AUTOMATIC LAYOUT OF OPTIMIZED PLA STRUCTURES

by

Howard A. Landman

Memorandum No. UCB/ERL M82/64

3 September 1982

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Automatic Layout of Optimized PLA Structures

by

Howard A. Landman

landman:Thu Jun 3 06:12:16 1982
cifplot* Window: -449 86249 -73200 449 --- Scale: 1 micron is 0.00787402 inches (200x)

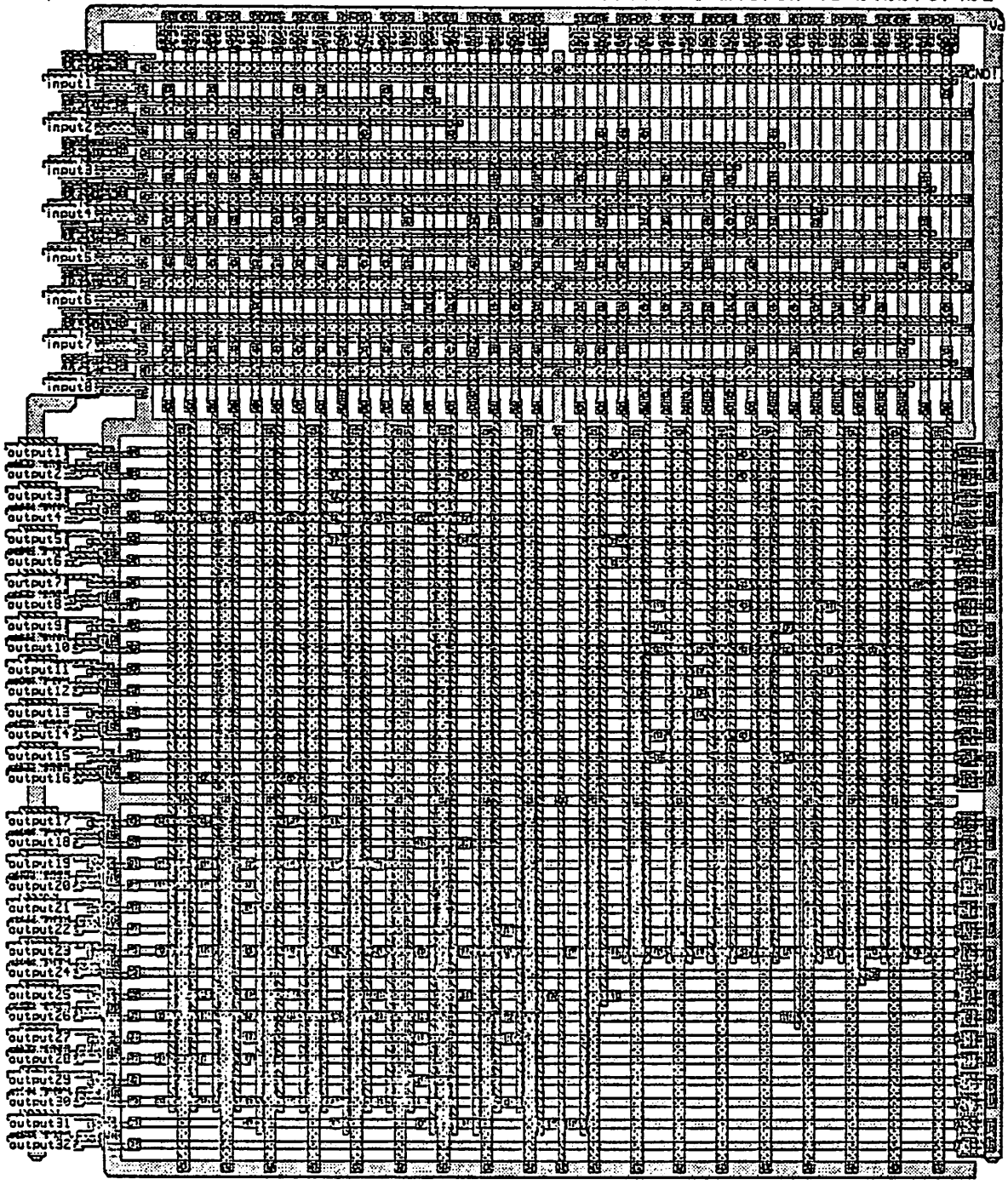


TABLE OF CONTENTS

1. INTRODUCTION

All digital systems can be viewed as being composed of two basic types of circuitry: *memory* and *logic*. This paper presents a certain style of designing logic subsystems for integrated circuits using *Programmed Logic Arrays* (PLAs), and the tools used to make it practical and economical (both in human and computational resources). These programs have been successfully used at Berkeley in the design of several integrated circuits, some as large as 44,000 transistors [Fitzpatrick81]. While most of our current tools were built with one particular technology in mind (silicon-gate NMOS with no buried contacts), the general approach is applicable in many technologies, and all the software that is technology-dependent could be adapted to, say, CMOS or I^2L without difficulty.

The fundamental observation that shapes this approach to VLSI design is that the time and effort of the designers is the limiting factor in most large integrated circuits being designed today, and that this limitation will become more severe as the chips get bigger. Having a way to transform logic equations quickly and reliably into working circuitry, with only limited human effort, frees up designers to spend more time on other (and probably higher level) aspects of the design. This can reduce the probability of design errors, and lead to improved performance of the whole system even if the automatically generated circuitry is non-optimal. In addition, since it is often possible for a program to perform simple optimizations that would be tedious to do by hand, generated circuitry may actually be faster, smaller or lower in power than what a human designer could produce in reasonable time.

A further difficulty in designing circuits is that large circuit blocks can have subtle problems with their electrical behavior (e.g. current limits due

to metal migration or I-R voltage drops). A designer of only moderate skill or experience may not be capable of handling these issues correctly. Thus it is highly desirable that an automatic layout system solve these problems itself, so that inexperienced designers can use it with confidence and experienced designers will usually not be tempted to "improve" the generated layout by hand.

Finally, since layout is only one part of the design process, it is important that a layout system have "hooks" that allow the resultant layout to be easily interfaced to other design tools. This includes automatic labelling of nodes (for plotting, circuit extraction, electrical rules checking and simulation) and naming the subcells in an appropriate format (for use with graphical layout systems like *caesar* and *kic*). Also, the layout program can minimize the number of geometric primitives generated, which decreases the time needed to plot, edit, or transmit the circuits which use the layout, or to make PG or MEBES tapes or photolithography masks from them. As we will see, the software system described here does all of the above.

2. IMPLEMENTING COMBINATIONAL LOGIC - FOUR APPROACHES

There are many approaches to implementing combinational logic beside PLAs. While a detailed discussion of these is outside the scope of this paper, a brief survey is in order. For further information see the excellent discussion in [Bell78], pages 42-46, most of which is applicable here.

2.1. Gate Arrays

Gate arrays are regular arrays of individual logic gates placed on a (usually rectangular) grid with space left for routing interconnect lines between the gates. Different logical functions can then be produced by "customizing"

the interconnect level. This can allow very fast turnaround for gate array fabrication; since the interconnect level occurs late in the fabrication process, an inventory of partially manufactured wafers can be maintained, and only one photolithography mask and a few processing steps are needed to complete a new circuit. In some gate array processes, even the exact nature of the gates themselves (AND, OR, XOR, etc.) can be programmed during this customization phase.

The problem is that gate arrays are generally of lower density than more custom approaches, often by a factor of ten. Thus the amount of circuitry that can be economically included in a single chip is about ten times less. Speed also suffers because the interconnections are longer than in a denser custom design. than with full custom design. Also, the advantage of fast turnaround is lost if one considers using a gate array as part of a larger circuit that includes some custom circuitry. The problems of interfacing the gate array portion of such a design to the custom portion could be difficult, and I know of no actual chips that have been made using a mix of gate array and full custom circuitry.

Gate arrays do have many advantages, however, and appear to be the most appropriate technology for some kinds of design. It is relatively straightforward to take an existing TTL or ECL gate-level design and convert it to a gate array, even if the design includes memory functions like latches or registers. Multi-level logic with three or more levels still maps easily. And if an entire chip is made as a gate array, and you have a captive fabrication facility, engineering changes can be implemented with rapid turnaround. Amdahl, DEC, Fujitsu, IBM, and Storage Technology have all used this approach to build large, complex products with great success.

2.2. Standard Cells

The *standard cell* approach involves developing a library of cells which implement a variety of logical functions but have uniform interconnect locations and identical heights. These cells can then be laid out in rows, and interconnected by a channel router which doesn't need to know the internal details of each cell. Since arbitrary custom subcells may be added to the library, this gives some of the advantages of full custom design, while still allowing arbitrary logical function and easy mapping from gate-level designs.

This approach also has density and speed problems, which stem primarily from the large number of interconnections required and the distances over which signals must travel. It shares most of the advantages of gate arrays except for fast turnaround. Some fairly large (and working) circuits have been built using this approach, and it is possible to make a chip that is part standard cell and part custom. A notable success in this regard was the Bell Labs echo canceller chip, reported in [Dutweiler80a] and [Dutweiler80b]. The Bellmac 32 reported at ISSCC 81 also used this approach.

2.3. Read-Only Memories

A simple, regularly structured way to implement combinational logic is to use a read-only memory or *ROM* to represent the entire truth table of the desired logic function. This has the advantage that it is easily implemented and modified, but the area grows geometrically in the number of inputs (proportional to $outputs \times 2^{inputs}$). Thus this approach is really not suitable for circuits with large numbers of inputs, even though it has been used commercially to a great extent. The recent 450,000 transistor microcomputer announced by Hewlett-Packard includes a 9,216 word by 38 bit ROM occupying about 40% of the chip surface [Beyers81]. Also, it is not possible to build

writeable memory from a ROM; all memory must be segregated from the ROM itself.

2.4. Programmed Logic Arrays

A *PLA* is a regular structure which can implement two level combinational logic. Usually this is thought of as consisting of an AND-plane and an OR-plane, though in practice we implement this as invert-NOR-NOR-invert. This AND-OR structure can implement any boolean function, since it can always be written in sum-of-products form. It can be used in place of a ROM simply by using the AND-plane as a full decoder, but it is also possible to reduce the number of "words" (product terms) by using terms that are not fully decoded, usually saving much space. Further, the reduced size may allow the PLA to be faster or to use less power than an equivalent ROM.

We chose to implement logic as PLAs for several other reasons. First of all, the necessary design of the basic circuit blocks had already been done in the desired technology (NMOS with Mead-Conway design rules), and tested in several working chips. Implementing a standard cell approach, on the other hand, would have required months of layout work in addition to the necessary programming, and user confidence in the correctness of the system would have had to wait for successful fabrication and testing of sample designs. Secondly, we saw that it would be possible to first program a simple PLA layout system which was computationally efficient, and then add optimization to it later. Gate arrays would have required us to solve the routing problem before the system was at all usable. Finally, we had had some experience with using PLAs in actual designs[1], and were certain that a PLA

[1]Several of the Berkeley projects in the MPC79 and MPC580 multi-project chip sets used PLAs built out of the library cells described in [Hon80].

generator could provide significant assistance to our designers without constraining their design style in any way.

