IMPLEMENTATION OF A VLSI LAYOUT TOOL ON PERSONAL COMPUTERS

A. G. Jost, P. Streatch[1] and D. B. Guptill
School of Computer Science
Technical University of Nova Scotia
P. O. Box 1000
Halifax, Nova Scotia, Canada B3J 2X4

ABSTRACT

This paper describes a VLSI layout editor implemented on the Macintosh and IBM personal computers. The editor is capable of manipulating integrated circuit designs of complexity exceeding 50,000 transistors on a standard Macintosh Plus computer. The paper discusses the design tradeoffs necessary to implement a program of this capability on small computers.

INTRODUCTION

VLSI design has traditionally been done on super-mini-computers with attached color graphics terminals, or more recently on high performance workstations. It is generally believed that only machines of this class are capable of managing the complex "artwork" required to specify the layout of an integrated circuit of VLSI scale (50,000+ devices). It may be more true to say that since major semiconductor manufacturers can afford expensive workstations, the software they need is written for such machines, and the size and complexity of these programs reflects the capabilities of the workstations they run on. The result is that a typical state-of-the-art commercial VLSI design package now consists of upwards of 30 megabytes of programs and support files, and requires several times that amount of space to carry the database for a reasonably complex chip design. This puts the tools required for VLSI design activity out of the reach of small to medium companies and many educational institutions. In fact, even in larger corporations employing many IC designers, the number of "seats" available (i.e. workstation screens) is often the limiting factor affecting the productivity of design teams.

It must be recognized that some of the steps involved in the design of integrated circuits are computationally very expensive. These tasks include design rule checking, circuit extraction, timing analysis and simulation. However, none of these tasks require graphical interaction, and it is perfectly feasible to perform them on a mainframe time-sharing computer. What the workstation graphics is required for is interactive editing of mask layout details and schematic diagrams. Schematic editors have existed for some time for personal computers: the missing piece that would enable the use of PC's in VLSI design is the mask layout editor. Accordingly, we will examine the feasibility of implementing a VLSI layout editor on "entry level" personal computers such as the Macintosh Plus.

VLSI LAYOUT EDITORS

A VLSI layout consists of a large number of complex geometric patterns for each of a dozen or so "mask layers" which are roughly analogous to the copper traces on printed circuit boards. These patterns are built up from a number of primitive geometric shapes, which historically have included rectangles, complex polygons, multi-segment "wires", and round flashes. One approach to building a VLSI editor is to produce a general graphics editor capable of manipulating hierarchical collections of all of these shapes. The problem with this approach, when viewed from the perspective of a small machine, is that for each primitive shape supported by the editor there is a corresponding data structure in the database, and for every manipulation permitted by the editor there must be supporting routines for each primitive type. Thus for each of rectangle, polygon, wire, etc. there must be separate routines to create, delete, read, write, select, deselect, alter, stretch, rotate, mirror, move, copy, and render (display) each primitive type. The total number of such support routines is the product of the number of primitive types times the number of editor operations to be supported, and any editor which attempts to be general and support all primitive types will inevitably be large, and it will build a large and complex database. Most existing layout editors, such as the KIC2 program [1] from Berkeley and almost all commercial packages, fall into this category.

One common constraint imposed on VLSI layouts is the "manhattan" property, which requires that every edge of every feature be parallel to either the X or Y axis. This can result in a layout density penalty of the order of 10%, which is in most cases quite acceptable. It can be shown that any complex set of manhattan mask features can be represented exactly in the form of a collection of abutting rectangles. It follows that a layout

[1] P. Streatch is now with Geovision in Ottawa, Ontario

editor which can manipulate only rectangles can be used to create and edit any manhattan design. Indeed, this approach was taken in the early 1980's with the Caesar editor from Berkeley [2], although the program was implemented on a VAX and the main advantage perceived was the resulting simplicity of the user interface and increased efficiency of related artwork analysis programs. The Berkeley RISC chip [3], the precursor of Sun Microsystem's SPARC architecture, was implemented with this editor. It is our conjecture that this approach is also a key to the implementation of a high performance layout editor on personal computers, since such an editor would only need support routines for a single type of primitive and would use a simple and compact database. We have thus implemented a layout editor which we call "RED" (Rectangle EDitor) which runs on a number of platforms including the IBM PC and the Macintosh, and this editor has been shown to be capable of effectively editing circuits exceeding 50,000 transistors on an entry level (1 MB with single floppy disk drive) Macintosh Plus. The remainder of this paper will discuss the implementation and user interface of this editor, with an emphasis on the important design trade-offs required to achieve high performance on small computers. Although the discussion focuses on the Macintosh version, most of the points raised are equally valid on other personal computers.

