MEXTRA: A MANHATTAN CIRCUIT EXTRACTOR

by

D. T. Fitzpatrick

Memorandum No. UCB/ERL M82/42

21 May 1982

MEXTRA: A MANHATTAN CIRCUIT EXTRACTOR

by

Daniel T. Fitzpatrick

Memorandum No. UCB/ERL M82/42

21 May 1982

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

MEXTRA: A Manhattan Circuit Extractor

Daniel T. Fitzpatrick

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTACT

This report describes the use and implementation of MEXTRA, a circuit extractor that reads a restricted subset of CIF and produces a circuit description suitable for switch level simulation. MEXTRA allows a designer to specify local and global symbolic names for nodes by placing labels in his layout description. MEXTRA has an approximately linear growth rate and is an order of magnitude faster than the CIFPLOT extractor.

May 15, 1982

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

MEXTRA: A Manhattan Circuit Extractor

Daniel T. Fitzpatrick

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

Introduction

In the design of large integrated circuit systems there are many opportunities to make mistakes. Simulation of the design can catch many mistakes and allows the mistakes to be fixed before the chip is sent for fabrication. If the design is done at the layout level, the underlying circuit must be extracted from the layout description. Simulation can then be driven from this extracted circuit. For large, complex designs this step is necessary to insure the correctness of the design.

This report describes MEXTRA, a circuit extractor that reads an integrated circuit layout description in Caltech Intermediate Form (CIF 2.0)[1] and creates a circuit description, which can be used to perform various electrical checks. The circuit description is directly compatible with MOSSIM, a switch level simulator, MOSERC, a static electrical rules checker, and POWEST, a program that estimates the power consumption of circuits[2]. SIM2SPICE can be used to convert the MEXTRA circuit description to a circuit description suitable for SPICE simulation.

MEXTRA is very similar to the CIFPLOT -X circuit extractor[2], but much faster. Its average growth rate is linear in the area of the circuit and the number of rectangles, and its execution speed is an order of magnitude faster than the CIFPLOT extractor. MEXTRA, however, is not as general as CIFPLOT. It handles only manhattan geometry, and it can read only a subset of CIF. Yet these restrictions seem to be well worth the significant speed up.

Usage

The command line to run MEXTRA looks as follows:

```
    mextra basename
```

MEXTRA will read the file *basename*.cif and create 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. Also it contains messages about possible errors. The '.al' file is a list of aliases for node names that can be used by MOSSIM. 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 basename.nodes file must be called first since it does not contain a CIF 'End' statement.

The '.sim' file is the circuit description. Contained in the '.sim' file is a list of transistors and capacitors. To do a static electrical rules check this file can be run through MOSERC. The form of the command is:

```
moserc basename.sim
```

This will print out various messages about possible electrical rule violations. To do a switch level simulation with MOSSIM, the command is of the form:

```
mossim basename.sim
```

(See [3] and [4] for a description of MOSSIM.)

Names

MEXTRA allows the user to specify his own names for nodes in the circuit. If no name is specified, a number is assigned to the node. The CIF extension command '94' is used to name nodes. The form of the CIF command is as follows:

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

This command attaches the name to the mask geometry on the specified layer crossing the point (x,y). If no layer is specified then any mask geometry crossing the point is given the name. The name may contain any ASCII character except space, tab, newline, double quotes, parenthesis, and semi-colon. To avoid conflict with extractor generated names, names should not be numbers or end in '#n', where n is a number. It is important that the (x,y) coordinate lie *within* the mask geometry. If the name lies right on the edge, it may not get attached to the desired node.

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 that is repeated many time. 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 named '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 starting at 0. In the shift register example, the names 'SR.in#0' through 'SR.in#7' are assigned. Global names never have suffixes appended to them. Thus unconnected nodes with the same global name will appear connected after extraction. Names are made local by ending them with a sharp sign, '#'. Names are made global by ending them with an exclamation mark, '!'. These terminating characters are not considered part of the name, however. Names that 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 has only one occurrence. It is illegal to have a name that is declared both local and global. The extractor will complain if this happens and make the name local. If a name is declared local or global, any unspecified occurrence takes on the specified type, i.e. local or global.

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 of the same type, then the extractor takes the shortest name. In the '.al' file the extractor lists the names of any node that has more than one name attached to it. Each line starts with an equal sign, which is followed by a list of names. The '.al' file is readable by MOSSIM so that it will understand these aliases.

Transistors

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

type gate source drain length width xloc yloc

Type can be one of three characters, 'e' for enhancement, 'd' for depletion, or 'u' for unusual implant. Unusual implant refers to transistors that are only partially in an implanted area. (These are not common and may be a design error. If this type of transistor was intended, it will be necessary to write a filter to replace these transistors with the appropriate model in terms of enhancement and depletion transistors.) *Gate, source,* and *drain* are the names of the gate, source, and drain nodes of the transistor. *Length,* and *width* are the channel length and width in CIF units. *Xloc,* and *yloc* are the x and y CIF coordinates of the bottom left corner of the transistor. (e. g. the minimum x and y coordinates of the transistor.)

The extractor guesses the length and width of a transistor by knowing the area, the perimeter, and the length of diffusion terminals. For rectangular transistors, and pullup transistors with a butting contact the reported length and width is correct. 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' files in the form:

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

The 'f' is followed by the type that is the same as the type for normal transistors, 'e', 'd', or 'u'. *Gate* is the poly node covering the transistor region. *Node1* through *nodeN* are the diffusion terminal nodes. *Xloc* and *yloc* are the x and y coordinates of the transistor. 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 diffusion transistors.