3. A GUIDE TO BERKELEY PLA TOOLS

3.1. Overview

Generating a PLA at Berkeley may involve the use of several different programs, collectively referred to as *PLA Tools*. Figure 3.1 shows the principal programs, their input and output formats, and the data flow among them. There are many advantages to having a set of small programs rather than one large program. Among them are:

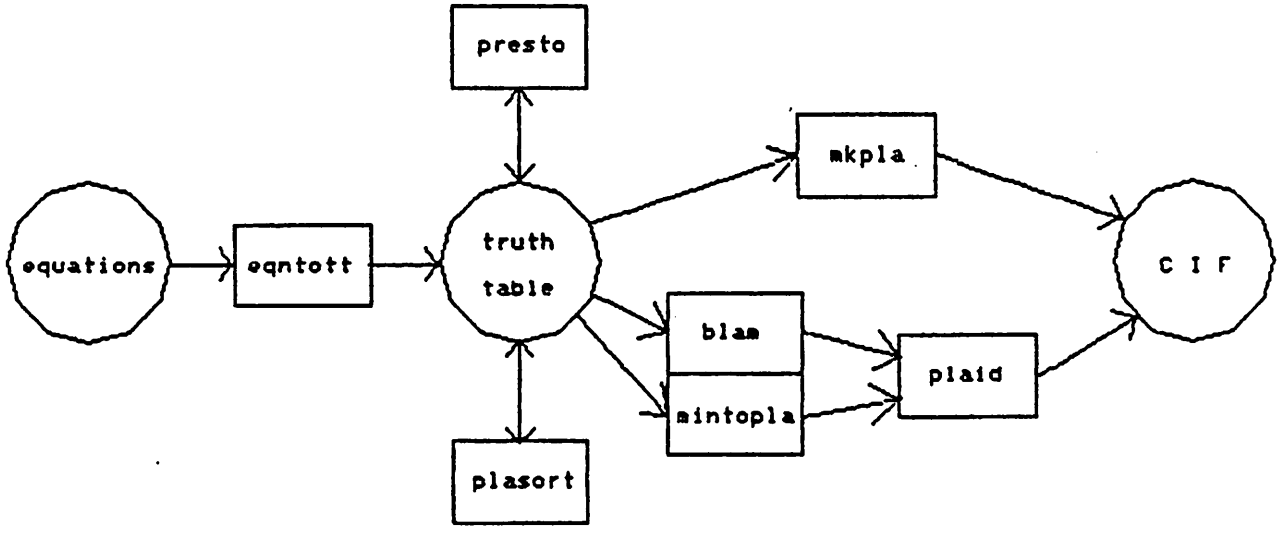


Figure 3.1 : Data flow among the principal PLA tools

The system can be usable with only a few vital portions present.
Programmers need not learn the entire system to enhance or maintain part of it.
Many programmers can work on the system without much need for coordination.
Debugging and maintaining each part of the system is simplified.
Enhancements can be made by adding programs instead of changing old ones.
A different language may be used for each program if desired.
Data structures can be chosen to best suit a single program's algorithms.
Pre-existing programs can be utilized for some tasks (e.g. sorting).
Users can begin using the system without understanding all of it.
Users can choose which capabilities of the system they need.
There is easy access to intermediate results.
Individual programs can be rerun with alternate options before proceeding.

There are programs not shown in Figure 3.1 that could be considered PLA tools, and while this paper is primarily concerned with the transformation of a logical description into a circuit layout, we will discuss related programs to show the relationship of the PLA tools to the rest of the Berkeley design environment.

3.2. Specifying Logic Equations

Many users will want to begin with a set of logic equations which define the function that the generated circuit is to perform. These equations can be written "by hand" using a text editor, or may be generated by a program from a higher-level description of the function or the circuit that contains it. For example, an instruction-decoding PLA might have its equations generated from a description of the instruction set to be executed and the control signals required.

The preferred format of these logic equations for input to the PLA Tools is called *.eqn format*[2]. It is mostly very similar to the way logical equations are written in the C programming language, which is the most widely used language at Berkeley. Some additional information is required to specify the

[2] This should not be confused with the format required by the program *eqn*, which formats equations to be printed by the typesetting programs *nroff* and *vtroff*. *Eqn* and *vtroff* were used in the production of this paper.

desired order of the inputs and outputs in the final layout. The precise definition of .eqn format is given in Appendix M, in the manual section for *eqntott*.

The important point to realize is that it is much easier to change a set of logic equations than to alter a circuit layout by hand, even if it means having to run the PLA Tools again to generate a completely new layout. During the design of the RISC-I microprocessor, the largest PLA on the chip was generated from equations which were changed more than 40 times in the last 2 months [Foderaro81]. Each time, a new layout was produced and integrated into the chip floor plan. To do this manually would have required at least a man-month of effort. Doing it automatically required only one or two man-days and a few hours of VAX CPU time.

3.3. Converting Logic Equations to Truth Tables

3.3.1. Eqntott (Robert F. Cmelik)

The *eqntott* program generates a truth table suitable for PLA programming (sometimes called a "personality matrix") from a set of Boolean equations in .eqn format that define the PLA outputs in terms of its inputs. If *eqntott* is run with the -R flag (as is normally done), it will attempt to reduce the size of the truth table by merging minterms, and to produce a truth table with no redundant minterms, but it does not attempt a full minimization of the logic functions.

3.4. Specifying the Truth Table

The truth table format describes both the logical function to be performed and a default method of implementing it as an AND-OR PLA. A truth table file begins with three lines giving the size of the PLA. They must each

begin with the character '.' (period). The required lines, which may appear in any order, and their meanings are:

.i i	The PLA has i inputs.
.p p	The PLA has p product terms.
.o o	The PLA has o outputs.

Each of i, p, and o must be an unsigned integer.

Following this is the programming pattern for the PLA, sometimes called its personality matrix. It is given on a pterm by pterm basis; for each pterm, first one program character is given for each input from 1 to i, and then one program character for each output from 1 to o. Tabs and spaces may be inserted anywhere to improve legibility. Since programs may in general handle quite large PLAs, and since there is an upper bound on line length on most computer systems, one pterm may be spread over multiple lines; a program that handles this format should keep reading until all $i+o$ characters have been read for each of the p pterms. The program characters are:

1 = Term utilizes predecessor.
0 = Term utilizes complement of predecessor.
- = Term does not utilize predecessor. No connection. Don't care what predecessor's value is.
x = Term does not necessarily utilize predecessor. Don't care whether there is a connection here or not. (Not used by most programs).

Note that a '0' cannot be used in the OR-plane since the complements of pterms are not available. Also, most programs currently treat an 'x' the same as a '-'.

Following the personality matrix is a command identifying the end:

.e End of PLA description.

See the examples of truth table files later in this paper (section 5).

3.5. Manipulating the Truth Table

While it is possible to feed the truth table produced by *eqntott* directly to a layout-generation program, in most cases the user will want to perform some optimizations first. There are two programs that read in a truth table, try to improve it, and produce another truth table which, while it specifies the same logical function, will result in a smaller and/or faster PLA.

3.5.1. Presto (Sheng Fang)

Presto is a combinational logic minimization program. It reads in a truth table and attempts to reduce the number of product terms, increase the number of don't care inputs, and also reduce the number of the output connections. The result is a new truth table in the same format which represents a (hopefully) smaller PLA than the original truth table did, thus saving silicon area and probably improving the speed of the PLA as well.

Since the general problem of completely minimizing truth tables is NP-complete[8], *presto* uses a heuristic algorithm to keep the computation from becoming prohibitively expensive. This does not produce quite as good results as some other known algorithms (e.g. MINI, described in [Hong74]), but requires less than 1% as much CPU time for a typical problem.

Presto is based on an earlier program of the same name by D. Brown of Tektronix, Inc. and the late A. Svoboda. There is a good description of the algorithms and data models used in [DeVries75].

3.5.2. Plasort (Jim Kleckner & Howard Landman)

Plasort, with reads a truth table and sorts the product terms to reduce the maximum delay through the PLA. It computes an approximate

[8]See [Garey79] for a detailed discussion of the implications of NP-completeness.

```
.i 3
.o 2
.p 7
001 10
010 10
-11 01
100 10
1-1 01
11- 01
111 10
.e
```

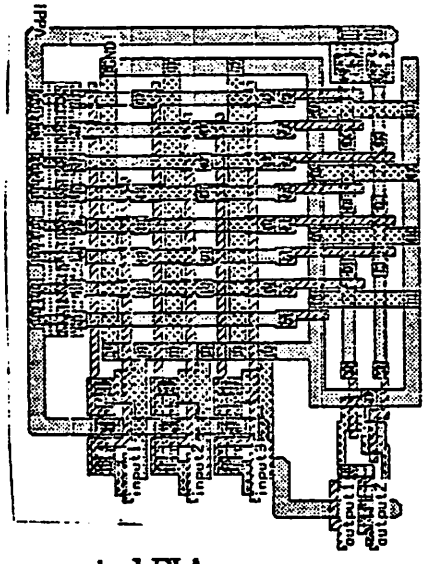


Figure 3.2(a) : An unsorted PLA

```
.i 3
.o 2
.p 7
001 10
010 10
100 10
111 10
x11 01
1x1 01
11x 01
.e
```

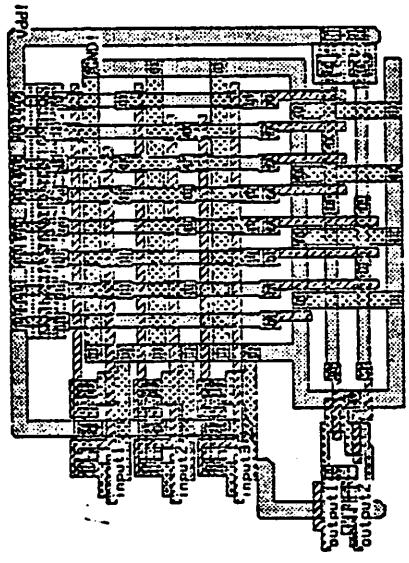


Figure 3.2(b) : The same PLA, sorted

"transmission line" delay for each product term based on typical NMOS electrical parameters, assuming that the layout will be generated by *mkpla* and that the -x option will not be used. It then reorders the product terms

so that the one with the longest horizontal delay is closest to the inputs of the PLA (and hence receives valid data soonest). Note that, because the product terms run orthogonally to the inputs and outputs, they may be permuted without changing the logical function of the PLA. This sorting reduces the variance of the times until the product terms become valid while leaving the average about the same. Thus, the worst case delay is reduced and the maximum operating speed of the PLA should be increased. Figure 3.2 shows a PLA before and after sorting.

To be fair, while we believe that sorting PLAs (and performing other optimizations) can speed them up significantly, this belief is based only on simple simulations at present. We plan to do more detailed simulations using SPICE, and measurements of actual chips, to better quantify the improvements achieved.

3.6. Topological Minimization

3.6.1. Blam (Mark Hofmann)

Blam takes a truth table and looks for ways to make a smaller circuit by altering the topology of the PLA so that, for example, inputs can enter from both sides of the AND-plane and utilize area that is not used by inputs on the opposite side. It can also generate structures with multiple AND-planes or OR-planes. These structures are output in a modified truth table format which can no longer be input to *presto*, *plasort*, or *mkpla*; instead, a program called *plaid* must be used to generate the layout. For details on the algorithms used, see [Hofmann80].

3.6.2. Plaid (Mark Hofmann)

Plaid takes a modified truth table produced by *blam* and generates a layout of the topologically optimized PLA. *Plaid* has options for producing a slower, lower power PLA and for placing extra ground lines in the AND- and OR-planes. Like *blam*, *plaid* is more fully described in [Hofmann80].

3.6.3. Mintopla (Mark Hofmann)

Plaid can generate layouts from the standard truth table format output by *eqntott*, *presto*, and *plasort* if they are first run through the format conversion filter *mintopla*, which has the same output format as *blam*. In this way *plaid* can be used to produce PLAs that have not been topologically altered.

3.7. Generating the Layout

There are two ways to get a PLA layout generated. If you have performed topological minimization (i.e. run *blam*), then you must use *plaid* to generate your layout as described in the previous section. If you have not, then either *plaid* or *mkpla* may be used, but *mkpla* is preferred because it handles electrical problems better, has more options, and is better integrated with the other design tools.

3.7.1. Mkpla (Howard Landman)

The *mkpla* program takes a truth table description of a logical function and produces a CIF file specifying the mask layout geometry for a circuit which performs that logical function. CIF, the Caltech Intermediate Form for describing layout, is described in [Mead80] and more extensively discussed in [Hon80].

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

The PLA structures produced are similar to those designed by Dick Lyon and described in [Hon80], but *mkpla* (optionally) optimizes several aspects of the layout to improve the performance and reliability of the PLA. For details see Appendices E-1 and E-2 and the manual entry in Appendix M.

3.8. Manipulating the Layout File

Sometimes the CIF layout produced needs to be modified slightly in order to be of more use. Some brief examples follow.

3.8.1. Editor scripts

Any standard text-editing program can be used to modify the CIF output file, since CIF is just Ascii text. This might be desirable if, for example, the user wants to give specific names to the inputs and outputs of the PLA.

While *mkpla* will only label the inputs with labels of the form "input1", "input2", etc., assignment of more meaningful names to these nodes is easily accomplished by using the editor to replace those strings with other names. If this replacement will have to be done more than once, a file containing the necessary editor commands can be saved and used to control the editor each time. Such a file of editor commands is called an *editor script*. If a high-level program is generating the logical equations for *eqntott*, it may be worthwhile to have it also generate an editor script for relabelling the input and output nodes.

