COMPUTER AIDS FOR THE DESIGN OF LARGE SCALE INTEGRATED CIRCUITS

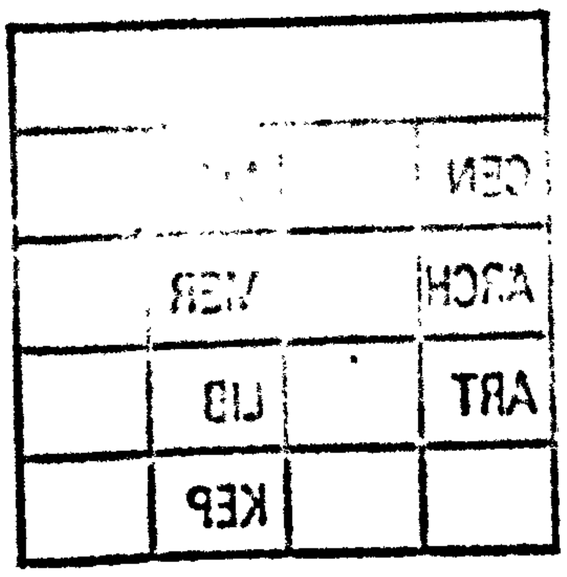
By

George Baxter Swan BSc with
First Class Honours in Electronic Engineering

A thesis submitted in partial fulfilment of the requirements of the Council for National Academic Awards for the Degree of Doctor of Philosophy (Ph.D)

School of Electronic and Electrical Engineering
Robert Gordon's Institute of Technology
Schoolhill
Aberdeen
AB9 1FR

December 1983



DECLARATION

I hereby declare that this thesis is a record of work undertaken by myself, that it has not been the subject of any previous application for a degree, and that all sources of information have been duly acknowledged.

During this research, the following courses were included in an approved programme of advanced studies :-

1. "Computer Aided Design"
   Robert Gordons Institute of Technology
   Aberdeen
   1980

2. Relevant seminars on integrated circuit design aids
   University of Edinburgh
   1980

3. "Interactive computer graphics"
   University of Manchester
   April 1981

4. "Integrated circuit design"
   University of Arizona"
   U.S.A.
   August 1981

5. "Industrial management"
   S.E.R.C Graduate School
   University of Durham
   April 1982

George B Swan

December 1983

INDEX

ACKNOWLEDGEMENTS

I would like to express thanks to my supervisor Dr. J. D. Eades for his encouragement and assistance throughout the project.

I also acknowledge the support of the SERC grant, and thank Mr. B. McK.Davidson for his help in the preparation of all the illustrations.

ABSTRACT

COMPUTER AIDS FOR THE DESIGN OF LARGE SCALE INTEGRATED CIRCUITS

by

George B. Swan

The work described in this thesis is concerned with the development of CADIC (Computer Aided Design of Integrated Circuits), a suite of computer programs which allows the user to design integrated circuit layouts at the geometric level.

Initially, a review of existing computer aids to integrated circuit design is carried out. Advantages and disadvantages of each computer aid is discused, and the approach taken by CADIC justified in the light of the review.

The hardware associated with a design aid can greatly influence its performance and useability. For this reason, a critical review of available graphic terminals is also undertaken.

The requirements, logistics, and operation of CADIC is then discussed in detail. CADIC provides a consise range of features to aid in the design and testing of integrated circuit layouts. The most important features are however CADIC's high efficiency in processing layout data, and the implementation of complete on-line design rule checking. Utilization of these features allows CADIC to substantially reduce the lengthy design turnaround time normally associated with manual design aids.

Finally, the performance of CADIC is presented. Analysis of the results show that CADIC is very efficient at data processing, especially when small sections of the layout are considered. CADIC can also perform complete on-line design rule checking well within the time it takes the designer to start adding the next shape.

CHAPTER 1

Introduction

1.1 Aim of project

The aim of this project is to produce CADIC (Computer Aided Design of Integrated Circuits), a suite of computer programs to aid in the design of integrated circuits. It is therefore useful to point out some of the problems faced by the integrated circuit designer, so as to form a clearer picture of how CADIC could aid circuit design.

Prior to circuit fabrication, it is the designer's job to produce the artwork required for each mask used in the fabrication process. A mask contains a unique pattern of opaque and transparent areas, and is used to control which areas on the silicon wafer will be doped (see later). This task is complicated by the fact that the physical size of the final circuit must be as small as possible. There are two main reasons for this :-

1. Smaller size increases circuit reliability, and production yield
2. Smaller size means more circuits on each wafer, which reduces fabrication costs

Designers must therefore ensure that the pattern of shapes on each mask are as compact as possible.

The first step in the fabrication process is to protect the silicon by growing a thin layer of silicon oxide over the surface of the wafer (Figure 1.1a). Next, a layer of photographic emulsion (photo-resist) is spread over the oxide then baked to make it photo sensitive. Finally, the mask plate is laid on the photo-resist, and the sandwich exposed to a strong source of ultra-violet light (Figure1.1b). The radiation causes molecular change in the exposed photo-resist, allowing the unexposed areas to be washed away easily.

Acid is then used to remove the unprotected oxide, leaving the bare silicon once again. This process is known as etching. Note that the exposed photo-resist and the silicon are unaffected by the acid. Now the pattern on the mask has been directly transferred onto the silicon (Figure 1.1c). The photo-resist has now served its purpose and is removed using strong organic solvents.

If the wafer is then placed into a temperature controlled furnace, and fed with for example, Boron gas, the exposed areas of silicon will start to absorb the Boron molecules (known as doping). Controlling the density of the gas, and the temperature of the furnace allows very accurate levels of doping to be achieved (Figure 1.1d).

The above process is now repeated using different masks and different chemical elements to produce the individual components. For example, the last two stages required to create a bipolar transistor are shown in Figures 1.1e and 1.1f.

By depositing metal over the entire wafer, and then selectively etching, the components can be connected to form the complete circuit (Figure 1.1g).

The problem with the fabrication process is that in practice, the chemical elements are absorbed as fast along the wafer, as they are absorbed into the wafer. This means that the previously well defined doped areas now contain curved 'walls' which travel underneath the oxide protection layer. For example, an actual transistor is shown in Figure 1.2. Should two areas be too close together, they may be seen to be separate on the mask, but in fact be joined in the silicon, thus leading to circuit failure.

The designer must therefore produce the masks with regard to a set of <u>design rules</u>. In the geometric sense, these rules set a minimum spacing between separate areas on any one mask, minimum spacing between areas on different masks, the amount of overlap required to ensure connectivity between areas, and so on. In this way, the design rules ensure that any mask layout which obeys them will be faithfully transferred onto the silicon.

Almost 85% of the total cost of producing integrated circuits is required to produce the first batch of circuits. Having to repeat the mask making, and fabrication stages just because the circuits were faulty is understandably very expensive in terms of time and money. However, increasing circuit complexity does increase this possibility. Therefore, to ensure that the circuits will operate correctly, stringent design rule checks must therefore be carried out while the mask layouts are still in intermediate form (i.e. stored on the computer).

At present, the design rule checks are performed after the masks are designed. Any violations detected means that the designers must return to the design stage, and edit the masks. Changing the layout often introduces new errors, therefore the design and checking stages

must be repeated several times before acceptable masks are obtained. Computer time is not cheap, for example to design rule check a large layout will typically cost between £10,000 and £25,000.

The problem of verifying mask layouts obviously gets worse as the layouts become larger, and more complex. New techniques are therefore required to handle large scale integrated circuits more efficiently.

1.2 Guide to thesis

Chapter two performs a critical review of existing design aids. Nowadays, the design and verification of integrated circuits is almost completely computer dependant. It is therefore very important to review the performance of existing computer aids before describing the development of CADIC. Advantages and disadvantages of each type of design aid is discussed in this chapter. Finally, the approach taken by CADIC is justified in light of the review.

The hardware associated with a design aid can also greatly affect the performance, reliability, and useability of a design system. For this reason, Chapter three critically reviews existing graphic terminals, and evaluates their performance when applied to integrated circuit design. Lastly, the graphic terminal used by CADIC is described in some detail.

The CADIC suite is split into four programs :-

1. MANCAD : Manual language compiler
2. CADIC1 : Interactive graphic aid
3. DRCCAD : Design rule language compiler
4. CADIC2 : On-line design rule checker

Chapter four through to seven discusses each program separately. The requirements of each program are discussed and logistics proposed. Finally the operation of each program is detailed. Chapter eight goes on to discuss the performance of these programs, with emphasis placed on CADIC1 and CADIC2, the most important programs in the CADIC suite.

Chapter nine concludes the thesis by giving an overview of the project. Certain weakpoints in the CADIC suite were identified, and possible improvements are proposed. Finally, this chapter outlines several areas of research that may be pursued in the future.

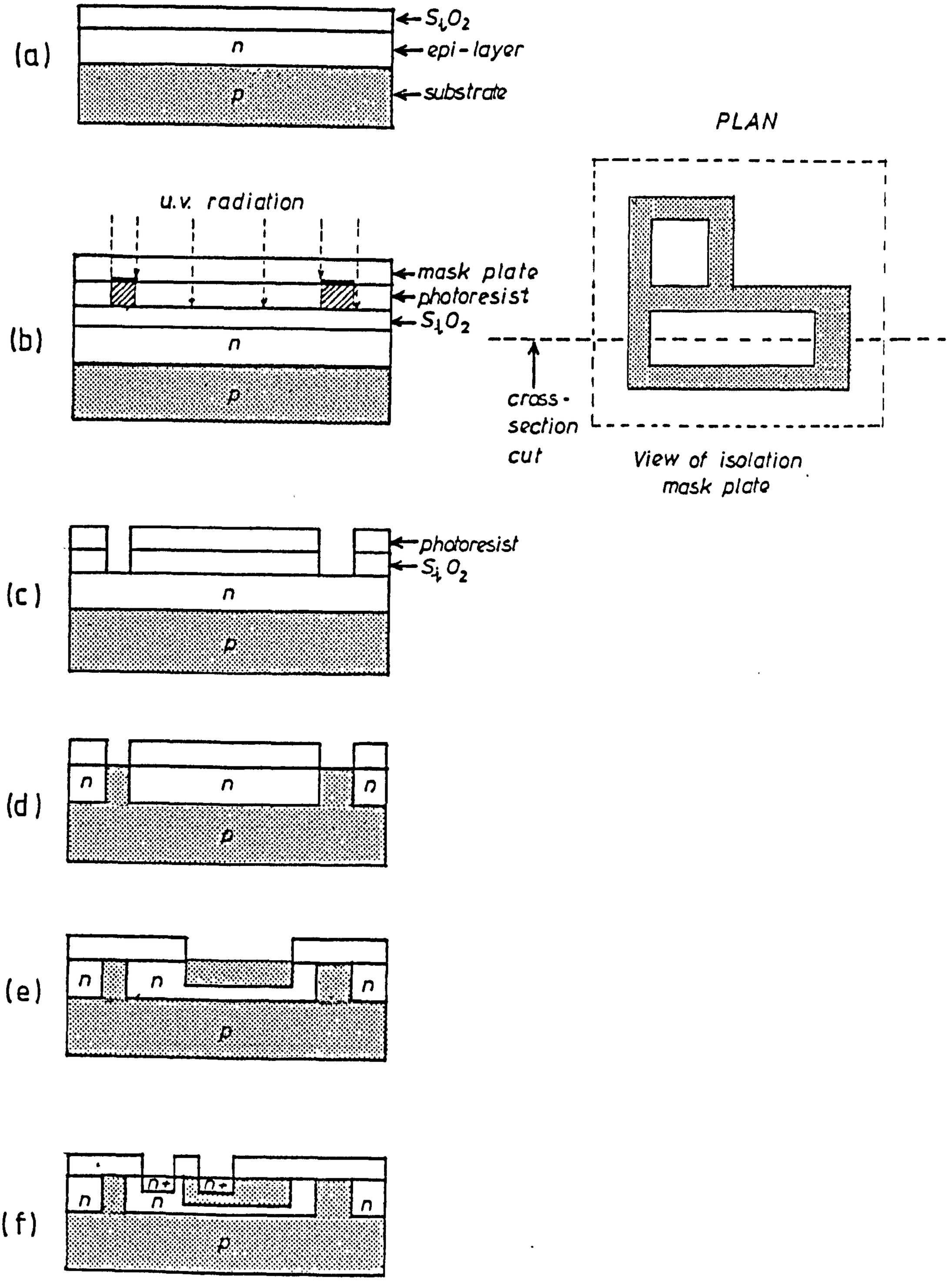


Figure 1.1 Cross-section view of Integrated circuit fabrication

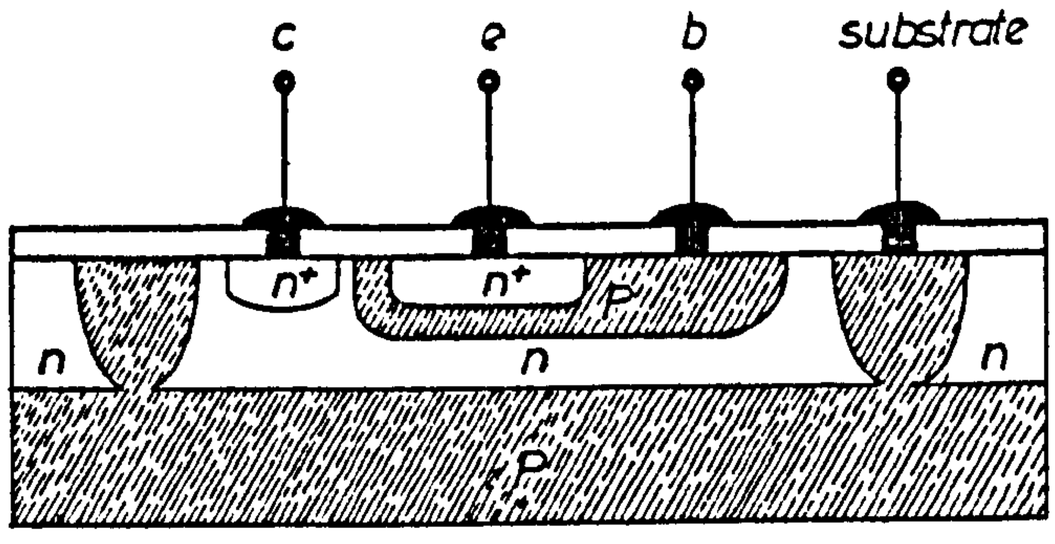


Figure 1.2 Cross section view of an actual transistor

CHAPTER 2

A review of computer aids for the design of integrated circuits

2.1 Introduction

Originally, the design of integrated circuits was performed entirely by humans. The layouts were designed by hand, and verified visually. Not surprisingly, this process was very time comsuming, therefore as layouts became larger, and computer time became cheaper, more and more of the design and verification workload became computer aided. Nowadays, the design and verification of integrated circuits is almost completely computer dependant. It is therefore important to review the performance of existing computer aids before developing CADIC, the new design aid formed as a result of this project. Two main categories of computer aid will be considered :-

1. Layout design
2. Layout verification

Advantages and disadvantages of different approaches within each category will be discussed below. Finally, the approach taken by CADIC will be justified in light of the review.

2.2 Layout design

Originally, the layout design was carried out at by hand. The designer produced a rough outline of the layout, which was then handed

to the draughtsman for tidying up. Around 80% of the design time was consumed in the latter stage, therefore the advent of computer graphics was welcomed. The designer could now design layouts with the inherent accuracy of the computer, and so dispense with the time-consuming draughting stage.

As computers became more powerful, it seemed a good idea to speed up the design time by automating the whole design process. Unfortunately, the 'intuitive' power of the computer was over estimated. Even today, a layout produced by automatic techniques is not as compact as a manually produced layout.

A compromise was therefore required. This produced systems which contained fully automatic routines to design a section of the layout. The designer could accept or reject the computer's decision, so by stepping through and/or repeating each stage of the design, was relieved of the repetitive work, but kept control of the design. Designers soon found that the layouts improved almost directly to the amount of human intervention applied. For this reason, a trend back to the manual approach occured.

Nowadays, layout design aids are numerous, spanning the range from manual to fully automatic. Which type of design aid a manufacturer will want to use will depend largely on how it will help reduce the cost of producing integrated circuits. This cost arises from two main factors :-

1. Fixed costs - the cost of designing the mask layouts
2. Variable costs - the cost of fabricating the circuits

It therefore pays the large-volume manufacturer to spend more time and money manually designing the layouts, to gain on layout compactness. Smaller physical size means that more circuits can be formed on each batch of wafers for the same fabrication cost. On the other hand, the small-volume manufacturer is better decreasing the design costs by automatically designing the layouts, at the expense of larger circuits.

2.2.1 Manual approach

Manual design is the technique whereby the designer primarily uses on-line interactive graphics to create the pattern of shapes which go to form the integrated circuit layouts. A manual design aid however often provides other features to help simplify the design problem. Typically, the design aid will consist of three main programs :-

1. Pre-processor
2. Graphic editor
3. Post-processor

Each type of program will now be discussed in more detail.

2.2.1.1 Pre-processor [1]

A pre-processing design aid accepts a 'user readable' description of the layout, and converts this description into a 'computer readable' description. In a design system incorporating several design aids, this 'computer readable' description will be the layout database which links the design aids together. Two main types of pre-processor exist :-

1. Digitiser
2. Compiler

A digitiser consists of a board about three feet long, by four feet wide, which has a grid of fine wires embedded into its surface, plus a scriber which is capable of emitting magnetic signals. When the button on the scriber is pressed, the wires detect the signal, and an accurate x-y coordinate is sent to the computer. Interfacing software is required to collect and process the information before it can be added to the layout database. In this way, whole layouts can be digitised very quickly and easily. The cost of the digitiser however often precludes this type of pre-processor in an integrated circuit design system.

A pre-processing compiler is conceptually similar to a software compiler in that a high level description of the layout is compiled down into the 'computer readable' description. Using a specially developed 'manual input language', the designer can sit at standard alphanumeric terminal, and create a file which contains geometric information about the shapes in the layout. Because the files containing the manual description are disc-based, standard text editors can be used to edit the layouts. Therefore libraries of basic elements for a particular technology can be built up, and stored for use in future designs. Entering a layout using a compiler tends to be much slower than using a digitiser. For this reason, compilers tend to be limited to entering small sections of layout at a time.

2.2.1.2 Graphic editor

A graphic editor allows the designer to interactively create/edit an integrated circuit layout. Two main types of editor exist, and are described below :-

Geometric [1-6] : A designer using the geometric approach deals with the actual shapes that are to appear on the final mask layouts. For example, the shapes required to form a NAND gate are shown in Figure 2.1. The design problem is three-fold in that the designer must :-

1. Specify the geometry of each shape
2. Place shapes as close together as possible, without violating any design rules.
3. Preserve correct layout topology

Since the designer has direct control over the artwork, mask layouts designed using the geometric approach tend to be very compact. In fact out of all the layout design techniques available (manual or automatic), the geometric approach is capable of producing the most compact layouts.

The main disadvantages with the geometric approach are that the time required to design layouts or design turnaround time is comparatively long, plus the finished layouts must be extensively design rule checked (see later) to ensure design correctness.

Symbolic [3,7-12] : A designer using the symbolic approach ultimately produces all the shapes on the mask layouts, just as in the geometric approach. The difference now is that the basic layout definitions (i.e. tracks, transistors, contacts) are represented by symbols. In this way, much of the geometric information can be ignored during layout design. Only once the layouts are complete does the design aid need to convert the symbolic layout into the geometric layout. Symbolic design therefore facilitates shorter design turnaround times, but the layouts tend to be larger than necessary.

Symbolic design exists in two forms; static and dynamic. Static design uses alphanumeric characters at specific geometric locations to represent the mask layouts (Figure 2.2). Layouts are designed on an alphanumeric terminal, and output on a line printer.

One of the advantages of this approach is the simplification of design rule checking. Using characters forces the layout geometry to a coarse grid. By defining the grid size to be equal to the resolution of the fabrication process, then correlating the design rules with the grid, the designer is less likely to make design rule errors. Note however that design rule errors can still occur, so the mask layout must be fully checked after the design is complete.

The main disadvantage is that mismatch between the grid spacing and design rule minimums forces layouts to be larger than is necessary. The ergonomics of static design are also very poor. The layouts are difficult to understand, the limited resolution of the alphanumeric screen restricts viewing options, and large layouts can only be checked by taping together sections of line printer output.

Dynamic design overcomes many of the problems associated with the static approach by using colour graphics plus 'spacing synthesis'. Spacing synthesis allows the designer to disregard all geometric information, therefore only topological information is required (Figure 2.3). After the design is complete, the design aid automatically converts the 'Stick diagram' into a geometric layout (as in the static approach) then compacts the geometric layout as much as possible.

The main advantage of this approach is that in theory, the final layout does not need to be design rule checked. The design rules are

built into the compaction routines, so correct shape relationships will always be observed. This concept breaks down when the designer wants to add some special geometric artwork, as is often the case in practice. Under these conditions, correct design cannot be assured, so the geometric layouts must be design rule checked, just as with all other manual design aids.

