) (0.038400 pf) affects: input to functions for the following nodes:      39      39
      32      32

out.4=H (vl=0.30 vh=0.80) (0.038400 pf) affects: input to functions for the following nodes:      51      51
      44      44

out.5=L (vl=0.30 vh=0.80) (0.038400 pf) affects: input to functions for the following nodes:      63      63
      56      56
```

At the end of (16), all the cells in the shift register were at Low. After setting the input to High and applying four clock cycles, the High input should have arrived at "out.4". The function "cycles" made it very easy to apply the clock cycles. The result is as expected.

(18) Open a log-file to record the activities of the RNL session. Define a node vector and change the report at the end of a simulation step using the vector.

```
(log-file  "shift.log")

49868


(defvec  '(bin  path  in  out.1  out.2  out.3  out.4  out.5
                          out.6  out.7  out.8  out.9  out.10))

(bin path <node in=H> <node out.1=H> <node out.2=H> <node out.3=H>
<node out.4=H> <node out.5=L> <node out.6=L> <node out.7=L>
<node out.8=L> <node out.9=L> <node out.10=L>)


(def-report  '("State of Shift Path: "  (vec  path)))

("State of Shift Path: " (bin path <node in=H> <node out.1=H>
<node out.2=H> <node out.3=H> <node out.4=H> <node out.5=L>
<node out.6=L> <node out.7=L> <node out.8=L> <node out.9=L>
<node out.10=L>))


s


Step begins @ 360 ns.
State of Shift Path:
Current time= 370
path=0b11111000000
```

The "log-file" command has the effect that the file *shift.log* is opened and that all subsequent terminal activities will be recorded in this file. You can later analyze this file or edit it to keep parts of your interactive RNL session for inclusion in a control file (*.l* file). The number returned is the file identification (ID) of *shift.log*. You can close the log-file with the command (log-file nil). Alternatively, the log-file will automatically be closed when you exit RNL (we will close the log-file in this way).

The "defvec" command defines a data structure called a vector. A vector is a list of nodes. The value of a vector is determined by the value of its nodes. For example, if a vector has three nodes with the values 0, 1, 0 (Low, High, Low), the value of the vector would be 010 binary, or 2 decimal. There are a number of commands operating on vectors, e.g. assigning a value to a vector (see RNL User's Guide). The vector definition has the format (defvec '(base name node_1 node_2 node_3 ...)). "base" is the base of a number system and can be bin for binary, oct for octal, hex for hexadecimal, and dec for decimal. When the value of the vector is printed, it is given as a number with the base given in "base" (see "def-report" below). "name" is the symbolic name of the vector, which is "path" in the example. "node_1", "node_2", "node_3", ... are the nodes of the vector. In the example they are "in", "out.1", "out.2", ... The list returned after "defvec" is the internal representation of the vector.

The "def-report" specifies a new format for the report at the end of a simulation step, thus overriding the "def-report" given in the *shift.l* file (however, this does not influence the reports on value changes of nodes marked with "chflag"). With (vec path) you specify the inclusion of the vector "path" in the report (for more variations of the "def-report", see RNL

User's Guide). A simulation step following the report definition illustrates the new report format. Since the base given in "defvec" is binary, the vector is printed as a binary number. The first two characters, 0b, indicate the base (they would be 0 for octal, 0x for hexadecimal, and none for decimal). The binary vector representation provides a good "visual" picture of the shift path. The input and the first four cells are High, the other cells are Low.

(19) Open a file to store RNL output ("behavior" file) for subsequent printing on a Printronix printer. Then shift an input signal through the shift register and exit RNL. (The RNL output in the "behavior" file will be analyzed with the program MTP).

```
openplot  "shift.beh"

nil

l in

(cycles 1)

h in

(cycles 1)

l in

(cycles 1)

(cycles 10)

exit
```

After the file *shift.beh* has been opened with the "openplot" command, information on all nodes whose change-flag is set will be stored in this file until the file is closed with "closeplot". Terminating RNL with "exit" will automatically close the file, in which case no "closeplot" is needed. ("openplot" returns "nil" after opening the plot file. The responses for the commands following "openplot" are reports similar to those already discussed and have therefore been omitted in the above text.)

Recall that we set the change flag for "nodes" with the control file *shift.1*. Later we interactively set the change-flag for "all_nodes". Therefore, information on the following nodes will be stored in *shift.beh*:

```
in  cl  out.1  out.2  out.3  out.4  out.5
        out.6  out.7  out.8  out.9  out.10
```

To obtain a signal that can be easily identified in the printout, we apply a pulse at the input of the shift register by setting the input Low, High, and Low again, each followed by a clock cycle. Then we shift this pulse through the register with ten more clock cycles. With the last command we exit RNL.

4.3. Defining Control and Stimulus Files the "Easy" Way.

The definition of control files for RNL is generally much more difficult than defining netlists for circuits. The reason is that many more lisp commands are necessary in the definition of control files. Two programs have been defined to substantially reduce the need of understanding lisp, although the user may benefit from understanding lisp. The first program is called 'GEN_CONTROL'. It is intended to create a set-up file for a circuit using answers to the questions asked by this program. It formats the answers so that the output file can be read by RNL.

The second program is called 'GEN_TIME'. This program uses a description of patterns in a very simple format, without the poisonous lace of parentheses and quotes and creates an output file which can be called by the control file created with the gen_control command.

A couple of conventions:

GEN_CONTROL will create a control or ".l" file: '*basename.l*'.

It expects a network by the name of '*basename*'.
Results will be stored in the '*basename.log*' file .
and the information for plotting is stored in the
'*basename.beh*'

Furthermore, the pattern information file will have
the name '*basename.time*'.

By the way, the user will be prompted for the actual 'basename' name.

GEN_TIME will use the '*basename.stim*' file as an input file and create a '*basename.time*' output file.

4.3.1. Using GEN_CONTROL to Generate a Control File.

Assume you created the shiftregister defined in section 3.2.3. and you have the network file "*shift*". Now run in the same directory:

% gen_control (no arguments).

The response is:

```
*
Interactive generation of control files for RNL.
UW/VLSI CONSORTIUM VERSION.
Filename extensions assumed : '.l', 'none', '.beh', '.log'   *
```

The first prompt is for the basename of the circuit; GEN_CONTROL will assume the conventional extensions as discussed before. Each response to the prompt is terminated with a **<CR>**. In this case type:

shift<CR>

The second prompt is for a comment line to be included in the control file for identification

purposes:

```
"Enter any comment line (80 characters max) you wish:"
```

For example type:

this is a shiftregister test circuit< CR>

The third prompt is for the duration of every simulation step in 0.1ns units.

```
"Enter the time scale for simulation in 0.1 ns units:
iner = "
```

For example respond with: **1000< CR>**

The fourth prompt is a definition of nodes to be defined as a unit in a vector. Vector definitions are used in the def-report and allow the user to print the state of clusters of nodes in a very compact manner. Note that its only use is for state reporting purposes at the end of a simulation step. It has no impact on the reporting on intermediate changes! Let us define a vector called "state" with nodes out.1 out.2 ..... out.10. The type of this vector is simulation). By the way, the type definition only affects the format of the report generated by RNL; it does not require that you use the same format when using 'invec'! The prompt is:

```
"Define input vectors in standard format, if any.
Reply with <CR> if no (more) definitions required.
Vectorname = "
```

So respond with:

state< CR>

the prompt then is for the type:

```
"Vector type (bit/bin/oct/dec/hex) = "
```

So respond with:

bit< CR>

the next prompt is for the nodenames in the vector:

```
"Elements of the vector definition (must be individual node names),
spaced one blankspace apart:
nodes in vector = "
```

respond with:

out.1 out.2 out.3 out.4 out.5 out.6 out.7 out.8 out.9 out.10< CR>

Now you will be prompted again for a vector definition:

```
"Vectorname = "
```

There is nothing more to define, just respond with:

< CR>

Now you will be allowed to define vectors in a simplified fashion; for example the vector 'state' if it could be re-baptized in 'out' can be easily defined as a single indexed vector, with starting index '1' and number of elements '10' and the function will assume vector elements out.10 ....out.1.

Since we already typed in the vector in the standard way, we will ignore this option and type **< CR>** .

Getting even more sophisticated, we can use double indexing of nodenames and create several defvec's all at one time. So a set of vectors are to be defined v_1 .... v_n (note no periods in names of vectors!) with nodenames v.i.1 through v.i.m is easily accomplished by properly

responding to the prompts for type, basename and values for n and m.

This option is not appropriate in this example and we type again <CR>.

Next you will be prompted for the format of the report you want RNL to provide. There are in fact two independent type of reports generated; one relates to the state of the network at the end of a simulation step and the other to intermediate changes occurring in the network. The answer to this prompt will only affect the report at the end of a simulation cycle. The report includes comments to be printed, nodenames to be reported on and vectornames to be reported on. The comment comes first, nodes and vectors in any sequence of choice. This format definition must always be present in an RNL definition, otherwise you may not get any reponse at all from running a simulation! The prompt is:

```
"Define the output nodes reported by RNL (required!)
Select standard type by responding with <CR>
Select indexed type by responding with <any char> <CR>
Selection = "
```

There can be two types of reports selected, one which is straight a set of nodenames and vectors and the other which allows sets of ectors to be defined. We ignore this latter option and type <CR>.

```
"Enter comment line (if any) "
```

First respond to the request for a comment:

response = <CR>
(just <CR> is fine if no comment is desired)
The next prompt is:

```
"Identify the nodenames and vectornames to be reported.
In case of a vector, type 'vec' and you will be prompted for its name.
Type a nodename or 'vec' : "
```

Now respond to the request for node and vector names to be reported on; we would like to show 'cl' and 'in':

cl in<CR>
The prompt becomes again:

```
"Type a nodename or 'vec' : "
```

since we still have the vector state to be included in the report format definition we type:

vec<CR>
GEN_CONTROL now asks for the vectorname:

```
"Name vector = "
```

respond with the name of the vector state

state<CR>

The program then asks for additional vector or nodenames:

```
"Type a nodename or 'vec' : "
```

Since we are done, we terminate the prompting for the report format by typing:

<CR>

Next we are prompted for logic analyzer style output. If affirmative, a lot of extra text is removed from the output report from RNL and the output becomes pretty much tabular. For example, instead of printing 'cl=0' only '0' is printed. RNL, before generating any output, will automatically print the order by which the reported states appear. The prompt is:

"Do you wish to specify logic analyzer output (no = <CR>) ? "

We respond by:

y<CR>

Next we are prompted by:

"Do you wish to turn on glitch detection (no = <CR>) ? "

If affirmative than the nodes which we will identify in the following prompt will only be reported if they are subject to more than one change in state during a simulation cycle.

Let us turn this on too (later try what happens if you leave it off):

y<CR>

Now we are getting to the definition of the second type of report which can be generated by RNL: reporting of intermediate changes in specified node states. When a node, which nodename was included in the 'chflag' list, changes state: its name and time of change is reported either in relative time (default) or in absolute time units. However, if the glitch-detection has been turned on, this only occurs if the node changes more than once; the latest change is the one that is reported.

Let us at least (with the glitch-detection on) report any glitches on the nodes indentified in our response to this prompt:

out.1 out.2 out.3 out.4 out.5 out.6 out.7 out.8 out.9 out.10<CR>

GEN_CONTROL comes back with a prompt for reporting intermediate changes. Like in the case of defvec's we can enable individual nodes to be defined or sets of single indexed nodes or even sets or double indexed nodes. We do not select this option here so only single nodenames are entered.

The prompt is:

"Define the nodes with transients to be reported.
Nodes can be defined individually,
by SINGLE INDEXED vector or DOUBLE INDEXED vector;
Type node name, 'vec', 'veci' or
<CR> if (no more) nodes to be defined.
Enter node name, 'vec' or 'veci' (or <CR>) : "

Let us enter just the nodes straight:

out.1 out.2 out.3 out.4 out.5 out.6 out.7 out.8 out.9 out.10<CR>

The next prompt is:

"Continue with individual node names or a <CR> : "

Terminate prompting for nodenames by:

<CR>

The next prompt again is:

"Enter node name, 'vec' or 'veci' (or <CR>) :

Respond with: <CR>

Another option available in RNL is define a logic trigger condition, at which time you execute commands in a special file. We do not select this option and we type <CR>.

We may want to define additional simulation set-up commands like print a message or whatever (now

you get into lisp!) (if you do not want anything just type a <CR>); the prompt for this is as follows:

```
"If desired, type additional RNL simulation SET-UP command lines:
(if not: just type <CR>; same after lines have been entered)"
```

We respond by asking that an announcement be printed when RNL has loaded all files and starts the actual simulation:

```
(printf "Let us start simulating now ....\n ")<CR>
```

There is no other prompt until we terminate the loop by typing another:

```
<CR>
```

Next we are prompted for a timing information file:

```
"Do you wish to specify a shift.time file      (no = <CR>) ? "
```

Most of the time, also in this case, you want one generated by GEN_TIME.

Respond with:

```
y<CR>
```

(In fact: any_character<CR> will do;
just a <CR> means no)

Finally there is a prompt for a wrap-up command:

```
"If desired, type additional WRAP-UP RNL command lines:
(if not: just type <CR>; same after lines have been entered)"
```

You can type in the 'exit' command if you want RNL to terminate immediately after executing the instructions in the '*time*' file (not recommended) or since we are fancy: a message saying simulation complete:

```
(printf ".....simulation done\n ") <CR>
exit<CR>
```

There are no prompts until an additional <CR> is typed:

```
<CR>
```

Finally: GEN_CONTROL indicates that it is terminating and that it has created a control file with the name *basename.l* (in this case the basename is '*shift*'):

```
"GEN_CONTROL COMPLETED.
Correct errors in shift.l using a standard text editor."
```

You should look at this file with your favorite text editor and make corrections using this text editor (GEN_CONTROL does not have any utilities for making corrections). You will note that all set-ups necessary for simulation are included, including the loading of libraries, reading the network and defining the report formats with the proper syntax (notice where the quotes and parentheses go!). We have now completed half of the work: the definition of a timing pattern remains to be done. That is the subject of the next section.

4.3.2. Generation of Stimulus Pattern Files.

Let us create a file called '*shift.stim*' using your favorite text editor. In this file we first have to define the number of simulation steps in units of the value of 'incr'. Let us assume we want to run 50 simulation steps, starting at 0:

The first non-comment line (comment lines start with a semicolon) contains:

```
time_range 0 50
```

Next we specify activities on the input nodes cl and in. Now remember that the first number after the nodename relates to the period of the signal applied to that node. This is also true for vector type

stimulus (invec and bitinvec). The clock has to run all the time, the minimum period therefore is 2; at time 0 the clock goes high and at time 1 the clock goes low again. The syntax is: nodename period state1 time1 state2 time2 .... (as many state changes you wish to define within the period). A value for periods equal to 0 means in reality infinity: this pattern is a one-time event cycle. For the clock signal this means the following:

cl 2 h 0 l 1

We have to specify an input pattern to in: let us start with setting in low at t = 0 and high at t = 15 and back low at t = 30. This results in the following command line:

in 0 l 0 h 15 l 30

Finally, we MUST define how often we want the state of the network reported. Let us assume we want a report every 2 time increments at the end of every 1st increment within the period:

report 2 1

(If you want a report at the end of every simulation step you type: **report 1 0**)

There are additional commands available for applying vectors, putting masks in to create bursts of signals and for including normal lisp commands. Refer to the documentation for those. Store the file in '*shift.stim*' and run the following command:

% gen_time shift.stim shift.time

This program terminates very quickly. Please inspect the *shift.time* file created by GEN_TIME. You will find that all signal activities are sorted by time increment and in a format suitable for RNL. In fact the '.time' is for control what the '.sim' file is for a netlist description.

You are now ready to run the simulation; the command line is:

% rnl shift.l

see what happens!

Since we included 'exit' in the control file, RNL terminates; if we did not include this command, RNL would be waiting for additional commands from the keyboard terminal. Options are:

additional RNL commands (fully interactive mode)

exit (terminate RNL simulation and store the results in '*shift.log*'
^Z to halt RNL to allow definition of an additional timing file to start at the end
of the current simulation cycle (e.g. 50); thereafter you restart RNL with 'fg'
and type: **(load "shift.time")**
DO NOT TYPE ^C which will terminate RNL, without however
storing the results of simulation in 'shift.log'.

4.3.3. Tying All the Programs Together With a Makefile.

We have identified several programs necessary to run a complete simulation. Rather than running these programs individually a very simple makefile can do the same for you.

Remember that in this section, source files that do not depend on outputs of programs are:

shift.net for the definition of the netlist description and
shift.stim for the stimulus description.

In the concept presented:

shift.l is generated in an interactive fashion by GEN_CONTROL.

The rest is generated from these input files. The make program keeps tabs on when those files have been updated and runs certain programs again, if necessary, to get a valid simulation result.

Include the makefile shown in the appendix in the same directory along with your other files needed for simulation, with a text editor in the makefile substitute *N = whatever* with *N = shift* (or another circuit basename). Back in the unix r' -ll type

```
% make
```

and all missing files will be generated and if the source files are not present, it will tell you so. There are certain arguments you may give to do only a portion of the generation of necessary simulation files. Refer to the summary command list in the index.

4.4. Printing RNL Output Using MTP

The program MTP has been developed for printing the output of RNL simulations on the Printronix dot matrix printer. MTP stands for Multiple Time-series Plot. From the behavior file produced by RNL at the end of the last section a printout of the signals can be obtained in three steps. You have to (a) create a file containing directives for MTP, (b) create a plot file and (c) send the plot file to the Printronix printer.

(a) MTP has been designed to plot a number of different kinds of behavior files in a number of different formats. However, for the purpose of plotting the output of RNL only a few directives need be supplied. These are the following

START time

The START directive tells MTP when to start plotting (in nanoseconds). If not supplied its default value is 0. Data is skipped on the behavior file until an event is found whose time is greater than or equal to the START time.

STOP time

The STOP directive tells MTP when to stop plotting (in nanoseconds). STOP has no default value and must be supplied. If the STOP time is greater than the time of the last event on the behavior file, the plot will be concluded with the last event.

SCALE time

The SCALE directive tells MTP how many time units to plot per inch on the plot (nanoseconds / inch). The default value is 100.0 which is an appropriate value for RNL.

Signal selection and trace format directives

MTP does not plot every signal in the behavior file but only those that are specifically requested. This permits experiments which generate a large number of traces to be analyzed selectively. MTP provides several trace formats which can be used for analog and data domain values but the simplest and most useful for RNL is the LOGICAL format. To select signals A, B and C for plotting in LOGICAL format the necessary MTP directives are

```
Logical A
Logical B
Logical C
```

The order of the traces on the plot is determined by the order of the selection directives

in the file. The first signal selected is plotted closest to the time axis. There can be a maximum of 20 signals selected on a given plot.

All MTP directives are case insensitive except for signal names and are free field, separated by blanks or CR.

For our example write the following into a directives file named *shift.dir*:

```
START 0.0
STOP  700.0
SCALE 100.0
LOGICAL in
LOGICAL cl
LOGICAL out.1
LOGICAL out.2
LOGICAL out.3
LOGICAL out.4
LOGICAL out.5
LOGICAL out.6
LOGICAL out.7
LOGICAL out.8
LOGICAL out.9
LOGICAL out.10
```

(b) To create the plot file, which is to get the name *shift.plot*, enter:

```
% mtp  shift.beh  shift.dir  shift.plot  <CR>
```

MTP will provide information on its progress and echo the content of the directives file:

```
Select and preprocess input data
START 0.0
STOP  700.0
SCALE 100.0
LOGICAL cl
LOGICAL out.1
LOGICAL out.2
LOGICAL out.3
LOGICAL out.4
LOGICAL out.5
LOGICAL out.6
LOGICAL out.7
LOGICAL out.8
LOGICAL out.9
LOGICAL out.10
Sort preprocessed events
Generate the plot
Rasterize for the Printronix
mtp complete, plot file is shift.plot
```

(c) To send the plot file to the Printronix printer enter:

```
% lpr -l shift.plot  <CR>
```

This will produce the printout shown on the following page. No signals are plotted before 370 ns (0.3700e+03), since we opened the behavior file only at that time. Remember, that at this time the outputs of the first four cells of the shift register were High, the other six outputs were Low. You can see this state of the register move through the output "out.10". You can also see the input signal move through the register immediately afterwards.

5. Summary and Outlook

The simulation of the inverter and the shift register are examples of what you might encounter if you attempt to model and simulate your particular application with NETLIST and RNL. We were not able to look at all of the commands available in RNL and NETLIST, and therefore concentrated on some of the most frequently used ones. You will find complete lists of commands for both NETLIST and RNL in the references listed in the appendix.

The LISP-like command interpreter used by both NETLIST and RNL provides the facilities, and enables you to create your own special tools, for simulating very complex circuits. There are several ways to tackle the intricacies of RNL LISP. In addition to studying the User's Guides, you may work through examples of elaborate simulations, such as the simulation of the microcode sequencer referenced in the appendix. Another possibility is to have a close look at the function definitions given in the files *uwstd.l* and *uwsim.l*. Also, especially if you are fond of languages, you may want to study LISP in its "pure" form, without commands particular to RNL and NETLIST.

Whatever you do, keep in mind that RNL is a simulator based on a model of the real circuit, and therefore it is wise to know the assumptions underlying the model as well as the limits of its applicability. Information about the theory of NETLIST and RNL is provided in Chris Terman's original User's Guides and his thesis referenced in the appendix.

Appendix 1 - Further References

You may need information from the following sources if you want to use NETLIST or RNL more extensively.

From UW/NW VLSI Consortium, VLSI Design Tools Reference Manual:

1. NETLIST User's Guide (Contains, among other information, a list of all NETLIST commands.)

2. PRESIM User's Guide (Contains, among other information, specifications for the *config* file.)

3. RNL User's Guide (Contains, among other information, a list of all RNL commands.)

and in addition,

4. User's Guide to NET, PRESIM, and RNL/NL, Christopher J. Terman, M.I.T. Laboratory for Computer Science.

5. Simulation Tools for Digital LSI Design, (Thesis), Christopher J. Terman, M.I.T. Laboratory for Computer Science.

6. Simulating a Microcode Sequencer Using RNL: An Annotated Example of RNL Usage, Robert J. Fowler, UW/NW VLSI Consortium.

7. LISP, P.H. Winston, B.K.P. Horn, Addison-Wesley Publishing Company, 1981.

8. Metamagical Themas, The Pleasures of LISP: the chosen language of artificial intelligence, D.R. Hofstatter, article in three parts published in Scientific American, March and April 1983. .RE .LP

Appendix 2 - Description of the .sim file of the example "inverter" (section 3.1.2).

The *inverter.sim* file produced in section 3.1.2. contains the following:

```
| units: 250.00   tech: ???   format: MIT
p in out Vdd 8.00 8.00 r 0 0 64.00
e in GND out 8.00 4.00 r 0 0 32.00
c out 3.000000e-02
```

Lines beginning with a vertical bar are considered comments by PRESIM, unless they have an entry "units:" or "format:". "units:" gives the conversion factor to centi-microns. "format:" is one of "MIT" or "UCB" (or, if no format is given, the old format originally used by the program is

assumed). "tech: ???" is the default comment indicating the technology used. You can change this comment with the -t option to NETLIST.

The following explanations relate to the MIT format, since the *.sim* file of the example has MIT format.

Lines 2 and three specify a p-channel and an e-channel transistor, respectively. Each of them is followed by the names of the nodes to which it is connected (gate, source, drain), by the length and width* of the gate area in units of lambda, by a geometrical descriptor for the gate area which is always set to r (rectangular) by NETLIST, by its layout positional coordinates (NETLIST always specifies 0 0), and finally by the gate area in square-lambdas.

The last line of *inverter.sim* specifies the output capacitance between out and GND, again with positional coordinates given as 0 by NETLIST, and a value of 0.03 pF.

(For more details on the *.sim* file formats read the on-line manual with the UNIX command **man 5 simfile**).

Appendix 3 - Preparation of a simple "config" File

The PRESIM User's Guide provides details on how to prepare configuration files. It lists all the possible parameters together with their default values. Generally, you need different configuration files for different technologies. Since we use a p-channel transistor, we need to use the configuration file to tell PRESIM the appropriate values for this device. The *config* used in the examples of this tutorial contains the following specifications:

```
resistance enh static 10 10 100000
resistance enh dynamic-high 10 10 10000
resistance enh dynamic-low 10 10 10000
resistance p-chan static 10 10 200000
resistance p-chan dynamic-high 10 10 20000
resistance p-chan dynamic-low 10 10 20000
```

Appendix 4 - The "alias" file of the example "shift" (section 3.2.3).

The alias file shift.al, created by NETLIST in section 3.2.3. has only one line:

```
= in out.0
```

This tells PRESIM that "in" and "out.10" have been connected and therefore are considered as representing the the same node.

* Note that the sequence of length and width is reversed in comparison to transistor specification with (ptrans ...), (etrans ...), etc.

Appendix 5 - Summary of Typical Commands Associated with Simulation Tools

netlist basename.net basename.sim

purpose: create a flattened netlist representation from an hierarchical representation.

files:

basename.net	(created by the user)
basename.sim	(is output of netlist.)

presim basename.sim basename config

purpose: create a binary circuit representation from the flat circuit representation.

files:

basename	(is created using NETLIST, input to PRESIM)
basename	(is output of PRESIM)
config	(is technology file provided to the class)

rnl basename.l

purpose: run the simulation

files:

basename.l	(created by the user, generally with GEN_CONTROL, calls out the other files for input and output: uwstd.l (standard library routines) uwsim.l (more standard library routines))
basename.log	(will contain all the stuff displayed on the terminal after exiting RNL using 'exit' or '(exit)')
basename	(binary network representation created using PRESIM)
basename.evl	(contain all plotting information created as a result of the chflag command.)

gen_control

purpose: set up a control file for a circuit for the first time.

files: none needed for input, however GEN_CONTROL will prompt for a basename and an output file basename.l will be created.

gen_time basename.stim basename.time

purpose: convert a simple timing file in a RNL compatible format.

files:

basename.stim	(input file for GEN-TIME; specifies input waveforms in terms of nodename, period and intra-period signal changes.)
basename.time	(output file which is "loaded" into the control/basename file.)

pla2net basename

purpose: generate a NETLIST macro for the pla from the truth table.

files:

basename.tt	(input file for PLA2NET, containing the truth table.)
basename.net	(output file created by PLA2NET containing a macro with the name 'basename': (macro basename out in))

Appendix 6 - Makefile Commands.

Step 1:

change the variable N to equal the basename of the circuit to be simulated in the 'makefile' file, unless you want to include in the command line every time 'N=basename'.

Step 2:

run the appropriate command:

make runs all programs using the basename.net, basename.stim and config files as the original source files only when necessary. Where files do not exist, the programs to create them will be run.

make c	remove all derived files: .sim, binary file, .time, .log, .al, and .beh files
make n	creates the .sim file running NETLIST
make p	creates the binary network file running PRESIM
make r	runs RNL
make l	create a control .l file for RNL running GEN_CONTROL
make t	create a timing .time file by running GEN_TIME
make filename	create the filename requested by running the appropriate programs.

Appendix 7 - A Simple Makefile