Capacitance

The '.sim' file also has information about capacitance in the circuit. Capacitance can come from two sources. Capacitance between a node and substrate, called node capacitance, and capacitance caused by poly overlapping diffusion but not forming a transistor, called poly/diff capacitance. Capacitance of transistors are not included since most of the tools that work on the .sim file calculate the transistor capacitance from the width and length information. The lines containing node capacitance information are of the form:

C *node1* GND *cap-value*

The form of the poly/diff capacitance information is:

C *node1 node2 cap-value* (*xloc yloc*)

Cap-value is the capacitance between the nodes in femto-farads. *Xloc* and *yloc* are the (x,y) coordinates of the bottom left corner of the capacitance area.

When calculating the node capacitance, the capacitance for each layer is calculated separately. The layer capacitance is calculated by taking the area of a node on that layer and multiplying it by a constant representing the capacitance per unit area. This is added to the product of the perimeter and a constant representing edge capacitance per unit length. The node capacitance is the total of the layer capacitances of the node. Node capacitances below a certain threshold are not reported. The default threshold is 50 femto-farads.

Poly/diffusion capacitance is calculated similar to layer capacitance. The area is multiplied by a constant and this is added to the perimeter multiplied by a constant. There is no threshold for the reporting of poly/diff capacitance. The default constants are given below.

layer	*area*	*perimeter*
metal	30	0
poly	50	0
diff	100	100
poly/diff	400	0

Area constants are in atto-farads (10^{-18} farads) per square micron. Perimeter constants are atto-farads per micron.

Changing Default Values

The extractor tries to do a reasonable job with a minimum of help from the user. Often, though, the values it chooses for certain parameters do not suit the needs of some users. This section discusses how to change the default setting of several parameters.

As part of its start up procedure MEXTRA reads two files, '~cad/.cadrc' and the '.cadrc' file in the user's home directory. MEXTRA reads these files to set up constants that it will use later. This allows several program constants to be changed without recompiling. Each line in the '.cadrc' starts with a keyword that tells the program how to process the line. The case of the keyword does not matter.

Unknown keywords are ignored. This allows several programs to share the '.cadrc' files.

One common set of parameters that a user would like to change is the constants used to compute the node capacitances. These values are dependent on the particular line that is to fabricate the chip. These values can be changed by entering in the .cadrc file lines of the following form:

  areatocap *layer value*

and

  perimtocap *layer value*

The value for area is in atto-farads per square micron and for perimeter it is in atto-farads per micron. By putting the following lines in the '.cadrc' file, the capacitor values are set to the values given in Mead and Conway[5].

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

As noted above, any node capacitance below 50 femto-farads is ignored. The user may want to change this value up or down depending on how important capacitance is in his circuit. This threshold value can be changed with an entry in the '.cadrc' file of the form:

  capthreshold *value*

Value is in femto-farads. A negative value is equivalent to setting the threshold to infinity.

By default, MEXTRA reports locations in CIF coordinates. Many designers, though, prefer to think in another set of units, such as microns or lambda units. While debugging a chip it is often necessary to look at the geometry near these coordinates. Finding these coordinates with a graphics editor usually involves a clumsy conversion from CIF units to the units used by the graphics editor. It is possible to change the coordinate system by including a line in the '.cadrc' file of the following form:

  units *scale*

This line makes the basic unit equal to *scale* centi-microns. The following line sets the basic unit to microns:

```
units 100
```

The coordinates in the '.nodes' file are kept in CIF units since it is a CIF file. The units can also be set on the command line using the -u option. The following command line sets the basic unit to microns:

```
mextra -u 100 file
```

Setting the units on the command line overrides the units specified in the '.cadrc' file.

Implementation

MEXTRA was written to fill a need for a fast circuit extractor for the RISC project. As such, very definite goals were set. It should be able to extract large chips in a reasonable amount of time. It should fit in well with the other tools. It should allow user supplied names. It should check for obvious errors, and report them in an easy to read fashion. Finally, it should be up and running as soon as possible.

In having such definite goals set, the generality of the extractor beyond the RISC project was not considered of major importance. Thus, it was decided that the extractor would handle only manhattan geometry and that it would read only a subset of the CIF language. This allowed the extractor to be faster, since checks for non-manhattan features could be eliminated. Also the parser could use a simple parsing scheme, which did not worry about the more troublesome features of CIF. Since the problem was simpler, implementation time was shortened. In addition, a simpler problem made it easier to concentrate on efficient algorithms and to find bottlenecks in the program.

MEXTRA is not a very big program; it is less than 6000 lines long, which can be broken down approximately as follows: 900 lines for utility routines, 350 lines for the scanner, 1300 lines for interpreter, 1400 lines for dealing with node names and numbers, 1000 lines for sorting and the geometry extractor, and the remainder is for control, initialization, and user interface. The utility routines, and the scanner/interpreter routines were made into libraries that have proved useful in a number of other programs, such as CIFPLOT, CIFSTAT, and CLIPCIF[2].

MEXTRA is actually made up of two programs. The first program reads the CIF file, sorts it, does the extraction, and then calls the second program. The second program replaces internal node numbers assigned during extraction, with either a user supplied name or a final node number. The next several paragraphs describe the implementation of MEXTRA.