A more serious problem with dynamic design is that the compaction routines [11,13] cannot compact the layout as well as humans can, therefore the final layouts are usually larger than necessary. Complex designs pose severe problems for even the best compaction routines, therefore at present, dynamic design aids are limited to producing only small layouts.

2.2.1.3 Post-processor [1]

A post-processor accepts a 'computer readable' description of the layout, and converts the description into a 'user readable' description, for example a scaled plot, or a manual input file. Post-processors therefore not only provide hard copies of the layout, but also provide a valuable feedback link within the manual design aid.

2.2.2 Automatic approach

Automatic design is the technique whereby the designer provides only a functional or behavioural description of the layout. The computer, running primarily in batch mode then takes this description, and produces the complete set of mask layouts. Two main approaches to automatic design exist, and these are described below.

Cell-based : The most common cell-based design aid used today relies on the concept of a standard cell [14-21]. A standard cell is basically a layout building block, which is rectangular in shape, and is defined as having input/output pins only on the top and bottom edges of the cell. A description of the layout is built up by choosing cells from a pre-defined library, and specifying the connections between cells. The design aid then takes this information and arranges the cells in a series of rows, then routes the connections in the intervening channels (Figure 2.4), such that the total wire length is at a minimum. Standard cell assemblies can be generated relatively quickly, so producing a cheap design system. The cells have been tried and tested in the past, therefore the layout is geometrically correct, even if built up by semi-skilled users.

The main disadvantage with the standard cell approach is that the finished layout consumes much more area than is necessary. This is due to the fact that the width along the entire length of each channel must be equal to the maximum required, even if this maximum is experienced only once in the channel. The cells themselves must be of constant height, which again is wasteful as the height must be that of the maximum required. The constraint of constant row height also makes it very difficult to include special cells such as ROM's or RAM's, which resricts ingenuity of design.

In the never ending search to achieve the excellent results produced by manual design aids, the restrictive standard cell approach was broken down to give the general cell approach [22-25]. The cell concept is still used, but now input/output pins can exist on any side of the cell, and the cells are given freedom of movement. The cell dimensions can now be optimized on an individual basis, and the routing

area in between the cells can be utilized much more efficiently (Figure 2.5). Consequently, layouts are more compact, and wire length reduced.

One never gets something for nothing, and in general cell assemblies, the penalties include more complex placement and routing routines which force an increase in computation time.

While placing the cells, the best a computer program can do is 'loosely' route the wires. Once all the cells have been placed, and the wires are to be 'hard' routed, the situation often arises in which some wires cannot be routed due to lack of space between the placed cells. Special cases must be made of these wires, and may involve several re-runs of the layout design package with human intervention.

Silicon compilation [26,27] : The concept behind the silicon compiler is that of an 'ultimate' layout design aid. The designer submits a high-level functional description of the layout required. The silicon compiler accepts this description, and automatically produces the geometric layouts, without the use of libraries, as with the cell-based approach.

To simplify the layout problem, all silicon compilers work on a 'target' architecture. For example, a typical target architecture [26] is shown in Figure 2.6. Silicon compilers are therefore limited to producing a particular type of circuit. Blocks within the architecture are usually filled using ROM and PLA generators [14]. With such a fixed format, layouts are not surprisingly much larger than necessary.

Silicon compilation is still very much a concept in, rather than an alternative to layout design. A non-trivial example has yet to be

published, and no working circuits have ever been produced [27]. The concept is however very appealing, therefore silicon compilers will undoubtably figure very strongly in automatic layout design in years to come.

2.3 Layout verification [28]

Humans are reliably error-prone, therefore any manual assistance given during the design of the mask layouts means that the masks must be checked to ensure validity. Originally the checks were done purely visually, but LSI and VLSI technology soon pushed the size of layouts outside the range that could be comfortably handled by humans. Nowadays, layout verification is in most cases totally computer automated, since much of the work involves mechanically repeating simple tests many times. In general, two types of check are required, both of which are described below.

2.3.1 Functional check

Functional checks ensure that the layout agrees with the original design specifications. Three types of functional check exist, and these are described below.

Device recognition [29-32] : Device recognition is becoming an important functional check. Analysis of the mask layouts allows the computer to identify the individual components, then extract information about the components. In this way, transistor characteristics, coupling capacitances, resistances, and so on can be reported to the designer. Comparison with the original design specifications will identify any errors, plus highlight possible problem areas not realised earlier in the design.

Connectivity [31,33,34] : As the title suggests, a connectivity check ensures that all the components in the circuit are interconnected correctly. The computer analyses the pattern of shapes on each mask layout, and builds up a list of how the components are interconnected. This connectivity list is then compared against a user-supplied list which described how the components should have been interconnected. Comparison of the two lists allows all connectivity violations to be identified.

Simulation : Simulation was the first functional check available, and it is still the most common functional check carried out today. The large range of simulators now available allow the integrated circuit design to be verified at various levels of abstraction, for example :-

1. Behavioural
2. Register transfer
3. Logical
4. Timing
5. Circuit

Using simulators in this way allows violations to be identified early in the layout design process. Behavioural simulators are used at the initial design stage, to verify the algorithms of the digital system to be produced. Computer software is often used to perform this task. Note that no details of the physical design are required at this stage.

Once the algorithms have been verified, a 'block diagram' of the layout can be formed. This 'block diagram' may then be tested using a register transfer level simulator [35,36]. Only crude timing information may be available, yet useful information such as congestion and hardware/firmware tradeoffs can often be identified at this stage.

The block diagrams can then be partitioned into low level building blocks, or logic gates. A logic simulator [37,38] may then be used to verify the logical circuit. Sophisticated delay models may be incorporated at this stage to obtain a more accurate picture of circuit operation.

Finally, the logic gates can be replaced by the actual transistors and interconnections which will appear on the integrated circuit layout. Accurate circuit simulation [39,40] can be performed for small sections of the layout using an circuit simulator. To limit the amount of CPU time required, larger sections of layout are often simulated in less detail using a timing simulator [41,42].

Some of the newer simulators allow different sections of a layout at different levels of abstraction to be simulated concurrently. These mixed-mode simulators [43,44] can give the effect of complete circuit simulation, yet allow the designer to minimise CPU time and memory requirements by taking advantage of fast high-level descriptions in less critical areas of the layout.

2.3.2 Geometric checks

In layout verification, of equal, if not greater importance, are the geometric or design rule checks. These checks ensure that the patterns on the masks will be correctly transferred onto the silicon during the fabrication process, so preserving layout topology. Limitations in the fabrication process are such that without design rules, two adjacent areas may be seen to be separate on the mask layouts, but in fact be merged together in the silicon. Circuit failure or a lowering of yield and reliability would probably ensue. The design

rule checks can be performed in two ways, both of which are described below.

Off-line [28,45-60] : The first point to note is that all mask layouts are presently checked this way. After the design is complete, the designer submits the mask layouts to the design rule checker, along with a file containing the design rules [61]. The rules are applied to each mask, or combinations of masks, and any violations identified are written to a report file.

Note that the combinatorial explosion caused by checking all the shapes against one another means that the design rule checks are very expensive to carry out (Typically £25,000).

Once the checks are complete, the designer must edit out the errors in the layout, using the information stored in the report file. Correction of one error may involve repositioning part of the layout, which could introduce new errors. Therefore, when the editing is complete, the layout must be checked again for design rule violations. In practice, this design - check cycle must be repeated three or four times before an acceptable layout is achieved.

On-line : In this approach, the design rule checker is integrated into the design aid, so that as each shape is added to the layout, it is checked against the the pre-defined set of design rules. If a violation occurs, then the shape is rejected, otherwise it is accepted. This approach is much cheaper in terms of CPU time, since the shape need only be checked against the existing layout.

Layouts checked on-line are correct at all times, so on completion of the design, the circuit is ready for fabrication. Without the

multiple design - check cycles present in off-line techniques, the design turnaround time is greatly reduced.

Ideally the checks should be performed within the time it takes the designer to start adding a new shape. Previous attempts at on-line design rule checking [3] have never achieved this, unless limited to very simple checks. New techniques to speed up the process are therefore required.

2.4 The approach taken by CADIC

The aim of this project is to produce CADIC (Computer Aided Design of Integrated Circuits), a new and more effective integrated circuit design aid. It is therefore important to justify the approach taken by CADIC, in the light of existing techniques.

A review of existing design aids shows that although automatic design is popular, manual design plays by far the major role in integrated circuit production. There are two main reasons for this :-

1. Manual aids are capable of producing the most compact layouts
2. Manual aids are required to produce the cells used in automatic design

Within manual design, there are two possible approaches; geometric and symbolic. When it was first introduced, symbolic design seemed to be the answer to the design problem. For a variety of reasons, symbolic design has not lived up to these expectations, so much so that many companies who changed to symbolic design when it first appeared have since returned to geometric design.

Geometric design, on the other hand, although very useful, is not without its drawbacks. The dependence on off-line design rule checking forces multiple design-check cycles, which create the lengthy design turnaround time normally associated with geometric design.

In conclusion, CADIC should be a manual design aid, which allows the designer to work at the geometric level. CADIC should also incorporate on-line design rule checking. In this way, the design-check 'bottleneck' found in existing geometric design aids can be broken, which will allow substantial reductions in design turnaround time. Other standard features such as pre-processors and post-processors should also be incorporated into CADIC as required.