3.8.2. Caesar (John Ousterhout), Cif2ca (Peter Kessler)

Caesar is an interactive color graphics editor which runs on an AED512 color terminal (with optional Summagraphics bitpad) and an additional text terminal. *Cif2ca* is a program to convert CIF files into *caesar* format. *Caesar* was used for the design of the RISC-I microprocessor, which contains four *mkpla* PLAs. Figure 2 of [Ousterhout81] shows three of these in context. The cells "barpla" (left center), "ccpla" (upper left), and "oppla" (center) were all generated by *mkpla*, except that the barpla also contains some hand-drawn geometry. Several interesting interface problems had to be solved to make *mkpla*'s output readily usable by *caesar*. We discuss two of them below.

Caesar produces layouts in which all edges are parallel to the coordinate axes; such 90-degree layouts are called *manhattan*. Earlier releases of *mkpla* had used non-manhattan features to slightly improve the conductivity at some junctions of metal lines, and to shorten the "random wiring" portions of the PLA. This turned out to be a false economy, since it made the resulting PLAs useless to any manhattan design tool. The solution was to rewrite *mkpla* to (optionally) produce all-manhattan layouts; in fact, this is now the default.

Another problem was that *caesar* keeps a separate file for each cell, with names of the form *cellname*.ca; since filenames in UNIX are restricted to 14 characters, cell names in *caesar* should not be longer than 11 characters. This creates a problem for a layout program like *mkpla* that has parameterized cells; it must give each variant a unique cell name to avoid confusion, and yet can use no more characters than *caesar* itself can. The solution here was to use the first three characters of the cell name to indicate that the cell was generated by *mkpla*, the fourth to specify which type of cell it is, and the

remaining seven to give the parameters that make the cell different from others of the same type. For example, a cell name of "plaI-Iw6" is given to an input buffer ("I") with input clocking ("I") and 6λ-wide power and ground busses ("w6").

For further details about *caesar* see [Ousterhout81].

3.8.3. KIC, Ciftokic, Kictocif (Ken Keller) *KIC* is an interactive color graphics editor which runs on an AED512 color terminal with Summagraphics bitpad. It handles a much wider range of geometries than *caesar*, including arbitrary polygons and paths. The conversion programs *ciftokic* and *kictocif* translate between KIC format and CIF. No special geometrical constraints had to placed on *mkpla* to interface it to *KIC*, but similar namelength limits had to be observed. A tutorial introduction to *KIC* is given in [Keller80].

4. RELATED PROGRAMS

4.1. Circuit Extraction

The circuit extractors in use at Berkeley take a CIF description of layout geometry and produce output files describing the transistors, nodes, and capacitances of the circuit, suitable for electrical rules checking or simulation. They read node labels from user extension "94" commands in the CIF file. Both *mkpla* and *plaid* produce the appropriate "94" commands to identify their input and output nodes to both of our circuit extractors. *Mkpla* can also produce any user extension desired for labelling, which would allow its output to be used with software using conventions other than Berkeley's. *Plaid* can't, but a simple text editor could be used for conversion.

4.1.1. Cifplot -X (Dan Fitzpatrick)

The first circuit extractor at Berkeley was developed as an option to the plotting program *cifplot*. It can handle arbitrary geometries, but is too inefficient to be used easily on very large circuits.

4.1.2. Mextra (Dan Fitzpatrick)

Mextra is a manhattan circuit extractor, i.e., it will only handle boxes whose edges are at multiples of 90 degrees. This is an unacceptable restriction for some applications, but we have found that it makes for a great simplification of the program, and a significant improvement in perfomance. Benchmarks show that *mextra* runs about ten times faster than *cifplot -X*.

4.2. Layout and Electrical Rules Checking

4.2.1. Layout Rule Checking

Several programs are in use or under development at Berkeley to check layout rules. To date, most production checking has been performed using *moslrc*, a version of the MIT design rule checker described in [Baker80]. Several bugs in early versions of *mkpla* were found by running *moslrc* on the generated layouts.

More recently, a new design rule checking program called *lyra* [Arnold81] has been used.

Except for some "technical" violations in the input and output drivers, which derive from Dick Lyon's original cells, neither *lyra* nor *moslrc* has found any errors in *mkpla*-generated layouts since April 1981. This kind of empirical evidence for correctness helps build user confidence in the PLA tools. Such tools, like other programs, could contain subtle or infrequent bugs which are hard to detect. Even if a procedural layout tool were

considered to be verified, any modification or enhancement might introduce new errors. We do not believe that correct procedural design tools can be built in the near future without a substantial verification effort.

4.2.2. Electrical Rules Checking

After a circuit has been extracted, a static verification of the electrical properties of the circuit is usually performed. The *moserc* program, a descendent of the *stat* program written at Stanford by Forest Baskett, will check that all nodes have paths to power and ground, and that there are no floating inputs or unused output values. It also checks the pullup/pulldown ratios of logic gates. For example, the extra pterm (or output line) generated by *mkpla* to make the total number even (if it was odd) is detected as a node which cannot be pulled low and whose value is never used by another node.

4.3. Simulation

4.3.1. Spice (Larry Nagel, Ellis Cohen, Andrei Vladimirescu)

The very popular simulation package *spice* can be used in conjunction with circuit extraction to do detailed electrical simulation of PLAs or other circuits. We plan to further study the effects of the electrical optimizations performed by *mkpla* in this manner.

4.3.2. Mossim (Chris Terman)

To verify the logical correctness of a digital circuit, it is not necessary (and often computationally infeasible) to run *spice* on the whole circuit. Instead, a logic-level simulation may be performed using *mossim*, a simulator which models enhancement MOSFETs as switches. The simplicity of this model allows very large circuits to be simulated with reasonable efficiency. One bug in an early version of *mkpla* (it was switching the true and

complemented inputs) was found by comparing the results of this simulation with evaluation of the logical expressions defining the PLA.

4.3.3. Slang (John Foderaro)

During the design of the RISC chip, a multi-level simulator called *slang* was developed in LISP to tie together high-level functional simulations with the lower-level simulation of *mossim*. *Slang* simultaneously ran an instruction-level simulation of the processor, functional simulations of the major subsystems of the chip, and a *mossim* simulation of the circuit net extracted from the actual layout. At appropriate times, it would then compare the results of the different levels of simulation, and report any discrepancies. Many subtle logical or conceptual errors can be discovered in this way, and the cost is not much greater than the cost of doing switch-level simulation only. The error described in the previous section was found in this manner, and would have been very difficult to find in any other way.

4.3.4. Plasim (Howard Landman)

A recent addition to the PLA Tools is *plasim*, a fast PLA simulator. It reads a truth table file for the description of the PLA to be simulated, and then takes input vectors on standard input and returns output vectors on standard output. It uses three-state logic (high, unknown, low) so that, for example, it could be used to analyze initialization problems in a finite state machine. It also produces a report evaluating how well the set of input (test) vectors "exercised" the PLA, and pointing out possible faults that might have escaped detection under this test.

5. A SIMPLE EXAMPLE OF PLA GENERATION

Let's take a small example and follow it through all the steps necessary to generate the PLA layout with appropriate labels for circuit extraction. The example we choose is a full adder with carry-in and carry-out. This is not the most compact way to build an adder, but it may be the easiest in terms of designer effort.

The logic equations we need are:

```
INORDER = C0 A1 B1;
OUTORDER = S1 C1;
S1 = (C0&!A1&!B1) | (!C0&A1&!B1) | (!C0&!A1&B1) | (C0&A1&B1);
C1 = (C0&A1) | (C0&B1) | (A1&B1);
```

Let's assume we have these in a file called "adder.eqn". Then the command "eqntott -R adder.eqn > adder.tt" will create a truth table in the file adder.tt:

```
.i 3
.o 2
.p 7
001   1 0
010   1 0
-11   0 1
100   1 0
1-1   0 1
11-   0 1
111   1 0
.e
```

Now we can feed the truth table to *presto* to see if we can optimize it further by doing "presto <adder.tt >adder2.tt". Note that *presto*'s input and output formats are identical, so this step is optional if you don't think that optimization will gain you anything, or if you want to save the CPU time. The

result is:

```
.i 3
.o 2
.p 7
001 10
010 10
-11 01
100 10
1-1 01
11- 01
111 10
.e
```

This is identical to the previous truth table, so that *presto* didn't succeed in reducing the size of this PLA. The next step, also optional, is to sort the product terms using the *plasort* program. This can speed up the PLA by putting pterms with larger RC delays closer to the inputs (and outputs if the pla is cis). The command "plasort -d adder2.tt >adder3.tt" produces:

```
.i 3
.o 2
.p 7
001 10
010 10
100 10
111 10
-11 01
1-1 01
11- 01
.e
```

Now we have optimized the truth table as much as possible, so it is time to generate the PLA. Running "mkpla < adder3.tt" causes the layout to be generated and written into the file "mkpla.out". This file contains all the symbol definitions, calls, and geometric primitives needed to completely

specify the layout of the PLA. Note that under UNIX you could have piped all of these programs together and run them as one command by doing "eqntott -R adder.eqn | presto | plasort -d | mkpla".

In terms of run time, the minimization routines in *eqntott -R* and *presto* take by far the largest portion. The other programs are all very efficient, as shown in Table 5.1.

6. DIRECTIONS FOR FURTHER WORK

This paper shows how combinational logic can be automatically implemented in Si-gate NMOS technology as Programmed Logic Arrays, using a simple and modular set of programs. We have also seen that integrating these tools into a larger design environment is a significant portion of the programming effort, and how the existence of standard formats like CIF and the truth table format makes this task much easier than it might otherwise be. This section outlines some areas in which further work might be productive.

	Adder	*Oppla*
i	(3)	(11)
o	(2)	(34)
eqntott	0.4	65.8
p	(7)	(703)
eqntott –R	0.4	365.0
p	(7)	(49)
presto	0.5	54.3
p	(7)	(36)
plasort –d	0.2	0.7
mkpla	1.7	6.8
blam	13.2	–
plaid	0.6	–

Table 5.1: Run times of the PLA Tools on two examples

Other types of PLA-related structures could be automatically generated. For example, it would be possible to generate complete Arithmetic-Logic Units using modified PLAs with multi-input decoding and additional logic at the outputs [Shmookler80]. The necessary cells to do this in NMOS have already been designed, but have not yet been tested, and no software to support them has been written.

The existence of these multi-input decoding cells raises interesting logical questions. It is easy to show that the number of product terms in a PLA need never increase, and can usually be decreased, by the use of such input drivers. But since the decoders themselves are larger than the normal PLA input buffers, it is not clear that the gain in product terms is always worth the price. And what is the best pairing of inputs for decoding? This question is not in general easy to answer.

The PLA Tools could be integrated with existing pad-placing and channel-routing software to produce a system with the capability of compiling logic equations directly into a complete chip. This system could compete very favorably with ROMs for logic functions with a large number of inputs.

It would be possible to generate multi-level (≥ 3) logic arrays. In some cases this might allow a reduction of the overall circuit size, especially for very complex functions. The question of when such a strategy pays off has not been well investigated.

Large PLAs sometimes appear to to not be very "dense", in the sense that the ratio of transistors/area is quite low, especially in the OR-plane. Mark Hofmann's work on folded and split PLAs is a first step toward dealing with this problem, but there are other things to try as well. It is often possible to break a large PLA into two or more smaller PLAs which occupy less

total area. This is rarely done in practice because there are no automatic tools to perform such "partitioning". Ron Ayres described a hierarchical PLA generator in his silicon compilation paper [Ayres79], but the program can only partition when hierarchy is explicitly designed into the logic specification. Thus it has nothing to say about the partitioning of general PLAs. Also, a perfect partition is not usually possible; some input buffers might have to be duplicated in the smaller PLAs, complicating the analysis.

A PLA is an example of a structure which occurs frequently in integrated circuits: a *heterogeneous array*, i.e., an array where not all the elements are the same. Other examples include ROMs, MUXs, decoders, encoders, and some kinds of gate array. No current layout system supports heterogeneous arrays. A general and flexible tool to lay them out would be an excellent base on which to build many different kinds of module generators.

Acknowledgements

Numerous people and organizations have contributed to the work described here during the past three years. Whether the support was technical, financial, or personal, without them this paper would have less to report.