```
# makefile for RNL
# the value of name should be replaced by the basename of the circuit
# to be simulated on the next line: e.g. N = shift for simulation
# of the shift register.
N = shift

# dependency information
$(N).log: $(N).l $(N).time $(N)
	rnl $(N).l

$(N): $(N).sim config
	- presim $(N).sim $(N) config

$(N).l:
	/usr/new/gen_control

$(N).time: $(N).stim
	/usr/new/gen_time $(N).stim $(N).time

$(N).sim: $(N).net
	netlist $(N).net $(N).sim


# datacalculation:
n: $(N).sim
p: $(N)
r: $(N).log
l: $(N).l
t: $(N).time

c:
    rm -f     core $(N).sim $(N) $(N).time $(N).log $(N.al)*
```

NETLIST/PRESIM/RNL - A TUTORIAL

Robert Daasch
Robert Fowler

UW/NW VLSI Consortium
Sieg Hall, FR-35,
University of Washington,
Seattle, WA 98195
TR #84-07-01

1. INTRODUCTION

The following document is intended for beginning to intermediate users of the following programs;

- NETLIST,
- PRESIM,
- RNL.

Some familarity with programming and the use of a computer terminal is assumed.

The approach used here is to provide examples that have been developed while using the programs here at the UW/NW VLSI Consortium. They are by no means exhaustive. Much of our attraction to these programs is their flexibility. This is particularly true in the RNL interpreter. When using this tutorial copies of the User's Guides (provided separately) should be available.

1.1. Ground Rules

The reader is encouraged to be sitting in front of a terminal that has these programs available. Much more can be learned by making the unavoidable mistakes when editing files and running these programs than by just reading. Error messages are at times cryptic and we make no effort here to wade through them. Readers are also encouraged to experiment and implement their own ideas. One very instructive method is to take these examples and modify and/or add extra capability. Learn by doing.

In sections where readers are expected to be editing; the text to be entered is in **bold face**. In the sections on the interactive use of RNL; user input will also in **bold face**. Program responses are in normal text. We recommend that an editor that supports Lisp (e.g. EMACS) be used if at all possible.

We make such a statement as both the NETLIST program and the RNL command interpreters are based on a Lisp syntax. That is to say program statements (commands) are surrounded with parentheses *( )*. A general template for a command is

(command_name arguments).

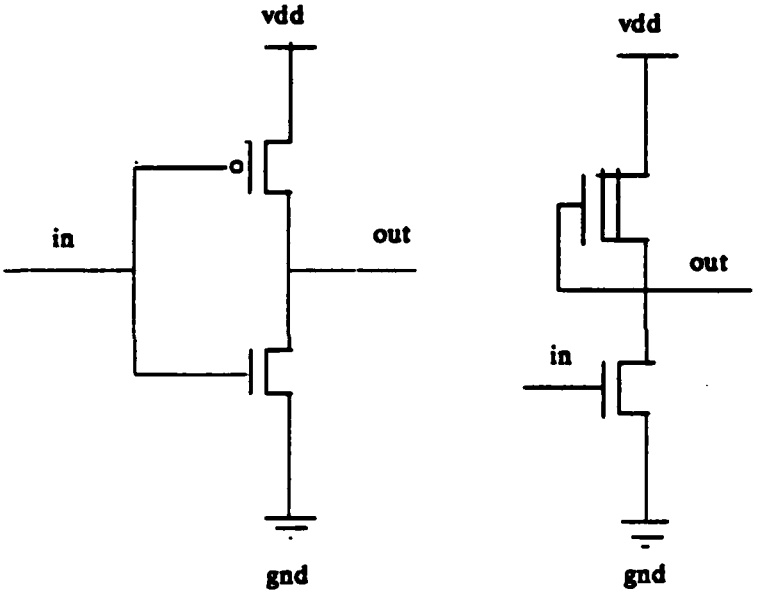
It should be assumed that all commands require the parentheses. It will be stated explicitly if they are not required.

1.2. Document Structure

We begin by discussing some of the basic statements used in writing NETLIST programs. This will be followed by the generation of the so-called .sim file of transistors. This file is the basic input for many simulators. Examples of using the program PRESIM, a .sim file preprocessor that generates input for the digital simulator RNL, are then presented. Simple interactive RNL experiments are shown and techniques for running RNL in batch mode are described. We end with some of the basic concepts of Lisp and a tour through some of the Lisp code that has been written locally to facilatate the use of RNL.

2. BUILDING A NETWORK DESCRIPTION USING NETLIST

For effective design it is important to establish that the design will work before layout is attempted. The program NETLIST allows the user to describe the circuit with a symbolic language. The NETLIST description is really a program which when run produces the list of transistors that make up the circuit. The following is a simple NETLIST program for a CMOS inverter. These commands will then be supplemented with others that will allow larger circuits to be partitioned.



CMOS inverter NMOS inverter

2.1. Simple Commands

```
; All text following a semicolon is a comment and is ignored          ; (1)
; A CMOS inverter p type device 2X width of n device                  ; (2)
(node in out)                                                        ; (3)
(ptrans in out vdd 8 8)                                              ; (4)
(etrans in gnd out 4 8)                                              ; (5)
(capacitance out 0.03)                                               ; (6)
```

1 As indicated by this line all text that appears after a semicolon (;) is considered a comment and is ignored by NETLIST.

3 This line declares the nodes *in* and *out*. You have to declare each node that you use. Nodes declared with the node command will be referred to as global nodes. Two global nodes that NETLIST knows about without your explicitly declaring them are *vdd* and *gnd*. Some programs are case sensitive and it is recommended you use them

as they are shown here. This redundant piece of information (after all, NETLIST can see that you are using a symbol as a node name when it builds the circuit) prevents spelling errors from causing unnecessary grief. Declarations are not as much trouble as they sound. Later a scheme will be presented that structures the NETLIST definition so that most nodes will be local to some module. Local nodes will be examined shortly. Using this technique only the few global nodes (usually clocks and i/o signals) have to be declared.

4 to 5 For simple circuits these are the two commands that do a lions share of the circuit description. They identify an individual transistor. They come in several types such as *etrans -> n type enhancement*[1], *ptrans -> p type enhancement and dtrans -> NMOS depletion transistor*. There are others and the interested reader is encouraged to examine the NETLIST User's Guide to find out more. In most CMOS designs etrans and ptrans should suffice. The template for any of the transistor types is

(type gate source drain width length).

The type is as described above. The gate, source and drain arguments are the terminals of the transistor being declared. In line 4 the p type transistor is gated by node *in*, its source is the node *out* and its drain is *vdd* (Note the case of *vdd* and *gnd* in lines 4 and 5). Source and drain in NETLIST is used solely to distinguish between the terminals of the transistor and do not imply anything about actual operating potentials. Width and length specify the size of the transistor. The values are given in a length parameter lambda. This allows for some technology independence in the network description. This unit is also used in layout programs. Typical values for lambda are 2-3 microns. In the inverter description above then, the n type transistor (line 5) is 1/2 the width of the p type.

6 Finally some capacitance is modeled on the node *out* with the use of the **capacitance** command. The user is relieved of specifying the second terminal on the capacitor because all are assumed to ground. The values (*0.03*) are in units of picofarads.

This file is then used as input to the program NETLIST. The actual running of this example is deferred momentarily as some additional NETLIST commands are investigated.

2.2. Additional Built-in Functions

Up to now we have used the transistor commands (**etrans** and **ptrans**) and the capacitance command. If this was all that was available life would indeed be tough. The general requirements for additional commands are function type, a technology and device sizes. Specification of the technology is important because NMOS uses depletion pullups whereas CMOS uses p type enhancements. This requires a slightly different handling of the signals. In NETLIST such commands exist and we will go through some now.

A CMOS inverter has the following template

(cinvert out (in width length))

Cinvert is the command name (like **etrans** above) and is followed by the argument declaring the output signal (*out*). The next element of the command may look a bit strange but in conveys a lot of information. It is in fact a data structure we will be seeing often, the details of which are deferred to section 5 of this tutorial. For this example, it is declaring the input signal to be *in* and it defines the size (*width* and *length*) of the n type transistor in the CMOS inverter. This nearly satisfies our requirements but note that in the **cinvert** command *(in width length)* only specified the size of the n type transistor. Where is the p device size declared?

---

[1] Historicaly NETLIST was written to describe NMOS circuits where there is just the one type of enhancement transistor.

This is a historical artifact of the NETLIST program. In NMOS the pullup is a depletion mode transistor which has its gate tied to the source. In the case of an NMOS inverter then the gate of the depletion device is in fact the output. (This is also true of nand and nor gates.) For the special case of NMOS design, we have a command that looks like

(invert (out width length) (in width length)).

As you can see the depletion pullup's size is declared on the output node. Similarly NMOS nand and nor gates have the same form[2]. In this context then the structure

(node_name width length)

specifies the size of the transistor gated the node node_name.

In CMOS the input gates both the p and n transistors. Moreover, nand and nor gates have equal numbers of pullup (p type) and pulldown (n type) transistors, the sizes of which could vary independently. Clearly some other solution must present itself.

The command **ratio** is the current solution to this dilemma. Its template is

(ratio value)

Ratio is the command name and *value* is a constant that is used to set the width of the p device. The p device's width is the product of *value* times the width of the n device ($p sub width "=" value "*" n sub width$). The default for *value* is 2.0. The lengths of the two transistors are assumed equal. This doesn't not allow for complete independence of device size but has worked well in practice.

Returning to our need for a CMOS inverter command, we are left with the following

```
(node in out)                    ; (1)
(ratio 2.0)                      ; (2)
(cinvert out (in 4 8))           ; (3)
```

Of course we still need the **node** command as before. The last two commands are equivalent to the transistor commands discussed earlier.

```
(ptrans in out vdd 8 8)
(etrans in gnd out 4 8)
```

Its hard to see the gain with this example but if we consider the two possiblities for CMOS nand and nor gates the advantages start to present themselves. Within this scheme one could guess the commands for a 3 input nand to be,

```
(node out in1 in2 in3)           ; (1)
(ratio 2.0)                      ; (2)
(cnand out in1 in2 in3)          ; (3)
```

Again input and output nodes have to be declared with **node**. By dropping the width and length arguments for the inputs we have assumed default sizes for the enhancement transistors (2 lambda x 2 lambda). The **ratio** command sets the p devices to be two times the width of their corresponding n type just as before. The equivalent transistor description in this case is

---

[2] For example a complete specification of 2 input nand and nor gates,

(nand (out width length) (in1 width1 length1) (in2 width2 length2))

(nor (out width length) (in1 width1 length1) (in2 width2 length2))

getting quite large

```
(node out in1 in2 in3 1 2)                ; (1)
(etrans in1 1 out)                        ; (2)
(etrans in2 2 1)                          ; (3)
(etrans in3 gnd 2)                        ; (4)
(ptrans in1 out vdd 4 2)                  ; (5)
(ptrans in2 out vdd 4 2)                  ; (6)
(ptrans in3 out vdd 4 2)                  ; (7)
```

1 to 3 If we explicitly describe the 3 input nand we have to declare 2 additional nodes. Nodes *1* and *2* are not particularly interesting as their only function is describing the connectivity to ground (*gnd*) from the output node *out* (lines 2 and 3). Note when we used the built-in function they needn't be declared. NETLIST recognized the need for these "local" nodes and generated their names automatically. NETLIST always uses numeric node names for local nodes and the user is strongly advised to avoid their use in **node** delcaration commands. In the next section will we see how these automatic nodes can be exploited even further.

5 to 7 The dual of the pulldown chain doesn't require any additional nodes but the 2 to 1 ratio in transistor width must be explicitly declared.

The same situation is encountered with CMOS nor gates[3]. Additional built-in functions of this nature are provided in NETLIST. The NETLIST User's Guide contains a brief description of each and in many cases contrasts the built-in function to its transistor equivalent.

2.3. User Defined Functions (Macros)

As shown in the previous section one of the main advantages of the built-in functions was the recognition and generation of the local nodes. The task of providing built-in functions for all the possible cases where they appear would be impossible but NETLIST provides an alternative. If user defined functions can be used as if they were built-in then specialized modules can be created as their need arises. An example of this would be if the device sizing of the gate functions was inappropriate for a design, a new function could be designed that fit the requirements.

User defined functions are built in the form of macros. One way to think about a macro is a replacement for a related set of commands. Macros can have calling arguments (much like FORTRAN subroutines) and their own "local" nodes. Several examples of user declared macros will be presented in this section.

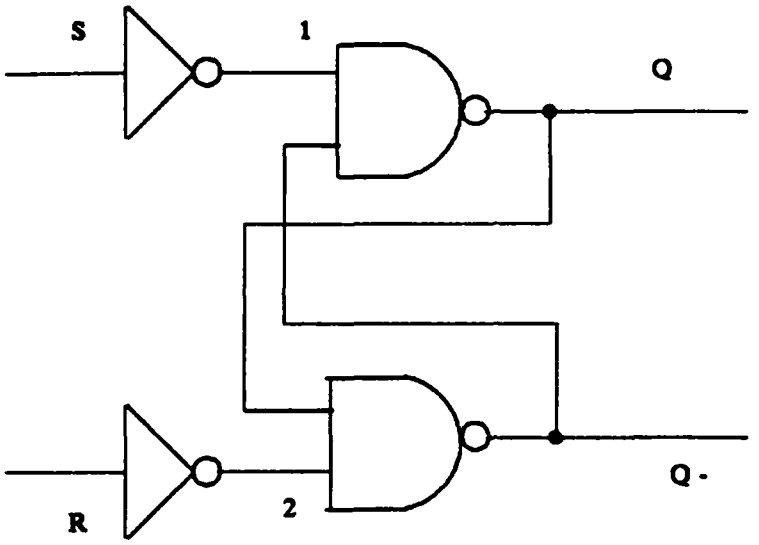
From a design point of view the macro of a SR latch shown below is not recommended. The choice was made to include an example where the circuitry needs little explination so that the important features of the macro are evident. As has been the pattern important features are descibed on a line by line basis.

```
; CMOS SRlatch macro                      ; (1)
; Inverters are ratioed @ 2.0             ; (2)
; Nand gates are ratioed @ 1.0            ; (3)
(macro SRlatch (m_S m_R m_Q m_Q-)         ; (4)
  (local h1 h2)                           ; (5)
```

---

[3] A 3 input CMOS nor gate with default size n transistors is

(cnor out in1 in2 in3)



SR latch

```
(ratio 2.0)                                  ; (6)
(cinvert h1 m_S)                             ; (7)
(cinvert h2 m_R)                             ; (8)
(ratio 1.0)                                  ; (9)
(cnand m_Q h1 m_Q-)                          ; (10)
(cnand m_Q- h2 m_Q)                          ; (11)
(capacitance h1 0.05)                        ; (12)
(capacitance h2 0.05)                        ; (13)
)                                            ; (14)
```

1 to 3 Comments just as before a begun with ";" and are ignored by NETLIST. They are useful especially as time goes by and you wonder what something really does.

4 This is the begining of the **macro** command. Its template is

(macro name (parameters)

A macro is called with

(name (calling_arguments))

Macros all begin with the left parenthesis and the word macro *"(macro"* followed by the macro's name (*SRlatch*). Following the name is the list of formal *parameters*. The number and type of *calling_arguments* must match the formal parameters. As mentioned earlier this similar to the parameters in a FORTRAN subroutine. For example if a parameter is used as a node name in the macro's definition then when the macro is called the corresponding calling argument must be a declared node name. The only way encountered so far to declare a node name is with the use of **node**. Line 5 introduces another way to declare a legal node name.

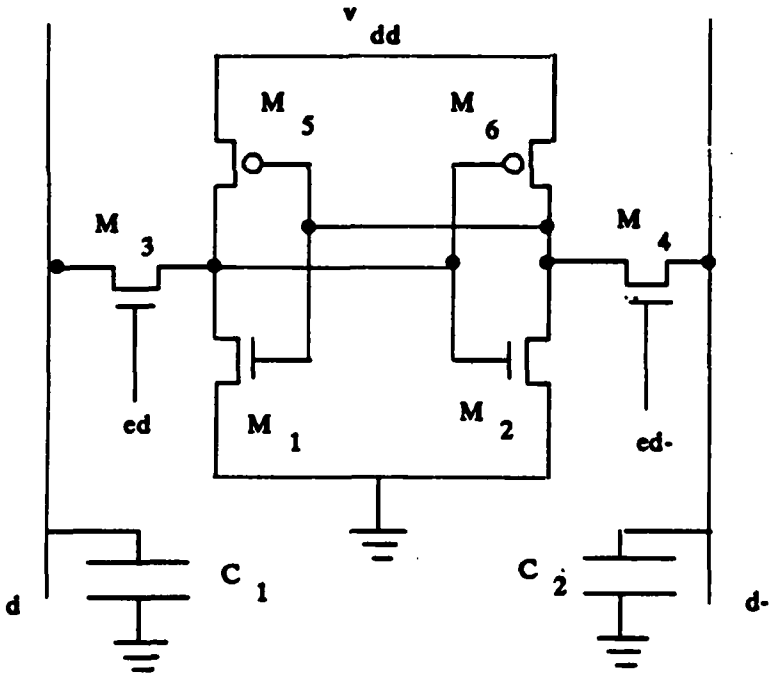
5 Following the declaration of the macros name and parameter list, any local nodes that are needed are specified by the **local** command. Its form is exactly like the **node**

command. This is useful because it helps in remembering what local does. It gives us a means of declaring the local names to a function and thus eliminates the need to specify them along with the important global signals (clocks, i/o, control) within the circuit. Every time the macro is called new nodes names are generated for these local nodes. Again we stress the fact that NETLIST uses numeric node names for these and one should avoid the use numeric node names in any other context.

6 to 13 This is the body of the macro. Any of the built-in functions can be used as a statement in the body. In addition any previously defined user macros can be statements in the body of the macro (shown in next example). Lines 7 and 8 also reflect that within the macro local nodes can be used just like the global nodes we have seen earlier. Lines 6 and 9 show the use of the ratio command. The scale values are set to 2.0 and 1.0 respectively. Upon exit from the macro the scaling value remains equal to 1.0. This can cause confusion and it is recommended that the ratio command be used frequently to ensure you are using the scale value you intended.

14 This right parenthesis ") completes the macro. Careful inspection will show that this matches the left parenthesis used in line 4, the begining of the macro declaration.

As a practical matter how and where does the use of the macro enter into the NETLIST program? For one thing the names in the parameter list must not conflict with the names of the global nodes. The following NETLIST program demonstrates the nesting of macros and hopefully provides a clear introduction to their use. We will also take this opportunity to introduce a command that allows repetition of a group of commands.



6 Transistor Memory

```
; macro for a 6 transistor memory cell, transistor sizing from          ; (1)
; "Analysis and Design of Digital Integrated Circuits,"                 ; (2)
; David A. Hodges and Horace G. Jackson.                                ; (3)
(macro mem_cell (m_ed m_ed- m_d m_d-)                                   ; (4)
  (local h1 h2)                                                         ; (5)
  (ratio 0.125)                                                         ; (6)
  (cinvert h1 (h2 16 2))                                                ; (7)
```

```
  (cinvert h2 (h1 16 2))                                    ; (8)
  (etrans m_ed m_d h1 4 2)                                  ; (9)
  (etrans m_ed- m_d- h2 4 2)                                ; (10)
)                                                           ; (11)
; end mem_cell                                              ; (12)
                                                            ; (13)
; macro for generation of maxword x maxbit memory           ; (14)
; requires macro mem_cell                                   ; (15)
; bit and word index begins at 0                           ; (16)
(macro mem_gen (maxword maxbit m_ed m_ed- m_d m_d-)         ; (17)
  (repeat r 0 maxword                                       ; (18)
    (repeat c 0 maxbit                                      ; (19)
      (mem_cell m_ed.r m_ed-.r m_d.c m_d-.c)                ; (20)
      (capacitance m_d.c 0.5)                               ; (21)
      (capacitance m_d-.c 0.5)                              ; (22)
    )                                                       ; (23)
  )                                                         ; (24)
)                                                           ; (25)
; end mem_gen                                               ; (26)
                                                            ; (27)
; Begin main body of memory generation                      ; (28)
(node ed d ed- d-)                                          ; (29)
(mem_gen 2 2 ed ed- d d-)                                   ; (30)
; end main                                                  ; (31)
```

1 to 11 The first macro declared is for a 6 transistor memory cell. The flip-flop is constructed from two cross-coupled inverters (lines 7 and 8). Note the use of local variables within the flip-flop. The enable lines control the reading and writing of bits or words into the flip-flop. In declaring the parameters to a macro one must use some care. Note in line 4 the prefix *m_*. This is used as a mnemonic for formal macro parameters and limits the possiblity of use the same name as global node name. Another point that should be made in this example is the use of a faction in the ratio command on line 6. This is perfectly reasonable when the drive must be very weak as is the case in such a memory cell. As noted in the comments a discussion of the transistor sizing can be found in "Analysis and Design..." The simulation of the 6 transistor memory cell is beyond the scope of this tutorial but is covered in section 4 of the RNL User's Guide. Finally, note the general formatting of the macro. The closing right parenthesis (line 11) is aligned with its corresponding left parenthesis. When writing macros some scheme like this should be employed. Some editors (EMACS) provide commands that will identify corresponding pairs of parentheses. As macro size increases use of an editor with this capability is strongly recommended.

14 to 25 This macro follows the same general scheme that has been outlined before. Important things to note here is the use of a new command and the nesting of one macro in another. When nesting one macro it is important to remember that a macro must be declared before it is executed. Line 18 provides a very useful capability when describing circuits. **Repeat** is much like the FORTRAN DO loop. It has the template

```
(repeat index initial final
  statements within repeat block
)
```

An index (line 18 *r* and line 19 *c*) is declared and followed by its initial and final values (e.g. *0* and *maxword* on line 18). As stated in the introduction, this command is contained within a set of parentheses (note: formatting can be very important during the debugging process). From the example it is obvious that repeat loops can be nested. Note how we used the repeat loop indicies in constructing new node names (line 20). This is a very powerful feature in NETLIST programs. This ability gives us a third kind of node name, global and local detailed earlier and *structured*. A structured name has two or more components separated by periods. The first component must be a declared node name. As we have seen node names can be declared with a local or a node command. Another way is as it used here where the declared name comes from one of the parameters to the macro (Recall that node names used in calling a macro must be declared names.); the remaining components can be other names or expressions. Here we have used the indicies from the repeat commands. You only have to declare non-numeric components of structured names -- for example, only "ed" must be declared, not "ed.1", "ed.2", etc. (see line 29). Full details can be found in the NETLIST User's Guide.

to 30 Finally we see that the main body of NETLIST program has been reduced to 2 statements. Declaring the global node names and calling memory generator macro **mem_gen**. Again note how a user defined macro is used just like the built-in functions described earlier.

2.3.1. Loading Macros

Up to now we have included all of the the macros that were required for the circuit descriptions in one file. However to use the **mem_cell** and **mem_gen** macros shown above it would be useful to be able to incorporate them into another design file without typing them in in their entirety. This can be done with the use of the **load** command. Its template is

(load "filename")

The actual filename is up to you but it must be surrounded by double quotes "". The practical effect is that any legal NETLIST commands and any the macros contained in the file are made available to the remainder of the program just as if they were typed in. An important requirement a macro must be loaded before it is executed. This is really the same requirement mentioned earlier, that a macro must be declared before it is used.

By putting the macros **mem_cell** and **mem_gen** in a separate file and with the **load** command, we could rewrite the program for generating the 2 x 2 memory.

```
(load "mem_primitives")                    ; (1)
(node ed d ed- d-)                         ; (2)
(mem_gen 2 2 ed ed- d d-)                  ; (3)
```

Where the file *mem_primitives* contains the NETLIST code for **mem_cell** and **mem_gen**.

2.4. Summary

This concludes the section on building circuit descriptions using NETLIST. We have discussed the following points

- Transistor descriptions (e.g. etrans, ptrans),
- Node name types (e.g. global, local and structured),
- Built-in functions (e.g. cinvert, cnand, cnor),
- User defined functions (i.e. macros),
- Repeating blocks of NETLIST statements (i.e. repeat),

* Loading previously defined macros (i.e. load).

In following sections details of generating transistor files for simulation will be investigated. A flexible command language for simulation and an elaborate example of simulating a large design will be described.

There are many more functions that can be used in writing NETLIST programs. Moreover, arithmetic functions and common Lisp functions are available. These features are beyond the scope of this tutorial but users are encouraged to refer often to the reference manuals.

3. GENERATION OF PRESIM AND RNL INPUT FILES

The below is a session of running the **netlist** and **presim** commands in the C shell of UNIX[4]. Following the session, comments are indexed by line number.