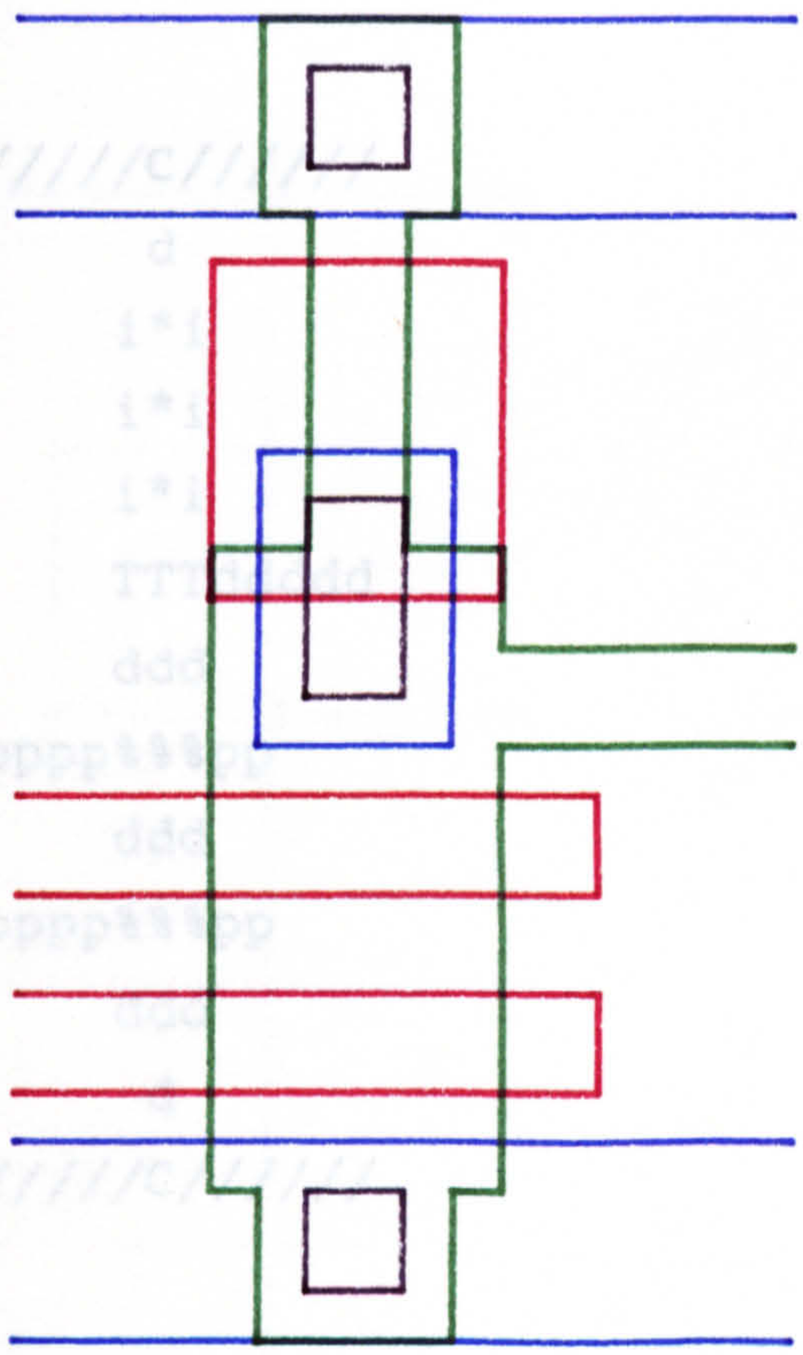


Figure 2.1 Geometric layout

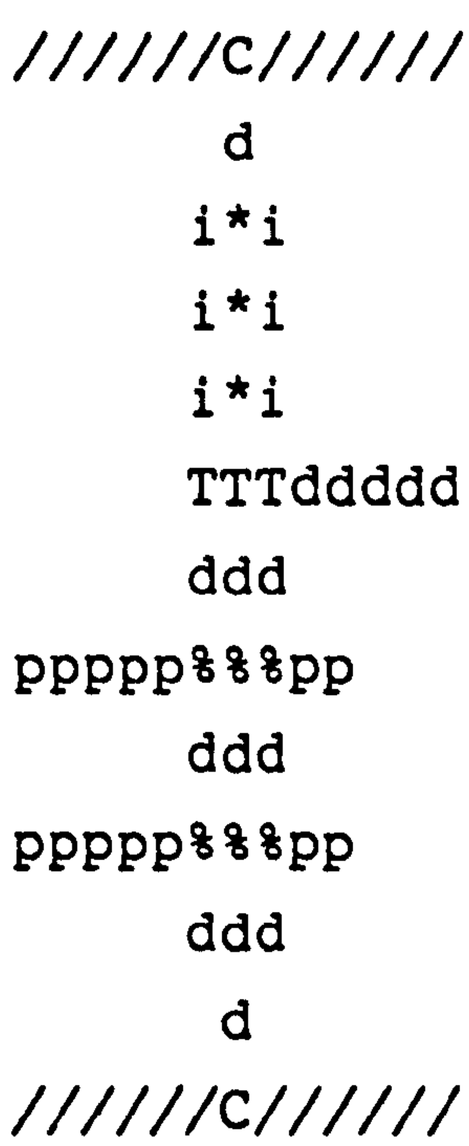


Figure 2.2 Static layout

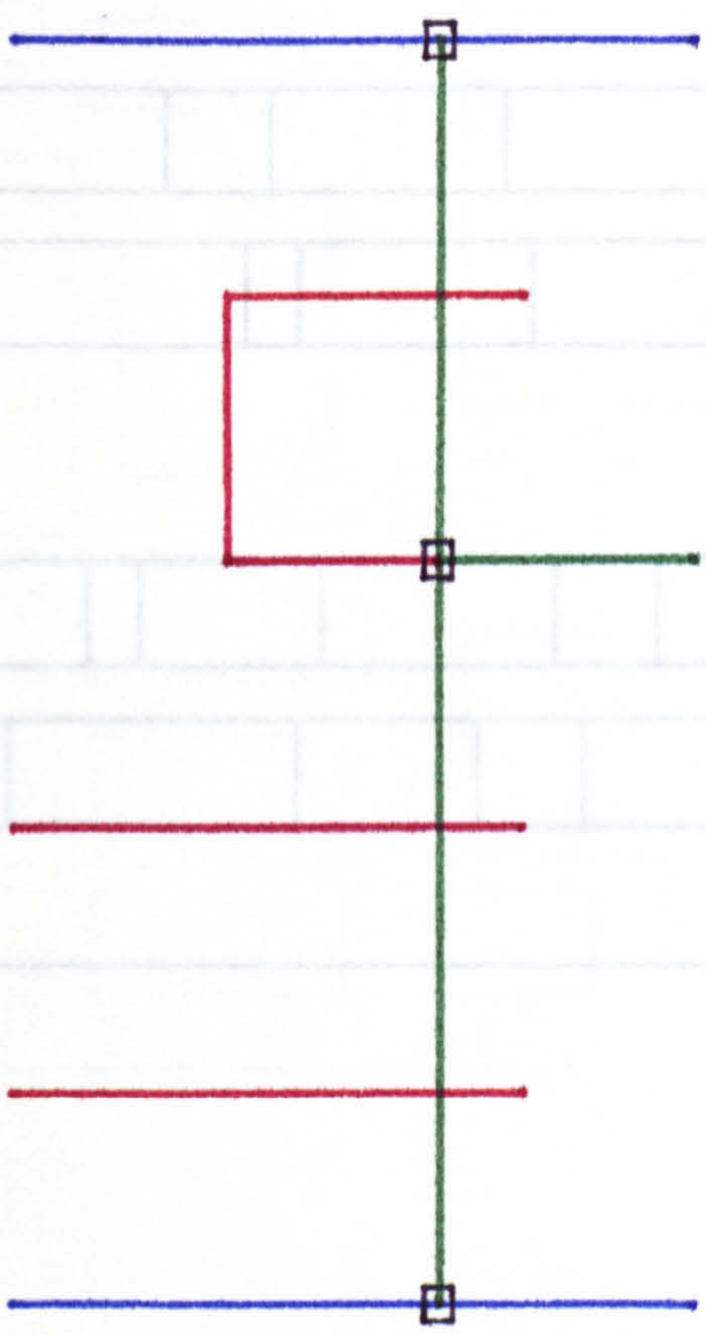


Figure 2.3 Dynamic layout

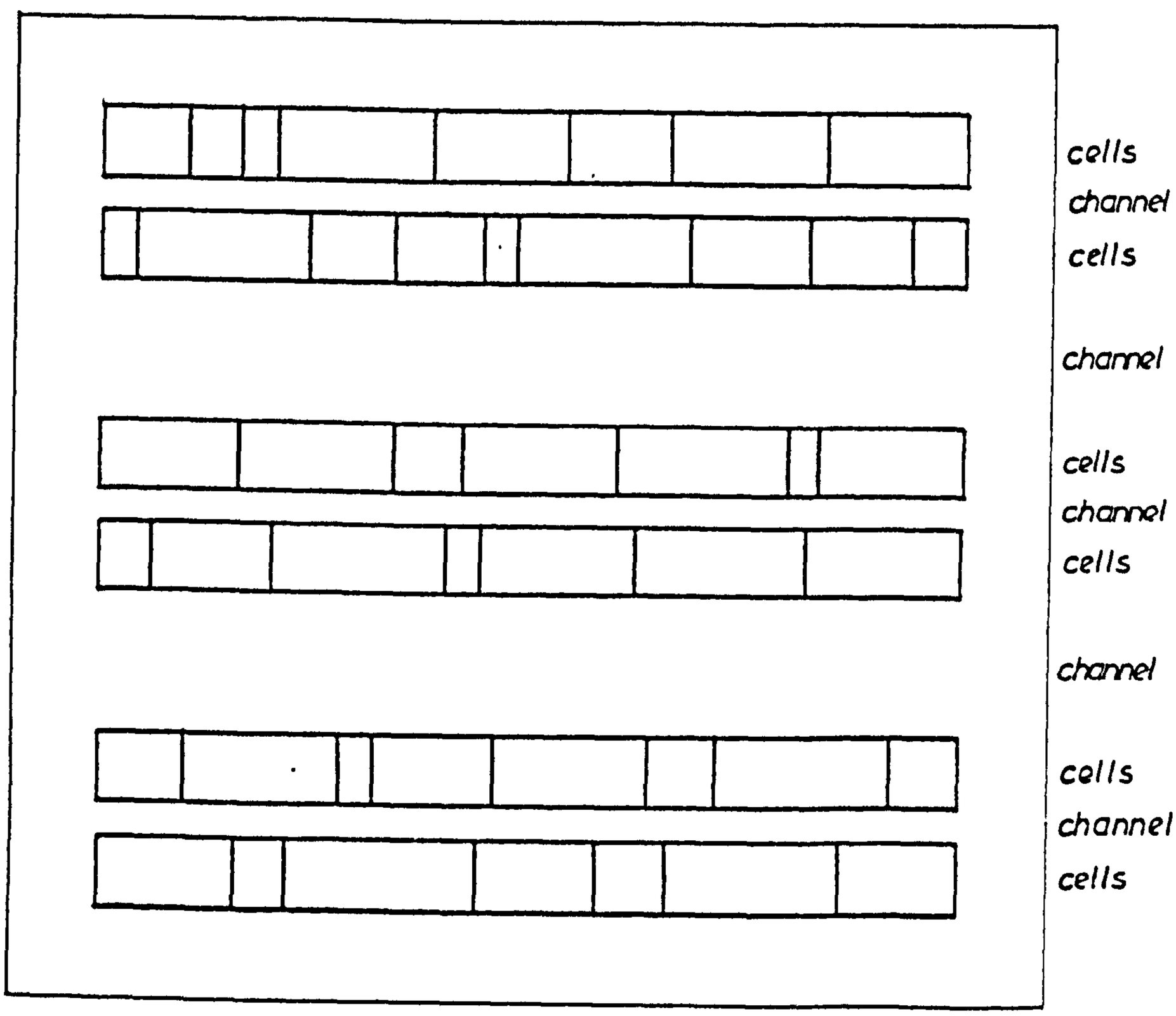


Figure 2.4 Standard cell assembly

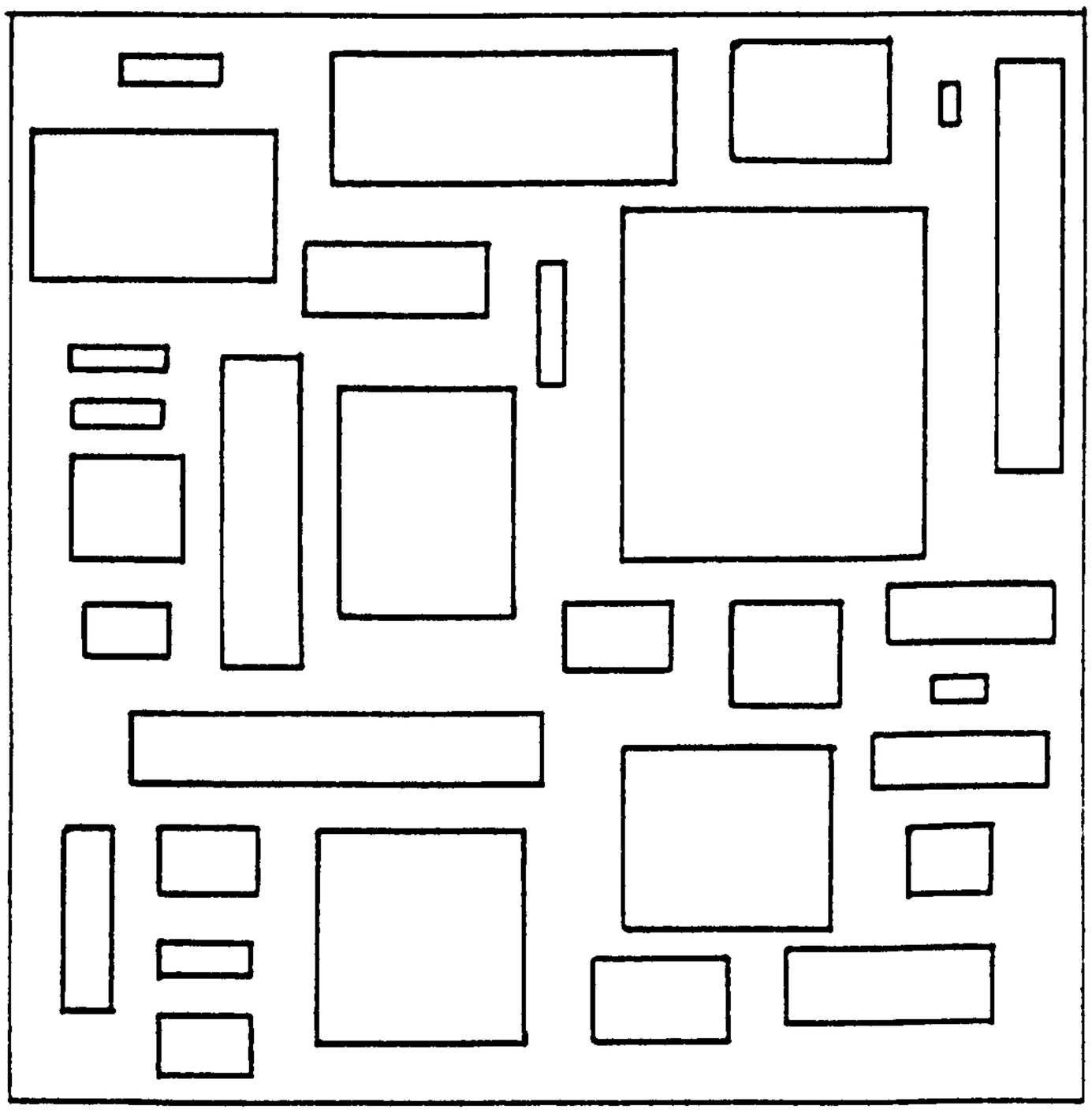


Figure 2.5 General cell assembly

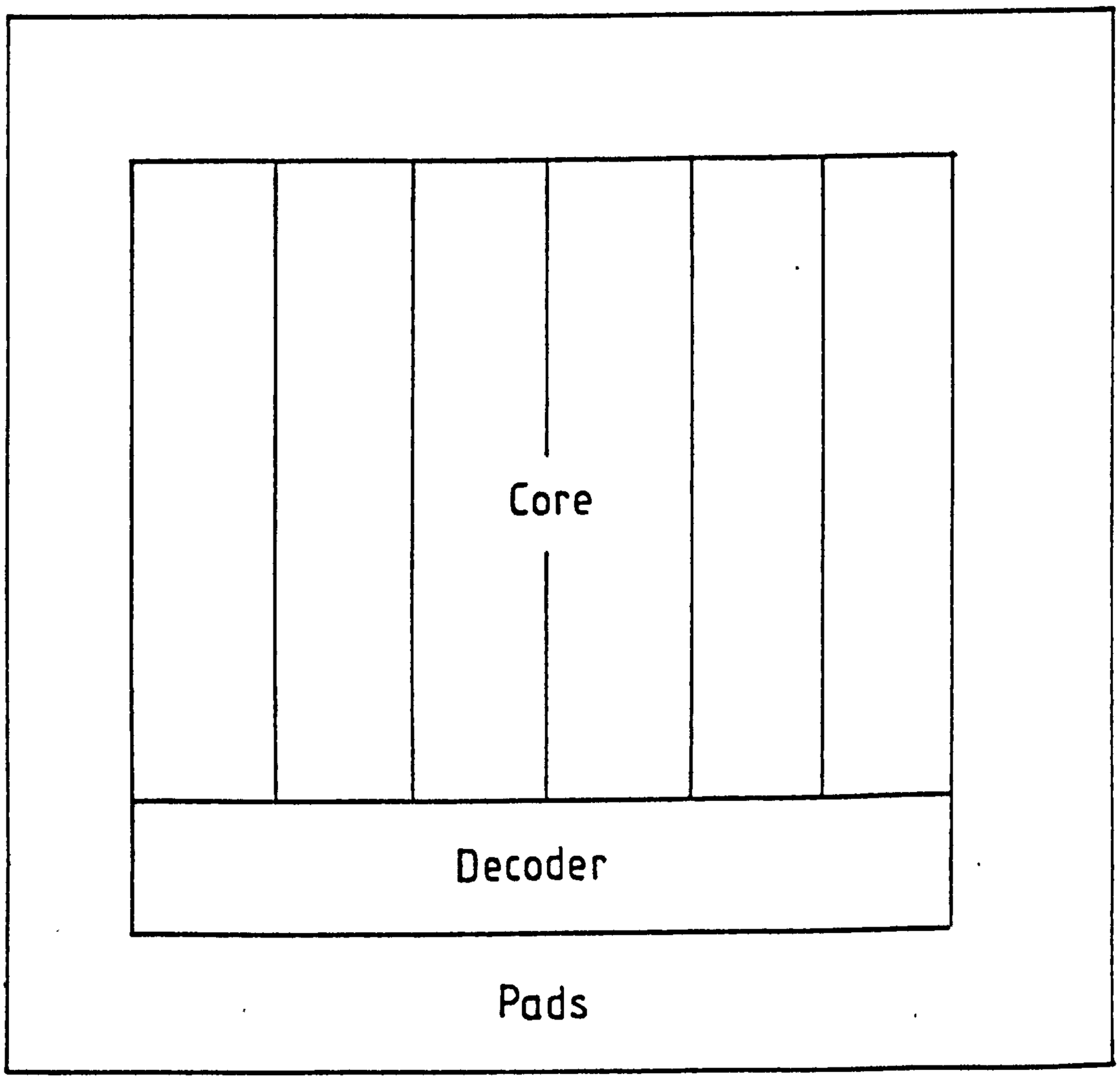


Figure 2.6 Bristle block target architecture

CHAPTER 3

A review of graphic terminals

3.1 Introduction

The way in which a graphic terminal is connected to the host computer, plus the distribution of intelligence between host and terminal can greatly influence the performance, reliability and useability of a system. For this reason, the whole of this chapter is devoted to describing graphic terminals and evaluating their use in integrated circuit design.

Originally, a graphics system consisted of a large mainframe computer controlling a simple display. The early displays were non-intelligent, which meant a large amount of computer power to do the simplest of operations. Understandably, the early systems were very expensive.

The addition of specialized hardware made displays more intelligent. Fundamental problems such as producing alphanumeric text and dashed lines could now be generated from the terminal itself. This helped reduce the load on the host computer, but a large amount of computing time was still spent driving the display.

As time-sharing became fashionable, the host computer could no longer provide enough dedication to the display. Mini-computers were used as a satellite, and so off-loaded much of the graphics software

previously stored in the host. The more powerful mini-computers became, the more graphic and non-graphic work they could accomplish.

As technology advanced, the micro-computer became as powerful as early mini-computers. Subsequently, micro-computers were soon to be found in the display terminal itself. The micro-computer thus helped reduce the mini-computer's workload, just as the mini-computer had done to the host computer several years earlier. Present intelligent terminals are highly sophisticated, with the use of these terminals in CAD packages becoming the rule rather than the exception.

Note that in general, even with good intelligence distribution, the amount of number crunching power required in integrated circuit design still forces the display to be connected to a mainframe computer.

3.2 Alphanumeric terminals [8,54,62]

The most basic, and certainly the cheapest type of graphics system, is to use the alphanumeric VDU (Video Display Unit) to display the plot. Software is required to convert conventional line drawing to raster-scan drawing, but different characters can be used to represent different regions. For example, if a manual design aid is constrained to plotting out a maximum of four masks at any one time, then all possible regions (hex 1 - F) can be displayed, if the individual masks are defined as hex 1,2,4,8 respectively.

The plots are however rather crude due to the limited resolution of the screen (Typically 80 x 24). Graphic interaction also tends to be very limited, therefore this type of terminal is often used purely an output device.

3.3 Plotters [63]

The very nature of plotters allows the user to produce a plot which can be taken away and studied at leisure. Consequently, plotters tend to be used purely as output devices. Some plotters have motor control buttons, plus a 'hit' button, which can be used to position the pen, then send the pen coordinates to the computer, so providing a limited interaction facility. Such an interactive system is very slow, and consequently its use is rather limited.

Probably the most popular type of plotter is the flat-bed plotter. This type of plotter uses pens, usually ink or fibre-tip, supported on a gantry over the plotter base (which holds the paper). Powered by stepping motors, the pen and its lifting mechanism move along the gantry in the Y direction, and the gantry moves in the X direction.

Changing the pens at pre-determined stages during the plot allows colour plots to be produced. A clever graphics package can also fill in the shapes, but this tends not to be done, due to plotter speed (Typically 25cm/sec), and striping, due to slight pen alignment errors.

Faster plots can be achieved by using a drum plotter. Now the pen may move along the axis of the drum (Y direction) with the paper and drum rotating under the pen (X direction). The drum and pen are both much lighter than the gantry system, and so speeds of 80cm/sec can be achieved.

Since raster graphics (see later) became popular, new types of plotters have been developed. These plotters handle normal 'outline' plots but are specifically designed to cope with 'shaded' and/or

'filled-shape' plots. Two main types exist ; matrix plotter, and ink-jet plotter.

A matrix plotter is similar to a conventional matrix printer, except that it uses a multicoloured ribbon to produce the coloured plots as a series of dots. The ribbon usually consists of three colours (yellow, cyan, and magenta), which by a system of overprinting can produce any one of eight standard colours. The quality of the output is however rather poor, due mainly to the limited number of dots per line (typically 700), plus misalignment of overprinted dots.

The ink-jet plotter is much more sophisticated in that liquid ink is sprayed onto the paper using three ink jets mounted on a gantry . Direction and amount of ink from each of the jets is controllable, allowing high resolution, and very high quality plots.

3.4 Direct View Storage Tube Terminals [1,2]

DVST terminals differ from the common VDU by requiring a special cathode ray tube as shown in Figure 3.1. Pictures are stored in the form of charge on the storage grid, using the main electron gun. The collector helps smooth out the flow of electrons from the flood gun, and the high potential screen accelerates these electrons through the storage grid, thus copying the image onto the screen.

For an input device, the DVST terminal uses a cross-hair cursor, in the form of a horizontal and vertical line extending across the screen. These lines are repeatedly drawn at an intensity just below that required to permanently store an image. Through the use of suitable controls on the keyboard, the cursor can be positioned anywhere on the

screen, and the cursor coordinates obtained on request.

DVST terminals can exist in two modes, alphanumeric and graphic, so allowing the user to carry out all types of interaction on the one terminal, providing the user with a relatively cheap graphics system.

The storage ability of DVST terminals means that the plot can be built up in stages, therefore they are ideal for use in time-shared environments. The plot does not flicker, and the screen resolution tends to be very high (Typically 4096 x 3071).

In general, the DVST terminal produces a monochromatic plot, so dashed-lines are required to differentiate between shapes on different masks. The shapes cannot be filled, therefore plots become rather confused when several masks are plotted at once.

The main disadvantage with the DVST terminal is that charge cannot be selectively removed from the grid, therefore no selective erasure of the screen is possible. Removal of part of the plot involves clearing the screen, then redrawing the complete plot, which can be time-consuming.

3.5 Vector scan Terminals [6]

A symbolic representation of a simple vector scan terminal is shown in Figure 3.2. Note that a standard cathode ray tube is used, so the phosphor on the screen excited by the electron beam will glow only momentarily. To produce a steady image on the screen, the plot must be redrawn or refreshed often enough so that the phosphor is re-excited before the glow disappears. In practice a refresh frequency of 50Hz is

usually chosen.

The picture description is stored in a display file, which is a list of drawing and control instructions. The display file can be regarded as a data structure, and can be dynamically updated, providing selective erasure and animation features such as rubber-banding, shape towing and so on.

Vector scan terminals usually offer very high resolution. Normally the image is monochrome, but the facility to dynamically vary the beam intensity does help to visually separate shapes on different masks. Shape fill is not possible, therefore complex layouts can become confused.

Colour vector scan terminals do exist. The tubes in these terminals contain three phosphor layers (red,green,blue) and the layer is selected by varying the potential on the gun anode (Figure 3.3). By this method, eight colours can be produced, but in general, the system involves specialized control hardware, and is very expensive.

From the computers viewpoint, a vector scan terminal requires 100% dedication. In a time-shared environment, this amount of dedication is of course not possible, so a satellite mini-computer must be present to carry out the graphic work.

Early terminals used the core memory of the mini-computer to store the display file. With such a system, 80% of the computer time was spent sending display information to the terminal. Therefore, on the advent of cheap memory, and intelligent terminals, the display file was soon to be found in the terminal itself.

Regardless of mini or micro computer efficiency, if the display file is so large so as to force a reduction in the refresh rate, the picture will begin to flicker, as the phosphor glow dies before being refreshed. This can be off-putting for the user, and so precludes its use in areas of graphics which continually require complex pictures. The design of integrated circuits falls directly into this category, and so a refresh terminal is best used if some sort of windowing constraint is imposed.

The input device most commonly used in conjunction with the vector scan terminal is the light pen. The pen is basically a photo-transistor which 'sees' over a limited region, and sends an interrupt signal to the terminal whenever the light from the electron beam enters the region. With knowledge of the scanning rate, the position of the light pen at time of interrupt can be calculated.

To be effective, the light pen must be held perpendicular to the screen. This is an unnatural and tiring position to hold for any length of time, plus the pen and/or the user's hand tends to hide part of the layout.

3.6 Raster Scan Terminals [3,4,5,64]

Raster scan terminals consider the screen to be divided up into a matrix of areas (Figure 3.4a). Each area, called a pixel has a value associated with it, and this value is stored in a pixel memory. Consider a simple system in which each pixel is represented by 1 bit of memory. On/off or black/white information is therefore stored (Figure 3.4b).

The picture is produced by computing which pixels to display, and which pixels to omit, then writing this information into the pixel memory. To display the picture, the pixel memory is continuously scanned row by row, hence the term raster. As each bit is read out, it is coverted to an analogue signal, and used to control the monitor.

As mentioned above, the simplest possible system would represent each pixel with 1 bit of memory, giving on/off or black/white information (Figure 3.5a). Better quality displays can be achieved if the each pixel is represented by more than 1 bit. In this situation, the pixel store is best visualized as a series of memory planes, each with equal resolution. The pixel representation is stored in parallel (1 bit per plane) and the outputs can be fed to a DAC to provide grey scale information. A 3-plane system would provide an eight level grey scale and is shown in Figure 3.5b.

The 3-plane system can give colour information if each bit is used to drive the red, green, and blue guns separately (Figure 3.5c). Eight colours (red, green, blue, yellow, cyan, magenta, white, black) are produced, but cannot be altered. A 9-plane system could provide eight levels of red, green, and blue, so producing a palate of 512 colours (Figure 3.5d). Note that only eight colours can be shown at any one time.

Simply increasing the size of the pixel store is really a brute-force solution to the colour palate problem. A better approach is to use a video look-up table (Figure 3.5e). The number of planes is now no longer restricted to a multiple of three, as the outputs from the planes provide the address for the table. Each address in the video look-up table specifies the colour number, and the memory contents

specify the levels of red, green, and blue to be associated with that colour number. The user can load up the table, and so define the colours as required.

Modern raster scan terminals have plane masking facilities, therefore planes may be written to selectively. In integrated circuit mask design, this is a useful feature. For example, if the diffusion layer is drawn on plane 0 and the colour 1 defined to be green, then the poly-diffusion layer is drawn on plane 1, and the colour 2 defined to be red, any intersection between the shapes on the layers will result in colour 3, which can be defined by the user to be a unique colour.

The resolution on raster scan terminals is low compared to other graphic terminals (Typically 512 x 512). This is due partly to the cost of memory, but mainly to the fact that standard T.V. monitors produce 625 lines in the Y direction. If the designer can keep the resolution within standard T.V. limits, then off-the-shelf components can be used in the terminal's manufacture.

At first sight, this seems to be a very poor resolution, but tests have shown that shapes which are in colour, and filled-in, can be identified on the screen as well as, if not better than similar shapes plotted on a monochrome terminal with four times the resolution [65]. Low resolution may cause the terminal to staircase non-orthogonal lines, but as the majority of integrated circuit artwork is Manhattan geometry, this problem is not critical.

3.7 Terminal used by CADIC

The terminal used by CADIC is a SIGMA 5000 microprocessor-based colour raster scan terminal. The schematic and physical layout is shown in Figure 3.6. The GOC (Graphics Option Controller) is linked directly to the host(a DEC2050 time-shared mainframe computer) and all communications pass through the controller. The GOC then directs the information to the downstream VDU's as required.

The GOC contains the microprocessor plus the pixel store, which consists of four display planes and two special planes (polygon, and fill). The display planes give 4-bit pixel representation, and so allows sixteen colours to be viewed simultaneously. Colours can be defined as required with the video look-up table, which provides a palate of 4096 colours.

The special planes are reserved for shape fill exercises. For example, the shapes to be filled are written to the 'polygon' plane. On receiving the 'FILL' command, the microprocessor copies all the pixels outside the shapes into the 'fill' plane. The zero-valued pixels in the 'fill' plane can then be copied into the display planes in any desired colour. A schematic diagram of the pixel memory is shown in Figure 3.7.

The GOC can exist in any one of three states :- Reset, Graphics, and Alphanumerics. The Graphics state is further divided into three modes :- Vector, Command, and Text. Transitions between states/modes only take place when the GOC receives the correct transition trigger. More detailed information on the SIGMA, is given in the user manual [66].

The microprocessor in the GOC provides a range of around a hundred functions, including plane enable, block mode (in which rectangles are specified only by the bottom left hand and top right hand corners), selective erasure, user-specified dashed line, point mode, and shape fill.

Communication carried out in the vector mode can be optimised by entering the Abbreviated Graphics State (AGS). In this state, the x and y coordinates are each represented by two bytes (Hi and Lo). The Hi byte gives coarse positional information, and the Lo byte gives the sensitive positional information. The GOC keeps a note of the last bytes sent, so should for example, a short horizontal line be required, then only the Lo-X byte need be sent. Data transmission savings range from 33% to 83%, with the greatest saving occuring when plotting out horizontal and/or vertical lines. Since integrated circuit layouts are made up of predominantly orthogonal geometry, the AGS is an invaluable facility.

The SIGMA uses a cross-hair cursor as an input device. The cursor is controlled by a hand held control box, which contains five keys :- up, down, left, right, and hit. CADIC programs the alphanumeric keys to replace the function of the hit button, so that on pressing any key, the relevant ASCII code, plus the cursor coordinates are sent to the host computer. CADIC then accepts the code as a command, and uses the coordinates accordingly.