Gelly Archibald
Michael Arnold
Bob Baldwin
Robert Cmelik
Lynn Conway
Sheng Fang
Dan Fitzpatrick
John Foderaro
Mohammad Hakam
Gordon Hamachi
Mark Hofmann
John Howes
Manolis Katevenis
Ken Keller
Jim Kleckner
Helen Landman
Louis Landman
Dick Lyon
Richard Newton
John Ousterhout
David Patterson
Jim Peek
Zvi Peshkess
Nirmal Ratnakumar
Ted Strollo
Don Scharfetter
Carlo Séquin
Robert Sherburne
Robert Tremain
Korbin Van Dyke
Anne Ver Steeg
Julie Ver Steeg

Comsat General Integrated Systems
SynMos Corporation
Xerox Palo Alto Research Center

Sponsored in part by

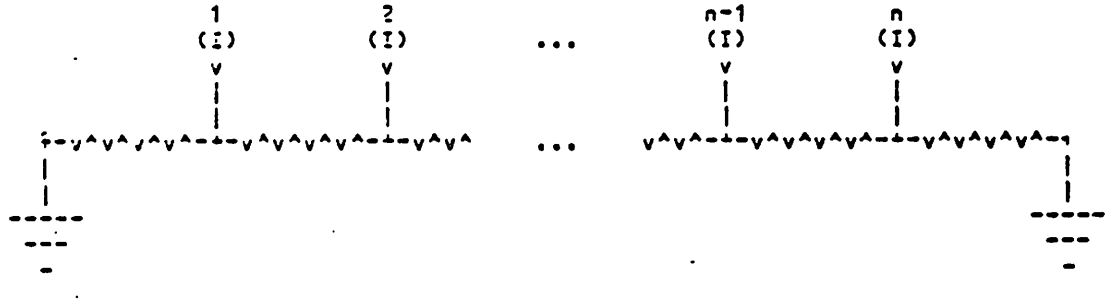
Defense Advance Research Projects Agency (DoD) ARPA Order No. 3803 Monitored by Naval Electronic System Command under Contract No. N00039-81-K-0251

Appendices E : Electrical Calculations for mkpla

Appendix E-1 : Extra ground lines in AND and OR planes.

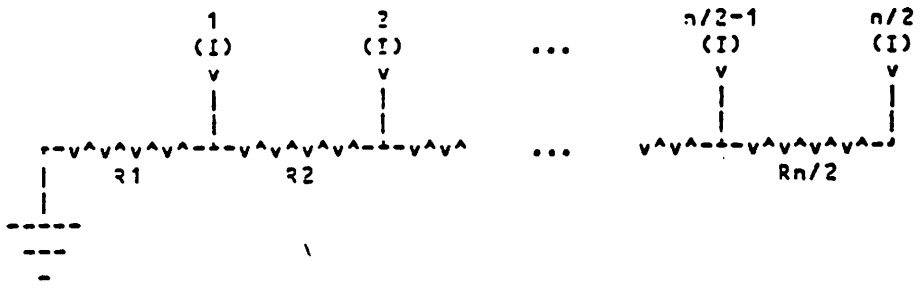
Very large PLAs require extra metal ground lines to make sure that the voltage doesn't get too high on the diffusion ground lines. Unfortunately, this problem gets worse as λ gets smaller if you also assume that V_{dd} doesn't scale down; hence, more of these extra ground lines are required. *Mkpla* automatically figures out how bad the problem could get in the worst case, calculates the number of extra ground lines needed in each plane, and inserts them (unless overriden by the user). It does this independently of the actual programming of the PLA for computational efficiency; it is probable that somewhat fewer lines would be needed if the program took the time to look at where the gates actually were.

Here is a worst case analysis of the problem, based in part on unpublished work done by Bob Baldwin at Xerox PARC. We can model a diffused ground line with pulldowns attached to it at regular intervals as a series chain of resistors, and the gates themselves as current sources. Since this is a DC analysis, we can ignore all capacitances. In the worst case, we must assume that all the metal lines crossing this diffused line are dumping current into it. Both ends of the diffused line are held to zero volts by a "good", i.e. 4λ wide metal, ground line. The picture is:



By symmetry, the highest voltage must be in the center, and the current must flow away from it in both directions. So we only need to look at

one half of the model. Assuming n is even for simplicity, we get $\frac{n}{2}$ current sources and $\frac{n}{2}$ resistors:



The voltage drop across r_i will be proportional to the current flowing through r_i, which is just $(\frac{n}{2} - i + 1)$ times the current from one gate. The voltage after $r_{\frac{n}{2}}$ is the sum of all the voltage drops,

$$\sum_{i=1}^{\frac{n}{2}} \left(\frac{n}{2} - i + 1 \right) = \frac{n^2}{4} - \frac{\frac{n}{2} \times \left(\frac{n}{2} + 1 \right)}{2} + \frac{n}{2} = \frac{n^2}{8} + \frac{n}{4}$$

or $\frac{n^2}{8} + \frac{n}{4}$ times the drop that one gate's worth of current would have going through one resistor. Since each current source generates about .1mA, and the resistance of the diffusion is about 60Ω (2 "squares" of diffusion at about $\frac{30\Omega}{\text{square}}$ sheet resistivity, a fairly conservative number), we get 6.0mV as this fundamental voltage drop.

It remains to ask: how high a voltage can be tolerated on the diffused line? Certainly it should be less than the enhancement transistor threshold voltage, which is typically .7V to 1.0V. I assume that .5V is acceptable, which should be enough since everything else in these calculations is pretty conservative. That means that we want:

$$\frac{n^2}{8} + \frac{n}{4} \leq \frac{0.5V}{0.006V}$$

$$n^2 + 2n \leq 667$$

$$n \leq 25$$

Unfortunately, this is not the whole story. The extra ground lines which are added have a maximum current-carrying capability, and this capability scales down quadratically with λ because the lines are not only getting narrower, but thinner as well. At $\lambda = 2.5\mu m$, this limit is about $\frac{1mA}{\mu m}$, or 10mA for a normal 4λ wide line. This means that we also want $n \leq \frac{10mA}{0.1mA} \times \frac{\lambda^2}{6.25\mu m^2} = 16\frac{\lambda^2}{\mu m^2}$ (with λ expressed in microns). Note that this imposes a scaling limit on this PLA structure, since n cannot be less than one; at $\lambda=.25\mu m$ the formula says we will need an extra metal ground line for every real signal line. To go beyond this limit will require widening the extra ground lines, using a metal with better current capacity than aluminum, or reducing V_{dd} to less than 5 volts. Even if V_{dd} scales down linearly with λ, there is a limit at $\lambda=.025\mu m$; however, this is well below the currently known scaling limits of NMOS transistors, and so doesn't need to be worried about in the near term.

Once we know how many extra ground lines we need, we could just stick in a ground line after every n pullups, but that would not distribute them evenly; the last one might come just before the end of the plane (where there is already a metal ground line) and be wasted. A better way is to figure out how many ground lines we need, and then distribute them as evenly as possible.

This is the method used by *mkpla*. A pair of functions called *groundsAbove* and *groundsLeftOf* determine how many extra ground lines should be placed above (in the AND-plane) or to the left of (in the OR-plane) a given pterm or output respectively. They are called both by the code that actually lays out the extra lines, and by the code that lays out program flashes to determine how much to offset them by. This approach guarantees that the flash placing code will never get confused about where the correct intersection is, even if we alter the algorithms used by these functions. The only possible exception to this rule is in the scaling limit, if we require more than one extra ground line every two signal lines. Using the present algorithm, this will certainly not happen with $\lambda \geq 0.5\mu$m.

Appendix E-2 : Widening V_{dd} and GND to avoid metal migration.

To calculate the linewidths necessary to eliminate any chance of metal migration problems and minimize the problem of voltage drops on metal V_{dd} and ground lines, *mkpla* makes the following conservative or worst case assumptions:

(1) V_{dd} will remain at 5V even though the dimensions of the circuitry may scale down. This is unrealistically conservative in the long run, but seems likely to be true for the next few years.

(2) Any part of the "random wiring" attached to pullups, inputs, outputs, or connect cells may have to carry the entire current load of the PLA. This is only realistic for about half of the lines widened, and it would be possible to save some space by calculating for each segment separately or by knowing where V_{dd} and GND are actually hooked up.

(3) The metal migration current limit at λ=2.0μm is about $1\frac{mA}{\mu m}$, and scales down linearly with λ because the thickness of the metal lines is assumed to scale down linearly with λ. Hence the metal migration limit expressed in $\frac{mA}{\lambda}$ scales down quadratically with λ.

(4) A 2λ by 2λ pullup (as used in the input & output drivers) sources 0.2mA; the "pullupPair" pullups source 0.1mA each.

(5) All outputs could be low simultaneously. However, an output is low only when the corresponding metal line in the OR-plane is high. This means that either the output or the corresponding pullup can be sourcing current, but not both at once.

(6) Only one of the two inverters in an input cell is sinking current at any instant.

The conclusion is that the metal width required (in λ) is:

$$\left[\frac{2.0\mu m}{\lambda}\right]^2 \times \frac{(total\ current)}{2mA/\lambda}$$

where (*total current*) is:

$$inputs*0.2mA + pterms*0.1mA + outputs*0.2mA$$

so that (assuming λ=2.0μm for the moment and hence ignoring the scaling factor which becomes 1):

$$width = \frac{(2*inputs + pterms + 2*outputs)}{20}$$

Now for a recent medium-sized PLA (RISC Gold chip instruction decoder) which has 11 inputs, 47 pterms, and 34 outputs, we get:

$$width = \frac{(22+47+68)}{20} = \frac{137}{20} = 6.85\lambda$$

which is considerably more than the 4λ default width. *Mkpla* rounds up to the next highest multiple of 2λ, so it would use 8λ wide lines for power and ground in this PLA.

Since some of the power and ground lines run through the center of the PLA and not just around the edges, it is necessary to *stretch* the cells that contain V_{dd} and GND busses to accomodate the wider lines, and to move the AND- and OR-planes apart so that cells still abut properly.

Appendices L : Large examples

LRU replacement algorithm

(provided by Carlo Séquin)

This example is particulary interesting because it was run through both presto and mini, and thus shows something about the relative performance of the two minimizers. The details of the algorithm itself are not of relevance here and are omitted. The function requires 7 inputs and 3 outputs. The naive formulation of the algorithm as a truth table has $128=2^7$ pterms as follows:

```
.i 7
.o 3
.p 128
0000000   000
0000001   101
0000010   011
0000011   001
0000100   101
0000101   000
0000110   001
0000111   001
0001000   011
0001001   001
0001010   000
0001011   000
0001100   010
0001101   000
0001110   000
0001111   000
0010000   111
0010001   101
0010010   011
0010011   001
0010100   111
0010101   001
0010110   001
0010111   001
0011000   110
0011001   001
0011010   001
0011011   000
0011100   100
0011101   000
0011110   000
0011111   001
```

```
0100000  010
0100001  101
0100010  011
0100011  011
0100100  111
0100101  000
0100110  011
0100111  011
0101000  011
0101001  001
0101010  010
0101011  010
0101100  010
0101101  000
0101110  010
0101111  010
0110000  011
0110001  111
0110010  011
0110011  011
0110100  111
0110101  101
0110110  011
0110111  011
0111000  011
0111001  110
0111010  011
0111011  010
0111100  010
0111101  100
0111110  010
0111111  011
1000000  100
1000001  101
1000010  011
1000011  101
1000100  101
1000101  100
1000110  001
1000111  100
1001000  110
1001001  101
1001010  000
1001011  100
1001100  100
1001101  100
1001110  000
1001111  100
1010000  101
1010001  101
1010010  111
1010011  101
1010100  101
```

```
1010101   101
1010110   110
1010111   100
1011000   110
1011001   101
1011010   011
1011011   100
1011100   100
1011101   100
1011110   010
1011111   101
1100000   110
1100001   111
1100010   111
1100011   011
1100100   111
1100101   110
1100110   110
1100111   001
1101000   110
1101001   110
1101010   110
1101011   000
1101100   110
1101101   110
1101110   110
1101111   000
1110000   111
1110001   111
1110010   111
1110011   101
1110100   111
1110101   111
1110110   110
1110111   100
1111000   110
1111001   110
1111010   111
1111011   010
1111100   110
1111101   110
1111110   110
1111111   011
.e
```

Presto reduces this to 39 terms:

```
.i   7
.o   3
.p  39
-011111  001
0011000  110
00-1001  001
0-11010  001
0010100  010
-01010-  001
00-011-  001
0-01-00  010
01-1-1-  010
0-0100-  001
0111-01  100
01-1--0  010
0-11111  001
-1--0-0  010
-1-0001  101
01-1000  001
-1100--  001
01-001-  011
10--10-  100
-0001-0  001
-01-100  100
10--1-1  100
101-110  010
1--0-0-  100
1--0010  011
10-00-1  101
-0-0100  100
11--00-  110
11-0-0-  010
11-0--0  110
1100-11  001
-11-010  001
-1110--  010
11--0-0  100
-10001-  010
1--0100  001
1-10---  100
1-10-0-  001
-11-00-  010
.e
```

MINI does even better, resulting in 29 pterms, but requires more CPU time:

```
.i 7
.o 3
.p 29
-1----0   010
01---1-   010
---0001   100
0--0-1-   001
0-10-00   010
--10-0-   001
01-1000   001
--11111   001
---00-1   001
-0--001   001
--1-010   001
1-10---   100
---0010   011
0-01-00   010
---0100   101
0-0100-   001
-000-10   001
11----0   100
11---0-   010
1-1--10   010
-111-1-   010
-10001-   010
-100-11   001
-11--01   100
-11-00-   010
1----0-   100
10----1   100
---1000   010
-01--00   100
.e
```

The validity of these minimized truth tables can be easily verified by running a complete set of 128 possible input vectors through *plasim*, using each of the truth tables in turn, and then comparing the output vectors which result. When this was done they were found to be identical.