The Scanner

The scanner is a simple, general purpose token scanner. The scanner assumes that the input is structured into a list of commands. Each command is terminated with a semi-colon. The command is made up of blank-separated tokens, where the first token is the keyword for the command. The lowest level is the *Input Module.* This module reads characters from the input file and puts them into the line buffer. This module also keeps track of the name of the file and the number of the line currently being read. Text enclosed in parenthesis is ignored. (Nesting of parenthesis is allowed.) A command is considered terminated upon reading a semi-colon or upon reaching the end-of-file.

The next level is the *Scanner Module.* This module breaks the line buffer into blank-separated tokens. (A blank may appear in a token only by enclosing text in a quote string. The quote marks are stripped off by the scanner.) This module maintains a KeywordTable that contains a list of keywords and associated functions. The first token in each command is compared against the keywords in the KeywordTable. If a match is found, the associated function is called with the number of tokens in the line and an array of character strings where the strings are the tokens of the command line. If no match is found, the default function is called. The KeywordTable is set by calling KeyScanner(keyword,function). Each call to KeyScanner places a new keyword and its associated function into the KeywordTable.

The Interpreter

The input module and the scanner module aid in breaking the input into tokens, yet neither module knows anything about CIF. The *Interpreter Module* contains the CIF specific information. The interpreter starts off by initializing the scanner with CIF keywords. The initialization routine looks like this:

```
InitScanner();
DefaultScanner(Unknown);
KeyScanner("DS",DefineSymbol);
KeyScanner("DF",DefineFinish);
KeyScanner("L",DefineLayer);
KeyScanner("B",DefineBox);
KeyScanner("P",DefinePolygon);
KeyScanner("W",DefineWire);
KeyScanner("C",DefineCall);
KeyScanner("94",DefineLabel);
KeyScanner("9",DefineSymbolName);
Scanner();
```

Each routine, DefineSymbol, DefineLayer, DefineBox, etc., takes appropriate action for the type of command it is to receive. The routine Unknown prints an error message.

Each object, boxes, polygons, calls, etc., is either in a definition or is called at the top level. Those objects called at the top level are placed in a pseudo-symbol. These objects can then be treated like any other object. A symbol consists of a name, number, bounding box, and a list of objects. Each symbol is placed in the SymbolTable according to its symbol number.

After reading the CIF file, the interpreter computes the bounding boxes of all the symbols, and checks to see that there are no recursive calls. The bounding box of the entire layout is also computed.

The extractor is not strongly tied into the CIF language. It would be relatively easy to retarget the scanner/interpreter for a different language than CIF, as long as the new language could be broken down into tokens. For instance, CAESAR format[6] would probably fit well into the scheme described above. Most of the changes needed would be in the interpreter module. It would have to load the keyword table with different strings. Special characters recognized by the scanner, such as parenthesis and semi-colon, could be changed by calls to the function

LoadCharClass(ch,class), which sets values in the character class table used by the scanner.

Instantiating the Geometry

The next step is to begin instantiating the geometry of the layout. There are three basic functions that do the instantiation: SortObject, EnumerateSymbol, and Instantiate. SortObject sorts the objects as it receives them by minimum y onto the ReadyList. SortObject is discussed in more detail later. The function EnumerateSymbol(n,trans,function) transforms each object in symbol n by trans and then calls function with the transformed object. When there are no more objects in symbol n, function is called with NIL.

The third function, Instantiate, does the bulk of the work. It takes the objects off the ReadyList in sorted order and breaks the objects down into more basic objects. If the object is a primitive object, such as a box or polygon, it breaks the object down into edges, and calls SortObject with each of the edges. If the object is an edge or a label, it is inserted to the EventList. If the object is a call, Instantiate calls EnumerateSymbol(n,trans,SortObject), where n is the number of the called symbol and trans is the call's transform. This replenishes the ReadyList.

Initially, of course, the ready list is empty. Therefore, Instantiate must prime the ReadyList. To do this Instantiate puts the objects not in symbol definitions onto the ReadyList. Since the interpreter puts all objects not in a CIF symbol into symbol #-1, Instantiate starts by calling EnumerateSymbol(-1,GlobalTransform,SortObject). GlobalTransform translates the CIF coordinates so that the transformed bounding box begins at the origin.

Instantiate continues taking objects off the ReadyList until the list is empty. When the ReadyList is empty, we know that the layout is fully instantiated. All edges and labels have been put on the EventList. Of course, if we didn't periodically empty out the EventList, large layouts would require unreasonably large amounts of the memory. To avoid this, whenever Instantiate advances in the Y-direction, it calls MoreEdges() which takes elements off the event list.

Sorting

Like many other programs, MEXTRA spends much of its time sorting. It is important, therefore, that sorting be done as efficiently as possible. The sorting method chosen almost exclusively in MEXTRA was bucket sorting. The number of buckets that are used is chosen according to the size of the chip.

The function SortObject uses a bucket structure for sorting. The buckets are 2000 CIF units high. Each bucket is simply an unsorted list, therefore, insertion into a bucket is quite easy. Whenever Instantiate wants a new object, it takes the first object in the lowest bucket. When the bucket is empty, it calls MoreEdges and then advances the pointer to the next lowest bucket.