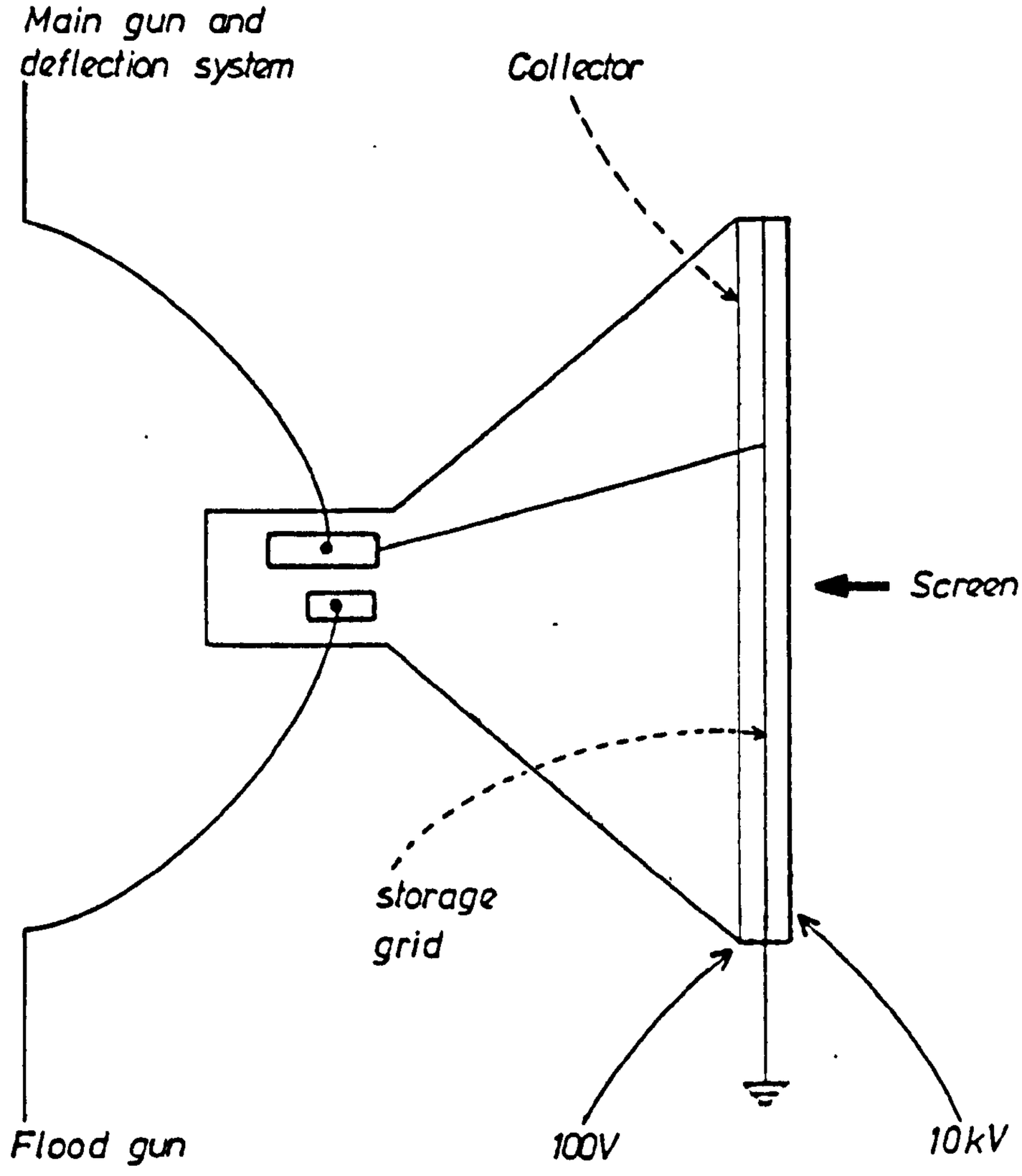


Figure 3.1 Direct view storage tube

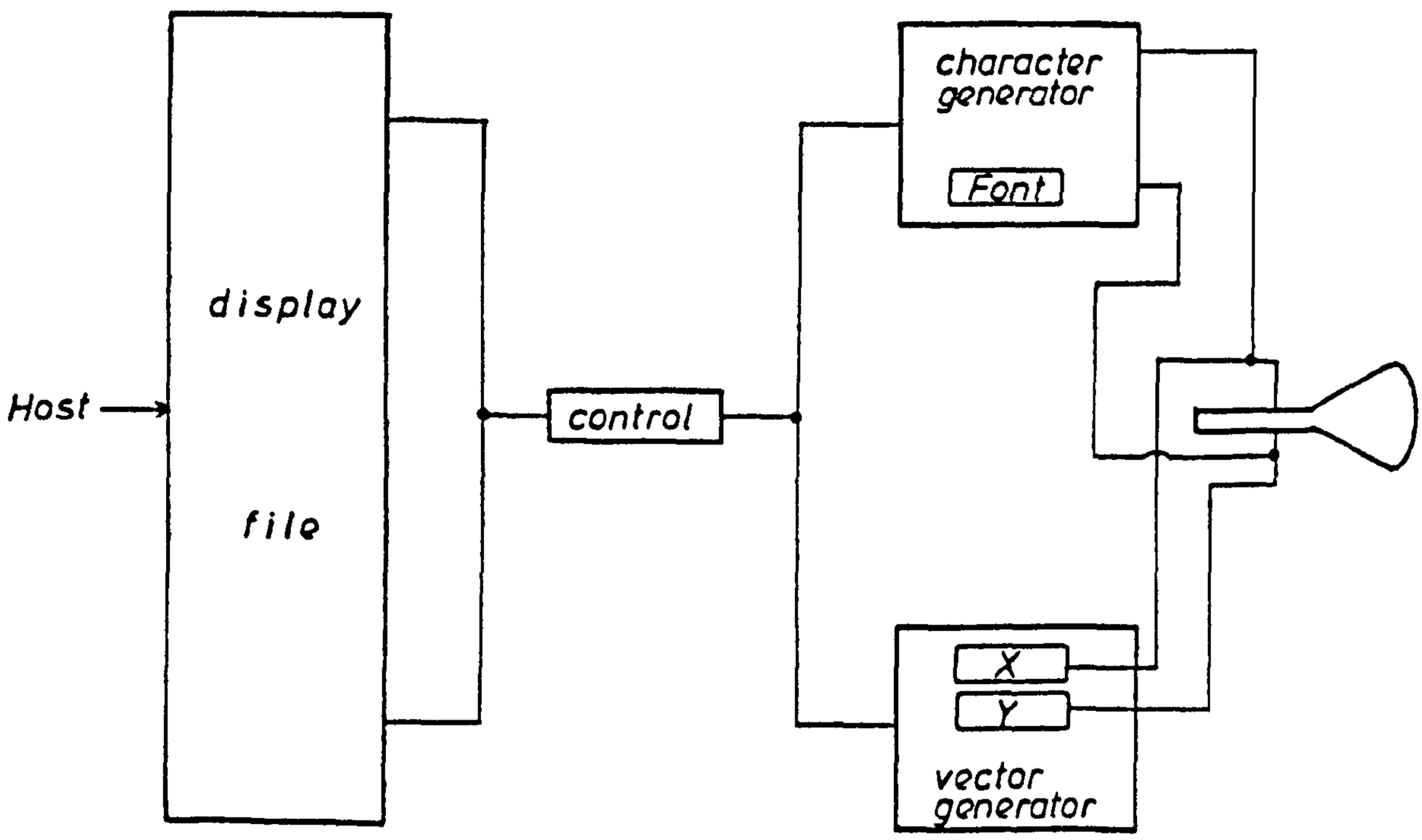


Figure 3.2 Vector scan display

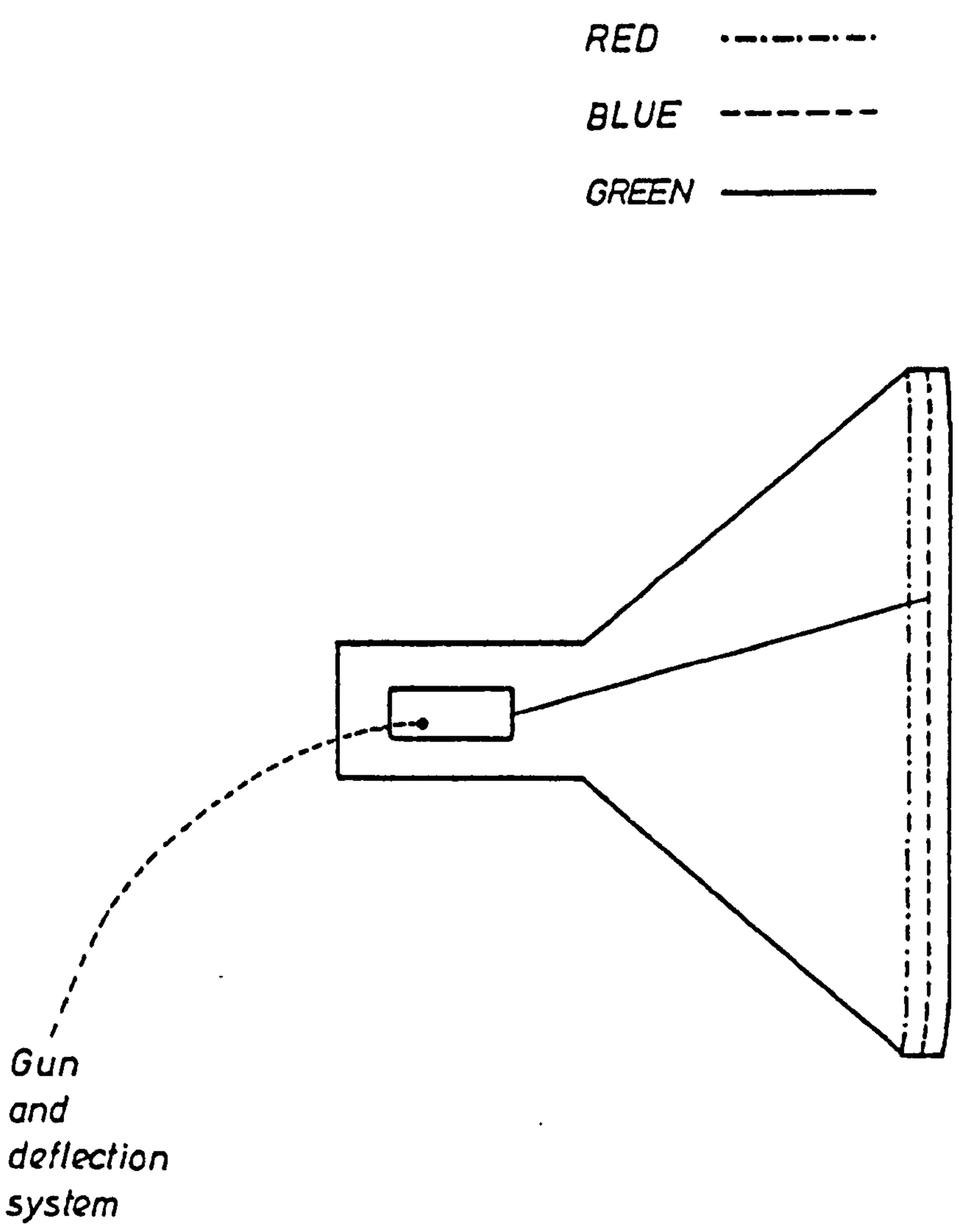
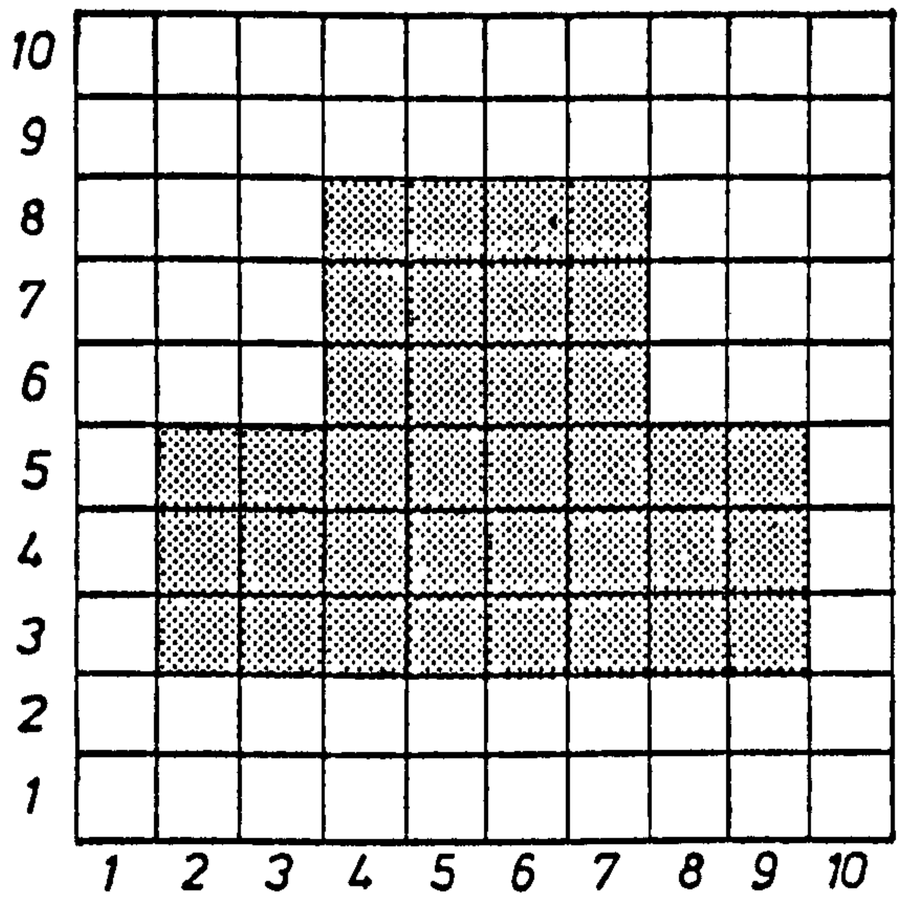


Figure 3.3 Penitron tube



(a)

Raster screen

(Resolution 100)

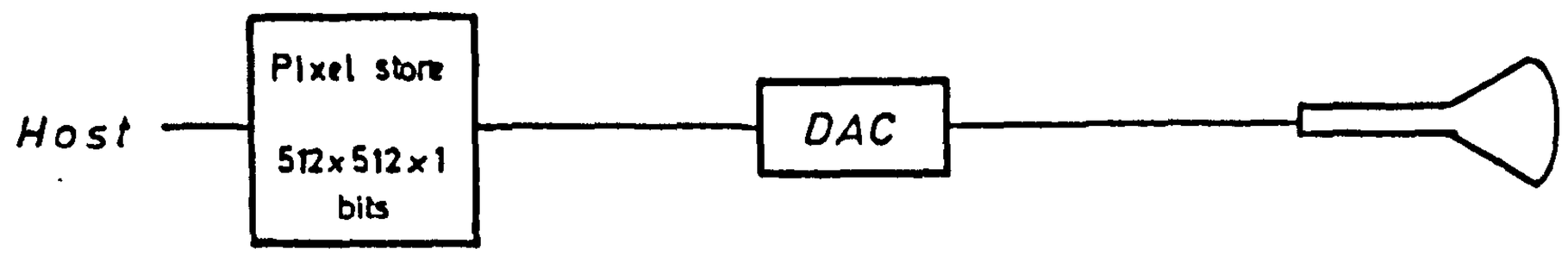
10	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0
8	0	0	0	1	1	1	1	0	0	0
7	0	0	0	1	1	1	1	0	0	0
6	0	0	0	1	1	1	1	0	0	0
5	0	1	1	1	1	1	1	1	1	0
4	0	1	1	1	1	1	1	1	1	0
3	0	1	1	1	1	1	1	1	1	0
2	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
	1	2	3	4	5	6	7	8	9	10

(b)

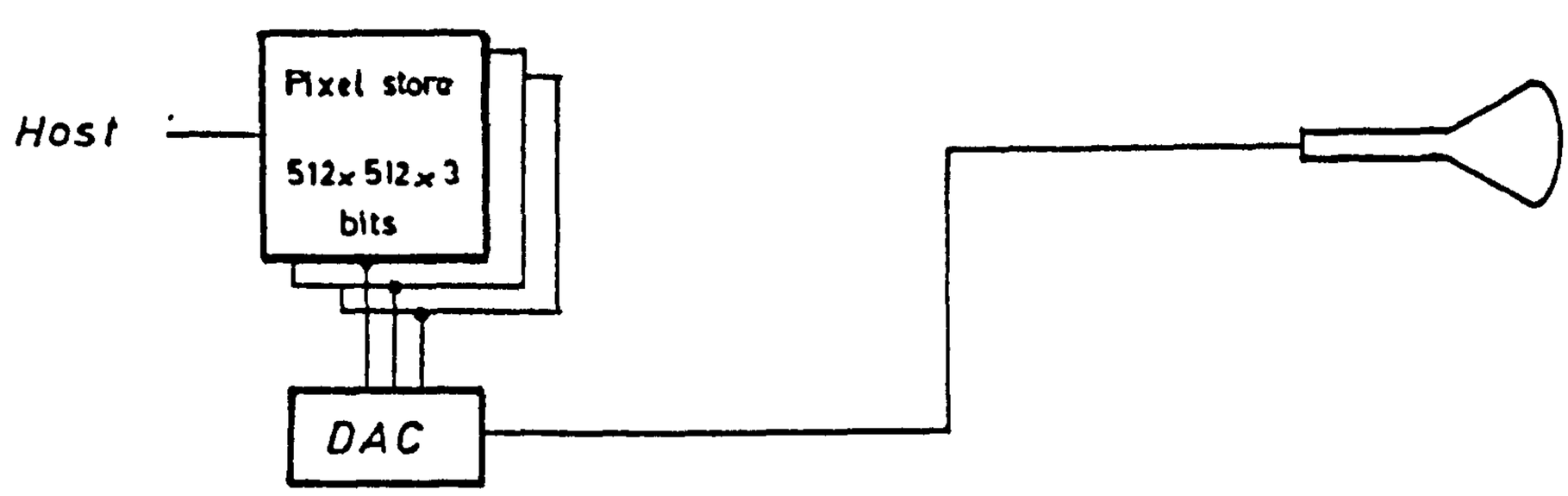
Pixel memory

(10 × 10 × 1 bit)

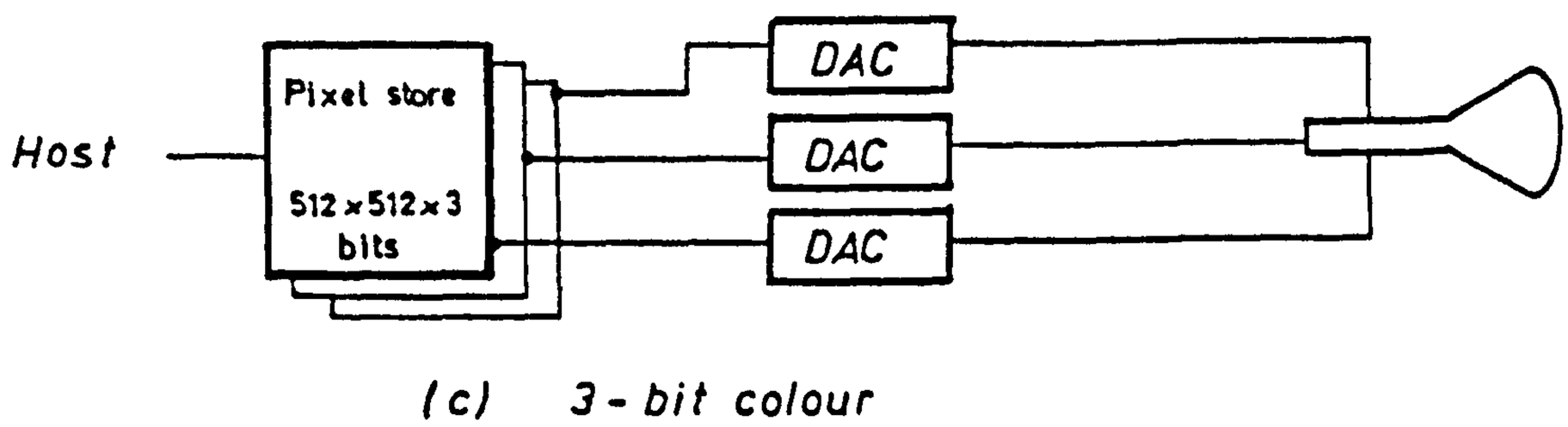
Figure 3.4 Storage of raster scan display