Hardware design-rule checker

One approach to checking geometric design rules involves rasterizing the layout and then checking properties of the resulting set of pixels. A 4x4 window is sufficient to check most design rules. (This approach was used in software at M.I.T.; see [Baker80] for details). Since NMOS uses no more than 7 mask layers, the logical fuction relating the errors to the pixel map requires no more than 112 (= 7x16) inputs to cover all possible combinations of layers. It would be nice if we could exploit the inherent parallelism of this problem by building a single chip that would check all the design rules at once.

One possible system architecture would have an external CPU writing a 32-bit word into the chip representing a 4x1 set of pixels. The three previous 4x1 sets of pixels would be kept on chip in a shift register arrangement; each time a new word was written, the oldest of the four would be discarded. The computation would take place in parallel and the system would then read back a 32-bit word which would be all zeros if there were no violations. The system thus consists of 112 bits of static storage (or 128 if we want to use the full 8 bits per pixel), plus a large block of combinational logic. We can implement the logic in a single large PLA if we desire, although in practice it might be better to have several smaller PLAs.

As a first investigation of the practicality of this approach, we coded all the design rules corresponding to single-layer width and spacing into .eqn format. Note that these rules have not been extensively verified and may still contain bugs.

```
INORDER =
d00 i00 b00 p00 c00 m00 g00 x00
d01 i01 b01 p01 c01 m01 g01 x01
d02 i02 b02 p02 c02 m02 g02 x02
d03 i03 b03 p03 c03 m03 g03 x03
d10 i10 b10 p10 c10 m10 g10 x10
d11 i11 b11 p11 c11 m11 g11 x11
d12 i12 b12 p12 c12 m12 g12 x12
d13 i13 b13 p13 c13 m13 g13 x13
d20 i20 b20 p20 c20 m20 g20 x20
d21 i21 b21 p21 c21 m21 g21 x21
d22 i22 b22 p22 c22 m22 g22 x22
d23 i23 b23 p23 c23 m23 g23 x23
d30 i30 b30 p30 c30 m30 g30 x30
d31 i31 b31 p31 c31 m31 g31 x31
d32 i32 b32 p32 c32 m32 g32 x32
d33 i33 b33 p33 c33 m33 g33 x33;


OUTORDER =
wd2 wi2 wb2 wp2 wc2 wm3 wg3 wx
sd3 si2 sb2 sp2 sc2 sm3 sg3 sx
sdp sie sbd sbp sbc sgd sgp sgc
xi xb xmc xmg;


wd2 = d11 &
      ((!d10&!d12) |
       (!d21&!d01) |
       (!d00&!d22&((d10&d01)|(d10&d12)|(d01&d21)|(d12&d21))) |
       (!d20&!d02&((d12&d01)|(d10&d12)|(d01&d21)|(d10&d21))) );


wi2 = i11 &
      ((!i10&!i12) |
       (!i21&!i01) |
       (!i00&!i22&((i10&i01)|(i10&i12)|(i01&i21)|(i12&i21))) |
       (!i20&!i02&((i12&i01)|(i10&i12)|(i01&i21)|(i10&i21))) );


wb2 = b11 &
      ((!b10&!b12) |
       (!b21&!b01) |
       (!b00&!b22&((b10&b01)|(b10&b12)|(b01&b21)|(b12&b21))) |
       (!b20&!b02&((b12&b01)|(b10&b12)|(b01&b21)|(b10&b21))) );
```

```
wp2 = p11 &
      ((!p10&!p12) |
       (!p21&!p01) |
       (!p00&!p22&((p10&p01)|(p10&p12)|(p01&p21)|(p12&p21))) |
       (!p20&!p02&((p12&p01)|(p10&p12)|(p01&p21)|(p10&p21))) );


wc2 = c11 &
      ((!c10&!c12) |
       (!c21&!c01) |
       (!c00&!c22&((c10&c01)|(c10&c12)|(c01&c21)|(c12&c21))) |
       (!c20&!c02&((c12&c01)|(c10&c12)|(c01&c21)|(c10&c21))) );


wm3 = (m11 &
       ((!m10&!m12) |
        (!m21&!m01) |
        (!m00&!m22&((m10&m01)|(m10&m12)|(m01&m21)|(m12&m21))) |
        (!m20&!m02&((m12&m01)|(m10&m12)|(m01&m21)|(m10&m21))) ) ) |
      ((m11&m12&m21&m22) &
       ((!m01&!m31) |
      (!m02&!m32) |
      (!m01&!m32) |
      (!m02&!m31) |
      (!m10&!m13) |
      (!m20&!m23) |
      (!m10&!m23) |
      (!m20&!m13) |
      (!m00&(m10|m20)&(m01|m02)&!(m03&m13&m23&m33&m32&m31&m30)) |
      (!m00&(m10|m20)&(m13|m23)&!(m33&m32&m31&m30)) |
      (!m00&(m31|m32)&(m01|m02)&!(m03&m13&m23&m33)) |
      (!m03&(m13|m23)&(m01|m02)&!(m00&m10&m20&m30&m31&m32&m33)) ) );
```

```
wg3 = (g11 &
        ((!g10&!g12) |
         (!g21&!g01) |
         (!g00&!g22&((g10&g01)|(g10&g12)|(g01&g21)|(g12&g21))) |
         (!g20&!g02&((g12&g01)|(g10&g12)|(g01&g21)|(g10&g21))) ) ) |
       ((g11&g12&g21&g22) &
        ((!g01&!g31) |
       (!g02&!g32) |
       (!g01&!g32) |
       (!g02&!g31) |
       (!g10&!g13) |
       (!g20&!g23) |
       (!g10&!g23) |
       (!g20&!g13) |
       (!g00&(g10|g20)&(g01|g02)&!(g03&g13&g23&g33&g32&g31&g30)) |
       (!g00&(g10|g20)&(g13|g23)&!(g33&g32&g31&g30)) |
       (!g00&(g31|g32)&(g01|g02)&!(g03&g13&g23&g33)) |
       (!g03&(g13|g23)&(g01|g02)&!(g00&g10&g20&g30&g31&g32&g33)) ) );


wx = ZERO;


sd3 = (!d11 &
        ((d10&d12) |
         (d21&d01) |
         (d00&d22&
        ((!d10&!d01)|(!d10&!d12)|(!d01&!d21)|(!d12&!d21))) |
         (d20&d02&
        ((!d12&!d01)|(!d10&!d12)|(!d01&!d21)|(!d10&!d21))) ) ) |
       ((!d11&!d12&!d21&!d22) &
        ((d01&d31) |
       (d02&d32) |
       (d01&d32) |
       (d02&d31) |
       (d10&d13) |
       (d20&d23) |
       (d10&d23) |
       (d20&d13) |
       (d00&(!d10|!d20)&(!d01|!d02)&
        !(!d03&!d13&!d23&!d33&!d32&!d31&!d30)) |
       (d00&(!d10|!d20)&(!d13|!d23)&
        !(!d33&!d32&!d31&!d30)) |
       (d00&(!d31|!d32)&(!d01|!d02)&
        !(!d03&!d13&!d23&!d33)) |
       (d03&(!d13|!d23)&(!d01|!d02)&
        !(!d00&!d10&!d20&!d30&!d31&!d32&!d33)) ) );
```

```
si2 = !i11 &
      ((i10&i12) |
       (i21&i01) |
       (i00&i22&
      ((!i10&!i01)|(!i10&!i12)|(!i01&!i21)|(!i12&!i21))) |
       (i20&i02&
      ((!i12&!i01)|(!i10&!i12)|(!i01&!i21)|(!i10&!i21))) );


sb2 = !b11 &
      ((b10&b12) |
       (b21&b01) |
       (b00&b22&
      ((!b10&!b01)|(!b10&!b12)|(!b01&!b21)|(!b12&!b21))) |
       (b20&b02&(
      (!b12&!b01)|(!b10&!b12)|(!b01&!b21)|(!b10&!b21))) );


sp2 = !p11 &
      ((p10&p12) |
       (p21&p01) |
       (p00&p22&
      ((!p10&!p01)|(!p10&!p12)|(!p01&!p21)|(!p12&!p21))) |
       (p20&p02&
      ((!p12&!p01)|(!p10&!p12)|(!p01&!p21)|(!p10&!p21))) );


sc2 = !c11 &
      ((c10&c12) |
       (c21&c01) |
       (c00&c22&
      ((!c10&!c01)|(!c10&!c12)|(!c01&!c21)|(!c12&!c21))) |
       (c20&c02&
      ((!c12&!c01)|(!c10&!c12)|(!c01&!c21)|(!c10&!c21))) );
```

```
sm3 = (!m11 &
       ((m10&m12) |
        (m21&m01) |
        (m00&m22&
       ((!m10&!m01)|(!m10&!m12)|(!m01&!m21)|(!m12&!m21))) |
        (m20&m02&
       ((!m12&!m01)|(!m10&!m12)|(!m01&!m21)|(!m10&!m21))) ) ) |
      ((!m11&!m12&!m21&!m22) &
       ((m01&m31) |
      (m02&m32) |
      (m01&m32) |
      (m02&m31) |
      (m10&m13) |
      (m20&m23) |
      (m10&m23) |
      (m20&m13) |
      (m00&(!m10|!m20)&(!m01|!m02)&
       !(!m03&!m13&!m23&!m33&!m32&!m31&!m30)) |
      (m00&(!m10|!m20)&(!m13|!m23)&
       !(!m33&!m32&!m31&!m30)) |
      (m00&(!m31|!m32)&(!m01|!m02)&
       !(!m03&!m13&!m23&!m33)) |
      (m03&(!m13|!m23)&(!m01|!m02)&
       !(!m00&!m10&!m20&!m30&!m31&!m32&!m33)) ) );
```

```
sg3 = (!g11 &
       ((g10&g12) |
        (g21&g01) |
        (g00&g22&
       ((!g10&!g01)|(!g10&!g12)|(!g01&!g21)|(!g12&!g21))) |
        (g20&g02&
       ((!g12&!g01)|(!g10&!g12)|(!g01&!g21)|(!g10&!g21))) ) ) |
      ((!g11&!g12&!g21&!g22) &
       ((g01&g31) |
      (g02&g32) |
      (g01&g32) |
      (g02&g31) |
      (g10&g13) |
      (g20&g23) |
      (g10&g23) |
      (g20&g13) |
      (g00&(!g10|!g20)&(!g01|!g02)&
       !(!g03&!g13&!g23&!g33&!g32&!g31&!g30)) |
      (g00&(!g10|!g20)&(!g13|!g23)&
       !(!g33&!g32&!g31&!g30)) |
      (g00&(!g31|!g32)&(!g01|!g02)&
       !(!g03&!g13&!g23&!g33)) |
      (g03&(!g13|!g23)&(!g01|!g02)&
       !(!g00&!g10&!g20&!g30&!g31&!g32&!g33)) ) );
```