```
% netlist inverter.net                                                    ; (1)
| units: 250      tech: ???     format: MIT                               ; (2)
p in out vdd 8.0 8.0 0 0 r 64.0                                           ; (3)
e in gnd out 8.0 4.00 0 0 r 32.0                                          ; (4)
c out 3.000000e-01                                                        ; (5)
% netlist inverter.net -tisocmos                                          ; (6)
| units: 250      tech: isocmos format: MIT                               ; (7)
p in out vdd 8.0 8.0 0 0 r 64.0                                           ; (8)
e in gnd out 8.0 4.00 0 0 r 32.0                                          ; (9)
c out 3.000000e-01                                                        ; (10)
% netlist inverter.net -tisocmos -s200                                    ; (11)
| units: 200      tech: isocmos format: MIT                               ; (12)
p in out vdd 8.0 8.0 0 0 r 64.0                                           ; (13)
e in gnd out 8.0 4.00 0 0 r 32.0                                          ; (14)
c out 3.000000e-01                                                        ; (15)
% netlist inverter.net inverter.sim -tisocmos                             ; (16)
% presim inverter.sim inverter                                            ; (17)
Version 4.2                                                               ; (18)
8 nodes; transistors: enh=1 intrinsic=0 p-chan=1 dep=0 low-power=0 pullup=0 resistor=0 ; (19)
Total transistors eliminated = 2                                          ; (20)
% presim inverter.sim inverter config                                     ; (21)
8 nodes; transistors: enh=1 intrinsic=0 p-chan=1 dep=0 low-power=0 pullup=0 resistor=0 ; (22)
Total transistors eliminated = 2                                          ; (23)
%                                                                         ; (24)
```

1 This command runs the program NETLIST. This is the simplest form of the command. That is, only an input file *inverter.net* is specified. The contents of inverter.net is the CMOS inverter described in the previous section. Output from **netlist** will be referred to as .sim file. With this usage of the command the .sim file (lines 2 to 5) is sent to standard output which in this case is the terminal.

2 A line that begins with a vertical bar (|) is generally a comment in the .sim file and is ignored. The only exception to this is when a .sim comment begins with a line of the form in 2. This line is required by programs in the Berkeley and UW tool sets. It contains a conversion number (units:) and a specification of the technology type (tech:) and finally the format: of the .sim file itself. Two formats are used within the UW/NW tools, the MIT format here and the UCB format produced by the layout extractor *mextra*.

---

[4] UNIX is a trademark of Bell Laboratories and the C shell is a command shell generally used with the 4.1BSD UNIX.

3 to 5 Lines 3 and 5 represent the transistors in the inverter circuit. The MIT template for these "transistor records" is the following;

type gate source drain length width xpos ypos shape area.

Type identifies transistor records as n type enhancement (e)[5], p type enhancement (p) or when designing in NMOS depletion (d). Gate, source and drain are of course the terminals of the transistor. Xpos and ypos would provide the location of the transistor if the .sim file was extracted from a layout. In this case when only the connectivity is being declared these values are set to 0. Length and width represents the size of the transistor. When a .sim file is obtained from a layout and if a transistor is oddly shaped, they are approximate values. If an actual transistor is something other than rectangular, it is indicated with the shape field. A rectangular shape ("r") is assumed when the .sim file is generated by NETLIST. Area is the total gate area. Line 29 reflects load capacitance declared in the NETLIST input file (in picofarads).

6 In this command the additional input parameter -t has been declared. Note there is no space between the option flag and the technology name. As can be seen the output (lines 7 to 10) is substainally the same except with the technology now being declared as isocmos.

11 This time we have added another option which sets the centimicron value of lambda to 200 (i.e. lambda = 2.0 centimicrons). Again there is no space between the option flag (*-u*) and the value. This option is not refected in the transistor sizes as both length and width are the same as the lines 7 through 10. Rather the units: parameter is now 200 rather than 250.

16 This is the usual form of the netlist command. Here we have specified an input file *inverter.net*, an output file *inverter.sim* and a technology *isocmos*. When an output file is desired it must appear as the second argument in the command.

17 The command presim prepares a binary file for input to RNL. Presim replaces the transistors in the .sim file with an equivalent sized resistors. The equivalent resistors are used with any capacitance on a node to compute an estimate of its transition delay time. The command must contain an input file (*inverter.sim*) and an output file (*inverter*). The defaults for this replacement are the source of no end of confusion for new users. The rule of thumb for the ratio of the conductivity of n type enhancements to p type enhancements is 2.0 to 2.5. Unfortunately, presim default assumes that all transistors have identical conductivity. To its credit presim does allow the user to parameterize the resistance and capacitance values by the use of the so-called config file (see PRESIM User's Guide and section 3 of the RNL User's Guide). The use of this file is shown on line 21 of this example.

19 to 20 Presim provides some additional output to the terminal. Line 19 contains information about the number of unique nodes encountered and a count, by type, of the transistors in the circuit. The node count of 8 is of no concern. The expected node count of 4 (*in*, *out*, *vdd* and *gnd*) has 4 nodes added to it that presim uses internally. Presim includes a network reduction scheme that improves execution speed during simulation. In effect it attempts to find a sum-of-products representation for as many of the nodes as possible. This has the practical effect of reducing the number of nodes (e.g. local nodes internal to the gate primitives) and transistors, line 20 reports the number of transistors involved in sum-of-products representations.

21 to 24 As mentioned earlier the configuration file is required when the resistor replacement operation needs parameterization. As can be seen the practical effect when issuing the presim command is to add an additional option (config). This option is the

[5] The notation for the transistor types reflects the history of this simulator. It was originally designed with NMOS circuits in mind where there is but one type of enhancement transistor.

name of the configuration file you wish PRESIM to use. A extremely simple configuration file that sets the conductivity ratio of n to p type transistors to 2/1 is shown below;

```
resistance enh static 10 10 10000
resistance enh dynamic-low 10 10 10000
resistance enh dynamic-high 10 10 10000

resistance p-chan static 10 10 20000
resistance p-chan dynamic-low 10 10 20000
resistance p-chan dynamic-high 10 10 20000
```

The template is

resistance t-type r-type l w value

Resistance is a keyword for presim. T-type identifies the transistor type and r-type the resistance type. The three values shown are discussed in section 2 of the RNL User's Guide. L and w are the length and width of the transistor that has a resistance given by the last parameter (value).

3.1. Review

We have run two very important programs for the simulation of digital circuits using RNL. Netlist is a program that generates the transistor representation for the circuit. Frequently used options were outlined. Many more exist and interested readers are encouraged to read the NETLIST User's Guide.

Presim is a preprocessor for the digital circuit simulator RNL. It performs a transistor resistor replacement that is used by RNL to give estimates of the signal delay times. Its major pitfall is the default value for the resistance of p type transistors. A simple "workaround" was presented by the use of the configuration file. Again much more can be done with the configuration file and this information can be found in the PRESIM and RNL User's Guides.

4. INTERACTIVE SESSION WITH RNL

The following is an interactive session with RNL. It includes loading additional Lisp interface functions, formatting the results of a simulation step and running the inverter test case.

```
% rnl                                                                    ; (0)
(load "uwstd.l")                                                         ; (1)
done                                                                     ; (2)
                                                                         ; (3)
(load "uwsim.l")                                                         ; (4)
Loading uwsim.l                                                          ; (5)
Done loading uwsim.l                                                     ; (6)
done                                                                     ; (7)
                                                                         ; (8)
(read-network "inverter")                                                ; (9)
; 8 nodes, transistors: enh=0 intrinsic=0 p-chan=0 dep=0 low-power=0 pullup=0 resistor=0 ; (10)
done                                                                     ; (11)
                                                                         ; (12)
(def-report '("Current state:" newline in out))                          ; (13)
("Current state:" "                                                      ; (14)
" in out)                                                                ; (15)
                                                                         ; (16)
```

```
(chflag '(in out))                                    ; (17)
(in out)                                              ; (18)
                                                      ; (19)
l in                                                  ; (20)
done                                                  ; (21)
s                                                     ; (22)
                                                      ; (23)
Step begins @ 0 ns.                                   ; (24)
in=0 @ 0                                              ; (25)
out=1 @ 0.6                                           ; (26)
Current state:                                        ; (27)
Current time= 100                                     ; (28)
                                                      ; (29)
in=0 out=1                                            ; (30)
done                                                  ; (31)
h in                                                  ; (32)
done                                                  ; (33)
s                                                     ; (34)
                                                      ; (35)
Step begins @ 100 ns.                                 ; (36)
in=1 @ 0                                              ; (37)
out=0 @ 0.6                                           ; (38)
Current state:                                        ; (39)
Current time= 200                                     ; (40)
                                                      ; (41)
in=1 out=0                                            ; (42)
done                                                  ; (43)
(exit)                                                ; (44)
%                                                     ; (45)
```

0 From the C shell (note the "% " prompt) issue the command **rnl**.

1 to 7 From within the RNL command interpreter issue the commands (**load "uwstd.l"**) and (**load "uwstd.l"**). The two files uwstd.l and uwsim.l provide users of RNL with a simple command interface. In general these files will be loaded every time RNL is used. The interpreter when finished with a **load** command returns "done" (lines 2 and 7). Many of the commands return this.

9 The **read-network** command is issued next. This reads the file prepared by **presim** and in effect builds the network that is to be simulated. Recall when **presim** was run it provided a summary of the network. Similarly read-network returns a summary (line 10) however, the node and transistor count now reflects only those remaining after presim. For this example there were no internal nodes and the node count is unchanged. Again in line 11 "done" is returned by the command interpreter.

13 **Def-report** is a function provided in the uwsim.l package that allows users to format a report that is printed at the end of each simulation step. In setting a format up it is important that it begin with *'(* and end with *)*. In this case we have a simple format that prints out the string "Current State:". A string of some kind is required the shortest one being the null string "". Following the string is a newline (newlines are interpreted as the sequence carriage return, linefeed). Then we request that the states of the nodes with the names *in* and *out* be printed. Def-report does not return "done" like before but some rather odd text (line 15) that for the purposes of the tutorial can be ignored.

17 **Chflag** is another function provided in the uwsim.l package that informs the simulator which nodes should have their transition times reported. This is done by declaring a quoted list of symbols that have the same print names as the node you are interested in (e.g. *in* and *out*). Quoted lists in Lisp begin with '(and end with). Alert readers will note that the def-report in line 13 is also a quoted list. The list is returned (printed) by chflag when it is finished.

The arguments to the RNL command interpreter formally are known as Lisp symbols. In most cases, these symbols will have the same print names as nodes in the circuit (i.e. you type the same string). We have already seen examples of this in the setup phase detailed above. In the following we will refer to the arguments of many of the commands as nodes. This is only to avoid clutter, they are in fact Lisp symbols. A discussion of the evaluation of Lisp symbols etc. is deferred to section 5 of this tutorial. In areas of potential confusion we will refer to them as Lisp symbols. In this same vein all the commands described for the rest of this example are provided in the **uwsim.l** package. This should be assumed unless it is noted otherwise.

We are now in a position to do some simulation of the inverter. The commands that are presented here are simple. With some experience with the Lisp interpretor much more elaborate commands can be written. That is not the subject here however and is deferred.

20 **l** (**h** and **u** described later) is a command that sets all of the nodes listed (e.g. *in*) to a logic low. Note this and related commands do not require surrounding *()* when run interactivly. This value will not change under any simulation conditions. Using this and related logic commands has the effect of declaring these nodes as inputs to the circuit.

20 to 25 **s** is the simplest simulation command available. It runs a single simulation step for a predetermined amount of time (default 100ns) or until the entire network being simulated has settled to a definite state. Recall in our set up phase we informed RNL (**chflag**) that the transition times of the nodes *in* and *out* should be reported. Lines 25 and 26 show these transition times relative to the current time reported in 24. Transitions of nodes that are set by **l** etc. are assumed to happen immediately hence the transition time of 0 in line 25 for node *in*.

27 to 31 When the simulation step is completed, **s** uses the def-report (line 13) to format the results of the step. Line 27 is of course the string we wanted to have printed, the current time (ns) is then given (this is always reported and need not be included in the format specification). Line 30 details the current states of all the nodes that were declared in the def-report. In this case *in* and *out* are indeed shown to be an inverted pair. "done" (line 31) is echoed to the terminal and this completes one simulation step. If a def-report was not specified a warning to that effect is displayed and then "done."

32 to 43 This sequence is very similar to the one just described. **h** is the analogous function for setting nodes to a logic high value. Not used in this example is the function **u**. It completes the set by declaring nodes to "unknown." The characteristics of nodes declared unknown are discussed in Section 4 of the RNL User's Guide. The results again show that the nodes of interest are indeed a inverted pair.

44 One exits the simluation by either typing on a newline the one of the strings (exit) or exit or by entering CNTL D. Exit is used here to return to the C shell of UNIX.

To release a node that has been declared an input with any of these commands one uses the **x** command followed by a list of nodes. Nodes that have been released will reflect their "true" state at the end of the next simulation step. The choice of **x** is a potential source of confusion as *X* (capital X) is used to represent the logic state of unknown. This is unfortunate but...

4.1. RNL Control File

For small simulations such as the one described in the previous section all of input can be entered while in interactive mode. As simulations become larger or as one iterates on a design the need for a set of frequently used commands to be entered without retyping grows rather quickly.

RNL provides such a capability by the use of the control file. The commands in this file are executed upon entry into simulation. When the end-of-file is reached control is returned to the user. That is to say commands like the ones we have been through can be entered. What are typical commands put in the control file?

In the cases we have just analyzed the following file would reduce our efforts considerably.

```
; All text appearing after a semicolon is a comment and is ignored
; Rnl control file for the inverter example
(load "nwstd.l")
(load "nwsim.l")
(read-network "inverter")
(chflag '(in out))
(def-report '("Current State:" newline in out))
```

This file is a collection of the setup commands that we issued first when running RNL interactivly. For detailed analysis of these commands the reader is referred to the previous section. Note the use of the comment lines. Comments are begun with a semi-colon (;) and all text appearing after it until a newline is ignored by RNL.

4.1.1. Using the RNL Control File

The only required modification to the start up of RNL is to add the name of the file that contains the RNL commands and looks like,

```
% rnl control_file
```

where you can substitute any filename that suits you for control_file. When using a control file RNL works rather quietly. Below you will see the output from the control file described above.

```
Loading nwsim.l
Done loading nwsim.l
; 8 nodes, transistors: enh=0 intrinsic=0 p-chan=0 dep=0 low-power=0 pullup=0 resistor=0
```

Clearly most of output that was generated interactively is gone. In fact we are left with only the fact that nwsim.l has been loaded and a summary of the preprocessed network from the read-network command. As shown below we can now set the input low and run a simulation just as was done before.

```
l in
done

s

Step begins @ 0 ns.
in=0 @ 0
out=1 @ 0.6
Current state:
Current time= 100
```

```
in=0 out=1
done
```

A useful technique to develop is one where the simulation experiment is verified interactively and is then followed by a larger simluation run in batch mode. This can be done with the use of the control file by adding simulation commands. Some care must be used here as some of the commands require a slightly different syntax when used in batch mode. For example when nodes are to be set at some input value (h, l etc.), in batch mode the nodes must be in the form of a quoted list (recall '( ))

```
(h '(n1 n2 n3)),
```

the command to run a simluation step must also have the special quoted list, the empty list

```
(s '()).
```

This is the result of the RNL Lisp interpreter having a special form only for interactive commands. The discussion of this is in section 5.

4.2. Useful Additions

To this point we have run through an example interactively and shown how one can condense the frequently used commands into an RNL control file. In this section, additional commands that can be used either interactively or through the control file will be explored.

4.2.1. Buses

A very common situation in design is to have a group of signals that work together (i.e. buses). When doing simulation it would be most useful to be able to give these signals a name and refer to the whole set by that name. The uwsim1 package provides this through a set of vector commands. An example (albeit overly simple) of this would be to define the nodes *in* and *out* in the inverter example as a vector. This is done with the following command,

```
(defvec '(bin inoutvec in out))
```

The template for this command is

```
(defvec '(radix name list_of_nodes))
```

Defvec is the command name and notice that a familar piece of syntax has appeared again, the quoted list (begins with '(and ends with)). Radix is the number base that the vector will be printed in when you request that it be displayed. The choices are the familar set, bin -> binary, oct -> octal, hex -> hexadecimal and dec -> decimal. List_of_nodes can be any number of nodes that defines the vector. In our case there are 2 *in* and *out*.

Vectors are a special data type that are composed of lists of nodes. There are functions provided for setting the vector's value (invec) and finding out how a vector is defined (vecnames). These functions are detailed in section 7.2 of the RNL User's Guide. The handiest place for vectors however is when they are used in def-report.

```
; All text appearing after a semicolon is a comment and is ignored      ; (1)
; Rnl control file for the inverter example                              ; (2)
(log-file "in.log")                                                      ; (3)
(load "uwstd.l")                                                         ; (4)
(load "uwsim.l")                                                         ; (5)
(read-network "inverter")                                                ; (6)
(chflag '(in out))                                                       ; (7)
```

```
(defvec '(bin inoutvec in out))                                    ; (8)
(def-report  '("Current State:" newline  (vec inoutvec)))          ; (9)
```

8 This control file is quite similar again to the one shown earlier but now on line 8 we have made use of the **defvec** command. In this case the radix has been set to binary and the name has been set to inoutvec. The signal names (nodes) *in* and *out* finish the definition of the vector.

9 The **def-report** command also reflects the use of defvec. Now the report uses vec which is defined to print the value of the named vector (e.g. *inoutvec*). Recall in declaring the vector we specified a binary radix. Vec reports the vector in this radix. This format replaces the individual node names and values we used earlier. Assuming that we have either given RNL a control file with this report or interactively declared a report the results of a step of simulation would be

```
l in                                                               ; (1)
done                                                               ; (2)
s                                                                  ; (3)
                                                                   ; (4)
Step begins @ 0 ns.                                                ; (5)
in=0 @ 0                                                           ; (6)
out=1 @ 0.6                                                        ; (7)
Current State:                                                     ; (8)
Current time= 100                                                  ; (9)
                                                                   ; (10)
inoutvec=0b01                                                      ; (11)
done                                                               ; (12)
```

11 The individual nodes and values have been replaced by the vector we declared. The name and the current value of the vector is reported. The prefix *0b* reflects the radix that was declared when the vector was defined. The other radix flags follow the UNIX convention (0 -> octal, 0x -> hex). One can also mix the printing of nodes and vectors thus

```
(def-report  '("Current State:" newline  in out (vec inoutvec)))
```

would be another way we could format the report.

4.3. Event files

A very useful command for displaying and interpreting the results of your simulation is **openplot**. This command has the following template

```
openplot "plot_file_name"
```

Note there are no parenthesis surrounding this command as there has been with the others that have been discussed. The effect of issuing this command is that all transitions that were requested with **chflag** are entered into the file plot_file_name. This file can then be used as input to programs that display the time series trace on either a 4010 compatible terminal or a Printronix raster printer. The details of the use of these programs are described in the man pages for **simscope** and **mtp**.

At the end of the simulation the file is closed with

```
closeplot "plot_file_name"
```

Again the parentheses are not used.

4.4. Local Network Walks

When working interactively, two commands that allow local areas of the network to be examined are presented. The first one allows the user to obtain all transistors that are gated by the node in question and any sum-of-products functions of which it is an input. This command will be referred to as a **foward reference**. The second command reports the list of transistors for which the node is either the source or drain and a summary of its sum-of-products representation if any. This will be referred to as **backward reference**. In the following example we will investigate the SR latch described earlier with these commands.

```
! S                                                                    ; (1)
S=H [NOTE: node is an input] (vl=0.30 vh=0.80) (0.019200 pf) affects:  ; (2)
input to functions for the following nodes:                           ; (3)
      1                                                               ; (4)
      1                                                               ; (5)
done                                                                  ; (6)
? 1                                                                   ; (7)
1=L (vl=0.30 vh=0.80) (0.062800 pf) is computed from:                 ; (8)
CMOS                                                                  ; (9)
  (nor (n-chan (and S=H )) resistance [1.00e+04, 1.00e+04]            ; (10)
       (p-chan (and S=H )) resistance [5.00e+03, 5.00e+03]            ; (11)
  )                                                                   ; (12)
done                                                                  ; (13)
! 1                                                                   ; (14)
1=L (vl=0.30 vh=0.80) (0.062800 pf) affects:                          ; (15)
input to functions for the following nodes:                           ; (16)
      Q                                                               ; (17)
      Q                                                               ; (18)
done                                                                  ; (19)
? Q                                                                   ; (20)
Q=H (vl=0.30 vh=0.80) (0.012800 pf) is computed from:                 ; (21)
CMOS                                                                  ; (22)
  (nor (n-chan (and 1=L Q-=L )) resistance [2.00e+04, 2.00e+04]       ; (23)
       (p-chan (and 1=L )) resistance [1.00e+04, 1.00e+04]            ; (24)
       (p-chan (and Q-=L )) resistance [1.00e+04, 1.00e+04]           ; (25)
  )                                                                   ; (26)
done                                                                  ; (27)
! Q                                                                   ; (28)
Q=H (vl=0.30 vh=0.80) (0.012800 pf) affects:                          ; (29)
input to functions for the following nodes:                           ; (30)
      Q-                                                              ; (31)
      Q-                                                              ; (32)
done                                                                  ; (33)
? Q-                                                                  ; (34)
Q-=L (vl=0.30 vh=0.80) (0.012800 pf) is computed from:                ; (35)
CMOS                                                                  ; (36)
  (nor (n-chan (and 2=H Q=H )) resistance [2.00e+04, 2.00e+04]        ; (37)
       (p-chan (and 2=H )) resistance [1.00e+04, 1.00e+04]            ; (38)
       (p-chan (and Q=H )) resistance [1.00e+04, 1.00e+04]            ; (39)
  )                                                                   ; (40)
done                                                                  ; (41)
! Q-                                                                  ; (42)
Q-=L (vl=0.30 vh=0.80) (0.012800 pf) affects:                         ; (43)
input to functions for the following nodes:                           ; (44)
```

```
    Q                                          ; (45)
    Q                                          ; (46)
done                                           ; (47)
```

1 to 6 This is the output of the forward reference command (*!*) for one of the inputs to the SR latch described earlier. Line 2 summarizes the parameters for node S. Several things should be observed here. First note that this node is considered an *input*. This is because the node has been set by of the command **h**. Similarly if **l** or **u** were used this messages would appear. The next parameters show the logic threshold values (voltage is normalized to 1.0) for the node. For a discussion of these values the reader is referred to section 2 of the RNL User's Guide. Next the total load capacitance is given in picofarads. Finally a summary of all nodes that this node is an input is reported. In this case only node *1* is affected. Recall that in building the SRlatch we used local nodes for the output of the inverters. The node *1* is one of the NETLIST generated nodes we talked about. If there were transistors that this node gated they would be reported next. The template for a transistor report is

type gate source drain resistance_values

All of these parameters have their usual meanings (e.g. resistance in ohms).

7 to 12 We can continue the local network walk with a backward reference (**?**) for node *1*. Line 8 is similar to the forward reference but note how this node is not an input. It is computed from the function shown in lines 9 to 12. This is what a summary of a CMOS inverter looks like. The technology is indicated in line 8. This is followed by a list of each of the product chains in the sum-of-products description. Each product chain is prefixed with the type of transistor (n-chan or p-chan). In this case each product chain consists of just the one input *S*. At the end of each list of inputs a summary of the total resistance computed by PRESIM is given. The first of the two resistance values is used in computing the state of the node and the second is for estimating the delay time of the transition if any. Details are given in the RNL User's Guide in section 2. Again resistance values are given in ohms.

14 Pressing on with the forward reference **!** of node *1* we find that it is the input to the node *Q* (see line 17).

20 Node *Q* definition is then obtained with a backward reference **?**. The function definition follows substaintially the same form. The pulldown product term (It is the pulldown product because it is formed from n type transistors.) contains 2 inputs *1* and *Q-*. The pullup is composed of the first two term sum-of-products encountered that is, the two single term pullup products are ored together (*1* or *Q-*). This is the general form for nand gates. The dual of this combination of p and n transistors would be the pattern for a 2 input nor gate. Again each term is followed by its resistance values.

28 This is the forward reference for the node *Q-*. And here we find the effects of the cross-coupled nand gates forming the flip-flop in the SR latch. Note that *Q-* is a function of *Q* and vise versa (see line 45). Also the other local node from the second inverter is an input to *Q-*. Again NETLIST used numeric node names for these nodes and such names should be avoided elsewhere.

From this example then we have shown how the forward and backward reference commands can be used to get a feel for where a node lives in the circuit. This is particularly useful when portions of the circuit appear to be misbehaving. These commands can ensure that it indeed is "wired up" correctly.

4.5. Summary

This completes an example for running RNL. This example is by no means exhaustive. In fact the Lisp command interpreter available in RNL makes the possibilities for elaborate simulation commands very attractive. Recall that one of the first things that we did during our session was to load the files uwstd.l and uwsim.l. These files represent some of the versatility of this interpreter. In the next section we will examine portions of these files for examples of how one might construct their own commands.

5. RNL LISP: SOME EXAMPLES

This section is a tutorial introduction by examples to programming in RNL Lisp. The particular examples chosen are taken directly frm the package *uwsim.l*. They were chosen to be instructive not only from the standpoint of being examples of Lisp functions, but also because they are prototypical of the kinds of functions that a user may want or need to write. We are deliberately encouraging users to write their own user interface functions by copying these examples and modifying them.

5.1. The RNL Lisp Interpreter.

In order to discuss the writing of Lisp functions we first present a brief introduction to the Lisp language and the Lisp interpreter. If at first you do not understand what is going on we recommend the following strategies: study the examples, reread this section carefully, and, most importantly, play with RNL interactively. Because Lisp is interactive you can often learn a lot more in a few minutes of hands on experimentation than in hours of staring at textbooks and manuals. The "real" Lisp systems that most resemble RNL Lisp are *MACLisp* (from MIT) and the *Franz Lisp* (from Berkeley). If you do consult other texts and manuals, ones that use either of these dialects will be the most useful.

It has been claimed that rather that standing for "LISt Processing" that Lisp is really an acronym for "Lots of Irritating Single Parentheses". Although the Lisp syntax is simple, elegant, and powerful, it has the unfortunate property that a user can easily wind up wasting time trying to balance parentheses and trying to understand poorly formated Lisp code. There are two things that one can do to minimize this problem. The first is to use care in formatting your Lisp functions. Don't put too much on one line and use a consistent indentation style. The second thing that you can do is to use a screen-oriented text editor with a Lisp mode (e.g. EMACS) that helps you to keep track of the parentheses and indentation.

5.1.1. Evaluation

Perhaps the most fundamental concept in Lisp is the *evaluation* of a Lisp object. Lisp objects are also known as *S-expressions* (The "S" stands for "symbolic".) and we will often just call them "expressions". The universe of Lisp objects is divided into two classes: primitive objects (also known as "atoms"), and lists.

In RNL Lisp there are several kinds of primitive object:

symbols are like variables in other programming languages. Each symbol potentially has a value, a functional definition, and a list of properties. In addition, it has a *print name* by which it is known, both on input and output. Usually print names are terminated by "white space" (spaces, tabs, and newlines), but print names containing these and other special characters can be entered by quoting the name with vertical bars (e.g. |long.name with white space| is a symbol).

numbers can be integers or floating point numbers.

strings are pieces of text surrounded by double quotes (e.g. "this string").

nodes correspond to the electrical nodes of the circuit you are simulating. This is a data type not found in other dialects of Lisp. The print names of nodes can resemble symbols or numbers. If a symbol or number is used where a node is expected then RNL automatically tries to convert to the node with a similar print name. In addition,

nodes can be named as lists of the form *(-struct- a b c etc)* where each of *a, b, c, etc.* are either symbols or integers. Lists like this name nodes with names like *a.b.c.etc.* This allows you to create hierarchically structured naming schemes for your circuits. If you try to enter *a.b.c.etc* then the RNL interpreter will convert it to a list rather than a symbol, so if you want to have symbols with periods in their print names you have to use the vertical bar convention, e.g. *|a.b.c.etc|*.

A *List* is a sequence (a list) of Lisp objects that is bracketed with parentheses. Lists can and do contain other lists. For historical reasons the first element of a list is known as the "car" of the list and there is a function, car that extracts it from a list. The list formed by removing the car is known as the "cdr" (pronounced "koo-der") and is extracted using the function cdr.

Example: ((a b) (c (d)) e)

This is a list of three elements. The first element (the car) is the list *(a b)*. The second element is the list *(c (d))*, and the last is the symbol *e*. The cdr is the list *((c (d)) e)*. Note that in the second element that *(d)* is a list of one element, not a symbol. Note also that the cdr of *((c (d)) e)* is the list *(e)* (i.e. *(c (d))* is one element of the original list). The cdr of (e) is the empty list (). The empty list is synonymous with the symbol *nil*.

That is all there is to Lisp data structures: atoms and lists. From these you build data structures, function definitions, and commands to the interpreter.

5.1.2. Evaluation

The central idea in the execution of Lisp programs is that of the *evaluation* of Lisp objects (expressions). Evaluation is a process that interprets a Lisp object and returns some other Lisp object, its value. The evaluation process can have side effects.

The evaluation of atoms is straightforward. When a symbol is evaluated the object the interpreter returns is the one that was most recently assigned to be the symbol's *value*. In other words, a symbol acts just like a variable in other programming languages. All the other types of atoms (numbers, strings, and nodes) are what is called "self-evaluating". That is, these objects and their values are identical.

The evaluation of lists is a little more complicated. In the interest of completeness we will give all the gory details here, but keep in mind that the simplest case is the most common. Usually lists are function calls, but sometimes they are what are called "special forms". In all cases, the car (first element) of the list being evaluated controls what will happen.

If the car of the list being evaluated is a *symbol* then the interpreter checks whether it has a function definition. If not, then the interpreter evaluates the symbol and uses the result as though it were the original car of the list.

If the car is an atom other than a symbol then this is an error because these types of atom cannot have function definitons.

If the car of the list being evaluated is itself a list then the interpreter first checks to see whether it is a function definition (see below). If it is not a function definition, then it is evaluated and the result is used as though it were originally the car of the list.

In this way, the interpreter repeatedly (recursively) evaluates the car of the list being evaluated until a function definition is found. Although this sounds complicated, the simplest case in which the car is a symbol with a function definition is the most common form and all other forms are extremely rare in circuit simulation applications.

5.1.3. Function Definitions

Function definitions come in two flavors. There are the built-in functions (including special forms) and there are user-defined functions. If you should print one of the former it would appear to be a funny symbol such as *#S66506*. This indicates that the symbol is a built-in function or special form.

A user defined function definition appears as a list that looks like:

```
(lambda (x y) (+ 3 (* x y)))
```

The symbol *lambda* is a special symbol that means "This list is a function definition. Do not continue evaluation". The list "(x y)" names the symbols that act as the formal parameters to the function. The remainder of the list is the body of the function and is composed of a sequence of Lisp expressions (objects) that are evaluated in left-to-right order when the function is evaluated. The particular function defined above returns a value three greater than the product of its two parameters.

Thus far in our narrative the interpreter has reduced the car of the list it is evaluating to a function definition. If this is an ordinary function definition then the next step is the evaluation of the arguments to the function. Each of the remaining elements of the list is interpreted as an argument to the function. They are evaluated in left-to-right order and the objects returned are used as the values of the formal parameters of the function.

Finally, the body of the function is evaluated using these parameter values and the value returned is the last value returned when the body is evaluated.

Let us look in detail at the evaluation of the following expression:

```
((lambda (x y) (+ 3 (* x y)) 8 k)
```

The interpreter goes through the following steps:

1. First the car of the list is found to be a user-defined definition of a function with two parameters called *x* and *y*.
2. The next element in the list is evaluated. It is the integer 8 and is thus self-evaluating.
3. The last element in the list is the symbol *k*. Suppose that its value is 5.
4. Because *x* and *y* are the formal parameters of the function, their values are set to 8 and 5 respectively when the function is entered. When the function is exited the values they held previously will be restored.
5. The body of the function "(+ 3 (* x y))" can now be evaluated:
   a. The symbol + is found to have a built-in function definition and the first argument evaluates to 3.
   b. The second argument to + is the list "(* x y)". When it is evaluated a built-in definition is found for multiplication, *x* has the value 8, and *y* has the value 5. The expression "(* x y)" therefore evaluates to 40.
   c. Both of +'s arguments are now evaluated so the addition can proceed, returning 43. This completes the evaluation of the body of the function.
6 Since the last value returned in the evaluation of the body of the function is 43, this is also returned as its own value.

5.1.4. Functions Versus Special Forms.

When a user-defined function is encountered as the car of a list all of the remaining elements of the list are evaluated and the results passed to the function as arguments. Furthermore, the number of formal parameters in the function definition and the number of arguments actually passed must agree.

The built-in function definitions do not have the same restrictions. Some functions can take a variable number of arguments. An example is the + function. It returns the sum of an arbitrary number of arguments. There are other built-in functions that do not evaluate one or more of their arguments. An example of this is the **setq** function that is used to set the value of a symbol. Evaluating the list

```
(setq  sym1 (foo a b c))
```

has the side effect of setting the value of its unevaluated first argument (in this case, the symbol *sym1*) to the result of evaluating its second argument. This is also the value returned by the **setq** function.

Other symbols have built-in definitions that do not act at all like functions. These are called "special forms". For example, the *lambda* symbol is a special form that indicates function definition. Other special forms are used to define program control structures. While these special forms may return values analagously to functions, their interpretation is very different from that described above. For example, evaluating the list

```
(defun crunch (x y) (+ 3 (* x y)))
```

has the side effect of setting the function definition of the symbol *crunch* to the list:

```
(lambda (x y) (+ 3 (* x y)))
```

In this case none of the elements of the list are evaluated as arguments. (The result returned in this example is the symbol *crunch*.)

The *quote* special form is especially useful. It returns its argument without evaluating it. That is, if you want an object rather than its value you can use *quote* to inhibit evaluation. This is so useful that it has its own special alternate input syntax . The form **'obj** is translated to **(quote obj)** where **obj** can be any Lisp object. For example, if you type the expression:

```
(setq a '(x y z))
```

to the interpreter (note the two single quotes), then it will return the object:

```
(quote (x y z))
```

Because the self-evaluating objects (numbers, strings, and nodes) do not need to be quoted, they are also sometimes referred to as "self-quoting".

5.2. The "Top-Level Loop"

Now that we have introduced the data structures and the idea of evaluation in Lisp, we proceed to introduce the user interface to the RNL Lisp interpreter. The interface is called the "top-level loop" and consists of the following:

1. Read a Lisp object from the current input.
2. Evaluate the object read in step 1.
3. If the current input is the terminal then print the result of step 2.
4. Go to step 1.

The input to the interpreter is buffered on a line by line basis. Thus, RNL does not see anything you have typed until you enter a "newline" and you can use the standard "within line" editing of the system keys to modify the input before you enter it. Even though you have completed a line, RNL might not have read a complete Lisp object and therefore might not respond to you. For example, suppose you enter the lines

```
(+ 3
 (* 8 5))
43
```

(We use the convention that user input is displayed in **bold** type.) After you enter the second line RNL responds by evaluating the expression and printing the result, the number 43. Note that it did nothing after you entered the first line because it had not read a complete Lisp object at that point.

In order to reduce the number of parentheses you have to type, RNL Lisp has a special alternative syntax you can use. If the first thing on a line is a symbol then RNL interprets it as a function name, creates a quoted list out of the rest of the line, and passes that list as the only argument to the function. For example, the following two lines of input are

interpreted identically because the "reader" of the top-level loop converts the second line into the first line. (**Length** counts the number of elements in a list.)

```
(length '(a b c (4 5) 3 25 8.0E6))
7
length a b c (4 5) 3 25 8.0E6
7
```

You should remain aware that if you are using this alternative syntax that the entire command must be on a single line. While this is convenient for invoking some kinds of functions it does make it awkward to find the value of a symbol. Consider the following sequence:

```
(setq a 23)
23
a
;illegal function object
23
(eval 'a)
23
```

Because of the alternate syntax, when we tried to get the interpreter to evaluate the symbol *a* as an expression, the "reader" actually created the list

```
(a '())
```

and that is what was evaluated. The function **eval** evaluates its argument an extra time, so passing a quoted symbol to it results in only one evaluation being done.

The discussion of evaluation in the previous section covers the case in which everything works. Inevitably, however, there will be errors such as the one above in which no function definition was found for the symbol *a*. Whenever an error like this is detected an error message is printed and all current attempts to evaluate Lisp objects are aborted. This leaves you back in the top-level interactive "read-eval-print" loop. A particularly troublesome aspect of this is that if an error is detected while you are using the **load** function to read a command file that the **load** itself is terminated by the error so that the remainder of the file will not be read.

Note that the result of evaluating a command is printed only when input is being taken from the standard input. This may make it difficult to locate errors in command files that are being "loaded".

6. EXAMPLES FROM *uwsim.l*

To make things clearer we proceed to look at some examples taken directly from *uwsim.l*. These examples are augmented with line numbers in square brackets at the left. These numbers are not part of the code but have been added for the purposes of this presentation.

While a large part of most command files is the definition of functions and data structures that will be used later, part of all command files is the initialization of "global" symbols that parameterize those functions.

```
[1]  (setq incr 1000)
[2]  (setq switch-level nil)
[3]  (setq relative-timing t)
```

These three commands use the **setq** function to set the default values of parameters that are used to control your simulation. Line 1 sets the value of the symbol **incr** to 1000 RNL time units (100.0 nano-seconds). Line 2 ensures that the simulator will use the RC model for simulation rather than the unit delay switch model and line 3 sets a flag that forces the **step** function (see below) to report simulation times relative to the start of the simulation step. These lines illustrate the concept of self-evaluating objects. The number 1000 evaluates to itself.

The two symbols **all** and **t** are predefined by the Lisp interpreter so that they are their own values. In addition, *nil* has the semantics of being identical to the empty list, ().

```
;; run a simulation step and print a report at the end
[1]  (defun s (dummy)
[2]    (step incr)
[3]    (wr-report)
[4]  )
```

This example illustrates the definition of the function **s** used in section 4. The function **s** does not add any real power to the user interface, rather it serves the purpose of reducing the amount of typing you need to do. This runs the simulation for a time increment of the current value of the symbol **incr** and then calls the function **wr-report** with no arguments to write out a summary report at the end of the step.

Note that **s** has a dummy parameter called **dummy**. This is because **s** is intended to be used interactively using RNL Lisp's alternate syntax. When you type *"s"* on a line by itself without any parentheses, the reader converts it to the list *(s '())*. In order for **s** to work correctly it must expect to see one argument even though that argument will be ignored.

```
(defun __prinnum (base num)    ; only called to bind the right val to base
    (princ num))
```

The function **__prinnum** is used by various other functions in the *uwsim.l* package. Its job is to print an integer in the radix specified by *base*. The Lisp printing routines use the current value of *base* to control the radix used for output. By naming one of the parameters of **__prinnum** *base* we use the argument binding mechanism of RNL Lisp to temporarily change the value of *base* to the desired value. When **princ** is called, this is the value it uses. When **__prinnum** returns the previous value of *base* is restored.

This example illustrates the principle of dynamic variable scoping in Lisp. Programming languages such as C or Pascal have what is known as a textual, or static, scope rule for the use of local variables. That is, the part of a program that sees a particular local variable is statically limited to the text of the routine in which it is defined. In contrast, Lisp uses a dynamic scope rule. When a Lisp special form (e.g. a function definition) uses a symbol as a "local variable" (e.g. a parameter), then the old value of the symbol is saved away and is restored only when the special form returns. Any functions that are called before the special form returns will see the new value of the symbol. Furthermore, any changes made to the value of the symbol will be lost when the old value is restored.

```
; chflag sets the STOPONCHANGE flag for the nodes in l
[1]  (defun chflag (l)
[2]    (do      ((here  l  (cdr here)))
[3]        ((null here) l)
[4]        (stop-on-change (car here) t)
[5]    ))
```

The function **chflag** takes as an argument a list of nodes. It sets the STOPONCHANGE flag for each of the nodes in the list. We use it as an example to introduce the **do** special form and, in particular, to show how a **do** is typically used to perform a function on each of the elements in a list. Note that although **chflag** has just the one argument (the list *l*), the list itself is not of fixed length.

Line 1 defines **chflag** to be a function with a single argument called **l**. Line 2 begins a **do** form. A **do** is a generalized iteration construct that allows you to define multiple local symbols to be used in the iteration. While a **repeat** increments its single local symbol on each iteration, a **do** allows arbitrary computations to be done on to get the new values of its local symbols.

The first thing that follows the **do** is the list of local symbol declarations. In this case there is only one element in that list, the declaration of *here*. A declaration is a list of one to three elements. The first element is a symbol, in this case *here*. The second (optional) element of the declaration is evaluated to get the initial value for the symbol, in this case returning the value of **l**. The third (optional) element in the declaration list is an expression that is evaluated at the beginning of each successive iteration and whose result then becomes the value of the symbol at the start of that iteration. In this case it is "(cdr here)". Thus, *here* is defined to initially be the original list of nodes and on each successive iteration *here* is shortened by dropping the current first element.

The next thing in a **do** following the declarations list is an "exit clause". This is on line 3. An exit clause consists of a list of expressions. The first expression (sometimes called the predicate) in this list is evaluated at the start of each iteration. In this case it is the expression *(null here)*. In Lisp "true" means "anything other than *nil*". When a built-in function has to return a value meaning "true" it uses the symbol *t*, but any non-*nil* Lisp object will do. When the value of the predicate of the exit clause becomes true (think of it as "non-*nil*") then all of the expressions in the rest of the exit clause are evaluated in left-to-right order and the loop is exited, returning the value of the last expression in the exit clause. In the example, the predicate (null here) will be true when *here* becomes *nil* (the null or empty list), that is, when we've removed all of the node elements. The last expression in the exit clause is the symbol *l*, so the loop returns its value. Since the **do** is the last (only) expression in the definition of the **chflag** function, the value of *l* is also the value returned by it.

The remainder of the **do** form is a sequence of expressions that are known as the body of the loop. The expressions of the body are evaluated on each iteration in which the predicate of the exit clause is **nil**. In the example the body is the single expression that is a call to the primitive function **stop-on-change** with the **car** (its first element) of *here* as the first argument and with *t* as the second. This has the effect of flagging the node so that the simulation is halted whenever the node changes state so that the event can be reported or some other special action can be taken.

```
; Run a simulation step, reporting transitions
[1]   (defun step (incr)
[2]         (printf "0tep begins @ %S ns.0 (/ (float current-time) 10.0))
[3]         (do ((stop-time (+ incr current-time)) (savex (* current-time 1))
[4]                   (n t))
[5]             ((null n))
[6]             (setq n (cond (switch-level (switch-step stop-time))
[7]                           (t (sim-step stop-time))))
[8]             (cond (n (dpy-node-trans n)
[9]                      (printf "@ %S0
[10]                       (/ (cond (relative-timing (- current-time savex))
[11]                                (t (float current-time)))
[12]                          10)))
[13]                  (t nil)))
[14]  )
```

The function **step** is interesting to look at for a couple of reasons. It is the function that one uses to simulate a circuit for a particular time increment and is therefore worth knowing about both for having an understanding of what is going on and also in case one wants to modify it. It also contains a more complicated example of a **do** as well as introducing the **cond** special form. The first thing done (line 2) is to print out the current elapsed simulated time in nanoseconds.

The RNL provided simulation primitives (**sim-step** and **switch-step**) both operate by advancing the simulated time until either the specified stop time is reached (returning **nil** in this case), or until some circuit node that has been flagged changes state (returning the node

that changed). The function **step** uses these primitives to run the simulation for the fixed time increment specified in the parameter **incr**. Once the beginning time is printed, the body of the function is loop implemented with a **do**. On lines 3 and 4 the local variables are defined. **Step-time** is initialized to be the time to stop the simulation step, **savex** is initialized to be the time at which the step started, and **n**, which will be used to hold the value returned by each call to the appropriate simulation primitive, is initialized to t. Since the simulation primitives return **nil** when the appropriate stop time is reached, (null n) is used as the predicate (line 5) of the exit clause for the loop.

Lines 6 and 7 are the call to the simulation step. A **cond** special form consists of the symbol **cond** followed by a number of lists of expressions known as clauses. These are similar to the exit clause of a **do**. In a **cond** the clauses are examined in left-to-right order. For each clause, the predicate is evaluated and if it is true (returns other than *nil*) then the rest of the expressions in that clause are evaluated and the **cond** is exited, returning the value of the last expression evaluated. In the **cond** on lines 6 and 7, if **switch-level** is not **nil** then **switch-step** is called, otherwise (since the value of *t* is *t*) **sim-step** is called. Since the value returned by a **cond** is the last value computed in the clause that is executed, the value returned by this **cond** will be the value returned by the appropriate simulation primitive. This value is either a node with its STOPONCHANGE flag set or *nil* and the **setq** on line 6 makes it the value of *n*.

The first clause of the **cond** on line 8 is executed if the value of **n** is not **nil**. It first passes **n** to the function **dpy-node-trans** and then prints the current time, either relative to the start of the step or in absolute terms, depending on the value of the symbol **relative-timing**. The default clause (line 13) is included just for readability.

To summarize, the body of **step** repeatedly calls a simulation primitive until that primitive returns a **nil** to indicate that the desired termination time has been reached. Each time the simulation primitive does not return **nil** it is because a flagged node changed state, and a reporting function is called.

```
;; run n clock cycles
[1]   (defun c (n)
[2]     (cond ((eq n nil) (setq n 1))
[3]           ((not (numberp n)) (setq n (car n))))
[4]     (do ((index 0 (1+ index))
[5]          (i 0))
[6]         ((== index n) (wr-report))
[7]       (set-node 'phi1 1)
[8]       (set-node 'phi2 0)
[9]       (step incr)
[10]      (set-node 'phi1 0)
[11]      (step incr)
[12]      (set-node 'phi2 1)
[13]      (step incr)
[14]      (set-node 'phi2 0)
[15]      (step incr))
[16]  )
```

The function **c** runs the simulation for one or more cycles of a two phase (four interval) non-overlapping clock defined using the predefined node names "phi1" and "phi2". This is written to take advantage of RNL Lisp's alternate input syntax. The symbol *n* will be the number of cycles to simulate.

The **cond** in line 2 is used to set *n to the correct value. The first clause sets the count to 1 if the argument to* **c** is the empty list (**nil**) as would be the case if the command were entered by typing "c" on a line by itself. The second clause is used to differentiate between the cases of calling **c** using the standard parenthesized syntax (e.g. "(c 5)") or the interactive syntax (e.g. "c 5" on a line by itself).

The rest of the body of **c** is a simple **do** loop that uses **index** as the loop index to count **n** clock cycles and call **wr-report** when it exits. Lines 7 through 15 handle the mechanics of raising and lowering the clock lines as appropriate and advancing the simulated time by calling **step**. If the circuit you are simulating uses a different clocking discipline then you should write your own "clock cycle" function analagous to **c**. The code for **c** provides a reasonable template to modify for your own needs.

```
; unchanged-since returns a list of the nodes that are unchanged since time.
[1]  (defun unchanged-since (time)
[2]    (prog (uc-list)
[3]      (walk-net '(lambda (n)
[4]                  (cond ((< (node-time n) time)
[5]                         (setq uc-list (cons n uc-list)))
[6]                        (t uc-list)
[7]                  ))
[8]      )))
```

The function **unchanged-since** introduces some new Lisp constructs and also illustrates the usefulness of the **walk-net** primitive when you want to examine all the nodes in your circuit. **Unchanged-since** returns a list of the nodes whose last transition occurred prior to the argument *time*.

The **prog** special form is used to define local symbols. After the symbol **prog** comes a list of symbols that are to be made local and following that list is a sequence of expressions. When a prog is evaluated the old values of the symbols in the list are saved, their current values are set to *nil*, and the sequence of expressions is evaluated. When the evaluation is done the old values of the local variables are restored and the result of the evaluation of the last expression in the sequence is returned as the value of the **prog**. In this case, the only local symbol is *uc-list* and the value of the **prog** will be the result of evaluating the **walk-net** function.

The **walk-net** function takes as its argument a function definition or a symbol that has a function definition. **Walk-net** then traverses the circuit and for each node it passes that node to the function and evaluates it. The value returned by **walk-net** is the value returned by the function the last time it was called.

In the example we didn't want to bother with giving a symbol a function definition, so we used the **lambda** special form to define an "anonymous" function definition. This anonymous function will build a list of nodes that have not been recently changed and keep it as the value of *uc-list*.

The predicate of the first clause of the **cond** beginning on line 4 tests (using the < predicate) whether the most recent node transition time of the parameter *n* was before the threshold *time*. If so, then it sets the value of *uc-list* to be the list consisting of *n* as the first element followed by the the list that was the previous value of *uc-list*. This is done by the **cons** function. Note that this returns the list as its value. If the node has been recently changed then the "default" clause (line 6) of the **cond** is executed, returning the unchanged value of *uw-list*. The function defined by the **lambda** thus always returns the current list. **Walk-net** therefore will return the final value of the list and therefore so will the **prog** and therefore so will **unchanged-since**.

6.1. Summary

We hope that by presenting these examples in detail that we have made a little less imposing the prospect of writing your own Lisp functions as part of your RNL simulation.

SPICE User's Guide

A.Vladimirescu, Kaihe Zhang,
A.R.Newton, D.O.Pederson, A.Sangiovanni-Vincentelli

Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, Ca., 94720

This manual corresponds to SPICE version 2G6

Acknowledgement: Dr. Richard Dowell and Dr. Sally Liu have contributed to develop the present SPICE version. SPICE was originally developed by Dr. Lawrence Nagel and has been modified extensively by Dr. Ellis Cohen.

SPICE is a general-purpose circuit simulation program for nonlinear dc, nonlinear transient, and linear ac analyses. Circuits may contain resistors, capacitors, inductors, mutual inductors, independent voltage and current sources, four types of dependent sources, transmission lines, and the four most common semiconductor devices: diodes, BJT's, JFET's, and MOSFET's.

SPICE has built-in models for the semiconductor devices, and the user need specify only the pertinent model parameter values. The model for the BJT is based on the integral charge model of Gummel and Poon; however, if the Gummel- Poon parameters are not specified, the model reduces to the simpler Ebers-Moll model. In either case, charge storage effects, ohmic resistances, and a current-dependent output conductance may be included. The diode model can be used for either junction diodes or Schottky barrier diodes. The JFET model is based on the FET model of Shichman and Hodges. Three MOSFET models are implemented; MOS1 is described by a square-law I-V characteristic MOS2 is an analytical model while MOS3 is a semi-empirical model. Both MOS2 and MOS3 include second-order effects such as channel length modulation, subthreshold conduction, scattering limited velocity saturation, small size effects and charge-controlled capacitances.

1. TYPES OF ANALYSIS

1.1. DC Analysis

The dc analysis portion of SPICE determines the dc operating point of the circuit with inductors shorted and capacitors opened. A dc analysis is automatically performed prior to a transient analysis to determine the transient initial conditions, and prior to an ac small-signal analysis to determine the linearized, small-signal models for nonlinear devices. If requested, the dc small-signal value of a transfer function (ratio of output variable to input

source), input resistance, and output resistance will also be computed as a part of the dc solution. The dc analysis can also be used to generate dc transfer curves: a specified independent voltage or current source is stepped over a user-specified range and the dc output variables are stored for each sequential source value. If requested, SPICE also will determine the dc small-signal sensitivities of specified output variables with respect to circuit parameters. The dc analysis options are specified on the .DC, .TF, .OP, and .SENS control cards.

If one desires to see the small signal models for nonlinear devices in conjunction with a transient analysis operating point, then the .OP card must be provided. The dc bias conditions will be identical for each case, but the more comprehensive operating point information is not available to be printed when transient initial conditions are computed.

1.2. AC Small-Signal Analysis

The ac small-signal portion of SPICE computes the ac output variables as a function of frequency. The program first computes the dc operating point of the circuit and determines linearized, small-signal models for all of the nonlinear devices in the circuit. The resultant linear circuit is then analyzed over a user-specified range of frequencies. The desired output of an ac small- signal analysis is usually a transfer function (voltage gain, transimpedance, etc). If the circuit has only one ac input, it is convenient to set that input to unity and zero phase, so that output variables have the same value as the transfer function of the output variable with respect to the input.

The generation of white noise by resistors and semiconductor devices can also be simulated with the ac small-signal portion of SPICE. Equivalent noise source values are determined automatically from the small-signal operating point of the circuit, and the contribution of each noise source is added at a given summing point. The total output noise level and the equivalent input noise level are determined at each frequency point. The output and input noise levels are normalized with respect to the square root of the noise bandwidth and have the units Volts/rt Hz or Amps/rt Hz. The output noise and equivalent input noise can be printed or plotted in the same fashion as other output variables. No additional input data are necessary for this analysis.

Flicker noise sources can be simulated in the noise analysis by including values for the parameters KF and AF on the appropriate device model cards.

The distortion characteristics of a circuit in the small signal mode can be simulated as a part of the ac small-signal analysis. The analysis is performed assuming that one or two signal frequencies are imposed at the input.

The frequency range and the noise and distortion analysis parameters are specified on the .AC, .NOISE, and .DISTO control lines.

1.3. Transient Analysis

The transient analysis portion of SPICE computes the transient output variables as a function of time over a user specified time interval. The initial conditions are automatically determined by a dc analysis. All sources which are not time dependent (for example, power supplies) are set to their dc value. For large-signal sinusoidal simulations, a Fourier analysis of the output waveform can be specified to obtain the frequency domain Fourier coefficients. The transient time interval and the Fourier analysis options are specified on the .TRAN and .FOURIER control lines.

1.4. Analysis at Different Temperatures

All input data for SPICE is assumed to have been measured at 27 deg C (300 deg K). The simulation also assumes a nominal temperature of 27 deg C. The circuit can be simulated at other temperatures by using a .TEMP control line.

Temperature appears explicitly in the exponential terms of the BJT and diode model equations. In addition, saturation currents have a built-in temperature dependence. The

temperature dependence of the saturation current in the BJT models is determined by:

$$IS(T1) = IS(T0)\left[\left(\frac{T1}{T0}\right)^{XTI} e^{\frac{q\,EG\,(T1-T0)}{k\,T1\,T0}}\right]$$

where k is Boltzmann's constant, q is the electronic charge, EG is the energy gap which is a model parameter, and XTI is the saturation current temperature exponent (also a model parameter, and usually equal to 3). The temperature dependence of forward and reverse beta is according to the formula:

$$beta(T1) = beta(T0)\left[\left(\frac{T1}{T0}\right)^{XTB}\right]$$

where T1 and T0 are in degrees Kelvin, and XTB is a user-supplied model parameter. Temperature effects on beta are carried out by appropriate adjustment to the values of BF, ISE, BR, and ISC. Temperature dependence of the saturation current in the junction diode model is determined by:

$$IS(T1) = IS(T0)\left[\left(\frac{T1}{T0}\right)^{\left(\frac{XTI}{N}\right)} e^{\frac{q\,EG\,(T1-T0)}{(k\,N\,T1\,T0)}}\right]$$

where N is the emission coefficient, which is a model parameter, and the other symbols have the same meaning as above. Note that for Schottky barrier diodes, the value of the saturation current temperature exponent, XTI, is usually 2.

Temperature appears explicitly in the value of junction potential, PHI, for all the device models. The temperature dependence is determined by:

$$PHI(TEMP) = \frac{k\,TEMP}{q\log\left(\frac{Na\,Nd}{Ni\,(TEMP)^2}\right)}$$

where k is Boltzmann's constant, q is the electronic charge, Na is the acceptor impurity density, Nd is the donor impurity den sity, Ni is the intrinsic concentration, and EG is the energy gap.

Temperature appears explicitly in the value of surface mobility, UO, for the MOSFET model. The temperature dependence is determined by:

$$UO(TEMP) = \frac{UO(TNOM)}{(TEMP/TNOM)^{1.5}}$$

The effects of temperature on resistors is modeled by the formula:

$$value(TEMP) = value(TNOM)\left[1 + TC1\,(TEMP - TNOM) + TC2\left[(TEMP - TNOM)^2\right]\right]$$

where TEMP is the circuit temperature, TNOM is the nominal temperature, and TC1 and TC2 are the first- and second-order temperature coefficients.

2. CONVERGENCE

Both dc and transient solutions are obtained by an iterative process which is terminated when both of the following conditions hold:

1) The nonlinear branch currents converge to within a tolerance of 0.1 percent or 1 picoamp (1.0E-12 Amp), whichever is larger.
2) The node voltages converge to within a tolerance of 0.1 per cent or 1 microvolt (1.0E-6 Volt), whichever is larger.

Although the algorithm used in SPICE has been found to be very reliable, in some cases it will fail to converge to a solution. When this failure occurs, the program will print the node voltages at the last iteration and terminate the job. In such cases, the node voltages that are printed are not necessarily correct or even close to the correct solution.

Failure to converge in the dc analysis is usually due to an error in specifying circuit connections, element values, or model parameter values. Regenerative switching circuits or circuits with positive feedback probably will not converge in the dc analysis unless the OFF option is used for some of the devices in the feedback path, or the .NODESET card is used to

force the circuit to converge to the desired state.

3. INPUT FORMAT

The input format for SPICE is of the free format type. Fields on a card are separated by one or more blanks, a comma, an equal (=) sign, or a left or right parenthesis; extra spaces are ignored. A card may be continued by entering a + (plus) in column 1 of the following card; SPICE continues reading beginning with column 2.

A name field must begin with a letter (A through Z) and cannot contain any delimiters. Only the first eight characters of the name are used.

A number field may be an integer field (12, -44), a floating point field (3.14159), either an integer or floating point number followed by an integer exponent (1E-14, 2.65E3), or either an integer or a floating point number followed by one of the following scale factors:

T=1E12 G=1E9 MEG=1E6 K=1E3 MIL=25.4E-6 M=1E-3 U=1E-6 N=1E-9 P=1E-12 F=1E-15

Letters immediately following a number that are not scale factors are ignored, and letters immediately following a scale factor are ignored. Hence, 10, 10V, 10VOLTS, and 10HZ all represent the same number, and M, MA, MSEC, and MMHOS all represent the same scale factor. Note that 1000, 1000.0, 1000HZ, 1E3, 1.0E3, 1KHZ, and 1K all represent the same number.

4. CIRCUIT DESCRIPTION

The circuit to be analyzed is described to SPICE by a set of element cards, which define the circuit topology and element values, and a set of control cards, which define the model parameters and the run controls. The first card in the input deck must be a title card, and the last card must be a .END card. The order of the remaining cards is arbitrary (except, of course, that continuation cards must immediately follow the card being continued).

Each element in the circuit is specified by an element card that contains the element name, the circuit nodes to which the element is connected, and the values of the parameters that determine the electrical characteristics of the element. The first letter of the element name specifies the element type. The format for the SPICE element types is given in what follows. The strings XXXXXXX, YYYYYYY, and ZZZZZZZ denote arbitrary alphanumeric strings. For example, a resistor name must begin with the letter R and can contain from one to eight characters. Hence, R, R1, RSE, ROUT, and R3AC2ZY are valid resistor names.

Data fields that are enclosed in lt and gt signs '< >' are optional. All indicated punctuation (parentheses, equal signs, etc.) are required. With respect to branch voltages and currents, SPICE uniformly uses the associated reference convention (current flows in the direction of voltage drop).

Nodes must be nonnegative integers but need not be numbered sequentially. The datum (ground) node must be numbered zero. The circuit cannot contain a loop of voltage sources and/or inductors and cannot contain a cutset of current sources and/or capacitors. Each node in the circuit must have a dc path to ground. Every node must have at least two connections except for transmission line nodes (to permit unterminated transmission lines) and MOSFET substrate nodes (which have two internal connections anyway).

5. TITLE CARD, COMMENT CARDS AND .END CARD

5.1. Title Card

Examples:

```
POWER AMPLIFIER CIRCUIT
TEST OF CAM CELL
```

This card must be the first card in the input deck. Its contents are printed verbatim as the heading for each section of output.

5.2. .END Card

Examples:

```
.END
```

This card must always be the last card in the input deck. Note that the period is an integral part of the name.

5.3. Comment Card

General Form:

```
* <any comment>
```

Examples:

```
* RF=1K      GAIN SHOULD BE 100
* MAY THE FORCE BE WITH MY CIRCUIT
```

The asterisk in the first column indicates that this card is a comment card. Comment cards may be placed anywhere in the circuit description.

6. ELEMENT CARDS

6.1. Resistors

General form:

```
RXXXXXXX N1 N2 VALUE <TC=TC1<,TC2>>
```

Examples:

```
R1 1 2 100
RC1 12 17 1K TC=0.001,0.015
```

N1 and N2 are the two element nodes. VALUE is the resistance (in ohms) and may be positive or negative but not zero. TC1 and TC2 are the (optional) temperature coefficients; if not specified, zero is assumed for both. The value of the resistor as a function of temperature is given by:

$$\text{value}(\text{TEMP}) = \text{value}(\text{TNOM})[1 + \text{TC}1(\text{TEMP} - \text{TNOM}) + \text{TC}2[(\text{TEMP} - \text{TNOM})^2]$$

6.2. Capacitors and Inductors

General form:

```
CXXXXXXX N+ N- VALUE <IC=INCOND>
LYYYYYYY N+ N- VALUE <IC=INCOND>
```

Examples:

```
CBYP 13 0 1UF
COSC 17 23 10U IC=3V
LLINK 42 69 1UH
LSHUNT 23 51 10U IC=15.7MA
```

N+ and N- are the positive and negative element nodes, respectively. VALUE is the capacitance in Farads or the inductance in Henries.

For the capacitor, the (optional) initial condition is the initial (time-zero) value of capacitor voltage (in Volts). For the inductor, the (optional) initial condition is the initial (time-zero) value of inductor current (in Amps) that flows from N+, through the inductor, to N-. Note that the initial conditions (if any) apply 'only' if the UIC option is specified on the .TRAN card.

Nonlinear capacitors and inductors can be described.

General form :

```
CXXXXXXX N+ N- POLY C0 C1 C2 ... <IC=INCOND>
LYYYYYYY N+ N- POLY L0 L1 L2 ... <IC=INCOND>
```

C0 C1 C2 ...(and L0 L1 L2 ...) are the coefficients of a polynomial describing the element value. The capacitance is expressed as a function of the voltage across the element while the inductance is a function of the current through the inductor. The value is computed as

$$\text{value} = C0 + C1 \cdot V + C2 \cdot V^2 + \ldots$$

$$\text{value} = L0 + L1 \cdot I + L2 \cdot I^2 + \ldots$$

where V is the voltage across the capacitor and I the current flowing in the inductor.

6.3. Coupled (Mutual) Inductors

General form:

```
KXXXXXXX LYYYYYYY LZZZZZZZ VALUE
```

Examples:

```
K43 LAA LBB 0.999
KXFRMR L1 L2 0.87
```

LYYYYYYY and LZZZZZZZ are the names of the two coupled inductors, and VALUE is the coefficient of coupling, K, which must be greater than 0 and less than or equal to 1. Using the 'dot' convention, place a 'dot' on the first node of each inductor.

6.4. Transmission Lines (Lossless)

General form:

```
TXXXXXXX N1 N2 N3 N4 Z0=VALUE <TD=VALUE> <F=FREQ <NL=NRMLEN>>
+                <IC=V1,I1,V2,I2>
```

Examples:

```
T1 1 0 2 0 Z0=50 TD=10NS
```

N1 and N2 are the nodes at port 1; N3 and N4 are the nodes at port 2. Z0 is the characteristic impedance. The length of the line may be expressed in either of two forms. The transmission delay, TD, may be specified directly (as TD=10ns, for example). Alternatively, a frequency F may be given, together with NL, the normalized electrical length of the transmission line with respect to the wavelength in the line at the frequency F. If a frequency is specified but NL is omitted, 0.25 is assumed (that is, the frequency is assumed to be the quarter-wave frequency). Note that although both forms for expressing the line length are indicated as optional, one of the two must be speci fied.

Note that this element models only one propagating mode. If all four nodes are distinct in the actual circuit, then two modes may be excited. To simulate such a situation, two transmission line elements are required. (see the example in Appendix A for further clarification.)

The (optional) initial condition specification consists of the voltage and current at each of the transmission line ports. Note that the initial conditions (if any) apply 'only' if the UIC option is specified on the .TRAN card.

One should be aware that SPICE will use a transient time step which does not exceed 1/2 the minimum transmission line delay. Therefore very short transmission lines (compared with the analysis time frame) will cause long run times.

6.5. Linear Dependent Sources

SPICE allows circuits to contain linear dependent sources characterized by any of the four equations

$i = g \cdot v \qquad v = e \cdot v \qquad i = f \cdot i \qquad v = h \cdot i$

where g, e, f, and h are constants representing transconductance, voltage gain, current gain, and transresistance, respectively. Note: a more complete description of dependent sources as imple mented in SPICE is given in Appendix B.

6.6. Linear Voltage-Controlled Current Sources

General form:

```
GXXXXXXX N+ N- NC+ NC- VALUE
```

Examples:

```
G1 2 0 5 0 0.1MMHO
```

N+ and N- are the positive and negative nodes, respectively. Current flow is from the positive node, through the source, to the negative node. NC+ and NC- are the positive and negative controlling nodes, respectively. VALUE is the transconductance (in mhos).

6.7. Linear Voltage-Controlled Voltage Sources

General form:

EXXXXXXX N+ N- NC+ NC- VALUE

Examples:

E1 2 3 14 1 2.0

N+ is the positive node, and N- is the negative node. NC+ and NC- are the positive and negative controlling nodes, respectively. VALUE is the voltage gain.

6.8. Linear Current-Controlled Current Sources

General form:

FXXXXXXX N+ N- VNAM VALUE

Examples:

F1 13 5 VSENS 5

N+ and N- are the positive and negative nodes, respectively. Current flow is from the positive node, through the source, to the negative node. VNAM is the name of a voltage source through which the controlling current flows. The direction of positive controlling current flow is from the positive node, through the source, to the negative node of VNAM. VALUE is the current gain.

6.9. Linear Current-Controlled Voltage Sources

General form:

HXXXXXXX N+ N- VNAM VALUE

Examples:

HX 5 17 VZ 0.5K

N+ and N- are the positive and negative nodes, respectively. VNAM is the name of a voltage source through which the controlling current flows. The direction of positive controlling current flow is from the positive node, through the source, to the negative node of VNAM. VALUE is the transresistance (in ohms).

6.10. Independent Sources

General form:

VXXXXXXX N+ N- < < DC> DC/TRAN VALUE> < AC < ACMAG < ACPHASE> > >
IYYYYYYY N+ N- < < DC> DC/TRAN VALUE> < AC < ACMAG < ACPHASE> > >

Examples:

VCC 10 0 DC 6

```
VIN 13 2 0.001 AC 1 SIN(0 1 1MEG)
ISRC 23 21 AC 0.333 45.0 SFFM(0 1 10K 5 1K)
VMEAS 12 9
```

N+ and N- are the positive and negative nodes, respectively. Note that voltage sources need not be grounded. Positive current is assumed to flow from the positive node, through the source, to the negative node. A current source of positive value, will force current to flow out of the N+ node, through the source, and into the N- node. Voltage sources, in addition to being used for circuit excitation, are the 'ammeters' for SPICE, that is, zero valued voltage sources may be inserted into the circuit for the purpose of measuring current. They will, of course, have no effect on circuit operation since they represent short-circuits.

DC/TRAN is the dc and transient analysis value of the source. If the source value is zero both for dc and transient analyses, this value may be omitted. If the source value is time-invariant (e.g., a power supply), then the value may optionally be preceded by the letters DC.

ACMAG is the ac magnitude and ACPHASE is the ac phase. The source is set to this value in the ac analysis. If ACMAG is omitted following the keyword AC, a value of unity is assumed. If ACPHASE is omitted, a value of zero is assumed. If the source is not an ac small-signal input, the keyword AC and the ac values are omitted.

Any independent source can be assigned a time-dependent value for transient analysis. If a source is assigned a time- dependent value, the time-zero value is used for dc analysis. There are five independent source functions: pulse, exponential, sinusoidal, piecewise linear, and single-frequency FM. If parameters other than source values are omitted or set to zero, the default values shown will be assumed. (TSTEP is the printing increment and TSTOP is the final time (see the .TRAN card for explanation)).

1. Pulse PULSE(V1 V2 TD TR TF PW PER)

Examples:

```
VIN 3 0 PULSE(-1 1 2NS 2NS 2NS 50NS 100NS)
```

parameter	default	units
V1 (initial value)		Volts or Amps
V2 (pulsed value)		Volts or Amps
TD (delay time)	0.0	seconds
TR (rise time)	TSTEP	seconds
TF (fall time)	TSTEP	seconds
PW (pulse width)	TSTOP	seconds
PER(period)	TSTOP	seconds

A single pulse so specified is described by the following table:

time	value
0	V1
TD	V1
TD+TR	V2
TD+TR+PW	V2
TD+TR+PW+TF	V1
TSTOP	V1

Intermediate points are determined by linear interpolation.

2. Sinusoidal SIN(VO VA FREQ TD THETA)

Examples:

```
VIN 3 0 SIN(0 1 100MEG 1NS 1E10)
```

parameter	default value	units
VO	(offset)	Volts or Amps
VA	(amplitude)	Volts or Amps
FREQ	(frequency)	1/TSTOP
TD	(delay)	0.0
THETA	(damping factor)	0.0

The shape of the waveform is described by the following table:

time	value
0 to TD	VO
TD to TSTOP	$V0 + VA\,e^{(-(\text{time}-TD)\Theta)}\sin(2\pi\,FREQ(\text{time}+TD))$

3. Exponential EXP(V1 V2 TD1 TAU1 TD2 TAU2)

Examples:

```
VIN 3 0 EXP(-4 -1 2NS 30NS 60NS 40NS)
```

parameters	default values	units
V1	(initial value)	Volts or Amps
V2	(pulsed value)	Volts or Amps
TD1	(rise delay time)	0.0
TAU1	(rise time constant)	TSTEP
TD2	(fall delay time)	TD1+TSTEP
TAU2	(fall time constant)	TSTEP

The shape of the waveform is described by the following table:

time	value
0 to TD1	V1
TD1 to TD2	$V1 + (V2-V1)[1-e^{(-(\text{time}-TD1)/TAU1)}]$
TD2 to TSTOP	$V1 + (V2-V1)[1-e^{(-(\text{time}-TD1)/TAU1)} + (V1-V2)[1-e^{(-(\text{time}-TD2)/TAU2)}]$

4. Piece-Wise Linear PWL(T1 V1 <T2 V2 T3 V3 T4 V4 ...>)

Example:

VCLOCK 7 5 PWL(0 -7 10NS -7 11NS -3 17NS -3 18NS -7 50NS -7)

Each pair of values (Ti, Vi) specifies that the value of the source is Vi (in Volts or Amps) at time=Ti. The value of the source at intermediate values of time is determined by using linear interpolation on the input values.

5. Single-Frequency FM SFFM(VO VA FC MDI FS)

Examples:

V1 12 0 SFFM(0 1M 20K 5 1K)

parameters	default values	units
VO	(offset)	Volts or Amps
VA	(amplitude)	Volts or Amps
FC	(carrier frequency)	1/TSTOP
MDI	(modulation index)	
FS	(signal frequency)	1/TSTOP

The shape of the waveform is described by the following equation:

$$value + VO + VA\sin((2\pi \cdot FC \cdot time) + MDI\sin(2\pi \cdot FS \cdot time))$$

7. SEMICONDUCTOR DEVICES

The elements that have been described to this point typically require only a few parameter values to specify completely the electrical characteristics of the element. However, the models for the four semiconductor devices that are included in the SPICE program require many parameter values. Moreover, many devices in a circuit often are defined by the same set of device model parameters. For these reasons, a set of device model parameters is defined on a separate .MODEL card and assigned a unique model name. The device element cards in SPICE then reference the model name. This scheme alleviates the need to specify all of the model parameters on each device element card.

Each device element card contains the device name, the nodes to which the device is connected, and the device model name. In addition, other optional parameters may be specified for each device: geometric factors and an initial condition.

The area factor used on the diode, BJT and JFET device card determines the number of equivalent parallel devices of a specified model. The affected parameters are marked with an asterisk under the heading 'area' in the model descriptions below. Several geometric factors associated with the channel and the drain and source diffusions can be specified on the MOSFET device card.

Two different forms of initial conditions may be specified for devices. The first form is included to improve the dc convergence for circuits that contain more than one stable state. If a device is specified OFF, the dc operating point is deter mined with the terminal voltages for that device set to zero. After convergence is obtained, the program continues to iterate to obtain the exact value for the terminal voltages. If a circuit has more than one dc stable state, the OFF option can be used to force the solution to correspond to a desired state. If a device is specified OFF when in reality the device is conducting, the program will still obtain the correct solution (assuming the solutions converge) but more iterations will be required since the program must independently converge to two separate solutions. The .NODESET card serves a similar purpose as the OFF option. The .NODESET

option is easier to apply and is the preferred means to aid convergence.

The second form of initial conditions are specified for use with the transient analysis. These are true 'initial conditions' as opposed to the convergence aids above. See the description of the .IC card and the .TRAN card for a detailed explanation of initial conditions.

7.1. Junction Diodes

General form:

DXXXXXXX N+ N- MNAME <AREA> <OFF> <IC=VD>

Examples:

```
DBRIDGE 2 10 DIODE1
DCLMP 3 7 DMOD 3.0 IC=0.2
```

N+ and N- are the positive and negative nodes, respectively. MNAME is the model name, AREA is the area factor, and off indicates an (optional) starting condition on the device for dc analysis. If the area factor is omitted, a value of 1.0 is assumed. The (optional) initial condition specification using IC=VD is intended for use with the UIC option on the .TRAN card, when a transient analysis is desired starting from other than the quiescent operating point.

7.2. Bipolar Junction Transistors (BJT's)

General form:

QXXXXXXX NC NB NE <NS> MNAME <AREA> <OFF> <IC=VBE,VCE>

Examples:

```
Q23 10 24 13 QMOD IC=0.6,5.0
Q50A 11 26 4 20 MOD1
```

NC, NB, and NE are the collector, base, and emitter nodes, respectively. NS is the (optional) substrate node. If unspecified, ground is used. MNAME is the model name, AREA is the area factor, and OFF indicates an (optional) initial condition on the device for the dc analysis. If the area factor is omitted, a value of 1.0 is assumed. The (optional) initial condition specification using IC=VBE,VCE is intended for use with the UIC option on the .TRAN card, when a transient analysis is desired starting from other than the quiescent operating point. See the .IC card description for a better way to set transient initial conditions.

7.3. Junction Field-Effect Transistors (JFET's)

General form:

JXXXXXXX ND NG NS MNAME <AREA> <OFF> <IC=VDS,VGS>

Examples:

```
J1 7 2 3 JM1 OFF
```

ND, NG, and NS are the drain, gate, and source nodes, respectively. MNAME is the model name, AREA is the area factor, and OFF indicates an (optional) initial condition on the device for dc analysis. If the area factor is omitted, a value of 1.0 is assumed. The (optional) initial condition specification, using IC=VDS,VGS is intended for use with the UIC option on the .TRAN card, when a transient analysis is desired starting from other than the quiescent operating point (see the .IC card for a better way to set initial conditions).

7.4. MOSFET's

General form:

MXXXXXXX ND NG NS NB MNAME <L=VAL> <W=VAL> <AD=VAL> <AS=VAL>
+ <PD=VAL> <PS=VAL> <NRD=VAL> <NRS=VAL> <OFF> <IC=VDS,VGS,VBS>

Examples:

```
M1 24 2 0 20 TYPE1
M31 2 17 6 10 MODM L=5U W=2U
M31 2 16 6 10 MODM 5U 2U
M1 2 9 3 0 MOD1 L=10U W=5U AD=100P AS=100P PD=40U PS=40U
M1 2 9 3 0 MOD1 10U 5U 2P 2P
```

ND, NG, NS, and NB are the drain, gate, source, and bulk (substrate) nodes, respectively. MNAME is the model name. L and W are the channel length and width, in meters. AD and AS are the areas of the drain and source diffusions, in sq-meters. Note that the suffix U specifies microns (1E-6 m) and P sq-microns (1E-12 sq-m). If any of L, W, AD, or AS are not specified, default values are used. The user may specify the values to be used for these default parameters on the .OPTIONS card. The use of defaults simplifies input deck preparation, as well as the editing required if device geometries are to be changed. PD and PS are the perimeters of the drain and source junctions, in meters. NRD and NRS designate the equivalent number of squares of the drain and source diffusions; these values multiply the sheet resistance RSH specified on the .MODEL card for an accurate representation of the parasitic series drain and source resistance of each transistor. PD and PS default to 0.0 while NRD and NRS to 1.0. OFF indicates an (optional) initial condition on the device for dc analysis. The (optional) initial condition specification using IC=VDS,VGS,VBS is intended for use with the UIC option on the .TRAN card, when a transient analysis is desired starting from other than the quiescent operating point. See the .IC card for a better and more convenient way to specify transient initial conditions.

7.5. .MODEL Card

General form:

.MODEL MNAME TYPE(PNAME1=PVAL1 PNAME2=PVAL2 ...)

Examples:

.MODEL MOD1 NPN BF=50 IS=1E-13 VBF=50

The .MODEL card specifies a set of model parameters that will be used by one or more devices. MNAME is the model name, and type is one of the following seven types:

type	description

NPN	NPN BJT model
PNP	PNP BJT model
D	diode model
NJF	N-channel JFET model
PJF	P-channel JFET model
NMOS	N-channel MOSFET model
PMOS	P-channel MOSFET model

Parameter values are defined by appending the parameter name, as given below for each model type, followed by an equal sign and the parameter value. Model parameters that are not given a value are assigned the default values given below for each model type.

7.6. Diode Model

The dc characteristics of the diode are determined by the parameters IS and N. An ohmic resistance, RS, is included. Charge storage effects are modeled by a transit time, TT, and a nonlinear depletion layer capacitance which is determined by the parameters CJO, VJ, and M. The temperature dependence of the saturation current is defined by the parameters EG, the energy and XTI, the saturation current temperature exponent. Reverse breakdown is modeled by an exponential increase in the reverse diode current and is determined by the parameters BV and IBV (both of which are positive numbers).

	name	parameter	units	default	example	area
1	IS	saturation current	A	1.0E-14	1.0E-14	•
2	RS	ohmic resistance	Ohm	0	10	•
3	N	emission coefficient	-	1	1.0	
4	TT	transit-time	sec	0	0.1Ns	
5	CJO	zero-bias junction capacitance	F	0	2PF	•
6	VJ	junction potential	V	1	0.6	
7	M	grading coefficient	-	0.5	0.5	
8	EG	activation energy	eV	1.11	1.11 Si 0.69 Sbd 0.67 Ge	
9	XTI	saturation-current temp. exp	-	3.0	3.0 jn 2.0 Sbd	
10	KF	flicker noise coefficient	-	0		
11	AF	flicker noise exponent	-	1		
12	FC	coefficient for forward-bias depletion capacitance formula	-	0.5		
13	BV	reverse breakdown voltage	V	infinite	40.0	
14	IBV	current at breakdown voltage	A	1.0E-3		

7.7. BJT Models (both NPN and PNP)

The bipolar junction transistor model in SPICE is an adaptation of the integral charge control model of Gummel and Poon. This modified Gummel-Poon model extends the original model to include several effects at high bias levels. The model will automatically simplify to the simpler Ebers-Moll model when certain parameters are not specified. The parameter names used in the modified Gummel-Poon model have been chosen to be more easily understood by the program user, and to reflect better both physical and circuit design thinking.

The dc model is defined by the parameters IS, BF, NF, ISE, IKF, and NE which determine the forward current gain characteristics, IS, BR, NR, ISC, IKR, and NC which determine the reverse current gain characteristics, and VAF and VAR which determine the

output conductance for forward and reverse regions. Three ohmic resistances RB, RC, and RE are included, where RB can be high current dependent. Base charge storage is modeled by forward and reverse transit times, TF and TR, the forward transit time TF being bias dependent if desired, and nonlinear depletion layer capacitances which are determined by CJE, VJE, and MJE for the B-E junction , CJC, VJC, and MJC for the B-C junction and CJS, VJS, and MJS for the C-S (Collector-Substrate) junction. The temperature dependence of the saturation current, IS, is determined by the energy-gap, EG, and the saturation current temperature exponent, XTI. Additionally base current temperature dependence is modeled by the beta temperature exponent XTB in the new model.

The BJT parameters used in the modified Gummel-Poon model are listed below. The parameter names used in earlier versions of SPICE2 are still accepted.

Modified Gummel-Poon BJT Parameters

	name	parameter	units	default	example	area
1	IS	transport saturation current	A	1.0E-16	1.0E-15	•
2	BF	ideal maximum forward beta	-	100	100	
3	NF	forward current emission coefficient	-	1.0	1	
4	VAF	forward Early voltage	V	infinite	200	
5	IKF	corner for forward beta high current roll-off	A	infinite	0.01	•
6	ISE	B-E leakage saturation current	A	0	1.0E-13	•
7	NE	B-E leakage emission coefficient	-	1.5	2	
8	BR	ideal maximum reverse beta	-	1	0.1	
9	NR	reverse current emission coefficient	-	1	1	
10	VAR	reverse Early voltage	V	infinite	200	
11	IKR	corner for reverse beta high current roll-off	A	infinite	0.01	•
12	ISC	B-C leakage saturation current	A	0	1.0E-13	•
13	NC	B-C leakage emission coefficient	-	2	1.5	
14	RB	zero bias base resistance	Ohms	0	100	•
15	IRB	current where base resistance falls halfway to its min value	A	infinite	0.1	•
16	RBM	minimum base resistance at high currents	Ohms	RB	10	•
17	RE	emitter resistance	Ohms	0	1	•
18	RC	collector resistance	Ohms	0	10	•
19	CJE	B-E zero-bias depletion capacitance	F	0	2PF	•
20	VJE	B-E built-in potential	V	0.75	0.6	
21	MJE	B-E junction exponential factor	-	0.33	0.33	
22	TF	ideal forward transit time	sec	0	0.1Ns	
23	XTF	coefficient for bias dependence of TF	-	0		
24	VTF	voltage describing VBC dependence of TF	V	infinite		
25	ITF	high-current parameter for effect on TF	A	0	•	
26	PTF	excess phase at freq = 1.0/(TF·2PI)	Hz	deg	0	
27	CJC	B-C zero-bias depletion capacitance	F	0	2PF	•
28	VJC	B-C built-in potential	V	0.75	0.5	
29	MJC	B-C junction exponential factor	-	0.33	0.5	
30	XCJC	fraction of B-C depletion capacitance connected to internal base node	-	1		
31	TR	ideal reverse transit time	sec	0	10Ns	
32	CJS	zero-bias collector-substrate				

		capacitance	F	0	2PF	*
	VJS	substrate junction built-in potential	V	0.75		
	MJS	substrate junction exponential factor	-	0	0.5	
	XTB	forward and reverse beta temperature exponent	-	0		
36	EG	energy gap for temperature effect on IS	eV	1.11		
37	XTI	temperature exponent for effect on IS	-	3		
38	KF	flicker-noise coefficient	-	0		
39	AF	flicker-noise exponent	-	1		
40	FC	coefficient for forward-bias depletion capacitance formula	-	0.5		

7.8. JFET Models (both N and P Channel)

The JFET model is derived from the FET model of Shichman and Hodges. The dc characteristics are defined by the parameters VTO and BETA, which determine the variation of drain current with gate voltage, LAMBDA, which determines the output conductance, and IS, the saturation current of the two gate junctions. Two ohmic resistances, RD and RS, are included. Charge storage is modeled by nonlinear depletion layer capacitances for both gate junctions which vary as the -1/2 power of junction voltage and are defined by the parameters CGS, CGD, and PB.

	name	parameter	units	default	example	area
1	VTO	threshold voltage	V	-2.0	-2.0	
2	BETA	transconductance parameter	A/V^2	1.0E-4	1.0E-3	*
3	LAMBDA	channel length modulation parameter	1/V	0	1.0E-4	
4	RD	drain ohmic resistance	Ohm	0	100	*
5	RS	source ohmic resistance	Ohm	0	100	*
6	CGS	zero-bias G-S junction capacitance	F	0	5PF	*
7	CGD	zero-bias G-D junction capacitance	F	0	1PF	*
8	PB	gate junction potential	V	1	0.6	
9	IS	gate junction saturation current	A	1.0E-14	1.0E-14	*
10	KF	flicker noise coefficient	-	0		
11	AF	flicker noise exponent	-	1		
12	FC	coefficient for forward-bias depletion capacitance formula	-	0.5		

7.9. MOSFET Models (both N and P channel)

SPICE provides three MOSFET device models which differ in the formulation of the I-V characteristic. The variable LEVEL specifies the model to be used:

LEVEL=1 -> Shichman-Hodges
LEVEL=2 -> MOS2 (as described in [1])
LEVEL=3 -> MOS3, a semi-empirical model(see [1])

The dc characteristics of the MOSFET are defined by the device parameters VTO, KP, LAMBDA, PHI and GAMMA. These parameters are computed by SPICE if process parameters (NSUB, TOX, ...) are given, but user-specified values always override. VTO is positive (negative) for enhancement mode and negative (positive) for depletion mode N-

channel (P-channel) devices. Charge storage is modeled by three constant capacitors, CGSO, CGDO, and CGBO which represent overlap capacitances, by the nonlinear thin-oxide capacitance which is distributed among the gate, source, drain, and bulk regions, and by the nonlinear depletion-layer capacitances for both substrate junctions divided into bottom and periphery, which vary as the MJ and MJSW power of junction voltage respecively, and are determined by the parameters CBD, CBS, CJ, CJSW, MJ, MJSW and PB. There are two built-in models of the charge storage effects associated with the thin-oxide. The default is the piecewise linear voltage-dependent capacitance model proposed by Meyer. The second choice is the charge-controlled capacitance model of Ward and Dutton [1]. The XQC model parameter acts as a flag and a coefficient at the same time. As the former it causes the program to use Meyer's model whenever larger than 0.5 or not specified, and the charge-controlled model when between 0 and 0.5. In the latter case its value defines the share of the channel charge associated with the drain terminal in the saturation region. The thin-oxide charge storage effects are treated slightly different for the LEVEL=1 model. These voltage- dependent capacitances are included only if TOX is specified in the input description and they are represented using Meyer's formulation.

There is some overlap among the parameters describing the junctions, e.g. the reverse current can be input either as IS (in A) or as JS (in A/m^2). Whereas the first is an absolute value the second is multiplied by AD and AS to give the reverse current of the drain and source junctions respectively. This methodology has been chosen since there is no sense in relating always junction characteristics with AD and AS entered on the device card; the areas can be defaulted. The same idea applies also to the zero-bias junction capacitances CBD and CBS (in F) on one hand, and CJ (in F/m^2) on the other. The parasitic drain and source series resistance can be expressed as either RD and RS (in ohms) or RSH (in ohms/sq.), the latter being multiplied by the number of squares NRD and NRS input on the device card.

	name	parameter	units	default	example
1	LEVEL	model index	-	1	
2	VTO	zero-bias threshold voltage	V	0.0	1.0
3	KP	transconductance parameter	A/V^2	2.0E-5	3.1E-5
4	GAMMA	bulk threshold parameter	$V^{0.5}$	0.0	0.37
5	PHI	surface potential	V	0.6	0.65
6	LAMBDA	channel-length modulation (MOS1 and MOS2 only)	1/V	0.0	0.02
7	RD	drain ohmic resistance	Ohm	0.0	1.0
8	RS	source ohmic resistance	Ohm	0.0	1.0
9	CBD	zero-bias B-D junction capacitance	F	0.0	20FF
10	CBS	zero-bias B-S junction capacitance	F	0.0	20FF
11	IS	bulk junction saturation current	A	1.0E-14	1.0E-15
12	PB	bulk junction potential	V	0.8	0.87
13	CGSO	gate-source overlap capacitance per meter channel width	F/m	0.0	4.0E-11
14	CGDO	gate-drain overlap capacitance per meter channel width	F/m	0.0	4.0E-11
15	CGBO	gate-bulk overlap capacitance per meter channel length	F/m	0.0	2.0E-10
16	RSH	drain and source diffusion sheet resisitance	Ohm/sq.	0.0	10.0

[1] A. Vladimirescu and S. Liu, "The Simulation of MOS Integrated Circuits Using SPICE2", ERL Memo No. ERL M80/7,Electronics Research Laboratory, University of California, Berkeley, Oct. 1980.

17	CJ	zero-bias bulk junction bottom cap. per sq-meter of junction area	F/m^2	0.0	2.0E-4
18	MJ	bulk junction bottom grading coef.	-	0.5	0.5
19	CJSW	zero-bias bulk junction sidewall cap. per meter of junction perimeter	F/m	0.0	1.0E-9
20	MJSW	bulk junction sidewall grading coef.	-	0.33	
21	JS	bulk junction saturation current per sq-meter of junction area	A/m^2	1.0E-8	
22	TOX	oxide thickness	meter	1.0E-7	1.0E-7
23	NSUB	substrate doping	$1/cm^3$	0.0	4.0E15
24	NSS	surface state density	$1/cm^2$	0.0	1.0E10
25	NFS	fast surface state density	$1/cm^2$	0.0	1.0E10
26	TPG	type of gate material: +1 opp. to substrate -1 same as substrate 0 Al gate	-	1.0	
27	XJ	metallurgical junction depth	meter	0.0	1U
28	LD	lateral diffusion	meter	0.0	0.8U
29	UO	surface mobility	$cm^2/V-s$	600	700
30	UCRIT	critical field for mobility degradation (MOS2 only)	V/cm	1.0E4	1.0E4
31	UEXP	critical field exponent in mobility degradation (MOS2 only)	-	0.0	0.1
32	UTRA	transverse field coeff (mobility) (deleted for MOS2)	-	0.0	0.3
33	VMAX	maximum drift velocity of carriers	m/s	0.0	5.0E4
34	NEFF	total channel charge (fixed and mobile) coefficient (MOS2 only)	-	1.0	5.0
35	XQC	thin-oxide capacitance model flag and coefficient of channel charge share attributed to drain (0-0.5)	-	1.0	0.4
36	KF	flicker noise coefficient	-	0.0	1.0E-26
37	AF	flicker noise exponent	-	1.0	1.2
38	FC	coefficient for forward-bias depletion capacitance formula	-	0.5	
39	DELTA	width effect on threshold voltage (MOS2 and MOS3)	-	0.0	1.0
40	THETA	mobility modulation (MOS3 only)	1/V	0.0	0.1
41	ETA	static feedback (MOS3 only)	-	0.0	1.0
42	KAPPA	saturation field factor (MOS3 only)	-	0.2	0.5

8. SUBCIRCUITS

A subcircuit that consists of SPICE elements can be defined and referenced in a fashion similar to device models. The sub- circuit is defined in the input deck by a grouping of element cards; the program then automatically inserts the group of elements wherever the subcircuit is referenced. There is no limit on the size or complexity of subcircuits, and subcircuits may contain other subcircuits. An example of subcircuit usage is given in Appendix A.

8.1. .SUBCKT Card

General form:

.SUBCKT subnam N1 < N2 N3 ...>

Examples:

.SUBCKT OPAMP 1 2 3 4

A circuit definition is begun with a .SUBCKT card. SUBNAM is the subcircuit name, and N1, N2, ... are the external nodes, which cannot be zero. The group of element cards which immediately follow the .SUBCKT card define the subcircuit. The last card in a subcircuit definition is the .ENDS card (see below). Control cards may not appear within a vsubcircuit definition; however, subcircuit definitions may contain anything else, including other subcircuit definitions, device models, and subcircuit calls (see below). Note that any device models or subcircuit definitions included as part of a subcircuit definition are strictly local (i.e., such models and definitions are not known outside the subcircuit definition). Also, any element nodes not included on the .SUBCKT card are strictly local, with the exception of 0 (ground) which is always global.

8.2. .ENDS Card

General form:

.ENDS < SUBNAM>

Examples:

.ENDS OPAMP

This card must be the last one for any subcircuit definition. The subcircuit name, if included, indicates which subcircuit definition is being terminated; if omitted, all subcircuits being defined are terminated. The name is needed only when nested subcircuit definitions are being made.

8.3. Subcircuit Calls

General form:

XYYYYYYY N1 < N2 N3 ...> SUBNAM

Examples:

X1 2 4 17 3 1 MULTI

Subcircuits are used in SPICE by specifying pseudo-elements beginning with the letter X, followed by the circuit nodes to be used in expanding the subcircuit.

9. CONTROL CARDS

9.1. .TEMP Card

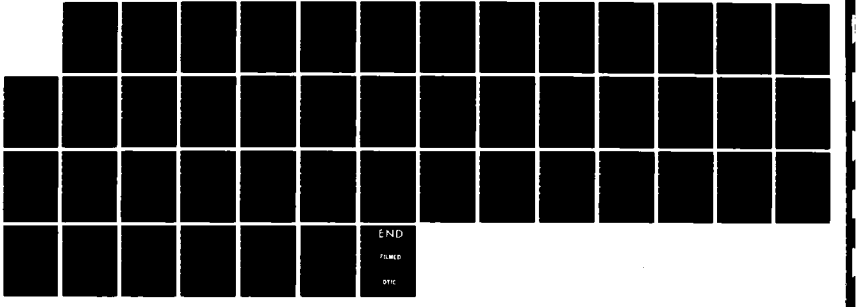
General form:

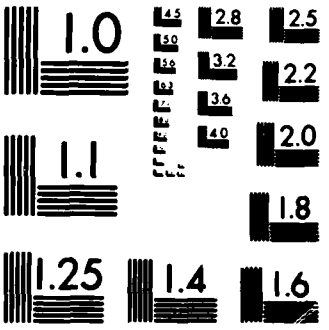
.TEMP T1 < T2 < T3 ...> >

Examples:

.TEMP -55.0 25.0 125.0

AD-A158 699 VLSI (VERY LARGE SCALE INTEGRATION) DESIGN TOOLS 5/5
REFERENCE MANUAL RELEASE 30(U) WASHINGTON UNIV SEATTLE
DEPT OF COMPUTER SCIENCE AUG 85 TR-85-07-03
UNCLASSIFIED MDA903-85-K-0072 F/G 9/5 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

This card specifies the temperatures at which the circuit is to be simulated. T1, T2, ... Are the different temperatures, in degrees C. Temperatures less than -223.0 deg C are ignored. Model data are specified at TNOM degrees (see the .OPTIONS card for TNOM); if the .TEMP card is omitted, the simulation will also be performed at a temperature equal to TNOM.

9.2. .WIDTH Card

General form:

.WIDTH IN=COLNUM OUT=COLNUM

Examples:

.WIDTH IN=72 OUT=133

COLNUM is the last column read from each line of input; the setting takes effect with the next line read. The default value for COLNUM is 80. The out parameter specifies the output print width. Permissible values for the output print width are 80 and 133.

9.3. .OPTIONS Card

General form:

.OPTIONS OPT1 OPT2 ... (or OPT=OPTVAL ...)

Examples:

.OPTIONS ACCT LIST NODE

This card allows the user to reset program control and user options for specific simulation purposes. Any combination of the following options may be included, in any order. 'x' (below) represents some positive number.

option	effect
ACCT	causes accounting and run time statistics to be printed
LIST	causes the summary listing of the input data to be printed
NOMOD	suppresses the printout of the model parameters.
NOPAGE	suppresses page ejects
NODE	causes the printing of the node table.
OPTS	causes the option values to be printed.
GMIN=x	sets the value of GMIN, the minimum conductance allowed by the program. The default value is 1.0E-12.
RELTOL=x	resets the relative error tolerance of the program. The default value is 0.001 (0.1 percent).
ABSTOL=x	resets the absolute current error tolerance of the program. The default value is 1 picoamp.
VNTOL=x	resets the absolute voltage error tolerance of the program. The default value is 1 microvolt.
TRTOL=x	resets the transient error tolerance. The default value is 7.0. This parameter is an estimate of the factor by which SPICE overestimates the actual

truncation error. IP "CHGTOL=x" 17 resets the charge tolerance of the program. The default value is 1.0E-14.

PIVTOL=x resets the absolute minimum value for a matrix entry to be accepted as a pivot. The default value is 1.0E-13.

PIVREL=x resets the relative ratio between the largest column entry and an acceptable pivot value. The default value is 1.0E-3. In the numerical pivoting algorithm the allowed minimum pivot value is determined by EPSREL = AMAX1(PIVREL·MAXVAL,PIVTOL) where MAXVAL is the maximum element in the column where a pivot is sought (partial pivoting).

NUMDGT=x is the number of significant digits printed for output variable values. X must satisfy the relation $0 < x < 8$. The default value is 4. Note: this option is independent of the error tolerance used by SPICE (i.e., if the values of options RELTOL, ABSTOL, etc., are not changed then one may be printing numerical 'noise' for NUMDGT > 4.

TNOM=x resets the nominal temperature. The default value is 27 deg C (300 deg K).

ITL1=x resets the dc iteration limit. The default is 100.

ITL2=x resets the dc transfer curve iteration limit. The default is 50.

ITL3=x resets the lower transient analysis iteration limit. The default value is 4.

ITL4=x resets the transient analysis timepoint iteration limit. The default is 10.

ITL5=x resets the transient analysis total iteration limit. The default is 5000. Set ITL5=0 to omit this test.

ITL6=x resets the dc iteration limit at each step of the source stepping method. The default is 0 which means not to use this method.

CPTIME=x is the maximum cpu-time in seconds allowed for this job.

LIMTIM=x resets the amount of cpu time reserved by SPICE for generating plots should a cpu time-limit cause job termination. The default value is 2 (seconds).

LIMPTS=x resets the total number of points that can be printed or plotted in a dc, ac, or transient analysis. The default value is 201.

LVLCOD=x if x is 2 (two), then machine code for the matrix solution will be generated. Otherwise, no machine code is generated. The default value is 2. Applies only to CDC computers.

LVLTIM=x is 1 (one), the iteration timestep control is used. if x is 2 (two), the truncation-error timestep is used. The default value is 2. If method=Gear and MAXORD> 2 then LVLTIM is set to 2 by SPICE.

METHOD=name sets the numerical integration method used by SPICE. Possible names are Gear or trapezoidal. The default is trapezoidal.

MAXORD=x sets the maximum order for the integration method if Gear's variable-order method is used. X must be between 2 and 6. The default value is 2.

DEFL=x resets the value for MOS channel length; the default is 100.0 micrometer.

DEFW=x resets the value for MOS channel width; the default is 100.0 micrometer.

DEFAD=x resets the value for MOS drain diffusion area; the default is 0.0.

DEFAS=x resets the value for MOS source diffusion area; the default is 0.0.

9.4. .OP Card

General form:

```
.OP
```

The inclusion of this card in an input deck will force SPICE to determine the dc operating point of the circuit with inductors shorted and capacitors opened. Note: a dc analysis is automatically performed prior to a transient analysis to determine the transient initial conditions, and prior to an ac small-signal analysis to determine the linearized, small-signal models for nonlinear devices.

SPICE performs a dc operating point analysis if no other analyses are requested.

9.5. .DC Card

General form:

```
.DC SRCNAM VSTART VSTOP VINCR [SRC2 START2 STOP2 INCR2]
```

Examples:

```
.DC VIN 0.25 5.0 0.25
.DC VDS 0 10 .5 VGS 0 5 1
.DC VCE 0 10 .25 IB 0 10U 1U
```

This card defines the dc transfer curve source and sweep limits. SRCNAM is the name of an independent voltage or current source. VSTART, VSTOP, and VINCR are the starting, final, and incrementing values respectively. The first example will cause the value of the voltage source VIN to be swept from 0.25 Volts to 5.0 Volts in increments of 0.25 Volts. A second source (SRC2) may optionally be specified with associated sweep parameters. In this case, the first source will be swept over its range for each value of the second source. This option can be useful for obtaining semiconductor device output characteristics. See the second example data deck in that section of the guide.

9.6. .NODESET Card

General form:

```
.NODESET V(NODNUM)=VAL V(NODNUM)=VAL ...
```

Examples:

```
.NODESET V(12)=4.5 V(4)=2.23
```

This card helps the program find the dc or initial transient solution by making a preliminary pass with the specified nodes held to the given voltages. The restriction is then released and the iteration continues to the true solution. The .NODESET card may be necessary for convergence on bistable or astable circuits. In general, this card should not be necessary.

9.7. .IC Card

General form:

.IC V(NODNUM)=VAL V(NODNUM)=VAL ...

Examples:

.IC V(11)=5 V(4)=-5 V(2)=2.2

This card is for setting transient initial conditions. It has two different interpretations, depending on whether the UIC parameter is specified on the .TRAN card. Also, one should not confuse this card with the .NODESET card. The .NODESET card is only to help dc convergence, and does not affect final bias solution (except for multi-stable circuits). The two interpretations of this card are as follows:

1. When the UIC parameter is specified on the .TRAN card, then the node voltages specified on the .IC card are used to compute the capacitor, diode, BJT, JFET, and MOSFET initial conditions. This is equivalent to specifying the IC=... parameter on each device card, but is much more convenient. The IC=... parameter can still be specified and will take precedence over the .IC values. Since no dc bias (initial transient) solution is computed before the transient analysis, one should take care to specify all dc source voltages on the .IC card if they are to be used to compute device initial conditions.
2. When the UIC parameter is not specified on the .TRAN card, the dc bias (initial transient) solution will be computed before the transient analysis. In this case, the node voltages specified on the .IC card will be forced to the desired initial values during the bias solution. During transient analysis, the constraint on these node voltages is removed.

9.8. .TF Card

General form:

.TF OUTVAR INSRC

Examples:

.TF V(5,3) VIN
.TF I(VLOAD) VIN

This card defines the small signal output and input for the dc small signal analysis. OUTVAR is the small-signal output variable and INSRC is the small-signal input source. If this card is included, SPICE will compute the dc small signal value of the transfer function (output/input), input resistance, and output resistance. For the first example, SPICE would compute the ratio of V(5,3) to VIN, the small-signal input resistance at VIN, and the small-signal output resistance measured across nodes 5 and 3.

9.9. .SENS Card

General form:

.SENS OV1 < OV2 ... >

Examples:

.SENS V(9) V(4,3) V(17) I(VCC)

If a .SENS card is included in the input deck, SPICE will determine the dc small-signal sensitivities of each specified output variable with respect to every circuit parameter. Note: for large circuits, large amounts of output can be generated.

9.10. .AC Card

General form:

.AC DEC ND FSTART FSTOP
.AC OCT NO FSTART FSTOP
.AC LIN NP FSTART FSTOP

Examples:

.AC DEC 10 1 10K
.AC DEC 10 1K 100MEG
.AC LIN 100 1 100HZ

DEC stands for decade variation, and ND is the number of points per decade. OCT stands for octave variation, and NO is the number of points per octave. LIN stands for linear variation, and NP is the number of points. FSTART is the starting frequency, and FSTOP is the final frequency. If this card is included in the deck, SPICE will perform an ac analysis of the circuit over the specified frequency range. Note that in order for this analysis to be meaningful, at least one independent source must have been specified with an ac value.

9.11. .DISTO Card

General form:

.DISTO RLOAD < INTER < SKW2 < REFPWR < SPW2> > > >

Examples:

.DISTO RL 2 0.95 1.0E-3 0.75

This card controls whether SPICE will compute the distortion characteristic of the circuit in a small-signal mode as a part of the ac small-signal sinusoidal steady-state analysis. The analysis is performed assuming that one or two signal frequencies are imposed at the input; let the two frequencies be f1 (the nominal analysis frequency) and f2 (=SKW2·f1). The program then computes the following distortion measures:

HD2 - the magnitude of the frequency component 2·f1 assuming that f2 is not present.

HD3 - the magnitude of the frequency component 3·f1 assuming that f2 is not present.

SIM2 - the magnitude of the frequency component f1 + f2.

DIM2 - the magnitude of the frequency component f1 - f2.

DIM3 - the magnitude of the frequency component 2·f1 - f2.

RLOAD is the name of the output load resistor into which all distortion power products are to be computed. INTER is the interval at which the summary printout of the

contributions of all nonlinear devices to the total distortion is to be printed. If omitted or set to zero, no summary printout will be made. REFPWR is the reference power level used in computing the distortion products; if omitted, a value of 1 mW (that is, dbm) is used. SKW2 is the ratio of f2 to f1. If omitted, a value of 0.9 is used (i.e., f2 = 0.9·f 1). SPW2 is the amplitude of f2. If omitted, a value of 1.0 is assumed. he distortion measures HD2, HD3, SIM2, DIM2, and DIM3 may also be be printed and/or plotted (see the description of the .PRINT and .PLOT cards).

9.12. .NOISE Card

General form:

```
.NOISE OUTV INSRC NUMS
```

Examples:

```
.NOISE V(5) VIN 10
```

This card controls the noise analysis of the circuit. The noise analysis is performed in conjunction with the ac analysis (see .AC card). OUTV is an output voltage which defines the summing point. INSRC is the name of the independent voltage or current source which is the noise input reference. NUMS is the summary interval. SPICE will compute the equivalent output noise at the specified output as well as the equivalent input noise at the specified input. In addition, the contributions of every noise generator in the circuit will be printed at every NUMS frequency points (the summary interval). If NUMS is zero, no summary printout will be made.

The output noise and the equivalent input noise may also be printed and/or plotted (see the description of the .PRINT and .PLOT cards).

9.13. .TRAN Card

General form:

```
.TRAN TSTEP TSTOP <TSTART <TMAX>> <UIC>
```

Examples:

```
.TRAN 1NS 100NS
.TRAN 1NS 1000NS 500NS
.TRAN 10NS 1US UIC
```

TSTEP is the printing or plotting increment for line-printer output. For use with the post-processor, TSTEP is the suggested computing increment. TSTOP is the final time, and TSTART is the initial time. If TSTART is omitted, it is assumed to be zero. The transient analysis always begins at time zero. In the interval <zero, TSTART>, the circuit is analyzed (to reach a steady state), but no outputs are stored. In the interval <TSTART, TSTOP>, the circuit is analyzed and outputs are stored. TMAX is the maximum stepsize that SPICE will use (for default, the program chooses either TSTEP or (TSTOP-TSTART)/50.0, whichever is smaller. TMAX is useful when one wishes to guarantee a computing interval which is smaller than the printer increment, TSTEP.

UIC (use initial conditions) is an optional keyword which indicates that the user does not want SPICE to solve for the quiescent operating point before beginning the transient analysis. If this keyword is specified, SPICE uses the values specified using IC=...

on the various elements as the initial transient condition and proceeds with the analysis. If the .IC card has been specified, then the node voltages on the .IC card are used to compute the intitial conditions for the devices. Look at the description on the .IC card for its interpretation when UIC is not specified.

9.14. .FOUR Card

General form:

```
.FOUR FREQ OV1 <OV2 OV3 ...>
```

Examples:

```
.FOUR 100K  V(5)
```

This card controls whether SPICE performs a Fourier analysis as a part of the transient analysis. FREQ is the fundamental frequency, and OV1, ..., are the output variables for which the analysis is desired. The Fourier analysis is performed over the interval <TSTOP-period, TSTOP>, where TSTOP is the final time specified for the transient analysis, and period is one period of the fundamental frequency. The dc component and the first nine components are determined. For maximum accuracy, TMAX (see the .TRAN card) should be set to period/100.0 (or less for very high-Q circuits).

9.15. .PRINT Cards

General form:

```
.PRINT PRTYPE OV1 <OV2 ... OV8>
```

Examples:

```
.PRINT TRAN V(4) I(VIN)
.PRINT AC VM(4,2) VR(7) VP(8,3)
.PRINT DC V(2) I(VSRC) V(23,17)
.PRINT NOISE INOISE
.PRINT DISTO HD3 SIM2(DB)
```

This card defines the contents of a tabular listing of one to eight output variables. PRTYPE is the type of the analysis (DC, AC, TRAN, NOISE, or DISTO) for which the specified outputs are desired. The form for voltage or current output variables is as follows:

V(N1<,N2>) specifies the voltage difference between nodes N1 and N2. If N2 (and the preceding comma) is omitted, ground (0) is assumed. For the ac analysis, five additional outputs can be accessed by replacing the letter V by:

VR - real part
VI - imaginary part
VM - magnitude
VP - phase
VDB - 20·log10(magnitude)

I(VXXXXXXX) specifies the current flowing in the independent voltage source named VXXXXXXX. Positive current flows from the positive node, through the source, to the negative node. For the ac analysis, the corresponding replacements for the letter I may be made in the same way as described

for voltage outputs.

Output variables for the noise and distortion analyses have a different general form from that of the other analyses, i.e.

OV<(X)>

where OV is any of ONOISE (output noise), INOISE (equivalent input noise), D2, HD3, SIM2, DIM2, or DIM3 (see description of distortion analysis), and X may be any of:

R - real part
I - imaginary part
M - magnitude (default if nothing specified)
P - phase
DB - 20·log10(magnitude)

thus, SIM2 (or SIM2(M)) describes the magnitude of the SIM2 distortion measure, while HD2(R) describes the real part of the HD2 distortion measure.

There is no limit on the number of .PRINT cards for each type of analysis.

9.16. .PLOT Cards

General form:

.PLOT PLTYPE OV1 <(PLO1,PHI1)> <OV2 <(PLO2,PHI2)> ... OV8>

Examples:

```
.PLOT DC V(4) V(5) V(1)
.PLOT TRAN V(17,5) (2,5) I(VIN) V(17) (1,9)
.PLOT AC VM(5) VM(31,24) VDB(5) VP(5)
.PLOT DISTO HD2 HD3(R) SIM2
.PLOT TRAN V(5,3) V(4) (0,5) V(7) (0,10)
```

This card defines the contents of one plot of from one to eight output variables. PLTYPE is the type of analysis (DC, AC, TRAN, NOISE, or DISTO) for which the specified outputs are desired. The syntax for the OVI is identical to that for the .PRINT card, described above.

The optional plot limits (PLO,PHI) may be specified after any of the output variables. All output variables to the left of a pair of plot limits (PLO,PHI) will be plotted using the same lower and upper plot bounds. If plot limits are not specified, SPICE will automatically determine the minimum and maximum values of all output variables being plotted and scale the plot to fit. More than one scale will be used if the output variable values warrant (i.e., mixing output variables with values which are orders-of-magnitude different still gives readable plots).

The overlap of two or more traces on any plot is indicated by the letter X.

When more than one output variable appears on the same plot, the first variable specified will be printed as well as plotted. If a printout of all variables is desired, then a companion .PRINT card should be included.

There is no limit on the number of .PLOT cards specified for each type of analysis.

10. APPENDIX A: EXAMPLE DATA DECKS

10.1. Circuit 1

The following deck determines the dc operating point and small-signal transfer function of a simple differential pair. In addition, the ac small-signal response is computed over the frequency range 1Hz to 100MEGHz.

```
SIMPLE DIFFERENTIAL PAIR
VCC 7 0 12
VEE 8 0 -12
VIN 1 0 AC 1
RS1 1 2 1K
RS2 6 0 1K
Q1 3 2 4 MOD1
Q2 5 6 4 MOD1
RC1 7 3 10K
RC2 7 5 10K
RE 4 8 10K
.MODEL MOD1 NPN BF=50 VAF=50 IS=1.E-12 RB=100 CJC=.5PF TF=.6NS
.TF V(5) VIN
.AC DEC 10 1 100MEG
.PLOT AC VM(5) VP(5)
.PRINT AC VM(5) VP(5)
.END
```

10.2. Circuit 2

The following deck computes the output characteristics of a MOS- FET device over the range 0-10V for VDS and 0-5V for VGS.

```
MOS OUTPUT CHARACTERISTICS
.OPTIONS NODE NOPAGE
VDS 3 0
VGS 2 0
M1 1 2 0 0 MOD1 L=4U W=6U AD=10P AS=10P
.MODEL MOD1 NMOS VTO=-2 NSUB=1.0E15 UO=550
* VIDS MEASURES ID, WE COULD HAVE USED VDS, BUT ID WOULD BE NEGATIVE
VIDS 3 1
.DC VDS 0 10 .5 VGS 0 5 1
.PRINT DC I(VIDS) V(2)
.PLOT DC I(VIDS)
.END
```

10.3. Circuit 3

The following deck determines the dc transfer curve and the transient pulse response of a simple RTL inverter. The input is a pulse from 0 to 5 Volts with delay, rise, and fall times of 2ns and a pulse width of 30ns. The transient interval is 0 to 100ns, with printing to be done every nanosecond.

```
SIMPLE RTL INVERTER
VCC 4 0 5
VIN 1 0 PULSE 0 5 2NS 2NS 2NS 30NS
RB 1 2 10K
Q1 3 2 0 Q1
RC 3 4 1K
.PLOT DC V(3)
.PLOT TRAN V(3) (0,5)
.PRINT TRAN V(3)
.MODEL Q1 NPN BF 20 RB 100 TF .1NS CJC 2PF
.DC VIN 0 5 0.1
.TRAN 1NS 100NS
.END
```

10.4. Circuit 4

The following deck simulates a four-bit binary adder, using several subcircuits to describe various pieces of the overall circuit.

```
ADDER - 4 BIT ALL-NAND-GATE BINARY ADDER

   *** SUBCIRCUIT DEFINITIONS

.SUBCKT NAND 1 2 3 4
   * NODES: INPUT(2), OUTPUT, VCC
Q1 9 5 1 QMOD
D1CLAMP 0 1 DMOD
Q2 9 5 2 QMOD
D2CLAMP 0 2 DMOD
RB 4 5 4K
R1 4 6 1.6K
Q3 6 9 8 QMOD
R2 8 0 1K
RC 4 7 130
Q4 7 6 10 QMOD
DVBEDROP 10 3 DMOD
Q5 3 8 0 QMOD
.ENDS NAND

.SUBCKT ONEBIT 1 2 3 4 5 6
   * NODES: INPUT(2), CARRY-IN, OUTPUT, CARRY-OUT, VCC
X1 1 2 7 6 NAND
X2 1 7 8 6 NAND
X3 2 7 9 6 NAND
X4 8 9 10 6 NAND
X5 3 10 11 6 NAND
X6 3 11 12 6 NAND
X7 10 11 13 6 NAND
X8 12 13 4 6 NAND
X9 11 7 5 6 NAND
.ENDS ONEBIT
```

```
.SUBCKT TWOBIT 1 2 3 4 5 6 7 8 9
   * NODES:  INPUT - BIT0(2) / BIT1(2), OUTPUT - BIT0 / BIT1,
   *         CARRY-IN, CARRY-OUT, VCC
X1 1 2 7 5 10 9 ONEBIT
X2 3 4 10 6 8 9 ONEBIT
.ENDS TWOBIT
.SUBCKT FOURBIT 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
 * NODES: INPUT - BIT0(2) / BIT1(2) / BIT2(2) / BIT3(2),
 *       OUTPUT - BIT0/BIT1/BIT2/BIT3,CARRY-IN,CARRY-OUT,VCC
X1 1 2 3 4 9 10 13 16 15 TWOBIT
X2 5 6 7 8 11 12 16 14 15 TWOBIT
.ENDS FOURBIT

   *** DEFINE NOMINAL CIRCUIT

.MODEL DMOD D
.MODEL QMOD NPN(BF=75 RB=100 CJE=1PF CJC=3PF)
VCC 99 0 DC 5V
VIN1A 1 0 PULSE(0 3 0 10NS 10NS   10NS   50NS)
VIN1B 2 0 PULSE(0 3 0 10NS 10NS   20NS  100NS)
VIN2A 3 0 PULSE(0 3 0 10NS 10NS   40NS  200NS)
VIN2B 4 0 PULSE(0 3 0 10NS 10NS   80NS  400NS)
VIN3A 5 0 PULSE(0 3 0 10NS 10NS  160NS  800NS)
VIN3B 6 0 PULSE(0 3 0 10NS 10NS  320NS 1600NS)
VIN4A 7 0 PULSE(0 3 0 10NS 10NS  640NS 3200NS)
VIN4B 8 0 PULSE(0 3 0 10NS 10NS 1280NS 6400NS)
X1 1 2 3 4 5 6 7 8 9 10 11 12 0 13 99 FOURBIT
RBIT0 9 0 1K
RBIT1 10 0 1K
RBIT2 11 0 1K
RBIT3 12 0 1K
RCOUT 13 0 1K
.PLOT TRAN V(1) V(2) V(3) V(4) V(5) V(6) V(7) V(8)
.PLOT TRAN V(9) V(10) V(11) V(12) V(13)
.PRINT TRAN V(1) V(2) V(3) V(4) V(5) V(6) V(7) V(8)
.PRINT TRAN V(9) V(10) V(11) V(12) V(13)

   *** (FOR THOSE WITH MONEY (AND MEMORY) TO BURN)
.TRAN 1NS 6400NS

.OPTIONS ACCT LIST NODE LIMPTS=6401
.END
```

10.5. Circuit 5

The following deck simulates a transmission-line inverter. Two transmission-line elements are required since two propagation modes are excited. In the case of a coaxial line, the first line (T1) models the inner conductor with respect to the shield, and the second line (T2) models the shield with respect to the out- side world.

```
TRANSMISSION-LINE INVERTER
V1 1 0 PULSE(0 1 0 0.1N)
R1 1 2 50
X1 2 0 0 4 TLINE
R2 4 0 50
.SUBCKT TLINE 1 2 3 4
T1 1 2 3 4 Z0=50 TD=1.5NS
T2 2 0 4 0 Z0=100 TD=1NS
.ENDS TLINE
.TRAN 0.1NS 20NS
.PLOT TRAN V(2) V(4)
.END
```

11. APPENDIX B: NONLINEAR DEPENDENT SOURCES

SPICE allows circuits to contain dependent sources charac terized by any of the four equations

$$i=f(v) \qquad v=f(v) \qquad i=f(i) \qquad v=f(i)$$

where the functions must be polynomials, and the arguments may be multidimensional. The polynomial functions are specified by a set of coefficients p0, p1, ..., pn. Both the number of dimensions and the number of coefficients are arbitrary. The meaning of the coefficients depends upon the dimension of the polynomial, as shown in the following examples:

Suppose that the function is one-dimensional (that is, a function of one argument). Then the function value fv is determined by the following expression in a (the function argument):

$$fv = p0 + p1 \cdot a + p2 \cdot a^2 + p3 \cdot a^3 + p4 \cdot a^4 + p5 \cdot a^5 + \ldots$$

Suppose now that the function is two-dimensional, with arguments a and b. Then the function value fv is determined by the following expression:

$$fv = p0 + p1 \cdot a + p2 \cdot b + p3 \cdot a^2 + p4 \cdot a \cdot b + p5 \cdot b^2 + p6 \cdot a^3 + p7 \cdot a^2 b + p8 \cdot a \cdot b^2 + p9 \cdot b^3 + \ldots$$

Consider now the case of a three-dimensional polynomial function with arguments a, b, and c. Then the function value fv is determined by the following expression:

$$fv = p0 + p1 \cdot a + p2 \cdot b + p3 \cdot c + p4 \cdot a^2 + p5 \cdot a \cdot b + p6 \cdot a \cdot c + p7 \cdot b^2 + p8 \cdot b \cdot c + p9 \cdot c^2 + p10 \cdot a^3 + p11 \cdot a^2 \cdot b + p12 \cdot a^2 \cdot c + p13 \cdot a \cdot b^2 + p14 \cdot a \cdot b \cdot c + p15 \cdot a \cdot c^2 + p16 \cdot b^3 + p17 \cdot b^2 \cdot c + p19 \cdot c^3 + p20 \cdot a^4 + \ldots$$

Note: if the polynomial is one-dimensional and exactly one coefficient is specified, then SPICE assumes it to be p1 (and p0 = 0.0), in order to facilitate the input of linear controlled sources.

For all four of the dependent sources described below, the initial condition parameter is described as optional. If not specified, SPICE assumes 0 the initial condition for dependent sources is an initial initial condition to obtain the dc operating point of the circuit. After convergence has been obtained, the program continues iterating to obtain the exact value for the controlling variable. Hence, to reduce the computational effort for the dc operating point (or if the polynomial specifies a strong nonlinearity), a value fairly close to

the actual controlling variable should be specified for the initial condition.

11.1. Voltage-Controlled Current Sources

General form:

GXXXXXXX N+ N- <POLY(ND)> NC1+ NC1- ... P0 <P1 ...> <IC=...>

Examples:

```
G1 1 0 5 3 0 0.1M
GR 17 3 17 3 0 1M 1.5M IC=2V
GMLT 23 17 POLY(2) 3 5 1 2 0 1M 17M 3.5U IC=2.5, 1.3
```

N+ and N- are the positive and negative nodes, respectively. Current flow is from the positive node, through the source, to the negative node. POLY(ND) only has to be specified if the source is multi-dimensional (one-dimensional is the default). If specified, ND is the number of dimensions, which must be positive. NC1+, NC1-, ... Are the positive and negative controlling nodes, respectively. One pair of nodes must be specified for each dimension. P0, P1, P2, ..., Pn are the polynomial coefficients. The (optional) initial condition is the initial guess at the value(s) of the controlling voltage(s). If not specified, 0.0 is assumed. The polynomial specifies the source current as a function of the controlling voltage(s). The second example above describes a current source with value

$$I = 10^{-3} \cdot V(27,3) + 1.5 \times 10^{-3} \cdot V(17,3)^2$$

note that since the source nodes are the same as the controlling nodes, this source actually models a nonlinear resistor.

11.2. Voltage-Controlled Voltage Sources

General form:

EXXXXXXX N+ N- <POLY(ND)> NC1+ NC1- ... P0 <P1 ...> <IC=...>

Examples:

```
E1 3 4 21 17 10.5 2.1 1.75
EX 17 0 POLY(3) 13 0 15 0 17 0 0 1 1 1 IC=1.5,2.0,17.35
```

N+ and N- are the positive and negative nodes, respectively. POLY(ND) only has to be specified if the source is multi- dimensional (one-dimensional is the default). If specified, ND is the number of dimensions, which must be positive. NC1+, NC1-, ... are the positive and negative controlling nodes, respectively. One pair of nodes must be specified for each dimension. P0, P1, P2, ..., Pn are the polynomial coefficients. The (optional) initial condition is the initial guess at the value(s) of the controlling voltage(s). If not specified, 0.0 is assumed. The polynomial specifies the source voltage as a function of the controlling voltage(s). The second example above describes a voltage source with value

V = V(13,0) + V(15,0) + V(17,0)

(in other words, an ideal voltage summer).

11.3. Current-Controlled Current Sources

General form:

```
FXXXXXXX N+ N- <POLY(ND)> VN1 <VN2 ...> P0 <P1 ...> <IC=...>
```

Examples:

```
F1 12 10 VCC 1MA 1.3M
FXFER 13 20 VSENS 0 1
```

N+ and N- are the positive and negative nodes, respectively. Current flow is from the positive node, through the source, to the negative node. POLY(ND) only has to be specified if the source is multi-dimensional (one-dimensional is the default). If specified, ND is the number of dimensions, which must be positive. VN1, VN2, ... are the names of voltage sources through which the controlling current flows; one name must be specified for each dimension. The direction of positive controlling current flow is from the positive node, through the source, to the negative node of each voltage source. P0, P1, P2, ..., Pn are the polynomial coefficients. The (optional) initial condition is the initial guess at the value(s) of the controlling current(s) (in Amps). If not specified, 0.0 is assumed. The polynomial specifies the source current as a function of the controlling current(s). The first example above describes a current source with value

$$I = 10^{-3} + 1.3 \times 10^{-3} \cdot (VCC)$$

11.4. Current-Controlled Voltage Sources

General form:

```
HXXXXXXX N+ N- <POLY(ND)> VN1 <VN2 ...> P0 <P1 ...> <IC=...>
```

Examples:

```
HXY 13 20 POLY(2) VIN1 VIN2 0 0 0 0 1 IC=0.5 1.3
HR 4 17 VX 0 0 1
```

N+ and N- are the positive and negative nodes, respectively. POLY(ND) only has to be specified if the source is multi- dimensional (one-dimensional is the default). If specified, ND is the number of dimensions, which must be positive. VN1, VN2, ... are the names of voltage sources through which the controlling current flows; one name must be specified for each dimension. The direction of positive controlling current flow is from the positive node, through the source, to the negative node of each voltage source. P0, P1, P2, ..., Pn are the polynomial coefficients. The (optional) initial condition is the initial guess at the value(s) of the controlling current(s) (in Amps). If not specified, 0.0 is assumed. The polynomial specifies the source voltage as a function of the controlling current(s). The first example above describes a voltage source with value

$$V = I(VIN1) \cdot I(VIN2)$$

12. APPENDIX C: BIPOLAR MODEL EQUATIONS

Acknowledgment: This section has been contributed by Bill Bider- mann at HP labs.

(G_{min} terms omitted)

12.1 D.C. MODEL

$$IC = \frac{IS}{QB}\left(e^{(NF)\frac{q\,\overline{VBE}}{kT}} - e^{(NR)\frac{q\,\overline{VBC}}{kT}}\right) - \frac{IS}{BR}\left(e^{(NR)\frac{q\,\overline{VBC}}{kT}} - 1\right) - ISC\left(e^{(NC)\frac{q\,\overline{VBC}}{kT}} - 1\right)$$

$$IB = \frac{IS}{BF}\left(e^{(NF)\frac{q\,\overline{VBE}}{kT}} - 1\right) + \frac{IS}{BR}\left(e^{(NR)\frac{q\,\overline{VBC}}{kT}} - 1\right) + ISE\left(e^{(NE)\frac{q\,\overline{VBE}}{kT}} - 1\right) + ISC\left(e^{(NC)\frac{q\,\overline{VBC}}{kT}} - 1\right)$$

NOTE: The last two terms in the expression of the base current IB represent the components due to recombination in the BE and BC space charge regions at low injection.

If IRB not specified

$$RBB = RBM + \frac{RB - RBM}{QB}$$

If IRB specified

$$RBB = 3(RB - RBM)\left[\frac{\tan(z) - z}{z\tan^2(z)}\right] + RBM$$

Where:

$$z = \frac{-1 + \sqrt{(144\,IB/(\pi^2 IRB) + 1)}}{24/\pi^2 \sqrt{(IB/IRB)}}$$

$$QB = \frac{Q1}{2}\left(1 + \sqrt{1 + 4Q2}\right)$$

$$Q1 = \frac{}{1 - \frac{\overline{VBC}}{VAF} - \frac{\overline{VBE}}{VAR}}$$

$$Q2 = \frac{IS}{IKF}\left(e^{(NF)\frac{q\,\overline{VBE}}{kT}} - 1\right) + \frac{IS}{IKR}\left(e^{(NR)\frac{q\,\overline{VBC}}{kT}} - 1\right)$$

NOTE: IRB is the current where the base resistance falls halfway to its minimum value. VAF and VAR are forward and reverse Early voltages respectively. IKF and IKR determine the high current beta rolloff with IC. ISE, ISC, NE and NC determine the low current beta rolloff with IC.

12.2 A.C. MODEL

$$CBE = \frac{\partial}{\partial \overline{VBE}}\left[\left(TFF\frac{IS}{QB}\left(e^{(NF)\frac{q\,\overline{VBE}}{kT}} - 1\right) + CJE\left(1 - \frac{\overline{VBE}}{VJE}\right)^{-MJ}\right]\right.$$

Where:

$$TFF = TF\left(1 + XTF \cdot \left(\frac{IF}{IF + ITF}\right)^2 \cdot e^{1.44\frac{\overline{VBC}}{VTF}}\right.$$

$$IF = IS\left(e^{(NF)\frac{q\,\overline{VBE}}{kT}} - 1\right)$$

$$CBI = CBC \cdot (1 - XCJC)$$

$$CB2 = CBC \cdot XCJC$$

$$CBC = TR\left(\frac{IS}{(NR)\frac{kT}{q}} e^{\frac{q\,\overline{VBC}}{kT}}\right) + CJC\left(1 - \frac{\overline{VBC}}{VJC}\right)^{-MJC}$$

$$CSS = CJS\left(1 - \frac{VCS}{VJS}\right)^{-MJS}$$

NOTE: all junction capacitances of the form $CO \cdot \left(1 - \frac{V}{\phi}\right)^{-M}$ revert to the form

$$\frac{CO}{(1 - FC)^M}\left(1 + \frac{M(V - FC \cdot \phi)}{\phi(1 - FC)}\right)$$

when V > FC·phi (For CSS assumes FC = 0)

12.3 NOISE MODEL

Thermal Noise :

$$\overline{IRB^2} = \frac{4kT}{R_B}\Delta f$$

$$\overline{IRC^2} = \frac{4kT}{R_C}\Delta f$$

$$\overline{IRE^2} = \frac{4kT}{2R_E}\Delta f + \frac{KF \cdot IB^{AF}}{f}\Delta f$$

Note: The first two terms are shot noise and the last term is flicker noise.

$$\overline{ICN^2} = 2qIC\Delta f$$

Note: This is shot noise.

12.4 TEMPERATURE EFFECTS

All junctions have dependences identical to the diode model but all N factors are considered equal 1.

BF and BR go as $(\frac{T}{TNOM})^{XTB}$ when NF=1. This is done through appropriate changes in BF , BR and ISE, ISC according to the following equations respectively:

$$\overline{BF} = BF \cdot (\frac{T}{TNOM})^{XTB}$$

$$\overline{BR} = BR \cdot (\frac{T}{TNOM})^{XTB}$$

$$\overline{ISE} = ISE \cdot (\frac{T}{TNOM})^{(XTI-XTB)} e^{\frac{qEG}{Nk} \cdot \frac{T-TNOM}{T \cdot TNOM}}$$

$$\overline{ISC} = ISC \cdot (\frac{T}{TNOM})^{(XTI-XTB)} e^{\frac{qEG}{Nk} \cdot \frac{T-TNOM}{T \cdot TNOM}}$$

12.5 EXCESS PHASE

This is a delay (linear phase) in the gm generator in AC analysis. It is also used in transient analysis using a Bessel polynomial approximation. Excess phase, PTF, is specified as the number of extra degrees of phase at the frequency

$$f = \frac{1}{2\pi \cdot TF} \text{Hertz}$$

12. APPENDIX D: ALTER STATEMENT AND THE SOURCE-STEPPING METHOD

The ALTER statement allows SPICE to run with altered circuit parameters.

General form:

```
.ALTER
ELEMENT CARDS (DEVICE CARDS, MODEL CARDS)
.ALTER (or .END CARD)
```

Examples:

```
R1 1 0 5K
VCC 3 0 10
M1 3 2 0 MOD1 L=5U W=2U
.MODEL MOD1 NMOS(VTO=1.0 KP=2.0E-5 PHI=0.6 NSUB=2.0E15 TOX=0.1U)
.ALTER
R1 1 0 3.5K
VCC 3 0 12
M1 3 2 0 MOD1 L=10U W=2U
```

```
.MODEL MOD1 NMOS(VTO=1.2 KP=2.0E-5 PHI=0.6 NSUB=5.0E15 TOX=1.5U)
.ALTER
M1 3 2 0 MOD1 L=10U W=4U
.END
```

This card introduces the element(s), device(s) and model(s) whose parameters are changed during the execution of the input deck. The vanalyses specified in the deck will start over again with the changed parameters. The .ALTER card with the cards defining the new parameters should be placed just before the .END card. The syntax for the element (device, model) cards is identical to that of the cards with the original parameters.

There is no limit on the number of .ALTER cards and the circuit will be re-analyzed as many times as the number of .ALTER cards. Subsequent ALTER operations employ parameters of the previous change. No topological change of the circuit is allowed.

The source-stepping method can enhance DC convergence. But it is slower than direct use of the Newton-Raphson method. Therefore it is best used as an alternative to achieve convergence of DC operating point when the circuit fails to converge by using the Newton-Raphson method. The source-stepping method is used by SPICE when the variable ITL6 in the .OPTIONS card is set to the iteration limit at each step of the source(s).

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.

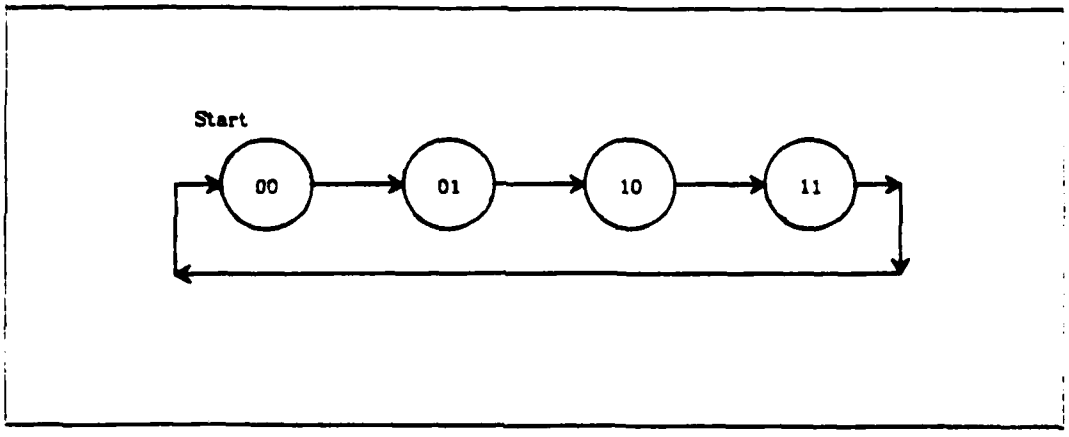


Figure 1: State Diagram for Example 1

```
--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

The INORDER and OUTORDER statements specify that the resulting PLA inputs and outputs, from left to right, are InSt0*, InSt1*, OutSt1*, and OutSt0*.