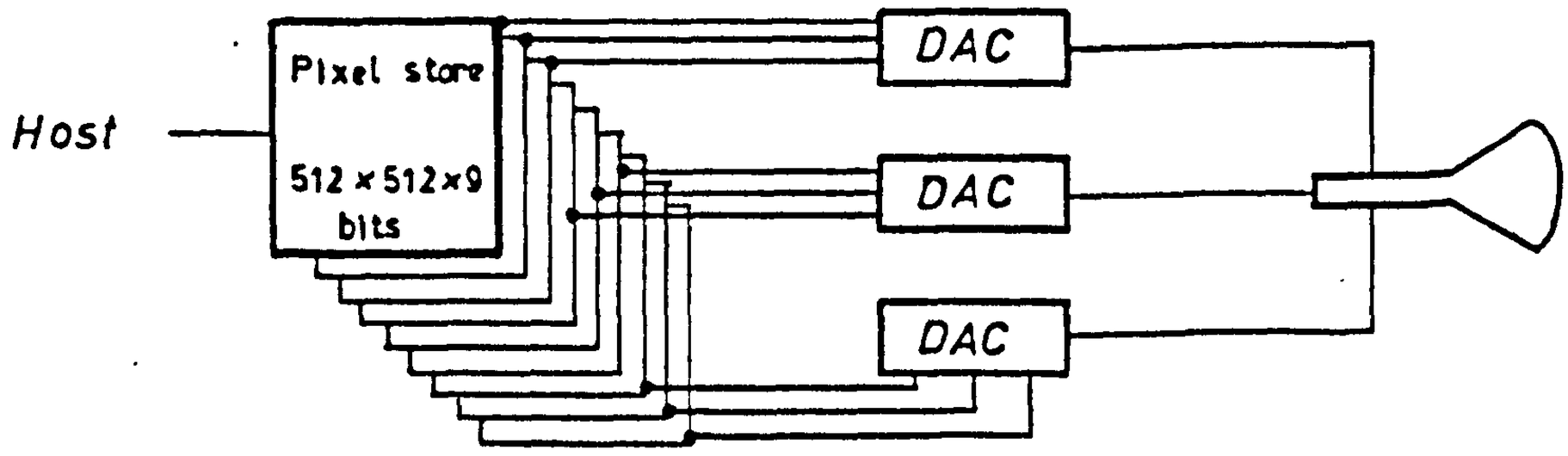
(a) Simple raster scan



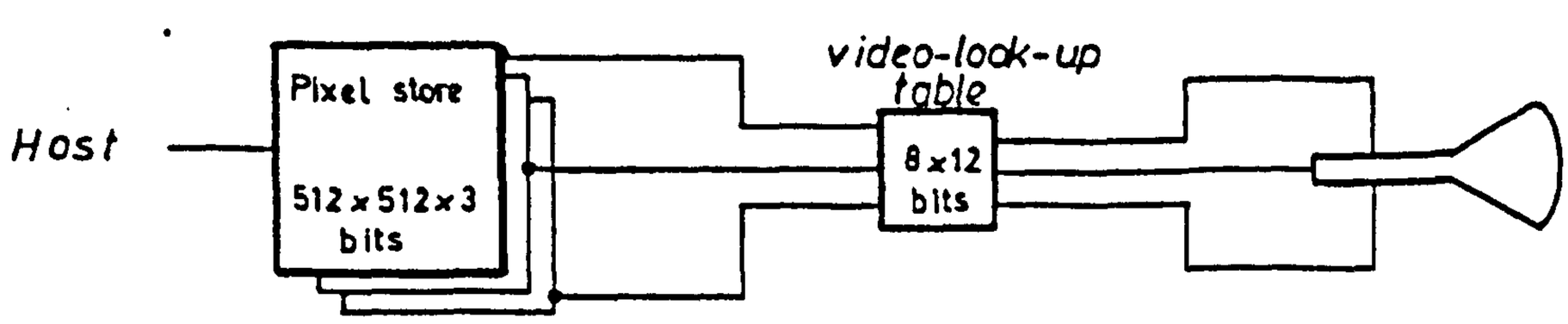
(b) 3-bit monochrome



(c) 3-bit colour

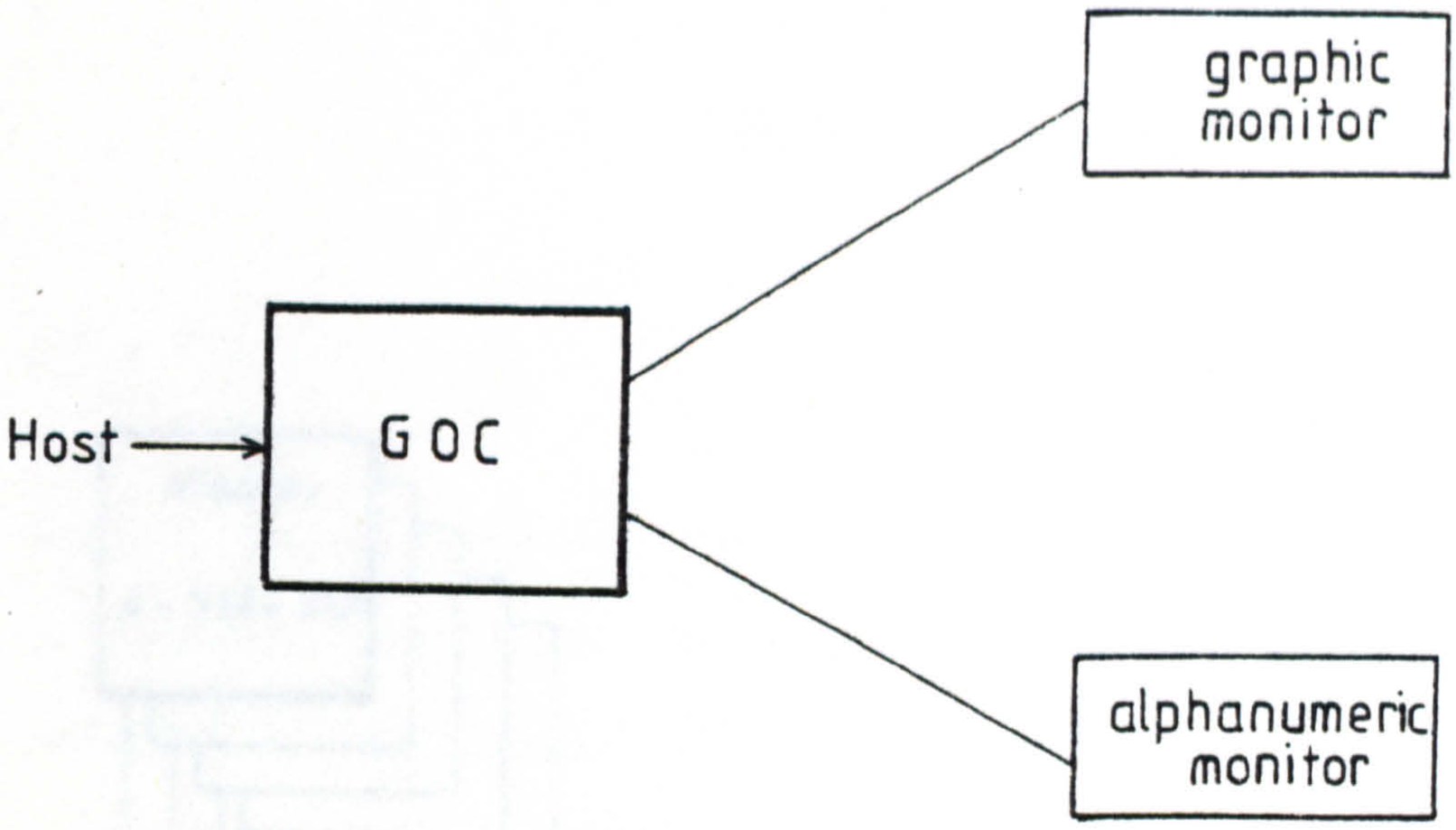


(d) 9-bit colour



(e) 3-bit raster scan with video look-up-table

Figure 3.5



(a) Schematic layout

(b) Physical layout

Figure 3.6 SIGMA 5000 workstation

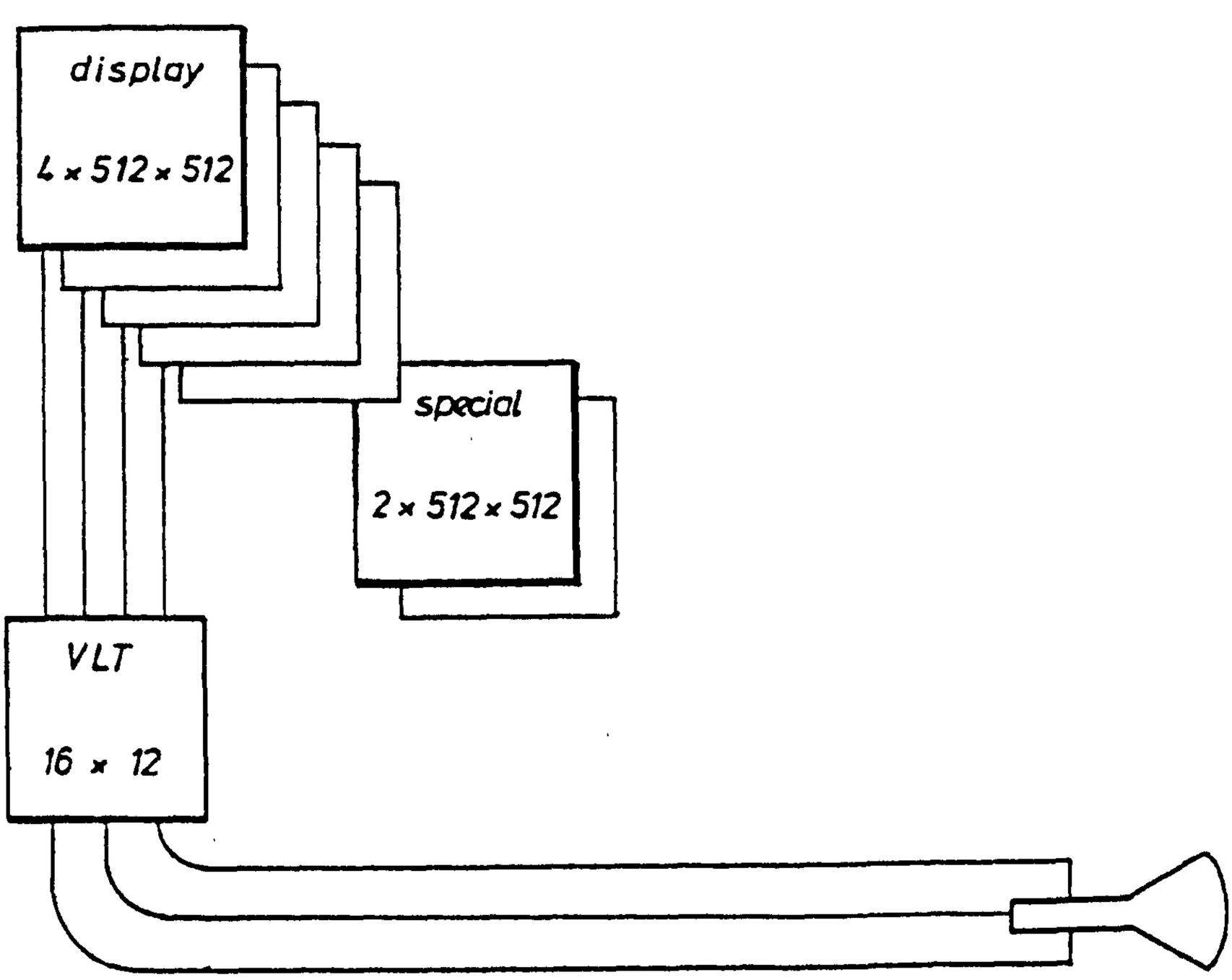


Figure 3·7 Schematic diagram of Pixel memory in the SIGMA system

CHAPTER 4

MANCAD

4.1 Introduction

MANCAD (MANual Computer Aided Design) accepts a manual description of an integrated circuit layout, and converts this description into a ring data structure readable by CADIC1 and CADIC2. This facility is very useful when a graphics terminal is not readily available. The designer can quickly 'code-up' a layout onto sheets of paper, then enter the data into MANCAD using a standard alphanumeric terminal. The graphics terminal is therefore only required to view and/or correct the layout.

The layout can also be design rule checked as it is being compiled using the routines developed in CADIC2 (See later). In this way, MANCAD also provides a very efficient batch-mode or off-line design rule checking facility.

4.2 Choice of manual input language

The ideology behind MANCAD is that a sketch drawn by the designer may be encoded by a non-technical assistant. This leaves the designer free to concentrate on more important areas of the design. The commands in the input language must therefore be easy to use and easy to remember. To save the user time and effort, the input language must also minimise the amount of data that is required.

Only the most optimistic user will expect to enter the layout completely correctly first time. More often, a certain amount of editing will be required, possibly months after the file was originally formed. A file purely consisting of numbers is going to be difficult to understand, so a simple format and a high degree of 'readability' is required in the language.

If MANCAD is to be used as an alternative to CADIC1, it is important that the manual input language allows the user to build up layouts using the same design philosophy. Therefore a format similar to a high level computer language seems an intuitive choice for the input language, since the designer builds up an integrated circuit in a very similar manner to the programmer writing software. For example, the designer collects together shapes (lines of code) to form group definitions (subroutines), and instances (calls) of the group definitions can be added to any other group definition, or main layout.

The deciding factor on the choice of manual input language came about as a result of the requirements of the project, rather than from MANCAD itself. To test the efficiency of the CADIC software, it was important to use large realistic circuit layouts. Compeda Ltd, Stevanage, and Wolfson Microelectronic Laison Unit, Edinburgh kindly provided suitable circuits. The problem is that the circuits were designed on the GAELIC system [1], and the GAELIC layout data structure is not compatible with the CADIC data structure.

To convert the GAELIC data structure to a CADIC data structure, two options were available :-

1. Write software which would convert the GAELIC data structure directly into a CADIC data structure
2. The GAELIC suite of programs contains a program which converts the data structure back into a GAELIC manual input file. If the MANCAD input language is designed to be similar to the GAELIC language, then the equivalent CADIC data structure can be obtained by compiling the manual input file through MANCAD.

It was decided to adopt the latter approach for two main reasons :-

1. No new software is required, thus saving development time
2. The one-stage conversion program can only be used to convert GAELIC data structures to CADIC data structures. Different data structures would require their own conversion program, therefore this approach is very limited in its practical use.

The proposed MANCAD language was therefore adjusted such that it was compatible with the GAELIC language. This language change enhanced MANCAD's qualities for the following reasons :-

1. Over the last few years, there has been a strong international effort to try and standardise systems and software related to computer aided design. By altering its manual input language, MANCAD has introduced a two-way link between systems, which otherwise would not exist.
2. The GAELIC language contains many useful commands, which did not exist in the proposed MANCAD language.
3. The GAELIC language has been used in industry for many years now, and seems to be well liked.

A detailed description of the GAELIC language can be found in the GAELIC user manual [67]. An important point to note is that the MANCAD language includes all the GAELIC commands, except the commands ; "CIRCLE","TEXT",and "LINE". The first two commands were dropped from the language simply because CADIC does not allow circles or text to be added to the layout. The reason for dropping the "LINE" command requires more explanation.

In GAELIC, shapes could contain only light segments. Therefore to produce a shape which contained dark segments (as in Figure 2.1), the designer had to use the "LINE" command to represent the shape as a series of lines. CADIC does however allow dark segments to be included in a shape, so the "LINE" command is no longer required.

As a result of allowing dark segments, another important difference exists between the MANCAD and GAELIC languages. In the MANCAD language, shape coordinates may be preceeded by the letter 'D', which will define the segment going to that point as being a dark segment. In all other cases, the segment is defined by default to be light.

The full range of commands available in the MANCAD input language therefore are as follows :-

"RECTANGLE".....Define coordinates of a rectangle

"POLYGON".....Define coordinates of a polygon

"TRACK".....Define coordinates of a track centre line

"NEWGROUP".....Initialise new group definition

"ENDGROUP".....Close present NEWGROUP command

"GROUP".....Call up instance(s) of group definition

"EXTENDGROUP".....Open existing group definition

"REPLACEGROUP"....Replace existing group definition

"DELETEGROUP".....Delete existing group definition

"REPEAT".....Repeat shapes and/or groups in X or Y directions

"ENDREPEAT".....Close present REPEAT command

"MATRIX".....Repeat shapes and/or groups in X and Y directions

"ENDMATRIX".....Close present MATRIX command

"FINISH".....End processing

Rather than give a detailed description of each command, a feel for the MANCAD language is provided by the following simple example. The example is in fact the manual file used to form the NAND gate layout shown in Figure 2.1. Note that only enough letters to uniquely identify a command are required in the input file.

```
"NEWGR" GATE;
        "RECT" (1) 5,14:4,8;
        "POLY" (2) S,0,5:12,2,-12,D-2;
        "POLY" (2) S,0,9:12,2,-12,D-2;
        "RECT" (2) 4,15:6,7;
        "POLY" (3) S,5,0:4,3,1,9,6,D2,-6,2,-2,7,1,4,-4,
                    -4,1,-7,-2,-13,1,-3;
        "RECT" (4) 6,1:2,2;
        "RECT" (4) 6,13:2,4;
        "RECT" (4) 6,24:2,2;
        "POLY" (5) S,0,0:16,D4,-16,D-4;
        "RECT" () 5,12:4,6;
        "POLY" (5) S,0,23:16,D4,-16,D-4;
"ENDGR";
"GROUP" GATE,0,16,111;
"FINISH";
```

4.3 Program operation

As described in the introduction to this Chapter, MANCAD converts or 'compiles' a manual description of a layout into the CADIC ring data structure. Originally, this was MANCAD's only function. It was soon realised however, that incorporating on-line design rule checking

techniques into MANCAD produced a very powerful off-line design rule checking facility. Both modes of MANCAD's operation are described below.

4.3.1 MANCAD : The compiler

After initialisation, MANCAD asks for the name of the file containing the manual input language, some details on the format of the file, then the name of the data structure to be created. Note that the data structure may already exist, in which case, the shapes in the input file will simply be appended to the specified data structure. In this way, a designer can rely on pre-defined library files to supply all the standard elements and/or layouts required in the new design. Lastly, MANCAD asks for the title to be associated with the layout.

During compilation, MANCAD performs extensive syntax checking on each command in the input file. If no errors are detected in a command, it is accepted by MANCAD, and fully processed. On the other hand, if an error is found, MANCAD will react in one of three ways, depending on the severity of the error.

1. The error will be automatically corrected by MANCAD, and the command accepted.

2. The erroneous command, accompanied by a descriptive warning message will be sent to the terminal, and the command ignored.

3. The erroneous command, and warning will be sent to the terminal, followed by a request by MANCAD to the user to correct the error immediately.

Once the manual file has been completely processed, MANCAD provides the user with three options :-

1. Close the files, and return to monitor level
2. Fetch another manual input file to be added to the data structure. In this way, library files can be loaded as required.
3. Enter data on-line, through the keyboard. Therefore if only a few shapes were rejected by MANCAD, they can be re-submitted correctly. This saves having to edit the relevant manual file, and start the possibly lengthy and involved 'compilation' from the beginning.

An example of compiling the manual input file shown in Section 4.2 is given below. Note that the manual file deliberately contains an error in one of the mask numbers, so as to show how MANCAD handles a typical error.

```
                    - MANCAD -

Program to convert manual input language into a ring data structure
Enter name of manual input file, or return to finish :- NANDG
Does the manual file contain line numbers ? NO
Do you want to include design rule checking ? NO
Enter name of existing ring data structure, or return :-
Enter name of the new ring data structure, or return to finish :- TEST

Enter the layout title : NAND gate

SYNTAX : Group GATE :"RECT" () 5,12:4,6;
         Mask information incorrect - shape ignored


Enter name of next manual file, or TTY for
keyboard input, or press return to finish :- TTY



Enter data - without line numbers
"EXTENDGR" GATE;
"RECT" (5) 5,12:4,6;
"ENDGR";
"FINISH";



Enter name of next manual file, or TTY for
keyboard input, or press return to finish :-

END OF EXECUTION
EXIT
```

4.3.2 MANCAD : The off-line design rule checker

This thesis goes into great detail to explain the advantages of on-line design rule checking over existing off-line techniques. Successful implementation of on-line design rule checking should make the off-line approach redundant. Why then, does the CADIC suite need to provide off-line design rule checking ?

The reason is that under certain circumstances, the off-line approach is the only way to check a circuit, even if it was designed using CADIC. These special cases are discussed below.

Firstly, layouts, or section of layouts chosen from a manual library file will have to be checked in an off-line fashion before being added to the new layout design. The reason for this is that the design rules may have changed since the library was first developed, therefore previously correct layouts may now contain violations.

Secondly, a designer may want to use a layout not designed on CADIC. As with the manual library files, the layout must be checked off-line before it can be used.

Lastly, once the layout is designed and tested, it is ready for fabrication. A large proportion of companies which design integrated circuits do not have an 'in-house' fabrication plant. Therefore these companies must send their designs to a 'silicon house' to be manufactured.

Before starting a design, the 'silicon house' will give the company details on the quality of fabrication possible, so that the design rules

can be defined. As long as the company always implements these rules, the 'silicon house' will be able to fabricate the integrated circuits.

If the company decides to use a different 'silicon house', say for second source or economic reasons, a new set of design rules will be defined. The layout will have to be checked against these design rules to ensure that no violations exist. Once again this process can only be performed off-line.

There are two ways in which the CADIC suite could incorporate off-line design rule checking as a design option :-

1. Classical approach - Develop an independent program to design rule check a finished layout design, using its data structure. Note that the data structure must be in CADIC format.
2. Simulated approach - Off-line design rule checking can be performed by checking the layouts as they are compiled into the CADIC data structure, using on-line design rule checking techniques.