This method of instantiating causes the event list to become full of edges that would normally fall into one 2000 CIF unit high bucket. Edges are sorted into the event list by the function `SortEdge`. The event list is actually a two level bucket structure. `SortEdge` first determines which of the 2000 possible first level buckets the edge falls into by taking the y-min of the edge modulo 2000. Each of these buckets are composed of sub-buckets, each 2000 CIF units wide. (Thus the number of sub-buckets is also determined by the size of the chip. One might think initially that this would take a great deal of storage, but it turns out that not all 2000 possible first level buckets are used. These buckets are only created when they are needed. For designs based on a lambda grid, where lambda is 200, the number of buckets used is only 10. If a half lambda grid is used twice the number of first level buckets are needed.) Each sub-bucket is a sorted list on x. The appropriate sub-bucket for an edge is determined by dividing the x-value of the edge by 2000. The edge is then sorted into the appropriate sub-bucket. It is now relatively easy to pull off edges in sorted order.

Node Propagation

A moving line algorithm is used to propagate node numbers along electrical paths. The chip is cut into horizontal swaths coinciding with the vertices of the geometry within the chip. For each swath we create *node segments*, which indicated horizontal runs that are covered by the geometry of the swath. Node segments are used to pass node information from one swath to the next. Given two adjacent swaths, by comparing the node segments of each swath we can propagate node information from one swath to the next. Node information stored with the segments at the bottom of the swath can be passed to the top of the swath by noting when segments in the two swaths overlap.

As an example, consider the geometry shown in figure 1a. The area between the three dashed lines represent two adjacent swaths. Figure 1b provides a closer look at the two swaths of figure 1a. Node numbers are passed up from swath A to swath B. By running down the segment list for swath A, we see that the first segment overlaps the first segment in B. We assign node number 17 to the first segment of B. We also find that the second segment in B is overlapped by the second and third segments of A. The node numbers 43 and 8 are merged and the second segment of B receives the smaller number, 8. Finally, the third segment of B is overlapped by no one. We generate a new node number for this segment.

During the node propagation phase we record the area and perimeter information for the layers metal, poly, diffusion, and gate. This information is used to compute node capacitance, and is used to determine the length and width of transistors. The gate layer is created whenever poly crosses diffusion (and there is no buried contact present). For each gate region we create a transistor record in which we store the node numbers of any adjacent diffusion and the overlapping poly. This gives us the connections to the drain, source, and gate of the transistor.

Merging Node Numbers

Often during node propagation we will find that we have assigned two node numbers to the same node. It is necessary to establish an equivalence between these node numbers. This is done by using a large array that has one entry for every node number used. This table is used to keep track of smallest node number merged with a node number. The index corresponds to a node number. If an entry in the array equals its index then we know that no smaller value has been merged with that node number. If the entry is smaller than the node number, then the node number is equivalent to the entry. This entry may in turn point to another entry. Eventually when we follow this path far enough we will find a number whose entry in the array equals itself. When two node numbers are merged we trace the two paths down until we find the two minimum numbers. We then make the larger of these two numbers point to the smaller. At the end of the extraction phase any reference to a node number is replaced with reference to the node number's minimum value.

As an example consider the array shown in figure 2a. To merge node number 10 with node number 9 we first find the minimum values for the node numbers. The minimum value for 10 is 7. The minimum value for 9 is 2. We then enter 2 into location 7. If we now trace for the minimum value of 10 we find 2, as shown in figure 2b.

Node Names

A node name is assigned to a node by associating a point with the name. Any node that crosses that point is assigned the name. During the node propagation phase, swaths are scanned to find points that overlap nodes. When an overlap is found a record is made of the name and the node number. After the node propagation phase has completed, in the second pass, node names are substituted for node numbers.

Performance/Experience

We have gained considerable experience with MEXTRA during the RISC[7] project. Soon after MEXTRA was first operating, it was put into use. Since the program was new, many bugs were found and fixed during this time. Also this proved to be a period of evolution in which a number of inadequacies and inconvenences were discovered. I believe that this rapid period of program development and user feedback help make MEXTRA a more usable and reliable program.

A major factor for MEXTRA's rapid acceptance was its speed. It was about 10 times faster than CIFPLOT for even small cases. We estimated circuit extraction for the entire RISC chip would take approximately 24 CPU hours using CIFPLOT, which in our heavily loaded computing environment was infeasible. Circuit extraction with MEXTRA, however, was found to take about one CPU hour. This was quite feasible, and the entire chip was, in fact, extracted many times.

The naming facilities of MEXTRA, though still imperfect, were extremely useful. The naming facilities helped find unfinished LSSD nets, and places where power and ground were connected backwards. In addition it allowed simulation to take place immediately after extraction without the need to make plots with the computer generated node numbers.

Mextra was usable immediately because it fit in with many of the other tools. Caesar-generated CIF could be read, and simulation files for Mossim, Moserc, and Powest were created. In addition, a list of Mossim aliases was generated. The file of node names and locations that was created could be integrated into the plot of the chip layout created by Cifplot or converted to Caesar format by using Cif2ca to be viewed in context interactively.

The table below lists the runtimes for Mextra on several layouts.