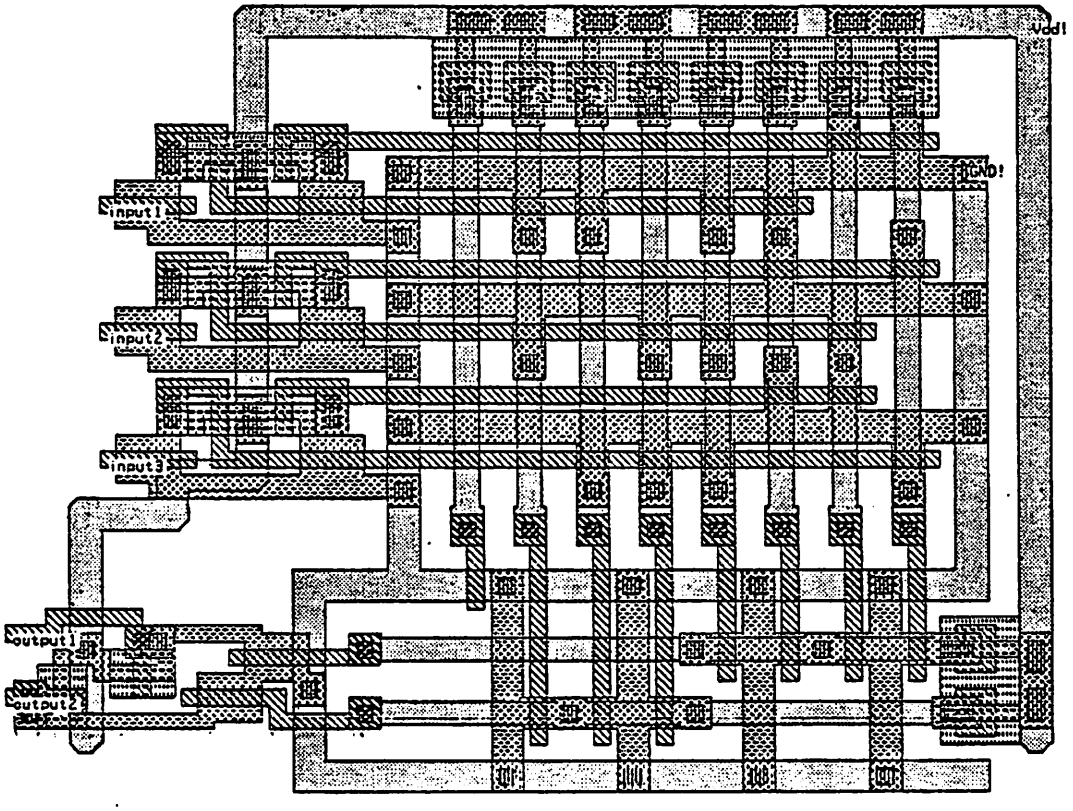
```
sx  = ZERO;
sdp = ZERO;
sie = ZERO;
sbd = ZERO;
sbp = ZERO;
sbc = ZERO;
sgd = ZERO;
sgp = ZERO;
sgc = ZERO;
xi  = ZERO;
xb  = ZERO;
xmc = ZERO;
xmg = ZERO;
```

The resulting truth table has 128 inputs, 620 pterms, and 28 outputs, and is too large to be usefully reproduced here: Optimizing this proves to be difficult due to the vast amount of computation required; even *presto* was unable to finish in 11 hours of CPU time. Partitioning the PLA into smaller modules would have helped, especially since some of them could have been identical (for example, a PLA to check 3-lambda spacing rules on a single layer could be used for diffusion, metal and overglass.) but it is also quite possible to go ahead and generate a PLA without optimization. *Mkpla* only requires 43.1 seconds of VAX CPU time to lay out this PLA, including issuing numerous error messages about unused inputs and unset outputs. The power and ground busses have to be made 46 lambda wide, providing a severe test of the cell-stretching mechanism, and extra metal ground lines are inserted into the AND and OR planes as needed.

Appendix O : Effects of Some of the Options to Mkpla

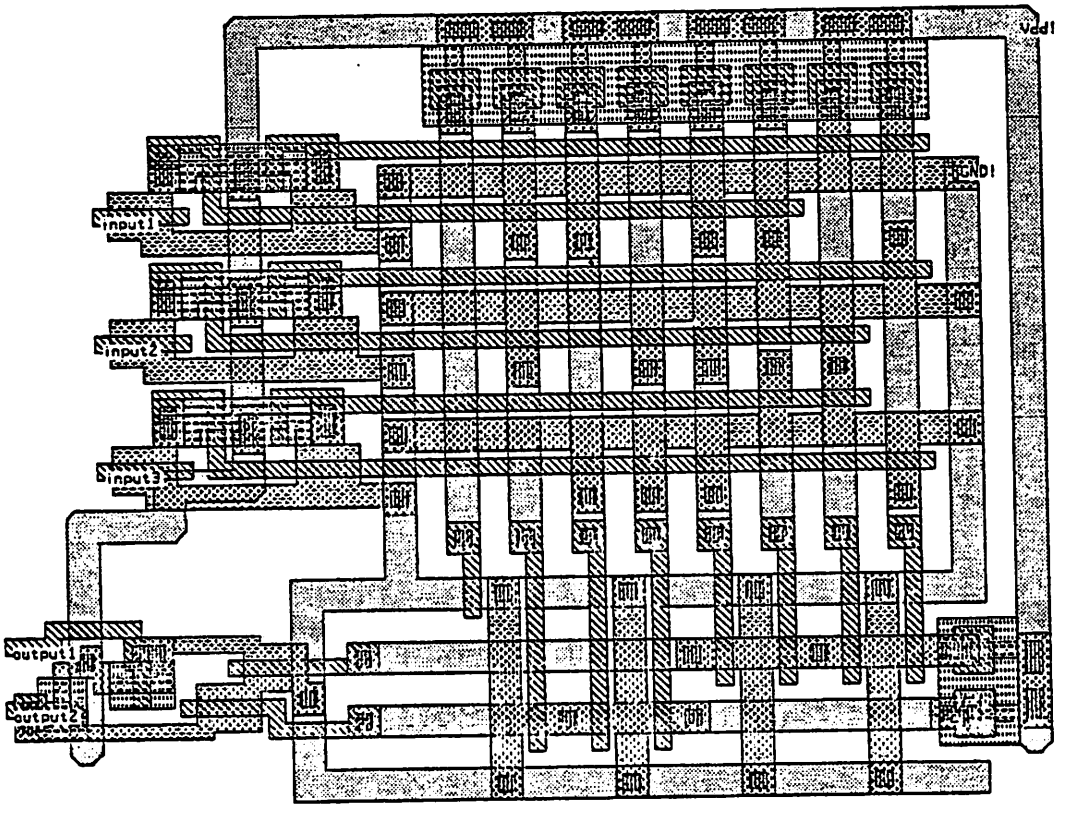
mkpla

This is the adder PLA generated with the default options. They include $\lambda = 200$ (2.0μm), truncated poly lines, and 3λ-wide metal lines in the planes. No extra ground lines are needed and the power and ground busses are not widened.



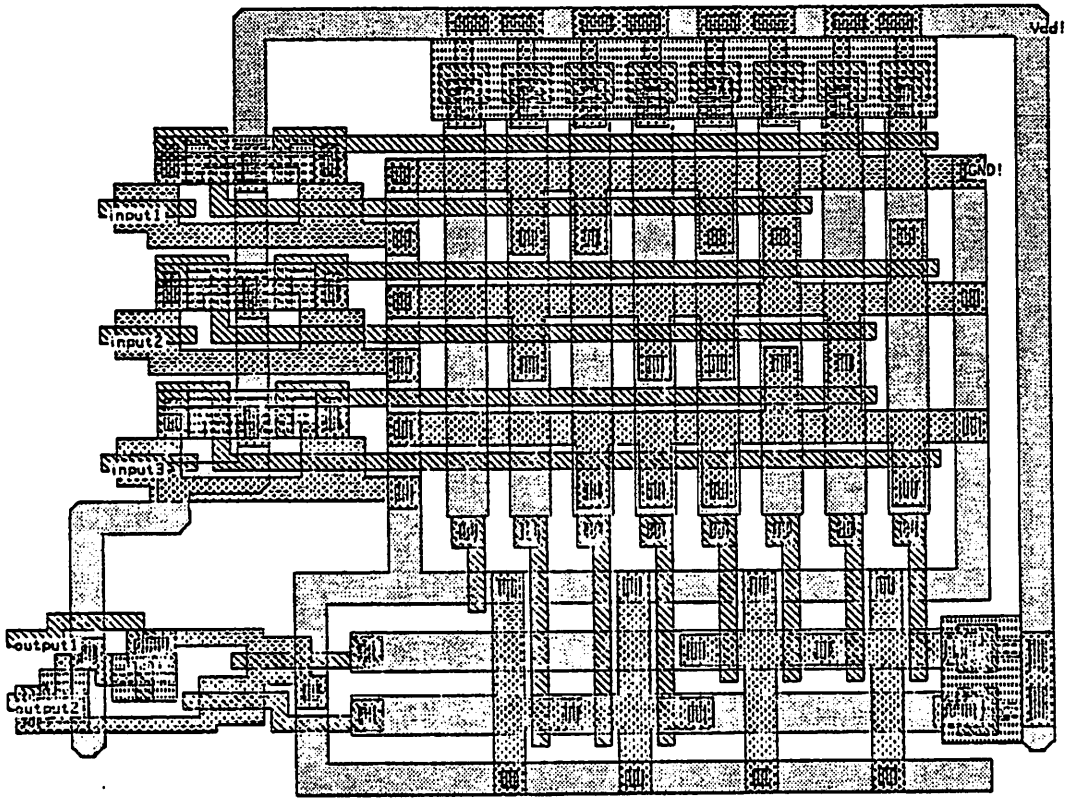
mkpla -C4

The adder PLA with 4λ-wide metal lines, as were found in the MPC79 PLA cells. This results in higher capacitance and hence slower operation but may be marginally more reliable.



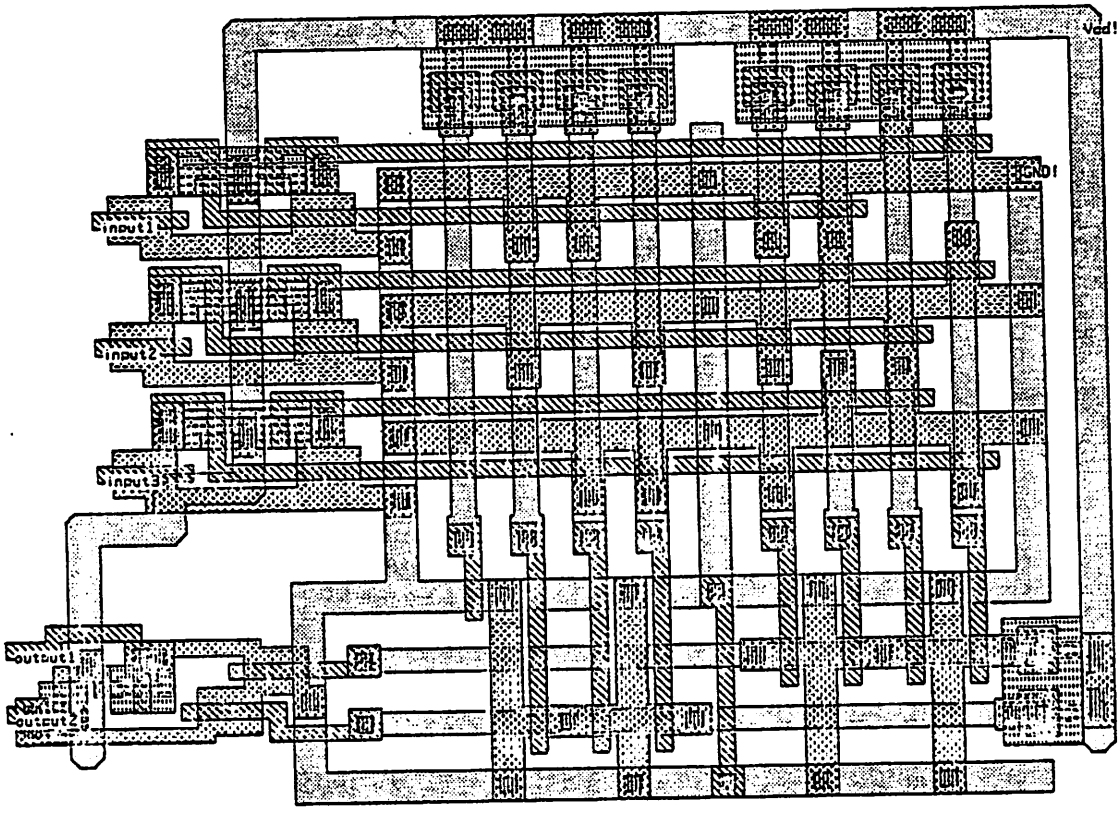
mkpla -C5

The adder PLA with 5λ-wide metal lines. There is no reason to use this option in an actual circuit, but this is as wide as the metal lines can get without violating the metal spacing design rule.