Note that the term 'simulated' in no way implies inferiority. In fact, this approach now shows many superior qualities. The simulated approach was adopted by the CADIC suite for a variety of reasons :-

1. Highly efficient routines, plus the implicit selectivity of the on-line approach allows the simulated approach to be much faster than the classical approach. Note that the layout is checked just as rigourously as any classical technique would check it.
2. The on-line design rule checking routines already exist.
3. Layouts not designed using CADIC almost certainely will not be compatible with the CADIC data structure. The layout must therefore be converted, before it can be checked. Earlier in this

Chapter, it was decided that the convertion problem should take the form of a two-stage process, using the manual input language as a common database. The second stage of this process involves using MANCAD to compile the manual description into a CADIC data structure, so it seems sensible to incorporate the design data structure into this stage.

Off-line design rule checking is incorporated into the MANCAD compilation by answering YES to the relevant question. If design rule checking is requested, then MANCAD will ask for the name of the data structure which contains the rules.

MANCAD processes the manual input file in exactly the same way as described in Section 4.3.1. The only difference now is that before each shape or group call can be added to the layout data structure, it must be design rule checked against the existing layout. The routines to do this are fully described in Chapter 7.

If a violation is identified, the relevant error message is printed out on the alphanumeric screen, along with information on the shape that caused the violation. Note that the shape is still accepted. On completion, the user can use the list of error messages to edit the layout as required.

An example of off-line design rule checking the NAND gate layout shown in Section 4.2 is given below. For sake of clarity, the layout description is assumed to contain no syntax errors. Also the set of design rules for the layout are not shown, but assume that one of the rules specify that the separation between unrelated shapes on mask (2) must be greater than 3 units. For example :-

```
RULE POLYSP;
      MASK PS IS RECT, POLY MASK 2
      FAIL 'Minimum spacing between unrelated poly' IF &
          SEPARATE (PS,PS) AND SPACING (PS,PS) < 3
END
```

N.B. For a description of the design rule input language, see Chapter 6.

The two polygons on mask (2) which form the inputs to the NAND gate do not satisfy rule POLYSP. To design rule check the layout, MANCAD therefore proceeds as follows :-

```
                              - MANCAD -

Program to convert manual input language into a ring data structure

Enter name of manual input file, or return to finish :- NANDG

Does the manual file contain line numbers ? NO

Do you want to include design rule checking ? YES

Enter name of file containing the design rules :- DRCRUL

Enter name of existing ring data structure, or return :-

Enter name of the new ring data structure, or return to finish :- TEST


Enter the layout title : NAND gate


          Minimum spacing between unrelated poly
DESIGN : Group GATE: "POLY" (2) S,0,9:12,2,-12,D-2;


Enter name of next manual file, or TTY for
keyboard input, or press return to finish :-

END OF EXECUTION
EXIT
```

CHAPTER 5

CADIC1 : The graphic design aid

5.1 Introduction

CADIC (Computer Aided Design of Integrated Circuits) is an interactive graphic design aid, which allows the user to design integrated circuit layouts at the geometric level. This approach was one of the first types of design aids made available to the designer, yet it can still produce more compact layouts than by alternative techniques. CADIC is split into two sections :-

1. CADIC1, which allows the designer to build up and/or edit the mask layouts.
2. CADIC2, which performs all the design rule checks on a newly added shape (if design rule checking is required).

These sections never work simultaneously. For example, after adding a shape using CADIC1, CADIC2 takes control and applies the design rule checks. Only when CADIC2 is finished can CADIC1 regain control, and allow another shape to be added. For the sake of clarity, each section is allocated it's own Chapter. CADIC1 is described in this Chapter, and CADIC2 is described in Chapter 7, after certain concepts about design rule checking have been discussed.

5.2 Requirements

CADICl's first requirement is that it must allow shapes to be added to the mask layouts. This feature is probably the most fundamental of them all, yet many design aids put heavy restrictions on the format of the shapes. For example, some design aids only accept rectangles [4]. Complex shapes must be segmented by the designer, which distracts his attention from the design problem. No restriction on shape format is therefore required in CADICl.

CADICl must also allow the designer to delete or move any shape in the layout, so that errors can be corrected.

In any problem, a designer, sometimes subconciously, will break the problem down into smaller, more manageable modules. In design of the integrated circuits, the same hierarchical process must be made available. CADICl should allow the designer to define a collection of shapes as a group definition, for example the circuitry that makes up a shift register cell (CELL). This group definition can then be used as a group instance in a group definition SHIFT REGISTER which includes several calls to CELL. The group facility therefore allows the designer to add and/or remove possibly complex sections of layout quickly and easily, so speeding up layout design.

As the artwork is built up, the user will almost certainly want to study the layout at a variety of scalings. For example a large scaling to check inter-shape dimensions, or a small scaling to examine the

overall topology of the layout. To accommodate such desires, CADIC1 must provide a large range of windowing facilities, in conjunction with a redraw facility.

The above requirements are the building blocks of any graphic design aid. Many other features can be added to ease the designers task, which are of lesser importance, yet very useful. For example draw out a set of axes to help the designer position shapes accurately, or find the nearest point in the layout, so that shapes can be 'tagged' on to it. It is the provision of these secondary features that often determines whether a graphic design aid is good or bad, so it is very important for CADIC1 to provide a concise range of this type of feature.

When comparing different graphic design aids, the quality of the output is also very important. For reasons mentioned earlier, the terminal chosen for CADIC1 was a SIGMA 5000 microprocessor-based colour raster scan terminal. A specialized graphics package is therefore required, so that the terminals unique features can best be utilized by CADIC1.

Even with computers becoming very common in everyday life, a fair amount of sceptisism exists when a user is introduced to a computer design aid. Any interactive graphic artwork package must therefore be as natural to use as pen to paper. Commands must be easy to use, and easy to remember. The design aid must also have as few restrictions as possible, so that the user can utilize his ingenuity to the fullest.

Program response to a command must be fast, or else the user will quickly become bored. The user with ten Cray 1's in his head, plus additional visual feedback can identify a point in a layout almost immediately, and expects a computer program to be able to do the same. Program response time is dependant on three main factors :-

1. Time sharing delays
2. Processing delays
3. File handling delays

Nothing can be done by the programmer to improve time-sharing delays, so the time must be made up elsewhere. Processing delays can be reduced with careful programming, such as performing integer arithmetic whenever possible. Reading data from a file on disc may be as much as 1000 times slower than if the data had been resident in core. The size of data stuctures required to store an integrated circuit layout force the use of disc storage, therefore the greatest improvements in response time can be achieved by efficiently handling the disc-based data.

In a time-sharing environment, data required by a program has more chance of being processed if the amount of data in core is limited to only a few pages at any one time. CADIC allocates six pages of core for data, and swaps information to and from the data file as required. While the pages are in core, the data can be accessed very quickly. Should a page not in core be required, then a page already in core must be written back to disc, and the required page read into core. This is termed page swapping or paging for short. Paging is very expensive in

terms of time, and so the data structure holding the layout information must be arranged in such a fashion so as to minimise the amount of page swaps required during processing.

5.3 Logistics

The point has been made that decreasing program response time requires more efficient data handling. Integrated circuit technology requires very large data structures to describe the layout. Therefore, it is worth considering ways in which information to be searched can be cut down.

One idea is to divide the board up into several areas, and store all shapes that lie in the same area together in the data structure, along with some sort of area identifier. This approach is of no benefit when plotting out the whole board, but if the user windows in to only a small section of the board, then the program can calculate the area(s) of interest, and plot out the shapes only in the relevant area(s).

The problem with this approach is how to define shapes that lie in two or more areas. A simple approach is to store all such shapes in a special area, for example area '0' [68]. When redrawing a section of the layout, only the areas in the window, plus area '0' must be processed. In large layouts, area '0' may contain many more shapes than any of the other areas, and so a lot of time is wasted checking it, possibly to no avail.

In an attempt to level out the distribution of shapes to an area, GAELIC [1] only stores shapes that lie in three or more areas in area '0'. Shapes that lie in two areas are stored in the area in which the shape's bottom left hand corner is positioned. Area '0' has certainly been reduced, but now shapes associated with an area can travel over the area's top and right-hand boundary. The effect of this is that when a small region is to be plotted (say within one area), then the area concerned must be checked, plus the three adjacent areas (left, below, and diagonal), then area '0'. Therefore a minimum of five areas must be checked.

To ensure that only the areas that actually enter the window need be searched, CADIC1 does not use the concept of area '0'. Instead shapes which cross area boundaries are treated in a new way.

The only way to ensure that a shape that enters more than one area is associated with an area and no others is to 'polygon clip' the original polygon into a number of sub-polygons, and store the sub-polygons as independent shapes. (See Figure 5.1).

The sub-polygons must now contain dark segments where they were cut by the area boundaries, so that these segments will not be seen by the designer. In this form, watching a layout being plotted out on a DVST terminal may prove confusing, especially if the transmission speed is low. A track which extends across most of the layout would be drawn out in sections, as the program processes each area in turn, rather than being drawn out all at once. Of course the problem disappears once the

plot is finished, as the layout is now identical to that achieved if the shapes had not been cut up.

Vector scan terminals may also have a problem with the sub-polygon approach, because the increase in individual polygons may cause the display to flicker, even if the layout appears to be rather sparse.

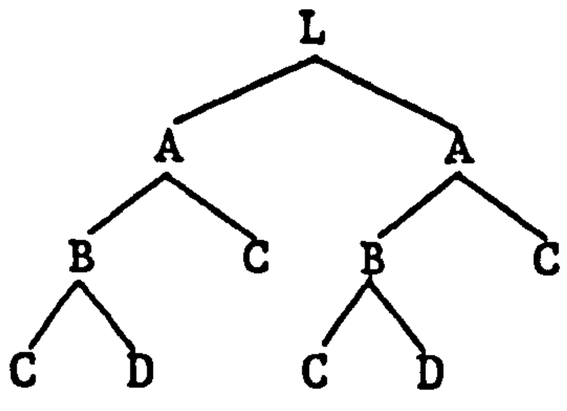
When using the SIGMA terminal in the 'FILL' mode, the layout is not drawn immediately onto the screen. Instead, each mask is drawn onto the 'invisible' polygon plane. The SIGMA then fills the shapes, then copies the 'filled' mask onto the screen. Because the mask only becomes visible once it is complete, segmentation of the polygons is never seen by the user.

Another way in which the amount of information to be checked can be cut down is to associate each shape in the layout with a bounding rectangle. The dimensions of this bounding rectangle defines the size of a rectangle that would be required to fully enclose the shape. Using this information, CADICl can often ignore a shape, and all its coordinates, just by checking the bounding rectangle.

Once the decision has been made as to what to put into the data structure, the problem of how to arrange the data becomes relevant. New data is always added to the end of the data structure. Therefore as the layout is built up, CADICl will find it increasingly difficult to keep useful pages in core for any length of time. As a consequence, excessive page swapping will occur, and CADICl's efficiency will

decrease. To overcome this problem, CADIC1 provides a 'CLEAN' command, which will re-organise the layout data such that data of a similar nature is stored on the same page, or consecutive pages. In this way useful information will remain in core longer, and so increase program efficiency.

The efficiency of data-handling is also dependant on the order in which the data is handled. For example consider a layout 'L' which contains group definitions 'A','B','C','D' arranged in the following hierarchy :-



While plotting out the whole layout, many design aids, on finding a group instance would jump to the group definition immediately, then plot out its shapes and instances and so on. This continues until every branch of the hierarchical tree has been processed. The order in which groups would be handled is thus :-

L A B C B D B A C A L A B C B D B A C A L

In a typical layout, CADIC's pre-defined core allocation of six pages will hold all the shape information for <u>one</u> group definition. Therefore each time a new group is processed, all the old information must be swapped for new information. In the above example, twenty group

transitions occurred, which would result in 120 page swaps (worst case). Excessively page swapping the same information is termed page thrashing, and is to be avoided at all times.

CADIC1 avoids this problem by obtaining a more global knowledge of the group hierarchy. If layout 'L' is to be plotted out, all the shapes in 'L' are plotted, then information about the group instances called from 'L' are stored in a temporary file. (In the above example, this would be two calls to 'A')

CADIC1 then identifies the first group instance in the file (Group 'A'), and brings the relevant group definition into core. All the shapes in group 'A' are then plotted out, and any group calls identified (in this case, one call to 'B', and a call to 'C') are added to the temporary file. The file is then searched to see if any other instances of group 'A' exist, so that the group definition information can best be utilized while in core.

Once the other instance of group 'A' is found in the file, all it's shapes are again plotted out, at the new position. Group calls 'B' and 'C' are then added to the temporary file. No more instances of 'A' are found, but the temporary file still contains two calls to 'B' and two calls to 'C'. CADIC1 therefore goes to the top of the file, identifies group B, then brings the relevant group definition into core. The above process is then repeated until all group instances in the temporary file have been plotted. Using this technique, the order in which CADIC1 would handle the group hierarchy in the above example would be :-

L A A B B C C C C D D

Note that now only four group transitions occurred, resulting in 24 page swaps (worst case). The saving in CPU time is therefore obvious.

5.4 Program operation

CADIC1 is a graphical design aid implemented in FORTRAN, which through the provision of simple commands allows the user to build up and/or modify an integrated circuit layout. The command structure used by CADIC1 is shown in Figure 5.2 and is described below in hierarchical order.

Initialization : The initialization stage simply sets up all the program variables and terminal conditions, then allows the user to specify which layout he wants to build/modify.

Main Command level : After initialization, CADIC1 enters the main command level. At this level the user cannot alter the layout in any way. However, the user can plot out the layout and/or set up the correct conditions in preparation for editing the layout. The available commands are briefly described below :-

ADJUST ... Adjust mask colour settings

AXIS ..... Draw axis on screen (Switch)

CHANGE ... Change name of group definition or instance

CLEAN .... Clean up the data structure

CURSOR ... Change cursor grid

DEPTH .... Change depth of group nesting to be plotted

EXIT ..... Exit from program

FILL ..... Fill in shapes (Switch)

GROUP .... Enter group definition

HELP ..... Write out this list of options

INFORM ... Inform user of all program settings

LIST ..... List out group names

MODIFY ... Modify layout/group definition

NET ..... Draw out a net of grid points (Switch)

ONLINE ... Perform on-line design rule checking (Switch)

ORIGIN ... Plot out group origins

PLOT ..... Plot out shapes on selected masks

SAVE ..... Save a copy of the data structure

SWITCH ... Switch off/on design rules

TRACK .... Change track width

WINDOW ... Change window dimensions

Note that only the first two letters of each command need be typed to uniquely identify a command.

Cursor Command level : On typing MODIFY at the main command level, the program drops down to the cursor command level, and the cross-hair cursor appears on the graphics screen. The user can now edit or inspect the layout, using the range of commands shown below :-

SPACE ... Return to main command level

- ..... Remove mask from plot list

0 => 9 .. Plot out mask (add mask to plot list)

? ..... Print this list

C ..... Add a collection or array of group instances

F ..... Find nearest point in layout (including groups)

G ..... Add a single group instance

I ..... Identify nearest point in layout (without groups)

J ..... Jump back to full layout

K ..... Kill shapes

L ..... Redraw last window used

M ..... Change mask to be worked upon

P ..... Add polygon

Q ..... Query distance between two points on screen

R ..... Add rectangle

T ..... Add track

U ..... Undefined zoom

V ..... Verify present cursor position

W ..... Redraw layout with present window size

Z ..... Defined zoom in

[..... Kill group instances and/or arrays

a ..... Draw axis once

n ..... Draw net once

w ..... Specify new window size

Subsequent cursor command level : Most cursor commands will perform their function then return to the cursor command level. On typing some commands, for example 'P' for Polygon, the program will drop down to the subsequent cursor command level. In the case of the 'P' command, the subsequent cursor command level is primarily concerned with adding points to the shape and/or modifying previous points if not correct. The subsequent cursor commands for adding polygons are given below :-

A ..... Add angled light segment

O ..... Add orthogonal light segment

X ..... End polygon with angled light segment

E ..... End polygon with orthogonal light segment

a ..... Add angled dark segment

o ..... Add orthogonal dark segment

x ..... End polygon with angled dark segment

e ..... End polygon with orthogonal dark segment

K ..... Kill shape

N ..... Finish segment on nearest point already in layout

S ..... Finish segment at new cursor position

..... Finish segment at point entered through the keyboard

? ..... Print this list

A detailed description of CADIC1 operation, plus all the available commands is given in the CADIC1 user manual (Appendix A).

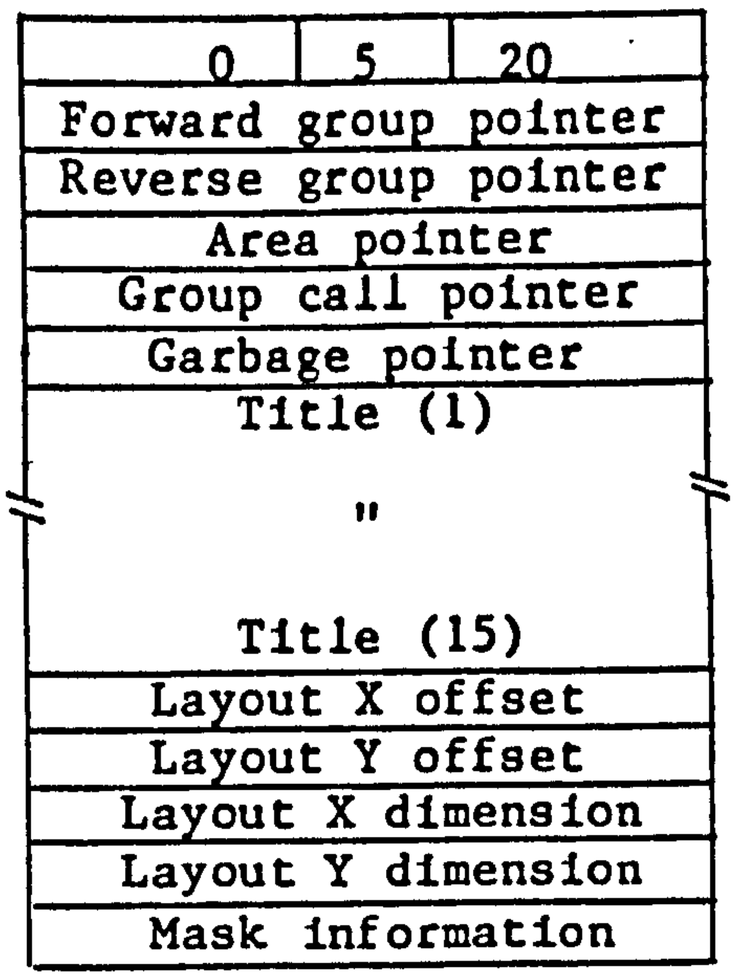
5.5 Data Structure

The data structure used by CADIC1 is classified as object orientated. This means that each shape is represented in the data structure by a block or bead of memory which stores the necessary information. Each bead is given a pointer which points to the next bead in the sequence. Searches involving all the possible occurrances of one particular type of bead area is therefore very selective if the pointer scheme is employed. By definition, the last bead points back to the first bead, and so forms a loop or ring. In this thesis, the data structure is therefore called a ring data structure.

Any interactive design aid will involve adding, deleting, and plotting shapes, so the data structure employed must be able to cope with these operations efficiently. A ring data structure satisfies all these conditions, and so was an ideal choice for CADIC1. For example, deleting a shape means removing the relevant bead, and simply involves adjusting the pointer in the previous bead, so that it now points to the bead after the one to be removed (See Figure 5.3). Adding shapes uses the reverse process.

A schematic representation of the ring data structure used by CADIC1 is shown in Figure 5.4. At first sight, it may look complex, so consider firstly a layout containing no group definitions (Figure 5.5).

Layout Headbead : This bead is the first bead in the data structure, and provides all the rings ready for adding area beads, group definitions, and group instances. It's form is as follows :-



The first byte in any bead gives information about the bead itself. To save on space, the byte is split into three sections or fields. Contained in the fields are NTYP, ND, NP. 'NTYP' is an integer such that beads with different properties can be identified. 'NP' defines the number of bytes in the bead that are used as pointers, and 'ND' defines the number of bytes in the bead that are used for data. Therefore the total size of the bead is 1+NP+ND.

The group rings will hold all the group definition headbeads (see later). The area ring holds all the area beads which are used to identify the position of shapes in the layout. If group definitions exist, then instances of these group definitions may exist in the layout. Information about these instances are held in group call beads on the group call ring

Beads which have been removed from the data structure are inactive as far as CADIC1 is concerned, but they still occupy space in the data structure. All inactive beads are therefore stored on the garbage ring, and re-used whenever possible. For example, if a bead is required, CADIC1 will first search the garbage ring to try and find if a suitable bead exists. If yes, then the bead is used, otherwise a new bead is formed at the end of the file.

Any layout designed may be filed away for later use, and so a facility for giving an identification title is provided in CADIC1. Fiveteen bytes of the layout headbead store text as two characters per byte, allowing a 30 character title. The layout bounding rectangle is also required by CADIC1, and this information is stored in the headbead, after the title. Lastly, the mask word is considered as 16 bits, 1 bit per mask. These bits are set to 1 if the relevant mask contains shapes, and 0 if not. By reading in the mask word, CADIC1 immediately knows whether searching for shapes on the required mask is going to be futile or not.