Name	*Number of Transistors*	*Time* (min:sec)	*Trans/sec*
padiotri.cif	24	:05	4.8
control.cif	231	:17	13.9
oppla.cif	369	:27	13.7
filtchip.cif	777	1:11	10.9
cherry.cif	881	:52	16.9
pcchip.cif	1116	1:48	10.3
alu.cif	2257	1:53	20.0
shift.cif	3070	3:45	13.6
fifo.cif	8276	7:52	17.5
scheme79.cif	9443	11:38	13.5
testram.cif	20480	15:26	22.1
rfile.cif	26560	21:18	20.8
chip.cif	35262	36:04	16.3
risc.cif	44424	54:25	13.6

From this table we can see that the growth rate of the algorithm is approximately linear. Throughout the development of the program, achieving linear performance was a major goal. Although many routines of the program have n^2 worst case performance we would not expect to see this worst case performance in practice. Worst case occurs when all the rectangles of the layout lay on top of one another, which is not likely for IC layouts. From the statistics gathered in [8] we expect the density of chip layouts to be distributed almost uniformly, very far from the worst case. The bin sorting method used by Mextra is an example of such behavior. Although this sorting method has $O(n^2)$ worst case performance, for IC layouts we would expect O(n) performance.

Conclusion

The rapid acceptance and use of Mextra, and the user feedback provided by the RISC project help make Mextra into a useful and reliable tool. By concentrating on efficent, linear expected time algorithms and by restricting designs to manhattan features, Mextra has achieved significant performance improvements.

Although the runtimes for Mextra are quite reasonable for current LSI circuits, the runtimes may become infeasible for the next generation circuits because of a 10 to 100 fold increase in circuit complexity. New methods, such as exploiting hierarchy, will need to be develop in order to keep pace with technological development.

Acknowledgements

I would like to thank Carlo Sequin for his advise and for carefully reading several drafts of this report, Dave Patterson for his encouragement, and the RISC designers their feedback and criticism.

References

[1] R. Hon and C. Sequin, *A Guide to LSI Implementation, Second Edition,* Xerox PARC, January 1980

[2] *Unix CAD Tool Box Manual,* On-line Documentation

[3] C. Baker and C. Terman, "Tools for Verifying Integrated Circuit Designs", *Lambda,* fourth quarter, 1980

[4] D. Fitzpatrick, "Circuit Analysis from CIF Layouts", available from author, January 1981

[5] C. Mead and L. Conway, *Introduction to VLSI Systems,* Addison-Wesley, 1980

[6] J. Ousterhout, "Editing VLSI Circuits with CAESAR", available from author, October 1981

[7] D. Fitzpatrick *et. al.*, "VLSI Implementations of a Reduced Instruction Set Computer", *VLSI Design,* fourth quarter, 1981

[8] J. Bentley, D. Haken, and R. Hon, "Fast Geometric Algorithms for VLSI tasks", *IEEE CompCon,* Spring 1980

Appendix A: Example Run

Figure 3a shows the design of a four bit shift register. There are two metal busses marked with the label 'Vdd!'. The metal bus through the center of the circuit is marked 'GND!'. Two other metal lines are marked 'Phi1' and 'Phi2'. The shifter is made up of eight calls to a basic shifter cell. In the basic cell the input is marked 'SR.in#', and the output is marked 'Sk.out#'. Since the cell is repeated eight times there are eight occurrences of the names 'SR.in#' and 'SR.out#'. The sharp sign (#) indicates that these names are local, and that unconnected nodes with these names should be assigned unique names. The exclamation point (!) after 'GND!' and 'Vdd!' indicate that these are global names; unconnected nodes with these names should receive the same name.

The '.cadrc' file has been set up with capacitance values given in Mead and Conway, and units set to 200 since this design was done with lambda = 2 microns. The '.cadrc' file follows.

```
areatocap poly  40
areatocap diff 100
areatocap metal 30
areatocap poly/diff 400
perimtocap poly  0
perimtocap diff  0
perimtocap metal 0
perimtocap poly/diff 0
units 200
```

The command line to run MEXTRA is:

```
% mextra shift8
```

After extraction it is important to check the '.log' file. The listing of 'shift8.log' is given below.

```
window: 0 76 -132 0 @ u=200
the label 'SR.out' has 8 occurrences
the global label 'GND' has 2 occurrences
the label 'SR.in' has 8 occurrences
the global label 'Vdd' has 2 occurrences
32 enhancement, 16 depletion
39 nodes
```

A plot of this circuit with node names is shown in figure 3b. Nodes with no name assigned to them are given numbers. Note that 'SR.in#' has generated the names 'SR.in#0' through 'SR.in#7'. There are seven places in the circuit where the names 'SR.in#' and 'SR.out#' conflict. These are listed in the file 'shift8.al' listed below.

```
= SR.in#0 SR.out#1
= SR.in#1 SR.out#2
= SR.in#2 SR.out#3
= SR.in#3 SR.out#4
= SR.in#4 SR.out#5
= SR.in#5 SR.out#6
= SR.in#6 SR.out#7
```

Simulation of the circuit can takc place using MOSSIM. The command line for using MOSSIM is:

```
% mossim shift8.sim shift8.al
```