DATABASE ISSUES

To accommodate large complex designs on a small computer, it is essential that memory be used as efficiently as possible. Because we support only a single primitive type, the database is essentially a hierarchical collection of rectangles. At the lowest level, all mask geometry is ultimately expressed as lists of rectangles, so the data structure for storing the rectangle is crucially important. For maximum flexibility, rectangles are organized in the form of separate linked lists for each mask layer: the layer number can be omitted from the data structure because it is implied by the list of which it is a member. Thus the rectangle structure requires only the x-y coordinates of diagonally opposite corners, and a pointer to the next rectangle on the list. A straightforward design would use 32 bit integers for each coordinate and a 32 bit link pointer, for a total of 20 bytes. However, careful choice of coordinate system permitted a significant space savings. In the final design, each unit corresponds to 1/4 micron, and 16 bit signed integers are used to store each coordinate. The range is thus plus or minus 32767 units, or 8192 microns. If a large design is properly centred, this permits the design of chips up to 1.6 centimetres square. The resultant rectangle data structure (figure 1) occupies only 12 bytes.

```
struct BOXES {
  short  int  x1,y1,  /* lower left corner   */
              x2,y2;  /* upper right corner */
  struct BOXES *next;
  }
```

Figure 1: Rectangle data structure (12 bytes total)

However, the situation is slightly more complex than this implies. Most system memory management facilities organize memory pools using linked lists of free blocks, and the memory management overhead typically involves two longwords per block allocated: a 32 bit integer to record the size of the block and a 32 bit pointer to the next block on the list. The Macintosh is no exception. The 8 byte overhead is not significant when allocating large structures, but in the case of the rectangle structure it increased the net memory requirement per rectangle to 20 bytes: a 67% penalty. To deal with this we implemented a specialized memory allocator for rectangles, which allocates space for 100 rectangles at a time and uses the internal link pointer to link unused rectangles, reducing the net overhead to less than 1% (8 bytes for each 1200 byte block). It also improved performance dramatically when starting up the program with large designs, since it turned out that the CPU overhead of the operating system's memory allocator was high.

Other data structures were required for cell definitions, cell instances, and labels, and these are all larger than the rectangle structure (Table I). However, in most complex designs these are much less numerous than the basic geometric primitives, so careful optimization is not so important. Table I also shows the incidence of each data structure for a complex CMOS chip designed at our institution: a digital time-domain beamformer [5] containing 42,313 transistors. With this entire design loaded into RED's internal data structures, there is still almost 500K bytes of free memory on a standard 1 MB Macintosh Plus.

Structure	Size (bytes)	Beamformer Freq.	Memory Use
rectangle	12	8873	106476
cell	358	146	52268
instance	44	755	33220
label	48	324	15552

Table I: Size in bytes of key RED data structures, and frequency of use in a representative complex chip design

HIERARCHY

Hierarchical design techniques are by now well established in the VLSI design community: they exploit the regularity of many key VLSI structures and permit the description of very complex circuits with relatively modest amounts of data storage. In RED hierarchical designs are represented as groups of related cells, each of which may include instances of other cells, organized logically in a tree structure with a single root. The depth of the tree is unlimited. Typically a cell contains either a few hundred rectangles, or a few instances of other cells with a small number of rectangles representing interconnection wiring. On disk, each cell is stored in a separate file, so that it is possible to edit individual cells without the need to load the entire design database into memory at once, although the current version of RED always loads at least the complete subhierarchy rooted at the cell being edited. The file format used is based on a subset of Caltech Intermediate Form (CIF) [4], and is largely compatible with the format used by the Berkeley KIC program [1]. It is pure ASCII, so it is easy to edit by hand and to interface to other tools. RED can also generate a complete CIF file when required for fabrication, design rule checking, extraction, etc. This is accomplished with the WriteCIF command, which performs a depth-first traversal of the hierarchy rooted at the current edit cell to establish the list

of required subcells and outputs them into the CIF file in an order which guarantees that each cell is defined before it is used, as required by CIF syntax rules [4].