Area beads : To increase program efficiency the layout is defined to be partitioned into areas. The area ring therefore holds the area beads which give information about which area the shapes are in. The form of an area bead is as follows :-

1	2	4
Area pointer		
Mask pointer		
Area X min.		
Area Y min.		
Area X max.		
Area Y max.		

The area pointer simply points to the next area bead on the ring. In an integrated circuit layout, each shape is placed on a specific mask layer. An area bead therefore contains a mask ring, which contains mask beads, and so describes which masks have shapes in the area defined by the area bead. Lastly, the X and Y coordinates in the area bead define the position of the area on the board.

Mask beads : A mask bead is as follows :-

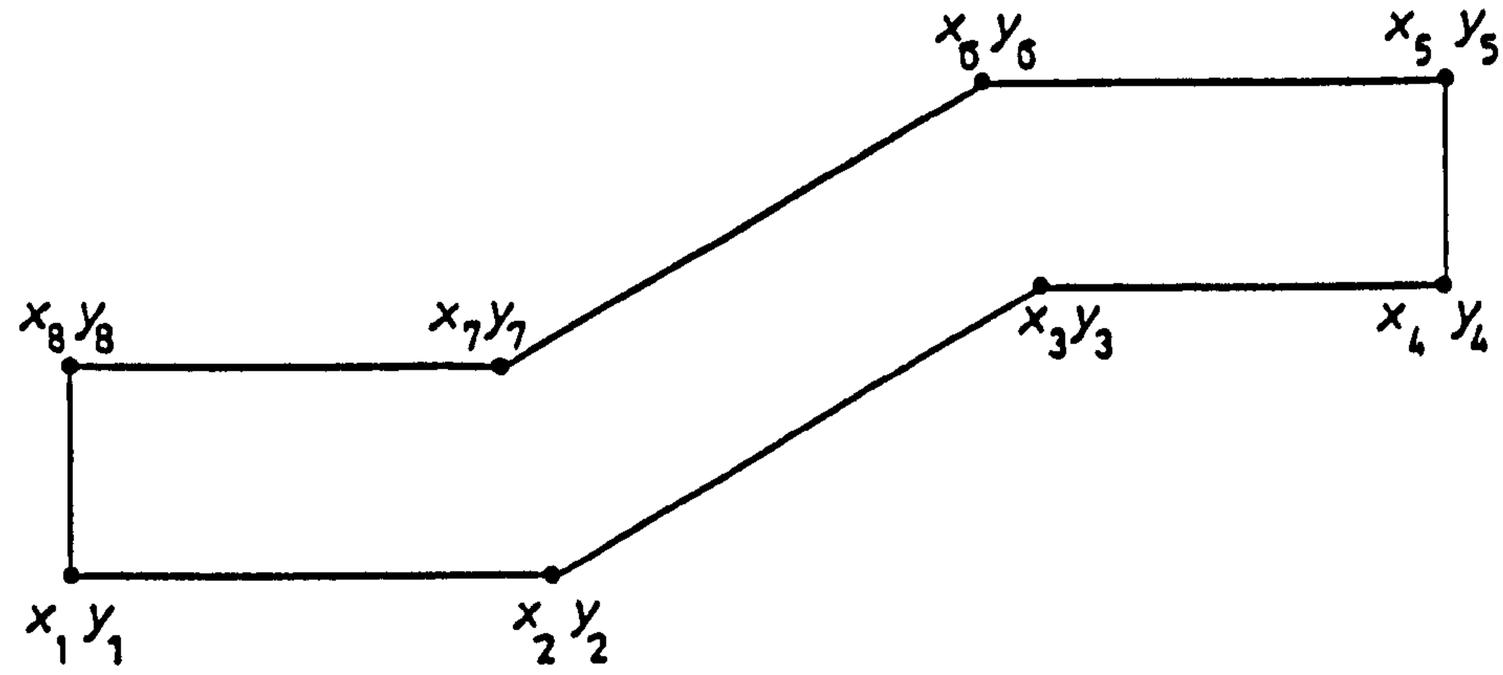
2	2	1
Mask pointer		
Shape pointer		
Mask number		

As described before, the mask pointer points to the next mask bead on the ring. The mask number defines the mask layer. At this stage, the shape information can be added on the shape ring, as the area and the mask layer have now been defined.

Shape beads : There are three main forms of shape bead used in the data structure :-

1. Long format polygons
2. Short format polygons
3. Rectangles

A long format polygon is the most general type of shape, and all shapes which contain angled segments come into this category. i.e. :-



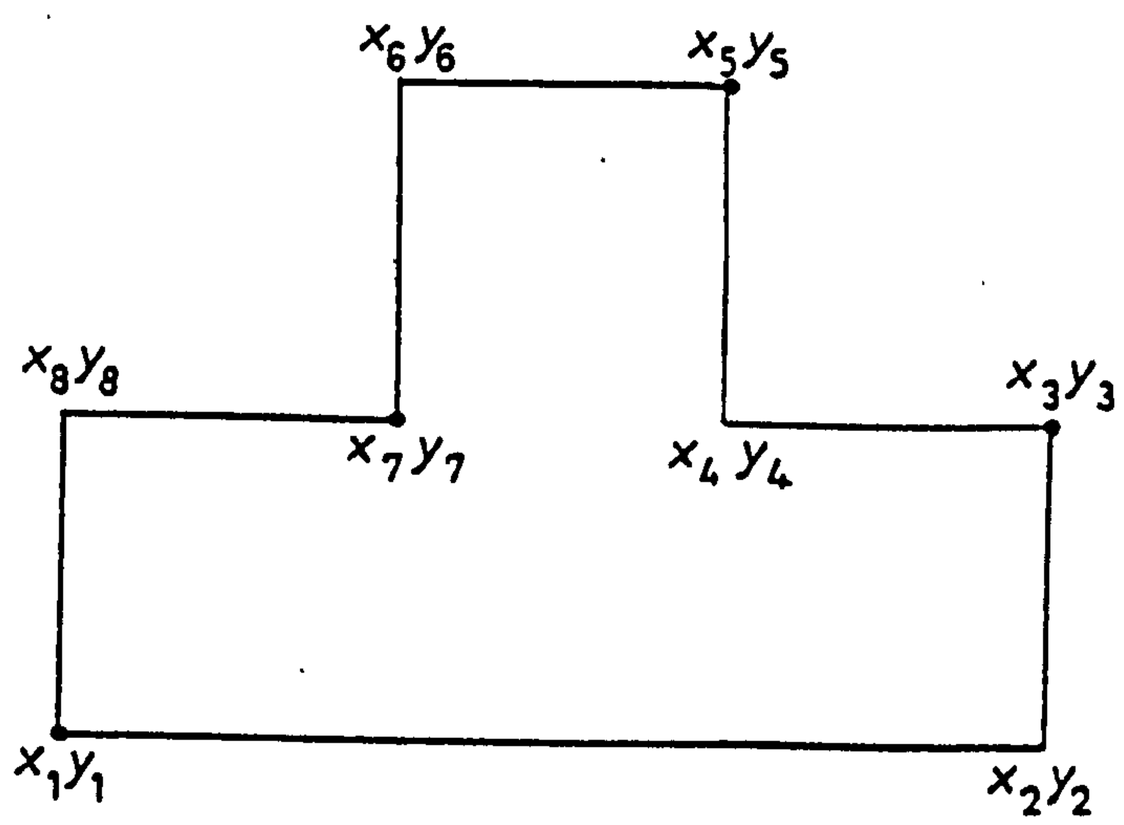
The long format shape bead is as follows :-

7	1	2n
Shape pointer		
B. Rectangle X min		
B. Rectangle Y min		
B. Rectangle X max		
B. Rectangle Y max		
xl		
yl		
"		
xn		
yn		

The shape pointer points to the next shape bead on the ring. Next comes the coordinates of the shape's bounding rectangle, followed by the coordinates of every point in the shape, one byte per coordinate. In CADIC1, all coordinates stored in the shape bead are actually those which define the shapes offset from the bottom left hand corner of the area, rather than the absolute coordinates of the shape. The reason for this is that CADIC1 has been designed for possible operation on a 16 bit computer. The range of coordinates required in an integrated circuit layout is now too large for 16 bit representation, therefore each

absolute coordinates would require two bytes. By storing the shape coordinates as offsets, and defining an area to be no larger than 64383 increments, then the shape beads only require one byte per coordinate. The saving in memory is therefore obvious.

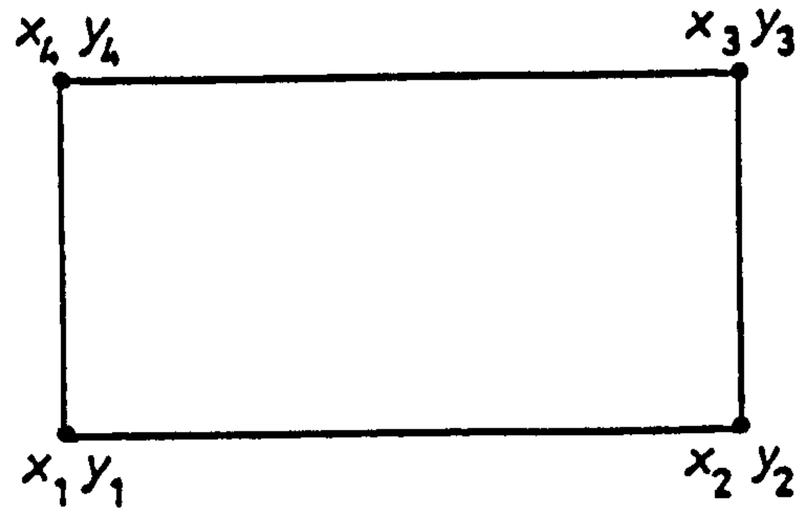
A special case of the long format polygon is the short format polygon. This type of shape contain only orthogonal segments, for example :-



In this type of shape, only every alternate coordinate need be stored, since (x2,y2) is also (x3,y1), and so on. Short format polygon beads therefore have the form :-

5	1	2n
Shape pointer		
B. Rectangle X min		
B. Rectangle Y min		
B. Rectangle X max		
B. Rectangle Y max		
x1		
y1		
x3		
y3		
"		
xm		
yn		

A special case of the short format polygon is the rectangle. The important coordinates now are the bottom left hand corner, and the top right hand corner.



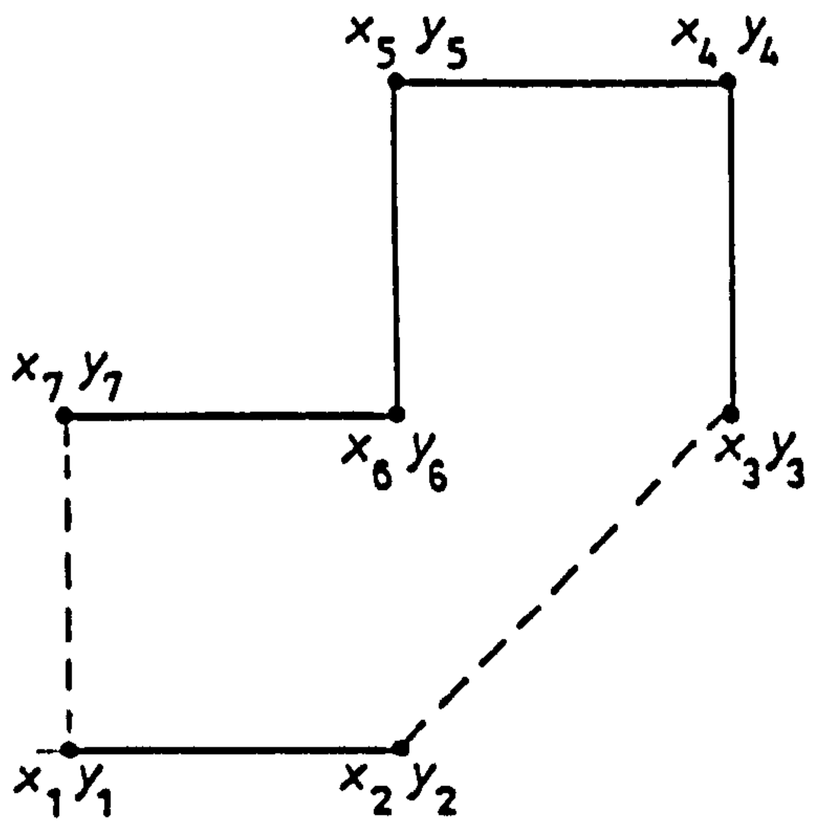
The reason for a separate bead type is that the rectangle coordinates are also the bounding rectangle coordinates, so obviously the bounding rectangle information is now no longer required. The form of a rectangle bead is as follows :-

3	1	4
Shape pointer		
x1		
y1		
x3		
y3		

Earlier it was described how polygons which lie over area boundaries are cut into sub-polygons. These sub-polygons or <u>open polygons</u> contain both light and dark segments, which must be represented somehow in the shape bead.

If the size of each area on the board is limited to 16383 increments, then CADIC1 can use the second most significant bit of each coordinate byte to store the information. If set, the segment is defined to be dark, otherwise the segment is defined to be light, for

example :-



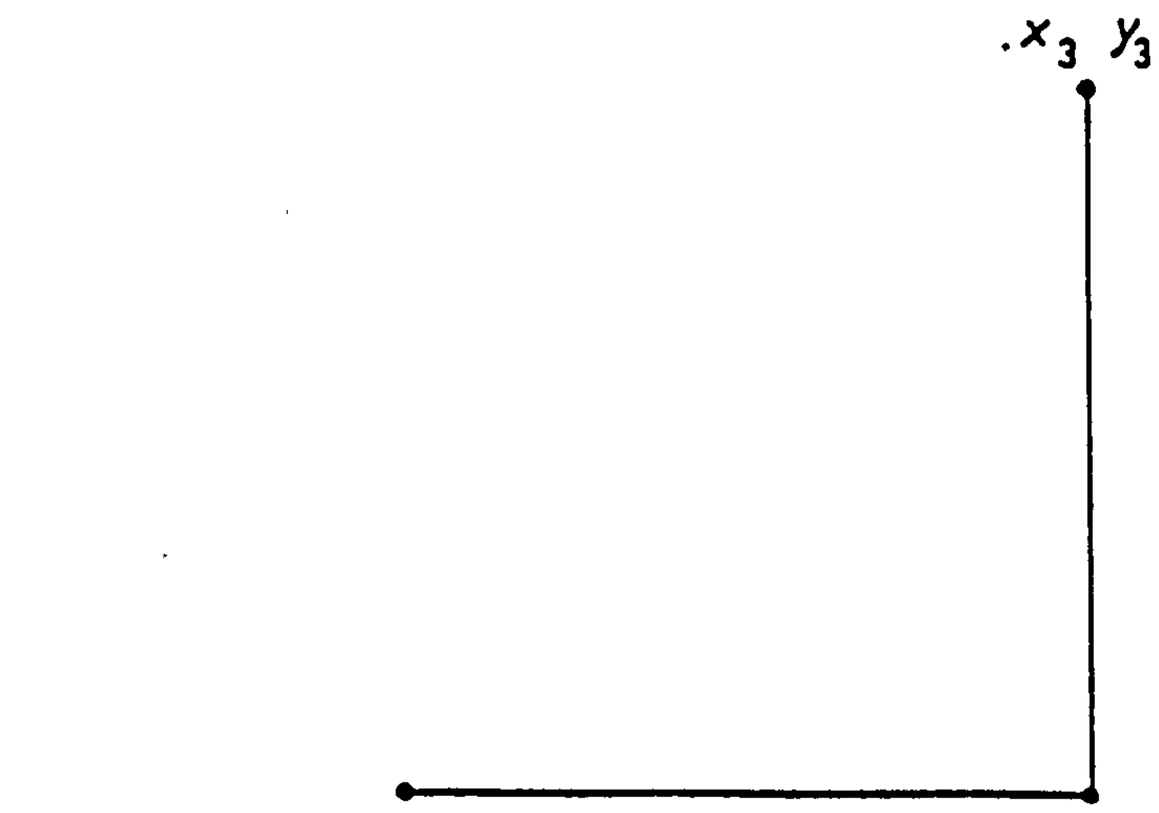
—— *light segment*

---- *dark segment*

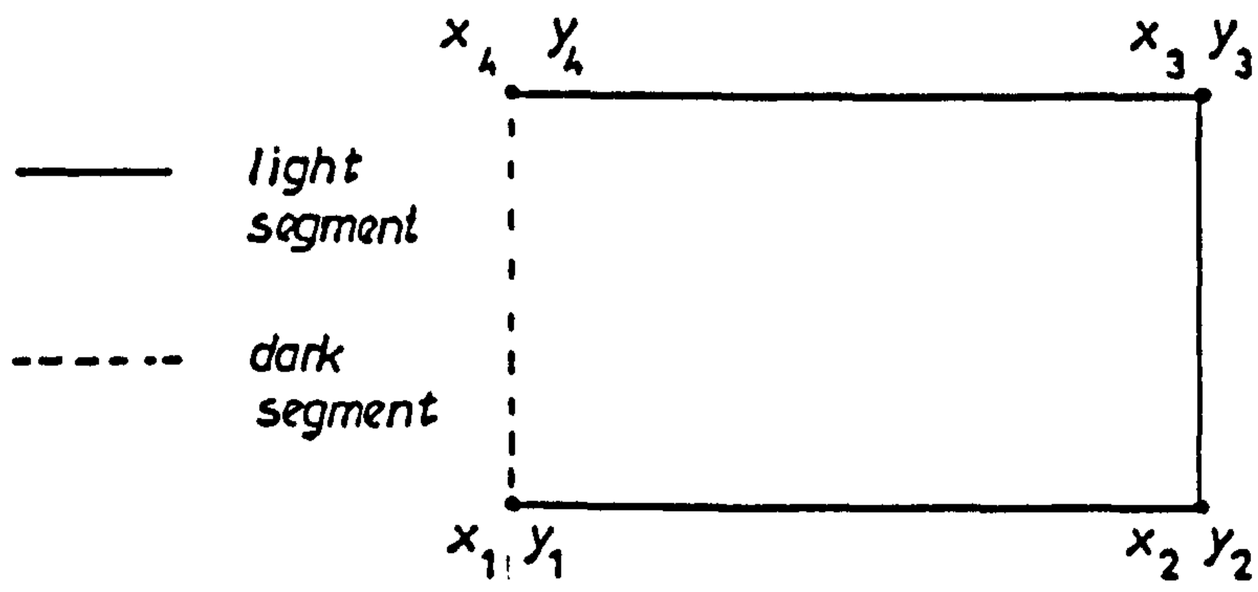
Bit	set
x1	Y
y1	
x2	
y2	
x3	Y
y3	
x4	
y4	
x5	
y5	
x6	
y6	
x7	
y7	

For long format polygons, the light/dark status of the segment is determined by the x-coordinate of the point that the segment goes to.

When the coordinate data is stored in a compact form, as in short format polygons and rectangles, the light/dark information must be similarly compact. Consider a section of a short format polygon as shown below :-



Remember that the point (x2,y2) is not stored in the bead, therefore the light/dark information must be stored totally in the point coordinates (x3,y3). Coordinate x3 holds the status for the first segment, and y3 holds the status for the second segment. An example for a rectangle is shown below :-



Bit set

x1
y1 Y
x3
y3

When a polygon is split up into several sub-polygons, the individual sub-polygons will be stored under different areas in the data structure. In commands such as deleting a polygon, all the sub-polygons which go to form the polygon must be found quickly. CADIC1 does this by connecting all sub-polygons (which represent the original polygon) together on a connectivity ring. This extra pointer requires a unique bead for the three types of open shape possible.

In, for example, plotting algorithms, CADIC1 processes the data structure in a top-down nature, so the area under concern is known before the shapes are processed. Remember that the shape coordinates are offset from the area origin, so the absolute coordinates of the shape are easily calculated. When deleting a polygon described by several sub-polygons, the program must chase round the connectivity ring to find all the coordinates, therefore the situation arises in which the

offset shape coordinates can be found, but the area origin is now no longer known. To obtain the area origin information quickly, a direct pointer to the area bead is also included in the open shape beads. The three types of open shape beads are shown below :-

4	3	4
Shape pointer		
Conn. pointer		
Direct pointer		
x1		
y1		
x3		
y3		

6	3	2n
Shape pointer		
Conn. pointer		
Direct pointer		
B. Rectangle X min		
B. Rectangle Y min		
B. Rectangle X dim		
B. Rectangle Y dim		
x1		
y1		
"		
xn		
yn		

8	3	2n
Shape pointer		
Conn. pointer		
Direct pointer		
B. Rectangle X min		
B. Rectangle Y min		
B. Rectangle X dim		
B. Rectangle Y dim		
x1		
y1		
"		
xn		
yn		

Group Definition Headbead : A group definition is built up in exactly the same way as the main layout. The only difference in terms of data structure is the form of headbeads used. In group definition headbeads, only three bytes are used to store the groupname, as opposed to fiveteen in the layout headbead. The form is as below :-

20	5	8
Forward group pointer		
Reverse group pointer		
Area pointer.		
Group Call pointer		
Garbage pointer		
Title (1)		
Title (2)		
Title (3)		
Group X offset		
Group Y offset		
Group X dimension		
Group Y dimension		
Mask word		

Note that all the group definitions are linked by a double ring, as opposed to the single ring used elsewhere in the data structure. If the group definitions can be arranged such that the newest group is first on the forward ring, one may think that the reverse ring is redundant, since in plotting out a layout, a top-down approach is required. This is correct, and in fact, the reverse group ring is seldom used by CADIC1. The moment when it is used is after the designer has added more shapes to a previously formed group definition, which is called by other group definitions higher up in the group hierarchy. Should the edited group definition now be larger, it may affect the size of the groups which call it. Increasing the size of the latter group definitions may, in turn, affect other group definitions, and so on.