The simulation file, 'shift8.sim', is listed below.

```
e Phi1 SR.in#7 158 2 2 45 -7
e 158 GND 156 2 4 18 -10
d 156 Vdd 156 8.5 2 13 -10
e Phi2 148 156 2 2 29 -15
d SR.in#6 Vdd SR.in#6 8.5 2 62 -18
e 148 SR.in#6 GND 2 4 56 -18
e Phi1 SR.in#6 138 2 2 45 -23
e 138 GND 136 2 4 18 -26
d 136 Vdd 136 8.5 2 13 -26
e Phi2 128 136 2 2 29 -31
d SR.in#5 Vdd SR.in#5 8.5 2 62 -34
e 128 SR.in#5 GND 2 4 56 -34
e Phi1 SR.in#5 118 2 2 45 -39
e 118 GND 116 2 4 18 -42
d 116 Vdd 116 8.5 2 13 -42
e Phi2 108 116 2 2 29 -47
d SR.in#4 Vdd SR.in#4 8.5 2 62 -50
e 108 SR.in#4 GND 2 4 56 -50
e Phi1 SR.in#4 98 2 2 45 -55
e 98 GND 96 2 4 18 -58
d 96 Vdd 96 8.5 2 13 -58
e Phi2 88 96 2 2 29 -63
d SR.in#3 Vdd SR.in#3 8.5 2 62 -66
e 88 SR.in#3 GND 2 4 56 -66
e Phi1 SR.in#3 78 2 2 45 -71
e 78 GND 76 2 4 18 -74
d 76 Vdd 76 8.5 2 13 -74
e Phi2 68 76 2 2 29 -79
d SR.in#2 Vdd SR.in#2 8.5 2 62 -82
e 68 SR.in#2 GND 2 4 56 -82
e Phi1 SR.in#2 58 2 2 45 -87
e 58 GND 56 2 4 18 -90
d 56 Vdd 56 8.5 2 13 -90
e Phi2 48 56 2 2 29 -95
d SR.in#1 Vdd SR.in#1 8.5 2 62 -98
e 48 SR.in#1 GND 2 4 56 -98
e Phi1 SR.in#1 38 2 2 45 -103
e 38 GND 36 2 4 18 -106
d 36 Vdd 36 8.5 2 13 -106
e Phi2 28 36 2 2 29 -111
d SR.in#0 Vdd SR.in#0 8.5 2 62 -114
e 28 SR.in#0 GND 2 4 56 -114
e Phi1 SR.in#0 18 2 2 45 -119
e 18 GND 16 2 4 18 -122
d 16 Vdd 16 8.5 2 13 -122
e Phi2 7 16 2 2 29 -127
d SR.out#0 Vdd SR.out#0 8.5 2 62 -130
e 7 SR.out#0 GND 2 4 56 -130
C Vdd GND 242
C Phi2 GND 84
C Phi1 GND 84
```

Appendix B: Simple CIF Format

CIF, the Cal-Tech Intermediate Form, is an interchange format for integrated circuit mask designs. As such, it should be quite simple and efficient to parse. Yet this is not the case. The syntax of CIF is very general, making it difficult to parse. Since CIF is not meant to be a design language, there is no need for this generality. Generality should give way to efficiency.

The CIF parser of CIFPLOT was designed to recognize any valid CIF construct. As a consequence the parser is slow. Yet by adding a little more structure to CIF it could be made tremendously more efficient. This section presents a structure to format CIF that is easier and faster to parse. This simple CIF structure is just a subset of CIF 2.0, so it is fully compatible with existing CIF programs. At the same time it does not sacrifice any of the expressive power of CIF.

A potential defect in building tools that recognize just this subset of CIF is that it might lock us into a closed environment, where our tools would only work on locally generated CIF files. CIF files from other universities that do not follow this standard could not be read. This is not a problem, however. CIFPLOT already has a full CIF parser, and it can output a file in simple CIF format. Thus, for any general CIF file it is possible to create an equivalent simple CIF file.

Each command in CIF 2.0 is made up of one or more tokens. The first token indicates the command type. Each command is separated by a semi-colon. For instance, the command for a box located at the origin with length 10 and width 4 is the following:

```
    B 10 4 0 0;
```

But this can also be expressed as any of the following:

```
    Box with Length = 10 Width = 4, Located @ 0,0;
    BOX 10,4/0.0;
    BIRD10CAT4X0ZZZ0;
    B10+4&0/0;
```

However, the following is illegal CIF:

```
    B10-4&0/0;
```

There is no inherent syntax in CIF that any low level scanner can exploit. In general every character must be passed directly to the parser to be interpreted.

Simple CIF places two basic restrictions on CIF syntax: first that every token must be separated by a blank or comma; second that all characters in a command, except blanks and commas, must be part of a token. The first restriction allows the scanner to break a command into separate tokens. The second restriction gets rid of extra characters, which are just noise, in the command line. These extra characters have no place in an interchange format, and only add to the complexity of the language parser. Some might argue that these extra characters increase readability, but no one should be reading CIF! These two restrictions allow a low level scanner to break the CIF into tokens before passing these tokens to the parser.

The CIF Delete Definition command is very hard to implement and is hardly ever used. The effect of Delete Definition can be achieved by a simple renumbering of the symbols in a CIF description. Since it adds no power to the language, is not widely used, and is difficult to implement, the Delete Definition command is not allowed in simple CIF. By the same argument, symbol redefinition is not allowed.

Language Extensions

This section discusses extensions not part of the official CIF language, but that can be added to our structured CIF without introducing incompatibility with standard CIF.

The constructs of CIF can be used to express the physical layout of an integrated circuit. Since CIF has become the standard language for expressing IC designs, it is often desirable to keep more information than just geometry. Text to be placed on checkplots is useful to help a designer understand his circuit. Node names are useful in programs like circuit extractors and wire routers in order to relate information back to the designer through names that are meaningful to him. Information about the size of lambda is useful for programs such as design rule checkers. Information about the last time a symbol was modified is useful for tools that do incremental checking on a circuit. Yet none of this information is necessary for fabrication. CIF provides for extensions to the language by user extension commands. Our structured CIF should establish a standard notation for user extension commands. No user extension command should make CIF incompatible for fabrication, however.