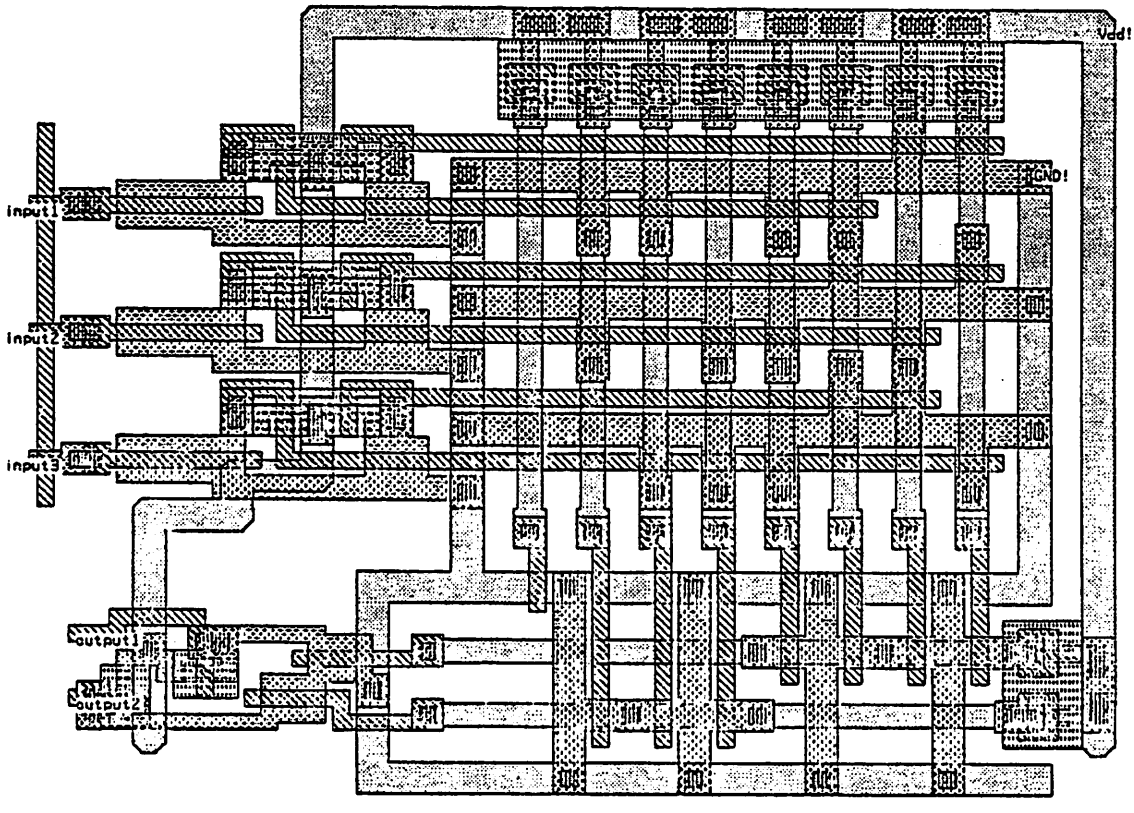
mkpla -G4

The adder PLA with extra ground lines forced every 4 terms. One extra ground line appears in the AND-plane.



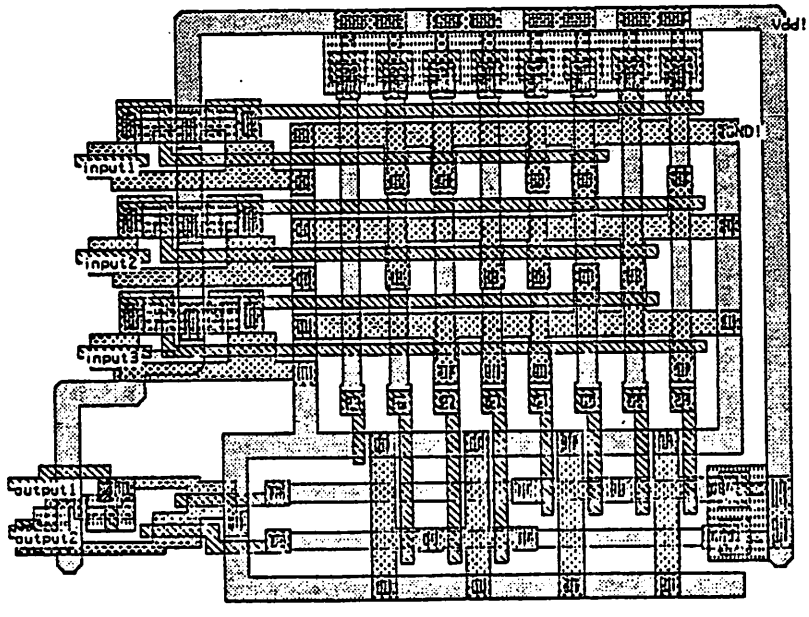
mkpla -i

The adder PLA with dynamically clocked inputs.



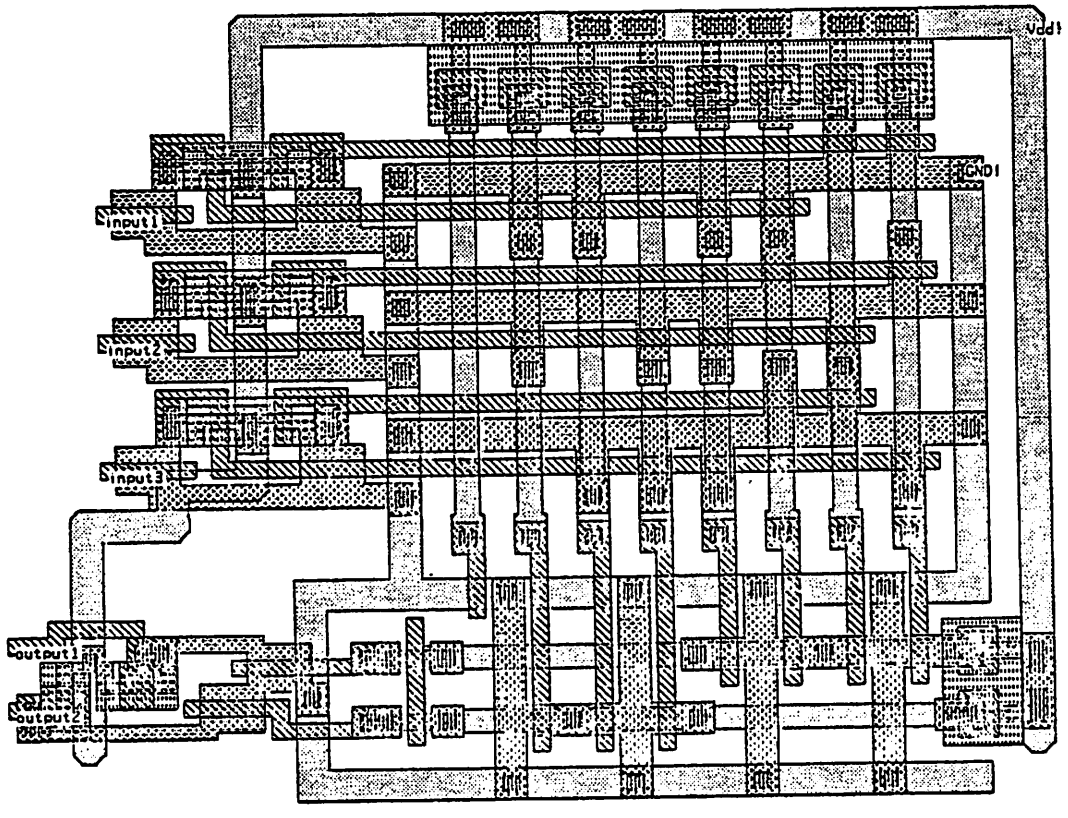
mkpla -l150

The adder PLA at a different (smaller) value of λ (150 CIF units = 1.5μm).



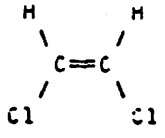
mkpla -o

The adder PLA with dynamically clocked outputs. Note that this doesn't change the size or shape of the PLA because sufficient space for the output clock lines is always reserved even if they are not used.

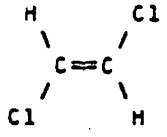


mkpla -t

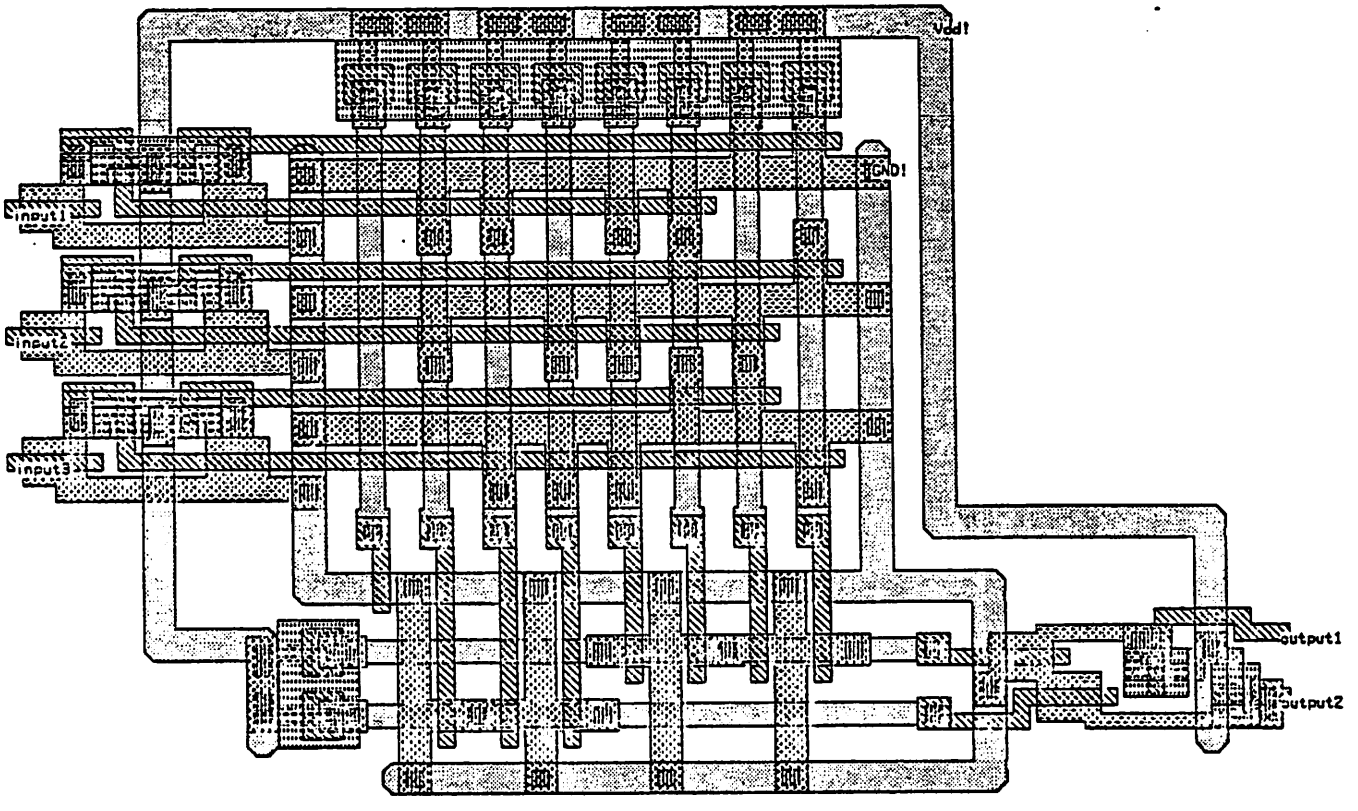
The adder PLA with the outputs coming out on the opposite ("trans") side. The terminology "cis" and "trans" is derived from organic chemistry, where it is used to distinguish geometric isomers, e.g.:



cis-dichloroethylene

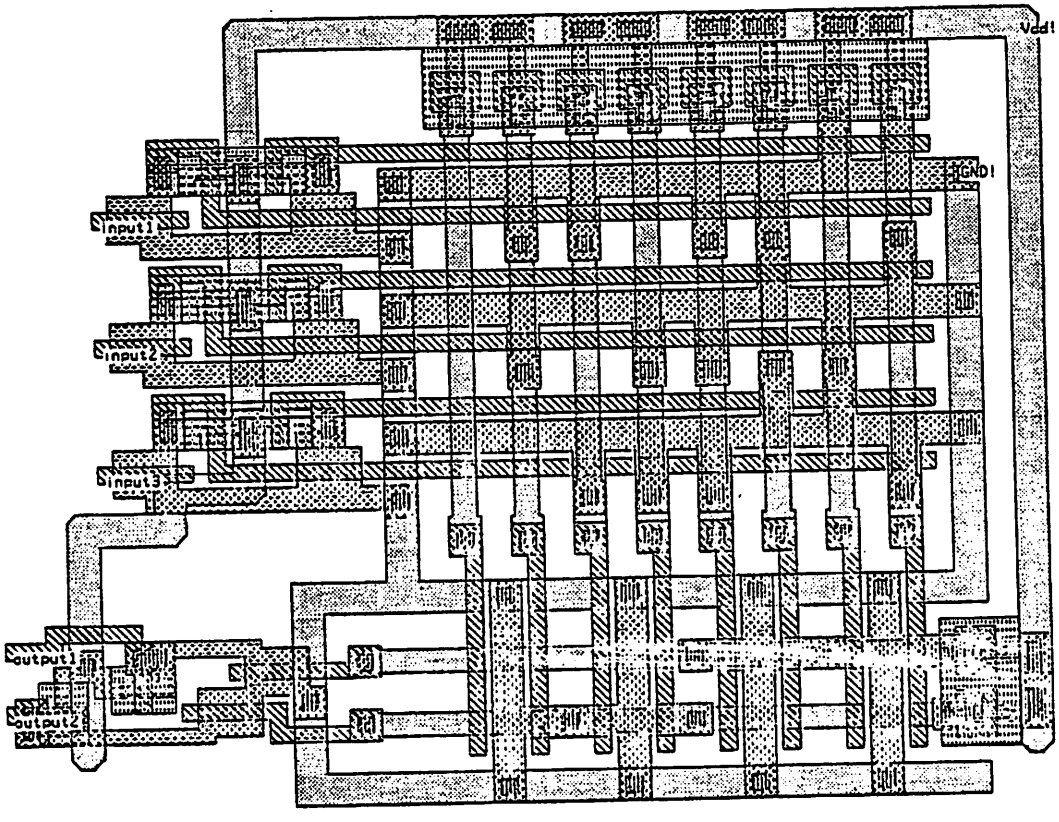


trans-dichloroethylene



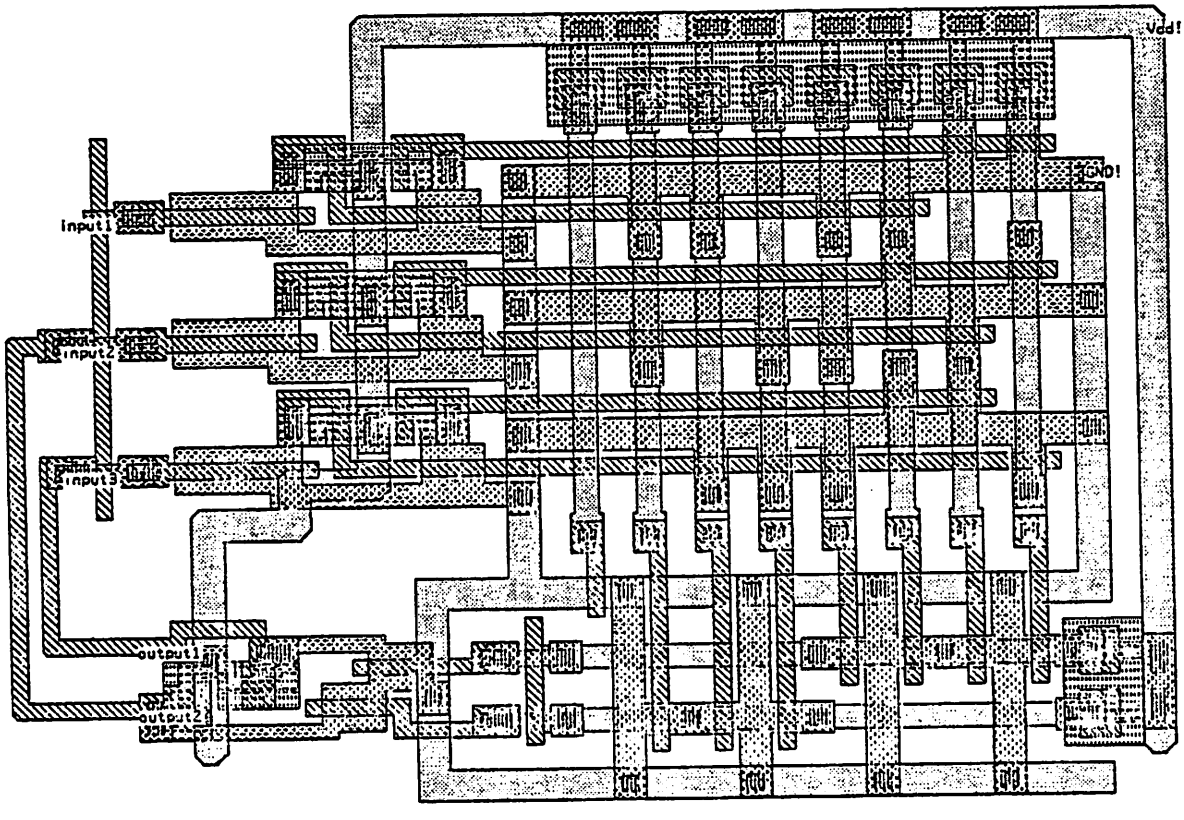
mkpla -x

The adder PLA with the poly lines extended. This option is useful if you need to use pterm or decoded input signals outside of the PLA, or if you want the PLA to look more like an MPC79-style PLA.



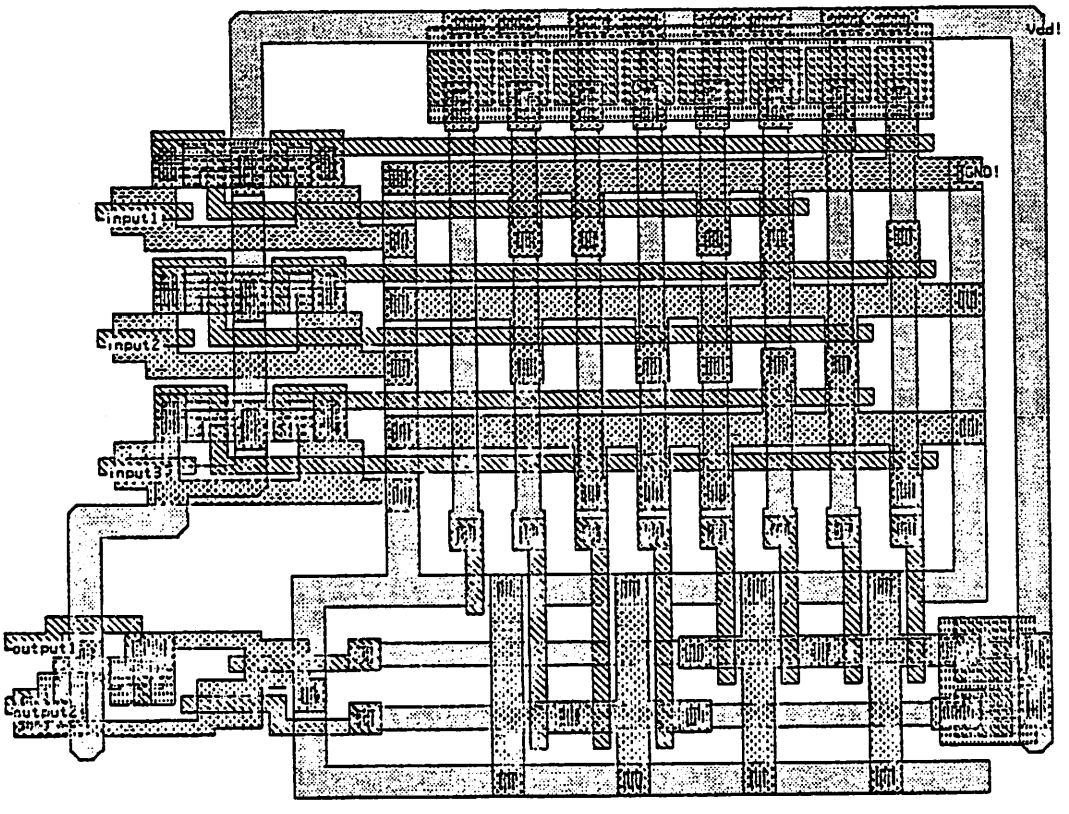
mkpla -i -o -y2

The adder PLA with two outputs fed back to the inputs to make a finite state machine. Note that clocking of the inputs and outputs should also be specified whenever the -y option is used, and that the -t option cannot be used if the -y option is.



mkpla -z

The adder PLA with the pullups lengthened to reduce power consumption. Only the AND- and OR-plane pullup pairs are affected; this option doesn't yet alter the input and output buffers.



Bibliography

[Arnold81]

Arnold & Ousterhout "Lyra: A New Approach to Geometric Layout Rule Checking", paper to appear at 1982 Design Automation Conference.

[Ayres79]

Ayres, Ron, "Silicon Compilation - A Hierarchical Use of PLAs", *Caltech Conference on VLSI*, January 1979

[Baker80]

Baker & Terman, "Tools for Verifying Integrated Circuit Designs", *Lambda* 1:3, 4th quarter 1980, pp. 22-30.

[Bell78]

Bell, Mudge, & McNamara, *Computer Engineering, a DEC view of hardware systems design*, Digital Press 1978

[Beyers81]

Beyers, Dohse, Fucetola, Kochis, Lob, Taylor, & Zeller, "A 32-Bit VLSI Chip", *IEEE Journal of Solid State Circuits* SC-16:5, October 1981, pp. 537-542.

[DeVries75]

DeVries & Svoboda "Multiple Output Optimization with Mosaics of Boolean Functions", *IEEE Transactions on Computers* C-24:8, August 1975, pp. 777-784.

[Duttweiler80a]

Duttweiler & Chen, "A single-chip VLSI echo canceler", *Bell System Technical Journal*, February 1980

[Duttweiler80b]

Duttweiler, Donald L., "Bell's echo-killer chip", *IEEE Spectrum* 17:10, October 1980, pp. 34-37.

[Fitzpatrick81]

Fitzpatrick, Foderaro, Katevenis, Landman, Patterson, Peek, Peshkess, Séquin, Sherburne, & Van Dyke, "VLSI Implementations of a Reduced Instruction Set Computer", *CMU Conference on VLSI Systems and Computations*, October 1981. Another version of this article, with better pictures and artwork, appeared as "A RISCy Approach to VLSI", *VLSI Design*, 4th qtr 1981, pp. 14-20.

[Foderaro81]

Foderaro, John K., personal communication, July 1981

[Garey79]

Garey & Johnson, *Computers and Intractability, a Guide to the Theory of NP-completeness*, W. H. Freeman & Co., San Francisco, 1979

[Hofmann80]

Hoffman, Mark, *A Method for Topological Compaction of Programmed Logic Arrays*, master's thesis and ERL memo, U.C. Berkeley, 1980

[Hon80]

Hon & Séquin, *A Guide to LSI Implementation, 2nd Ed.*, Xerox Palo Alto Research Center, 1980

[Hong74]

Hong, Cain, & Ostapko, "MINI: A Heuristic Approach for Logic Minimization", *IBM Journal of Research and Development*, 18:5, September, 1974, pp. 443-457.

[Keller80]

Keller, Ken *Tutorial for KIC 2 - A Graphics Editor for Integrated Circuits*, master's report, U.C. Berkeley, 1980

[Mead80]

Mead & Conway, *Introduction to VLSI Systems*, 1981

[Ousterhout81]

Ousterhout, John K. "Caesar: An Interactive Editor for VLSI Layouts", *VLSI Design*, 4th qtr 1981, pp. 34-38. Note that figures 1b) and 1d) were switched in printing.

[Schmookler80]

Schmookler, Martin, "Design of Large ALUs Using Multiple PLA Macros", *IBM Journal of Research and Development*, 24:1, January 1980, pp. 2-14.