At the cell level, mask data is organized as separate linked lists of boxes for each layer. Multiple selection had to be implemented to support efficient editing of details at the box level, and a novel approach to this problem was to use an additional set of head pointers to partition each box list into two sections: the "select pointer" points to a particular box on the cell's box list and all boxes beyond this point on the list are considered to be selected. Selecting/deselecting boxes involves moving a data structure from one end of the list to the other. Deselecting all boxes is accomplished simply by setting the select pointer(s) to NULL.

The cell definition structure is the largest used in RED, and although it could be optimized somewhat, it does not account for a large percentage of memory use. The cell instance structure is much smaller and contains only a pointer to the appropriate cell definition, a link pointer, and the parameters required to define the transformation to be applied to the particular instance. Cell instances are very common in typical VLSI designs.

GRAPHICS

Most mask layout editors use colors to distinguish individual mask layers on the screen. On a monochrome screen such as the Macintosh's, mask layers are displayed using unique stipple patterns drawn in a manner which shows key overlaps such as polysilicon over diffusion as composite patterns. Red also supports color displays, on which masks can be displayed with any combination of colors and patterns, and if the hardware supports the concept of "bit-planes" transparent solid colors are possible. The program can also use AND, OR, and XOR drawing modes to enhance the visual transparency effect. These attributes are all user-configurable via a configuration file.

The first version of RED [6] used a generalized graphics interface package modelled on the ACM CORE graphics proposal. Arithmetic was performed in floating point and the graphics subsystem took care of all required window transformations and clipping. Although it functioned correctly, performance was barely acceptable, especially when viewing complex layouts. Since raster displays use integer coordinates, it was decided to rewrite the graphics code in RED to use only fixed point arithmetic (in fact floating point variables were eliminated from the program entirely). The result was dramatic: drawing speed improved by an order of magnitude, and the size of the executable program was reduced substantially. Restricting the program to working with rectangles also paid big dividends in graphics performance, since graphics clipping is much more efficient for rectangles than for more general primitives such as polygons, particularly when using manhattan geometry. Both of these factors turn out to be very significant, considering that the display of a fully expanded view of a large chip can require the clipping and rendering of hundreds of thousands of rectangles.

To achieve portability and performance at the same time, we defined a new graphics interface package, which we call Dlib (Table 2), to be the sole interface to the screen, keyboard, and mouse. For the most part the Dlib graphics routines are stateless: instead of setting graphics modes which are "remembered" in subsequent drawing calls, most calling sequences include all information relevant to the operation being performed. It is arguable that this implies additional calling overhead which would impair performance, but it is interesting to note that the majority of bugs that arose in the graphics code were eventually traced to careless reliance on state information in the original implementation. The conversion of RED to interface with Dlib and eliminate floating point arithmetic required a substantial overhaul of the original code, but the end result was well worth the effort. Dlib drivers for new display hardware can usually be written in a couple of days (the IBM PC driver was a notable exception).

Initialize	D_init (width, height, colors, planes, buttons)
Terminate	D_term()
Draw line	D_line (x1, y1, x2, y2, color, pattern, planes, mode)
Draw box	D_rect(x1,y1,x2,y2,color,style,pattern, planes,mode)
Set text font	D_setfont (name, charwidth, charheight)
Display text	D_text (x, y, string, color, planes, mode)
Create RGB color	D_define_color (color, red, green, blue)
Create pattern	D_define_pattern(patno,pattern)
Set event mask	D_eventmask(mask)
Get input event	D_getevent(type, data, x, y)
Get mouse location	D_readlocator(x, y)
Alert	D_Beep()
Flush output	D_flush()
Flush and wait	D_sync()

Table 2: "Dlib" graphics interface summary

USER INTERFACE

Supporting only a single geometric primitive made it easy to keep the user interface simple and intuitive. There are approximately 35 commands in RED, most of which are available on a single menu on the right margin of the screen as shown in figure 2, which shows a typical monochrome display on the Macintosh computer. Each command can also be executed by typing a single keystroke on the keyboard (for example "b" initiates the AddBoxes command), which speeds up operation considerably for the experienced user. The menu items "light up" at all times that a command is active, providing useful operator feedback.