Experience with Simple-CIF

As an example of the efficiency gains achievable from simple-CIF, let us compare the programs CIFPLOT and MEXTRA. CIFPLOT recognizes the full CIF language; MEXTRA recognizes only simple-CIF. After comparing the times needed by each program to read in several CIF files, the scanner of MEXTRA was found to be about seven times faster than CIFPLOT. Considering the size of CIF files this is a very significant speed up. In addition, the scanner/parser of MEXTRA is less than 400 lines of code; the scanner/parser of CIFPLOT is about 1200 lines of code. By being smaller, the scanner/parser code for MEXTRA was written, tested, and debugged much more quickly than the code for CIFPLOT. Thus, the scanner/parser modules of MEXTRA occupied a lesser percentage of the implementation time.

The importance of this is easy to overlook. The amount of effort that is put into developing a CAD tool versus the payoff is an important consideration. There always seems more work to do, than there are people to do it. It is wasteful to spend time on developing complicated parsers for clumsy interchange formats when there are many useful tools that need to be developed.

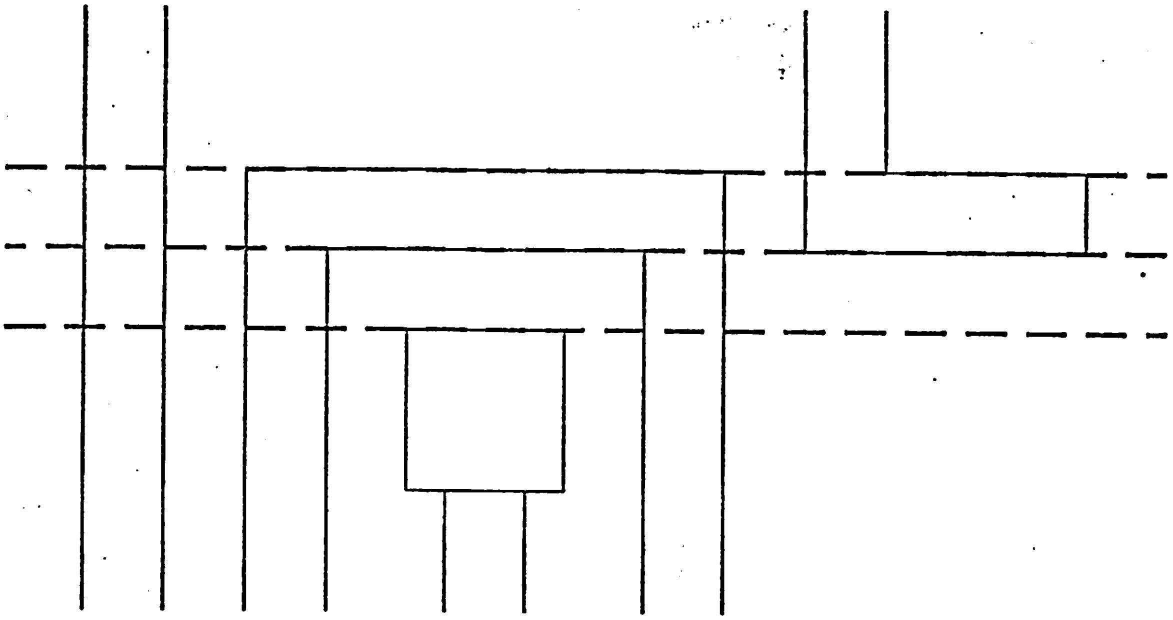


Fig. 1a. Chip Geometry

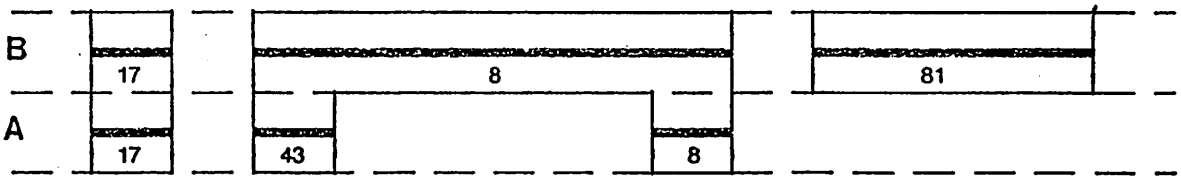


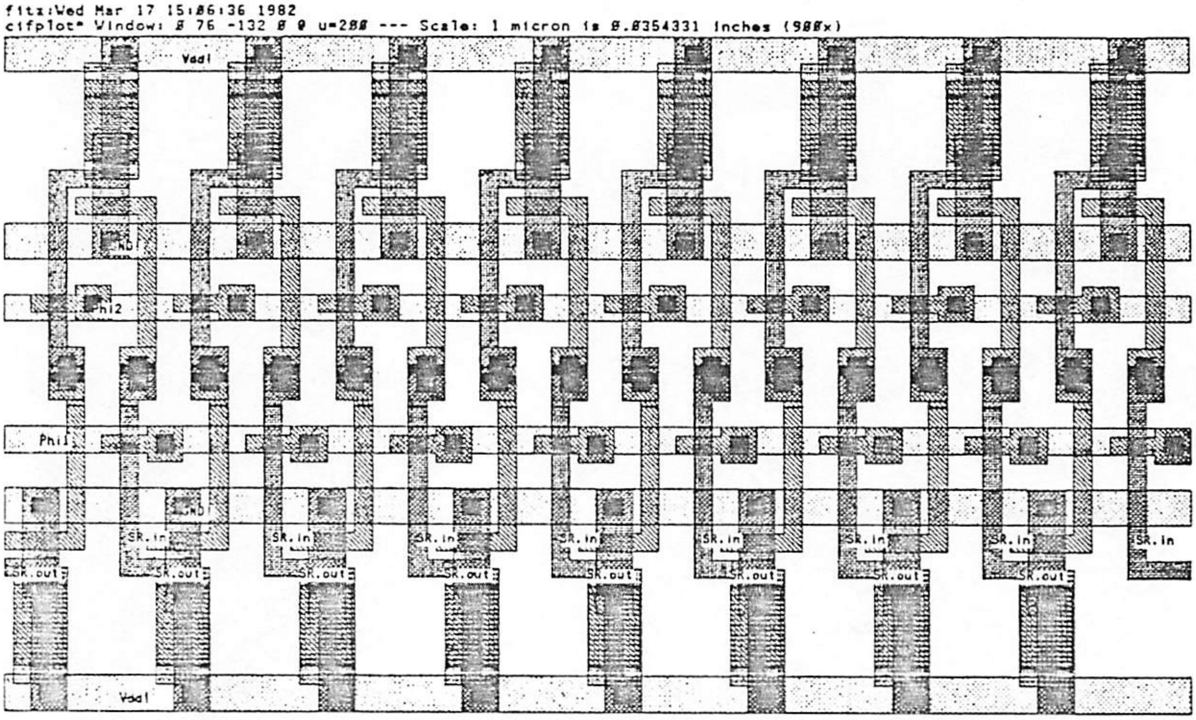
Fig. 1b. Node Propagation

11	7
10	8
9	4
8	7
7	7
6	4
5	5
4	2
3	3
2	2
1	1

Fig. 2a. Merge Table

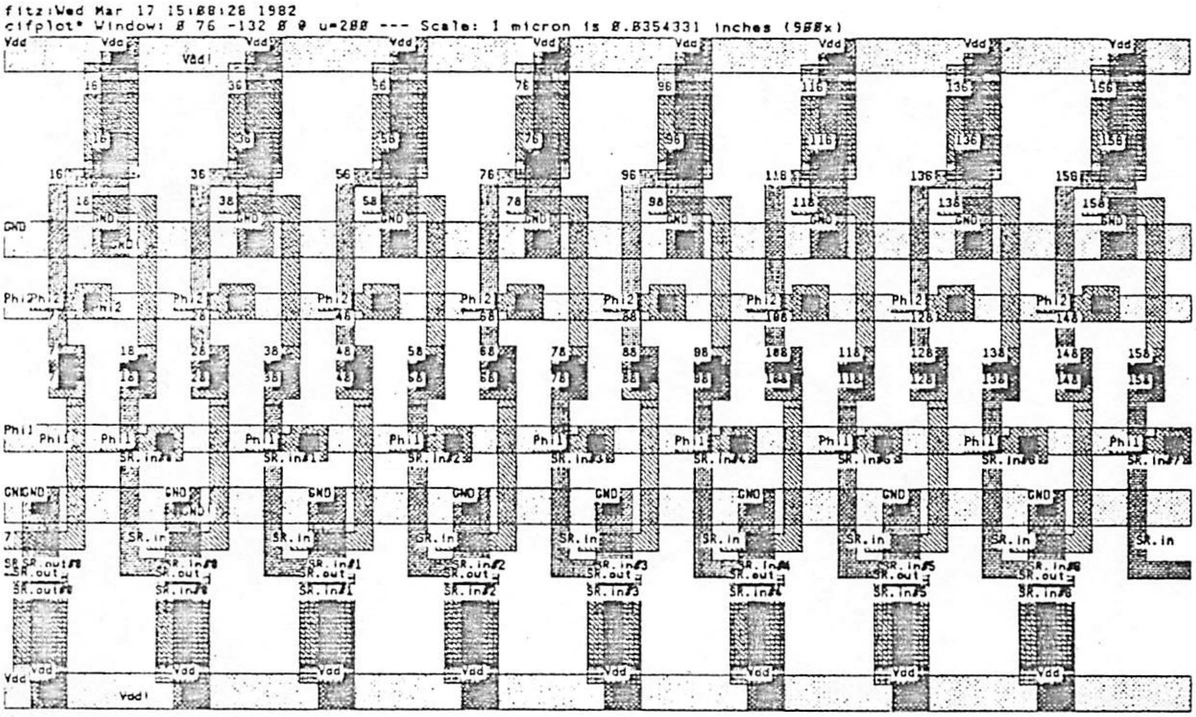
11	7
10	8
9	4
8	7
7	2
6	4
5	5
4	2
3	3
2	2
1	1

Fig. 2b. MergeTable after Merging 9 & 10



```
fitz:Wed Mar 17 15:06:36 1982
cifplot* Window: 0 76 -132 0 0 u=200 --- Scale: 1 micron is 0.0354331 inches (900x)
```

Fig. 3a. Eight Bit Shift Register



```
fitz:Wed Mar 17 15:08:28 1982
cifplot* Window: 0 76 -132 0 0 u=200 --- Scale: 1 micron is 0.0354331 inches (900x)
```

Fig. 3b. Eight Bit Shift Register after Extraction