Following the OUTORDER statement are the logic equations for the two output variables, OutSt1* and OutSt0*. The exclaimation mark "!" indicates logical negation. The ampersand "&" signifies the logical *AND*, while the vertical bar "|" signifies a logical *OR*.

3.2. Truth Table

The -t option generates a truth table for the finite state machine. This truth table is written to the file *peg.summary*. The truth table for example 1 is shown in figure 4.

INPUTS:		s00:	InSt0* (msb)	
		s01:	InSt1* (lsb)	
OUTPUTS:		n01:	OutSt1* (lsb)	
		n00:	OutSt0* (msb)	
State Table	s	s	n	n
	0	1	1	0
	0	0	1	0
	0	1	0	1
	1	0	1	1
	1	1	0	0

Figure 4: Truth Table for Example 1.

Labels across the top of the truth table identify its columns. The mapping from column labels to actual signal names is given in the lists of input and output signals which preceed the truth table. To the right of the truth table are the names of the states described by the rows of the table.

4. Another Example

The second and more complex example shows the state diagram and corresponding *PEG* program for a FSM which recognizes the 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

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.

```
-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*>**] ;

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*.

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.

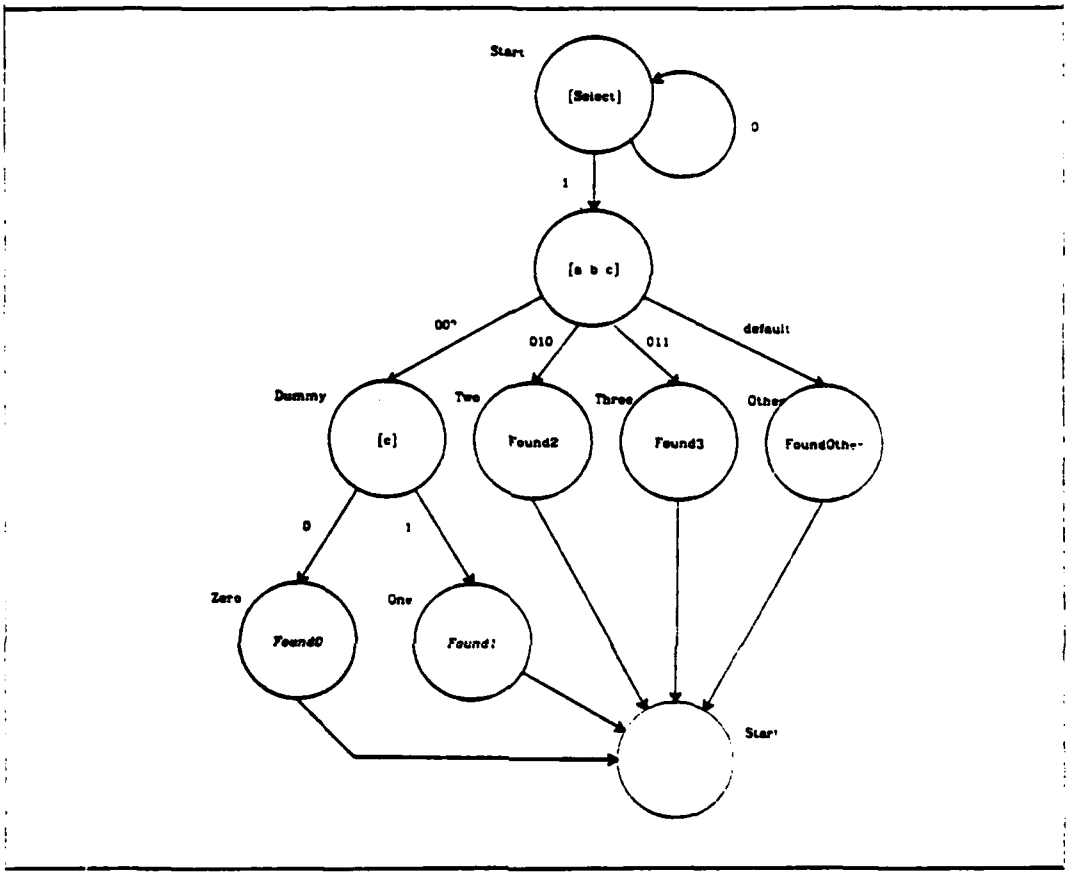


Figure 9: State Diagram for Example 3

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 ?.

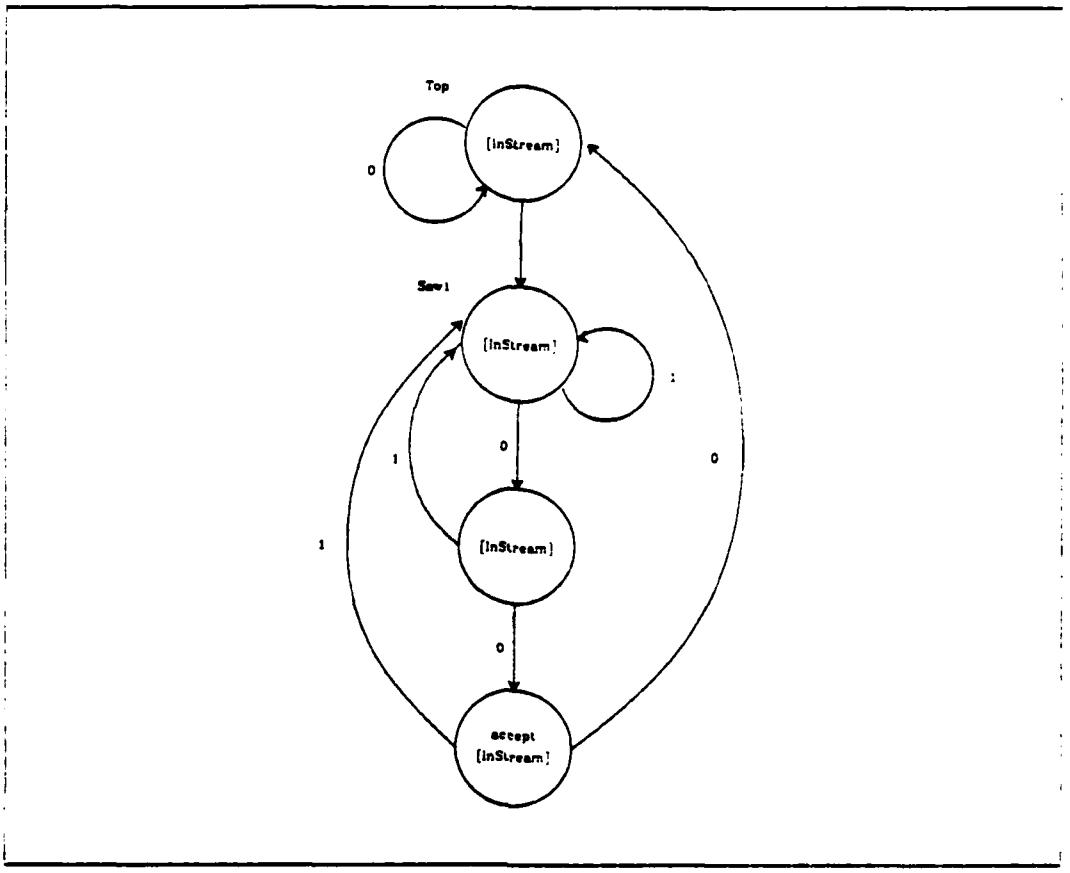


Figure 5: State Diagram Recognizing (1|0)*100

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

```
<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._]*
```

INPUTS: i00: RESET
i01: InStream
s00: InSt0* (msb)
s01: InSt1* (lsb)

OUTPUTS: n01: OutSt1* (lsb)
n00: OutSt0* (msb)
o00: Accept

State Table		i 0	i 1	s 0	s 1	n 1	n 0	o 0
	1	-	0	0	0	0	-	Top
	0	0	0	0	0	0	-	Top
	0	1	0	0	1	0	-	Top
	1	-	0	1	0	0	-	Saw1
	0	0	0	1	0	1	-	Saw1
	0	1	0	1	1	0	-	Saw1
	1	-	1	0	0	0	-	Saw1+1
	0	0	1	0	1	1	-	Saw1+1
	0	1	1	0	1	0	-	Saw1+1
	1	-	1	1	0	0	1	Saw1+2
	0	0	1	1	0	0	1	Saw1+2
	0	1	1	1	1	0	1	Saw1+2

Figure 8: Truth table for Example 2.

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.

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

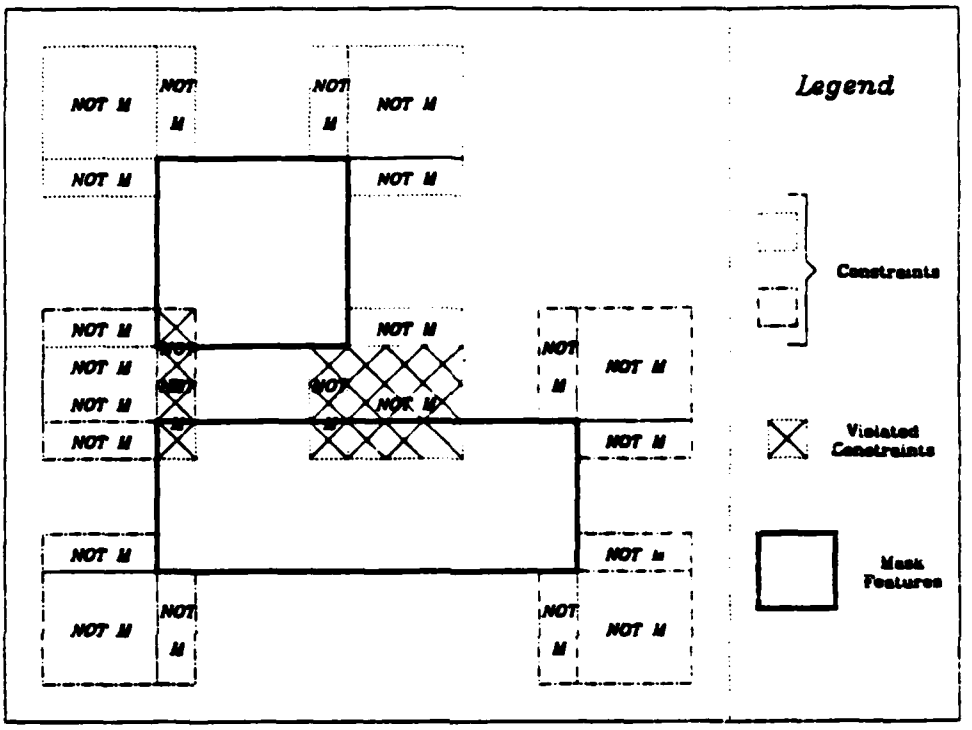


Figure 1. Metal Spacing Constraints.

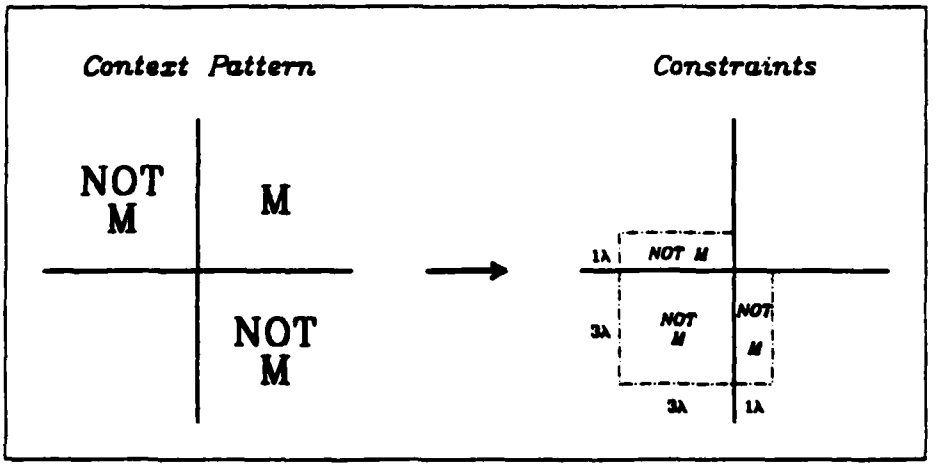


Figure 2. Metal Spacing Rule.

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

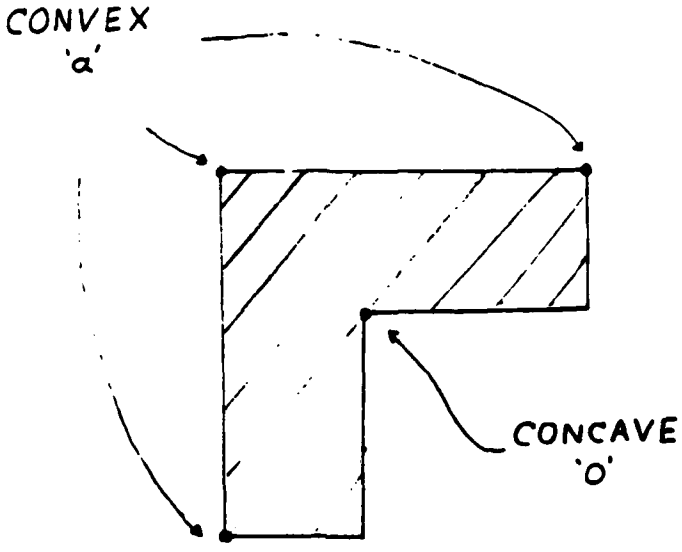


Figure 3. The Two Corner Types.

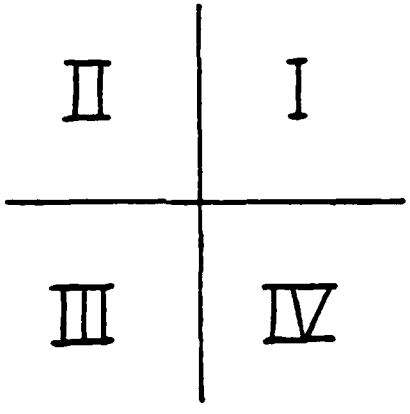


Figure 4. Quadrant Numbering.

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 applys 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.

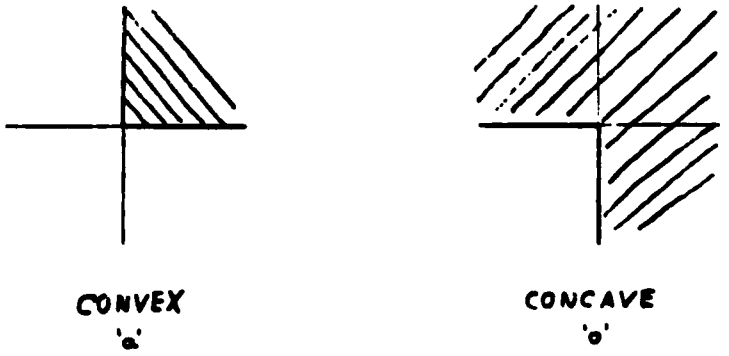


Figure 5. Canonical Corner Orientation.

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: ...**)]**

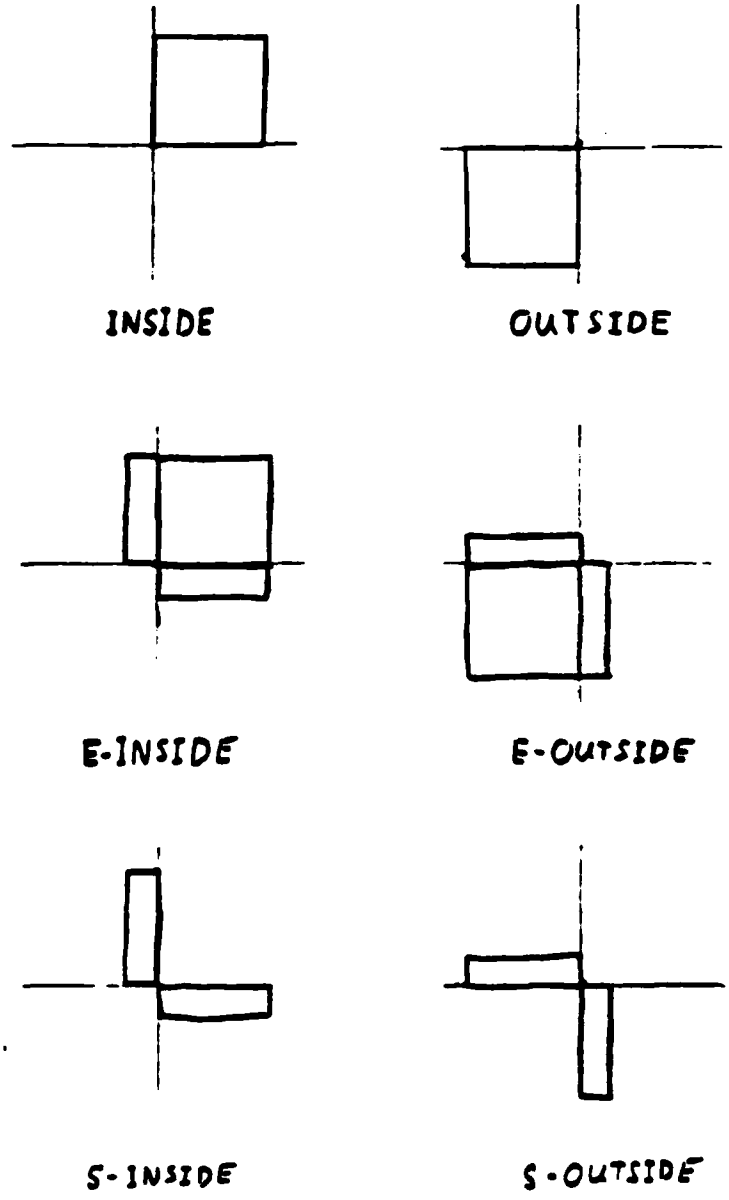


Figure 6. Constraint Macros.

(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

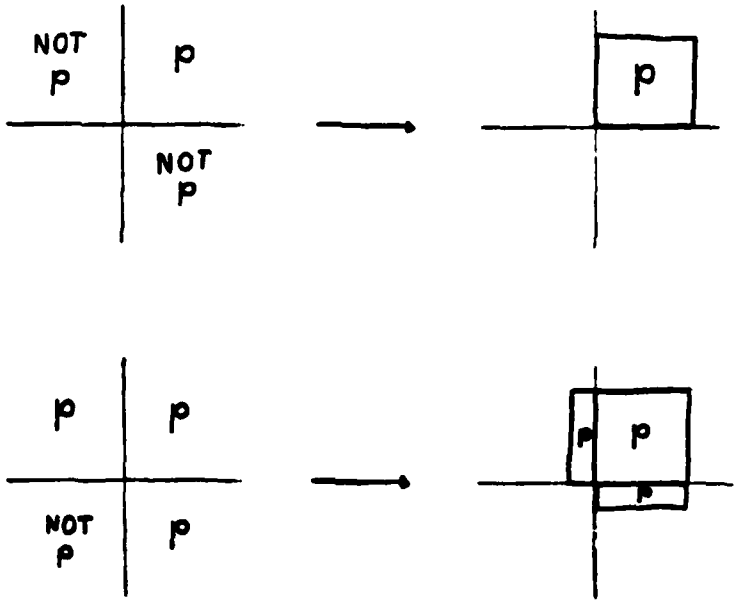


Figure 7. Width Rules. (width)

<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_s")))

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

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.

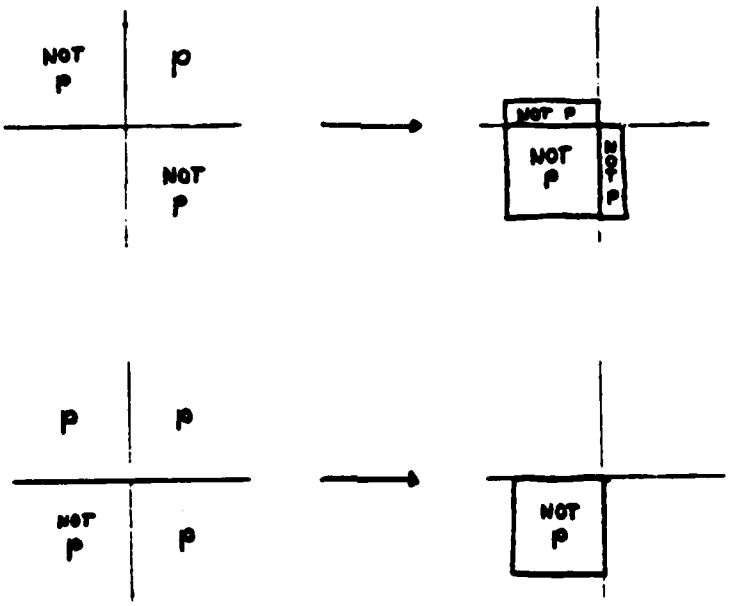


Figure 8. Single Layer Spacing Rules. (ss)

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.

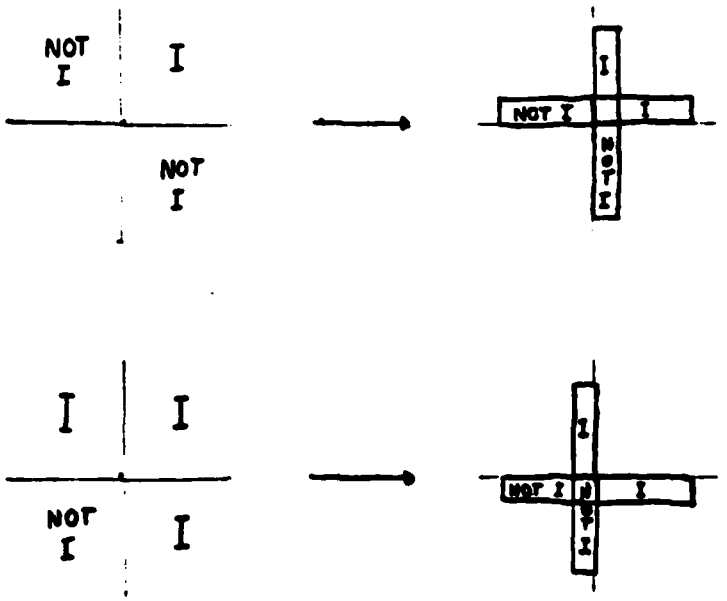


Figure 9. Parallel Edge Rules. (pe)

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.

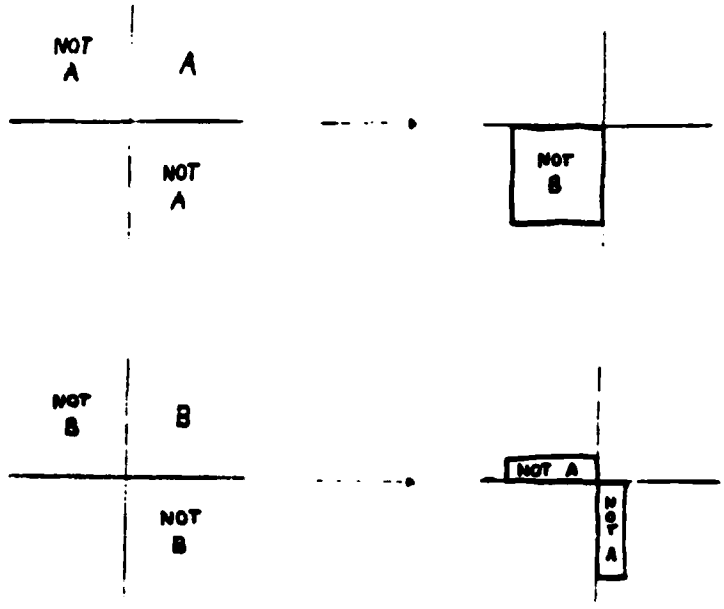


Figure 10. Interlayer Spacing Rules. (sep)

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

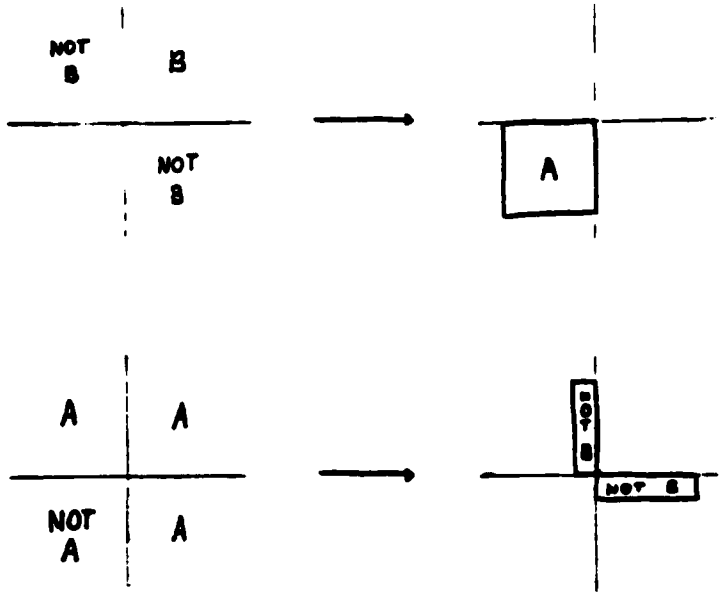


Figure 11. Enclosure Rules. (ext)

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.

END

FILMED

10-85

DTIC