The user interface of RED assumes that the workstation has a mouse or tablet with at least three buttons, and on machines which have fewer than this, modifier keys are used to achieve the same effect. For example, on the Macintosh, which has only one button on the mouse, holding the "Option" key while pressing the button is treated as "left button". The middle button is used for all menu selection and drawing/editing operations, and most graphical operations provide "rubber-banding" (to a user-settable grid) to preview their effect. The right button is used to initiate a zoom operation (which shows a rubber-band rectangle to indicate the area to be zoomed); if the operation is finished with the right button it is treated as a zoom-in: if it is finished with the middle button it is a zoom-out. Panning is performed with the left button in a similar fashion. Both zoom and pan can be performed at any time without disturbing other commands in progress, which is very useful when drawing very long wires and boxes. For convenience, the keyboard keys comma, space, and period are considered to be equivalent to left, middle, and right buttons respectively. At any particular time during operation of the program, one mask layer is considered to be the "current layer" and the designation of this layer is controlled via a second menu across the bottom of the screen (figure 2). This menu shows the available mask layers with their graphics rendition and CIF name, and it is user-configurable through the configuration file. The "current layer" determines the mask used during drawing operations and is also used to restrict selection operations. The special layer called "***" is a wild card layer which permits selection of geometry on any layer. It cannot be used for drawing.

The editing operations supported include deletion, moving, copying, rotation, and flipping either vertically or horizontally. It is also possible to stretch rectangles (or groups of rectangles), by "grabbing" either an edge or a corner. Prior to performing any of these operations, one or more primitives must first be selected by means of the selection commands. The set of operations provided has proved to be entirely satisfactory, even for experienced designers used to more "sophisticated" editors.

CONCLUSIONS

RED has become the workhorse layout editor at our institution, even on our Unix workstations, on which it also runs (in color). The undergraduate VLSI design class does all its layout work on Macintoshes. The Macintosh binary is only about 70K bytes, which leaves most of memory free to support the database. The disk drive is active only when the user loads or saves a cell, with the result that overall performance is about the same with either floppy or hard disk. We believe that it would have been possible for the Berkeley RISC chip to be designed on a Macintosh with RED: we estimate it would

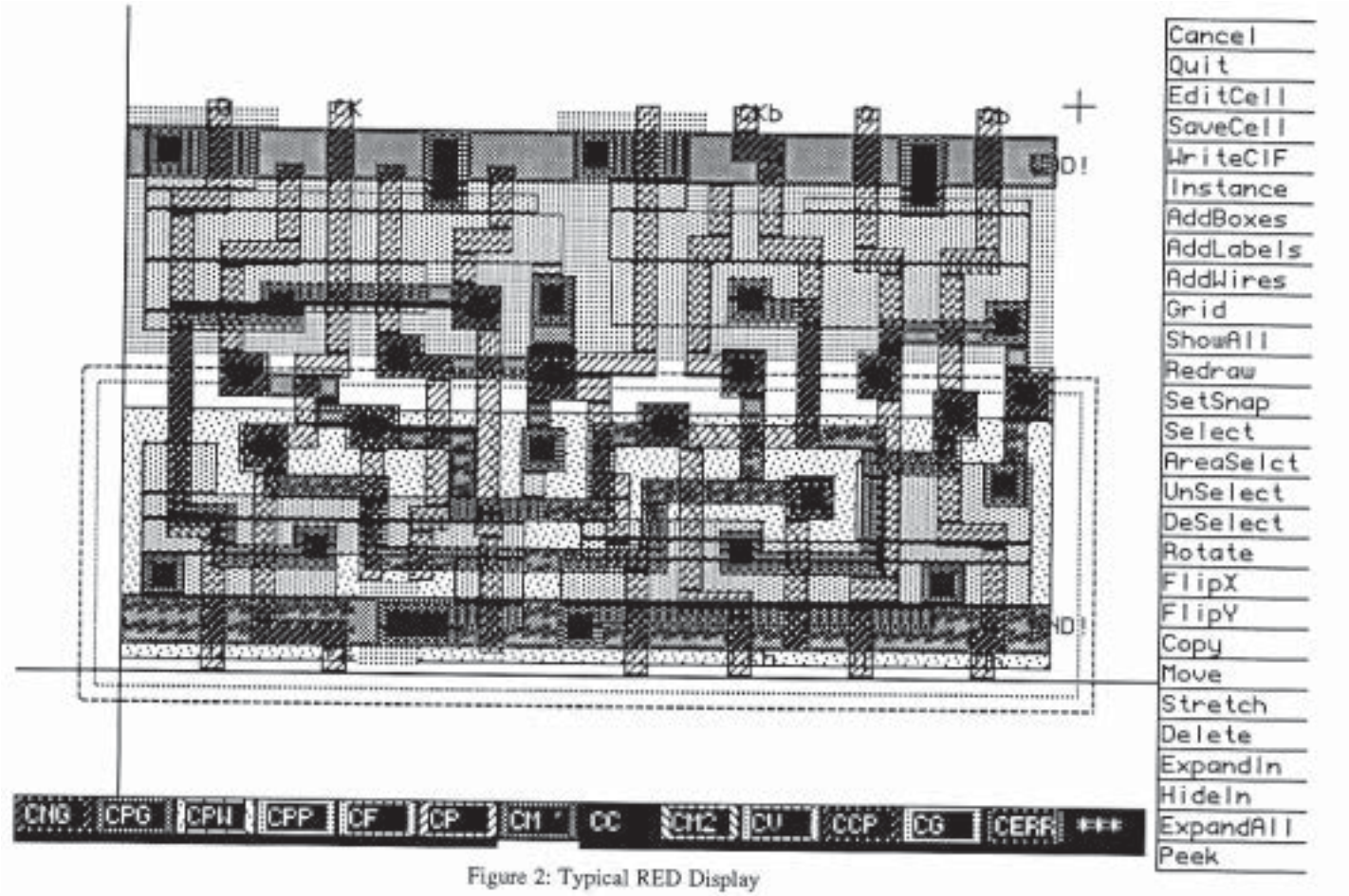

Figure 2: Typical RED Display

require only about 550k bytes of memory to hold the entire design.

Our experience with the project has taught us several valuable lessons which would be valid for other projects of a similar nature:

1) It pays to resist the temptation to fill your machine's memory with your program by adding myriads of nice but unnecessary features. Keeping the program small leaves valuable room for more important things, in our case to support a large database.
2) The restriction to the rectangle primitive and careful attention to database design resulted in a simple, elegant user interface and efficient use of storage without the need to resort to virtual memory techniques which would have brought the disk drive(s) into play and degraded performance.
3) Color is nice, but you can achieve a lot by intelligent use of monochrome techniques.
4) Fixed point arithmetic seems to be a lost art, but the price of floating point can be high for certain kinds of applications.
5) Hierarchical data structures are an excellent way of using resources efficiently when working with designs containing many repeated patterns, as is typical in VLSI.
6) Vendor-supplied system software support, which is of necessity designed to be general, is not always suitable for specialized applications which use system resources in a manner differing from that foreseen by the vendor (in our case it was the memory management that proved to be a bottle-neck).

REFERENCES

1. K. Keller, "KIC: A Graphics Editor for Integrated Circuits," Master's thesis, Univ. of Calif., Berkeley, 1981.

2. J.K. Ousterhout, "Caesar: An Interactive Editor for VLSI Layouts," *VLSI Design,* Vol. 2, No. 4, 1981, pp. 34-38.

3. M. G. H. Katevenis, "Reduced Instruction Set Computer Architectures for VLSI," Ph.D. dissertation, Univ. of Calif., Berkeley, October 1983.

4. R.F. Sproull and R.F. Lyon, in C. Mead and L. Conway, *Introduction to VLSI Systems Design,* Addison Wesley, Reading Mass., 1980.

5. A.G. Jost, B.J. McKinnon and W. Robertson, "Implementation of a CMOS Single Chip Time Domain Beamformer," *Proc. 1988 Canadian Conference on Very Large Scale Integration*, pp. 40-48, October, 1988.

6. P. Streatch, *RED: An Interactive Mask Level Layout Editor,* Master's Project Report, Technical University of Nova Scotia, May 1987.