The forward group ring is pointing in the wrong direction to process the groups in such a bottom-up manner, therefore the reverse group ring was included to serve this purpose.

Group Call beads : Two types of group calls can be achieved. The first type is a single group instance, and the second type is an array instance, containing many group instances.

The group instance bead is as below, and is placed on the group call ring of the relevant layout or group definition headbead.

15	2	8
Group Call pointer		
Direct pointer		
Orientation		
X offset		
Y offset		
B. Rectangle X min		
B. Rectangle Y min		
B. Rectangle X max		
B. Rectangle Y max		
Mask word		

To save on memory space and search time, the address of the group definition headbead is stored in the group instance bead rather than the group-name. Next in the bead comes the orientation of the group instance, relative to the layout/group definition, plus the group instance position relative to the origin of the layout/group definition. To help cut down the processing required by CADIC1 to handle all the group instances, the bounding rectangle plus the masks used in the instance are stored in the group call bead. Therefore, a group instance is only considered if it is inside the window, and contains shapes on the mask(s) required.

The array instance saves space and time by being defined as an array of group instances. The array bead looks like :-

16	2	11
Group Call pointer		
Direct pointer		
Orientation		
X offset		
Y offset		
X number		
X spacing		
Y number		
Y spacing		
B. Rectangle X min		
B. Rectangle Y min		
B. Rectangle X max		
B. Rectangle Y max		

As can be seen, the array bead is identical to the group instance bead except for four bytes. These bytes simply store the number of instances required in the X and Y directions, and the spacing between each instance in the respective directions.

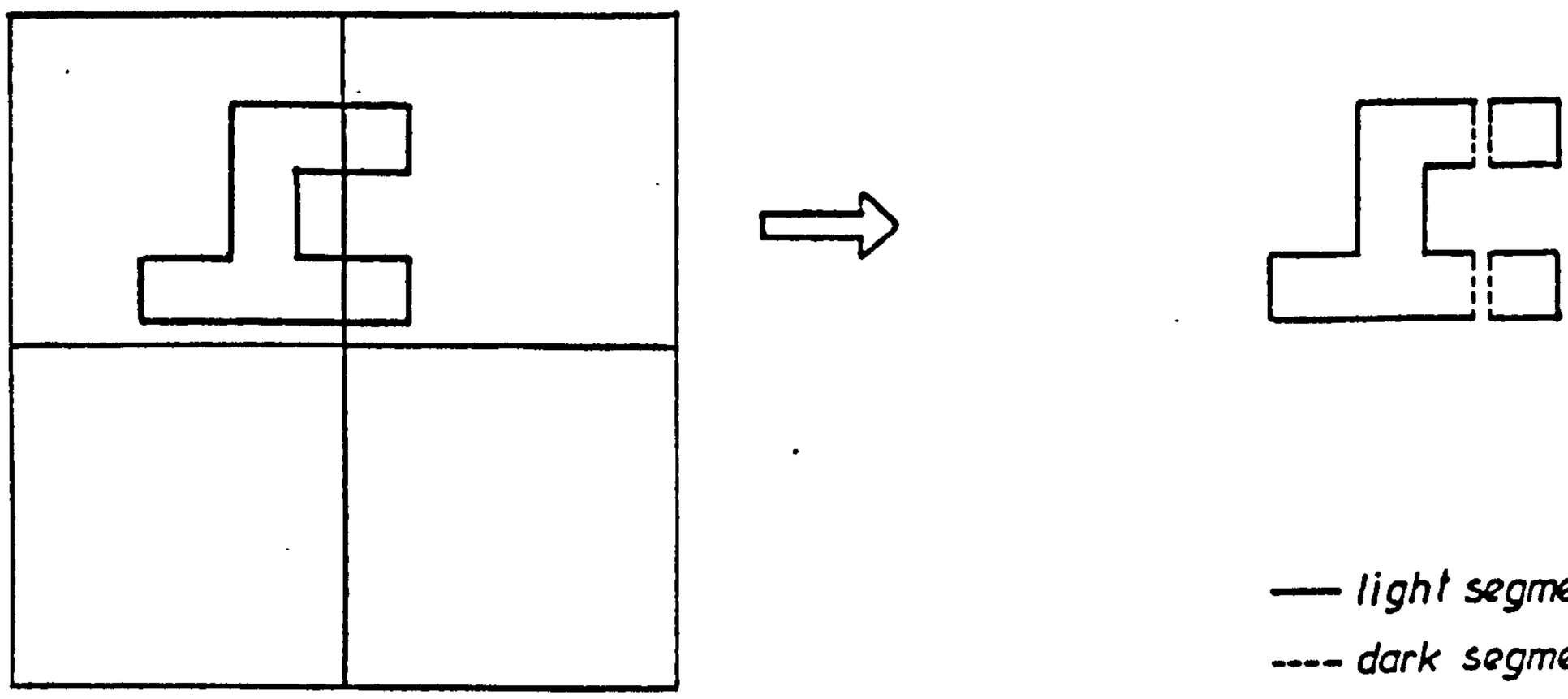


Figure 5.1 Segmentation of shapes which cross area boundaries

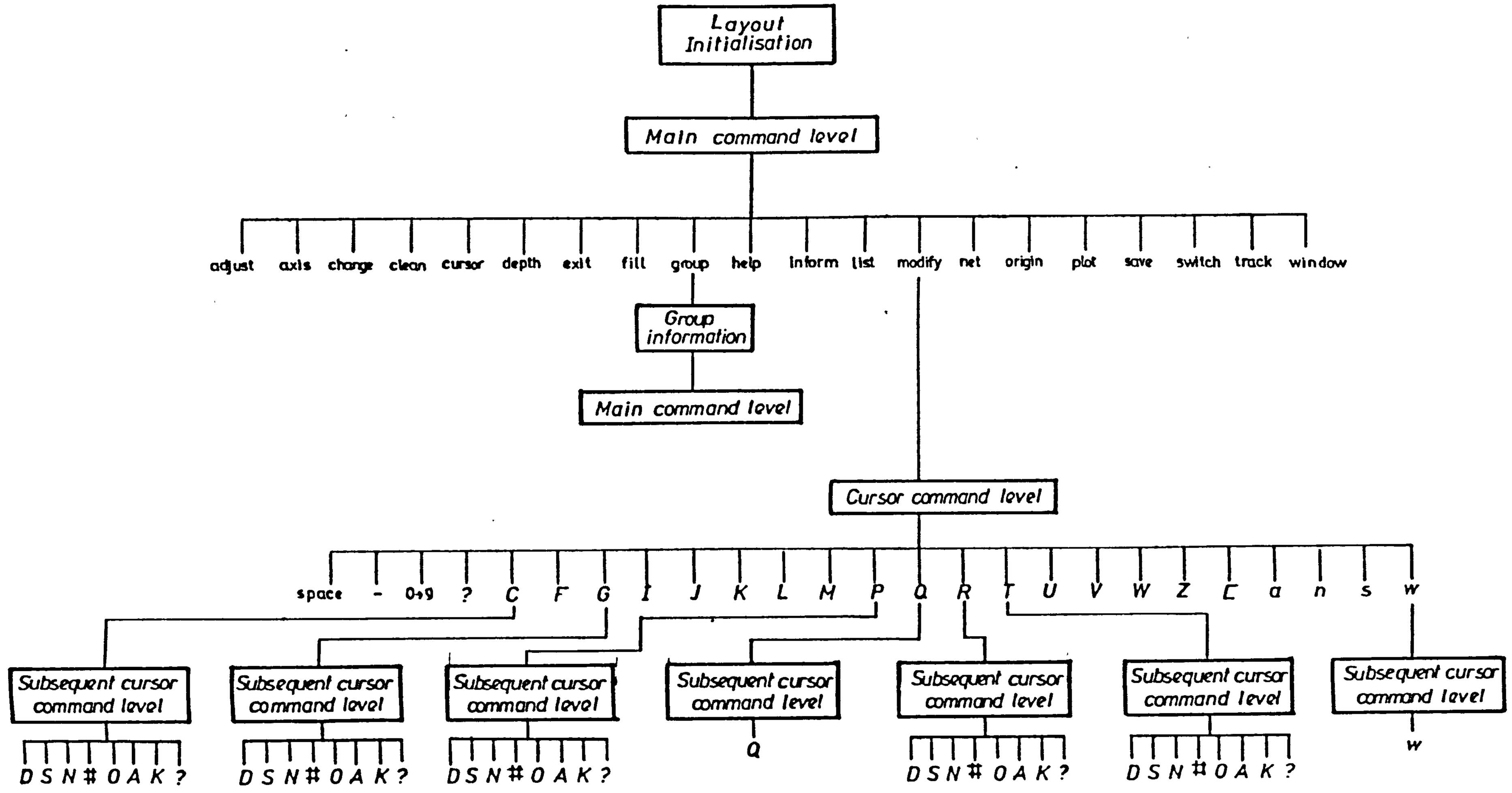
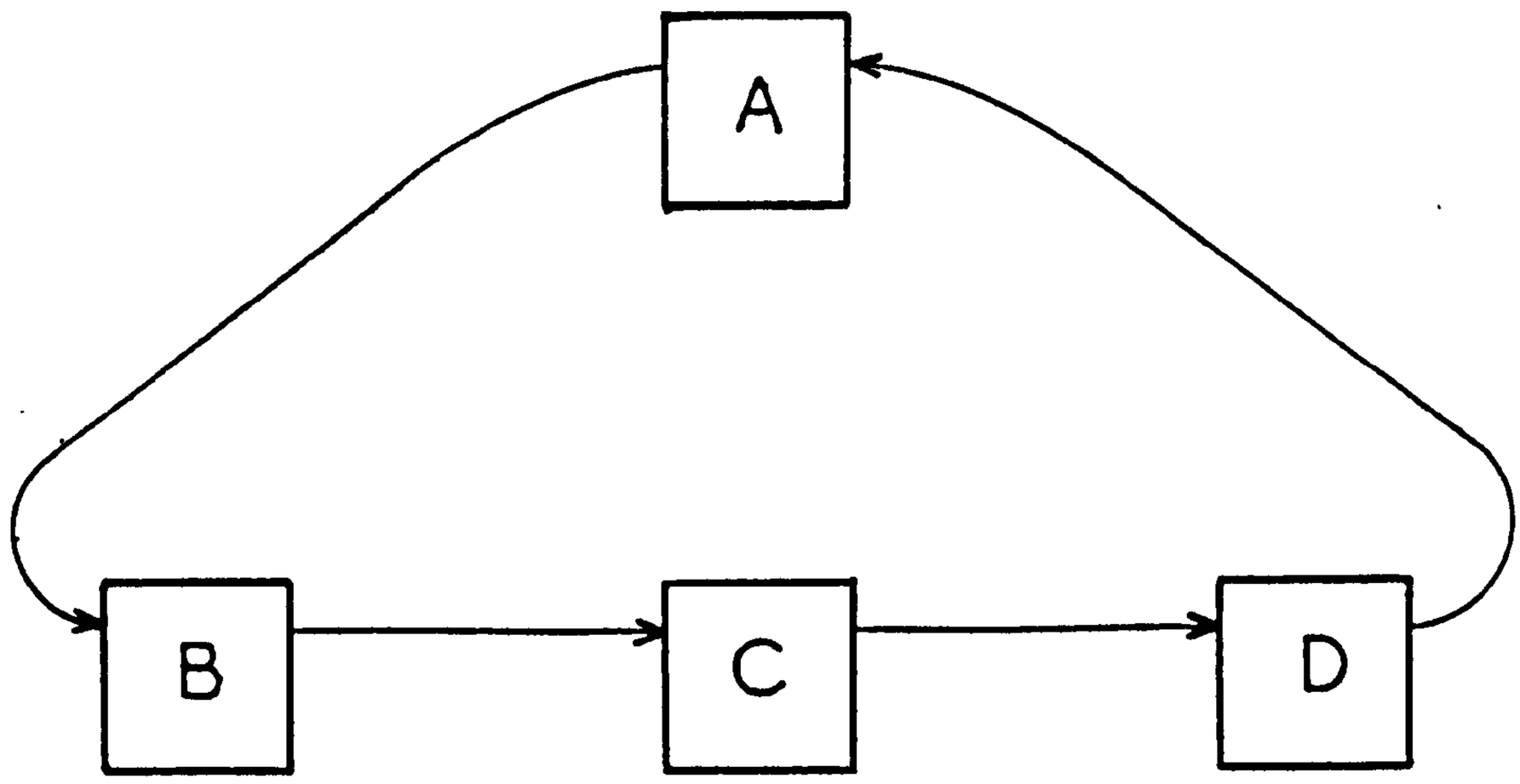
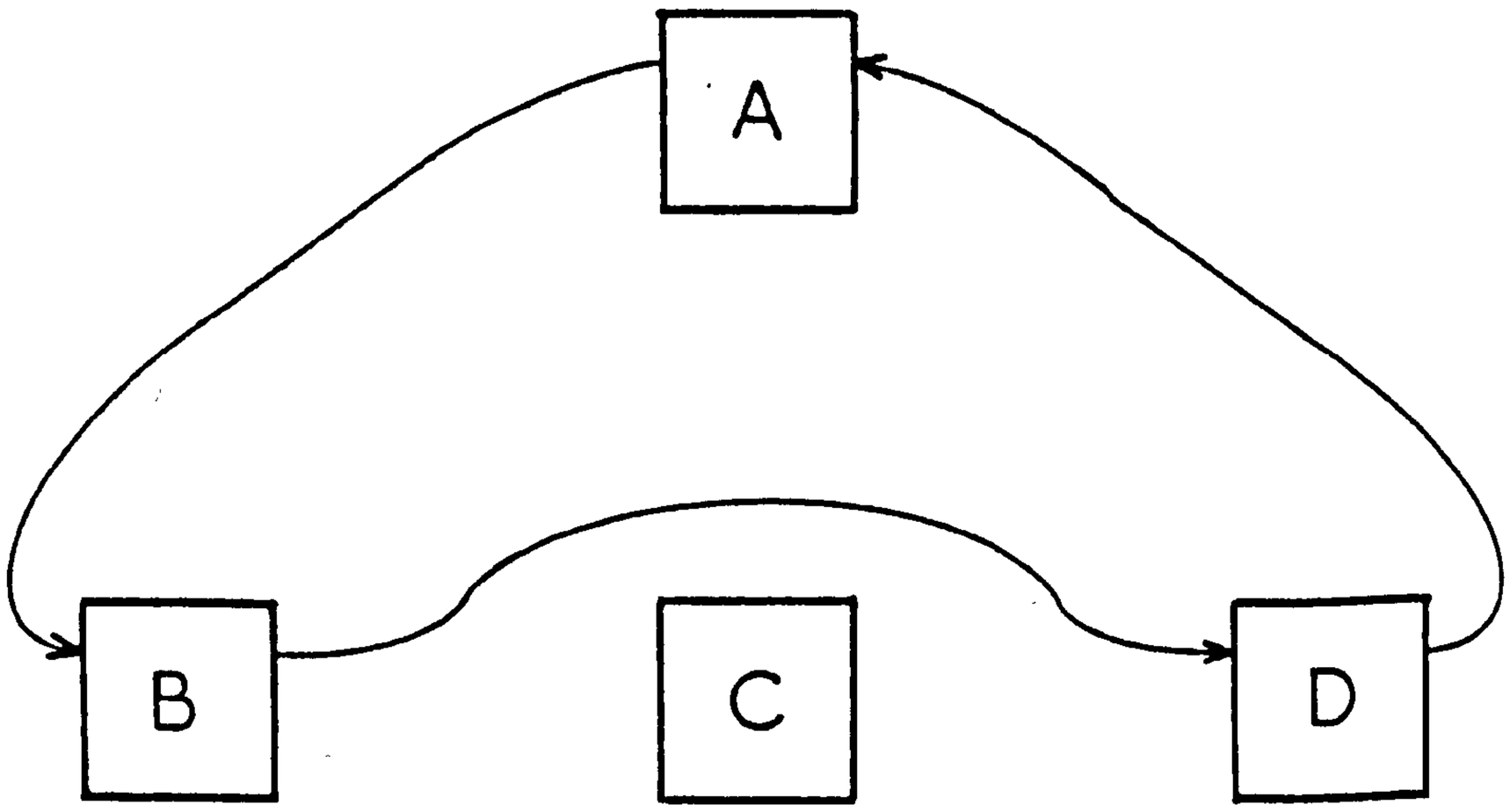


Figure 5.2 Hierarchical structure of CADIC commands



(a) Ring before removing bead C



(b) Ring after removing bead C

Figure 5.3 Removing a bead from ring

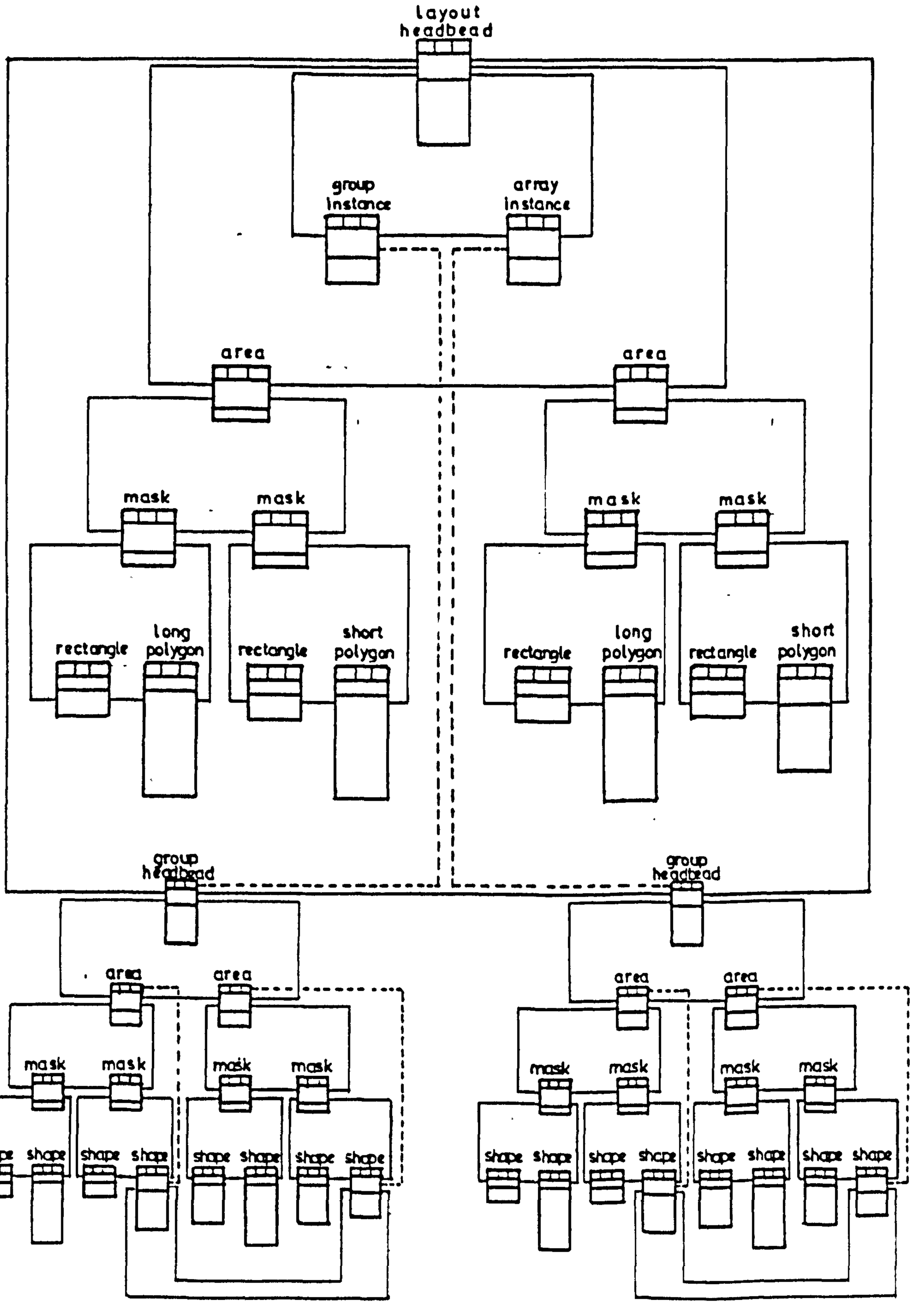


Figure 5.4 Layout ring data structure

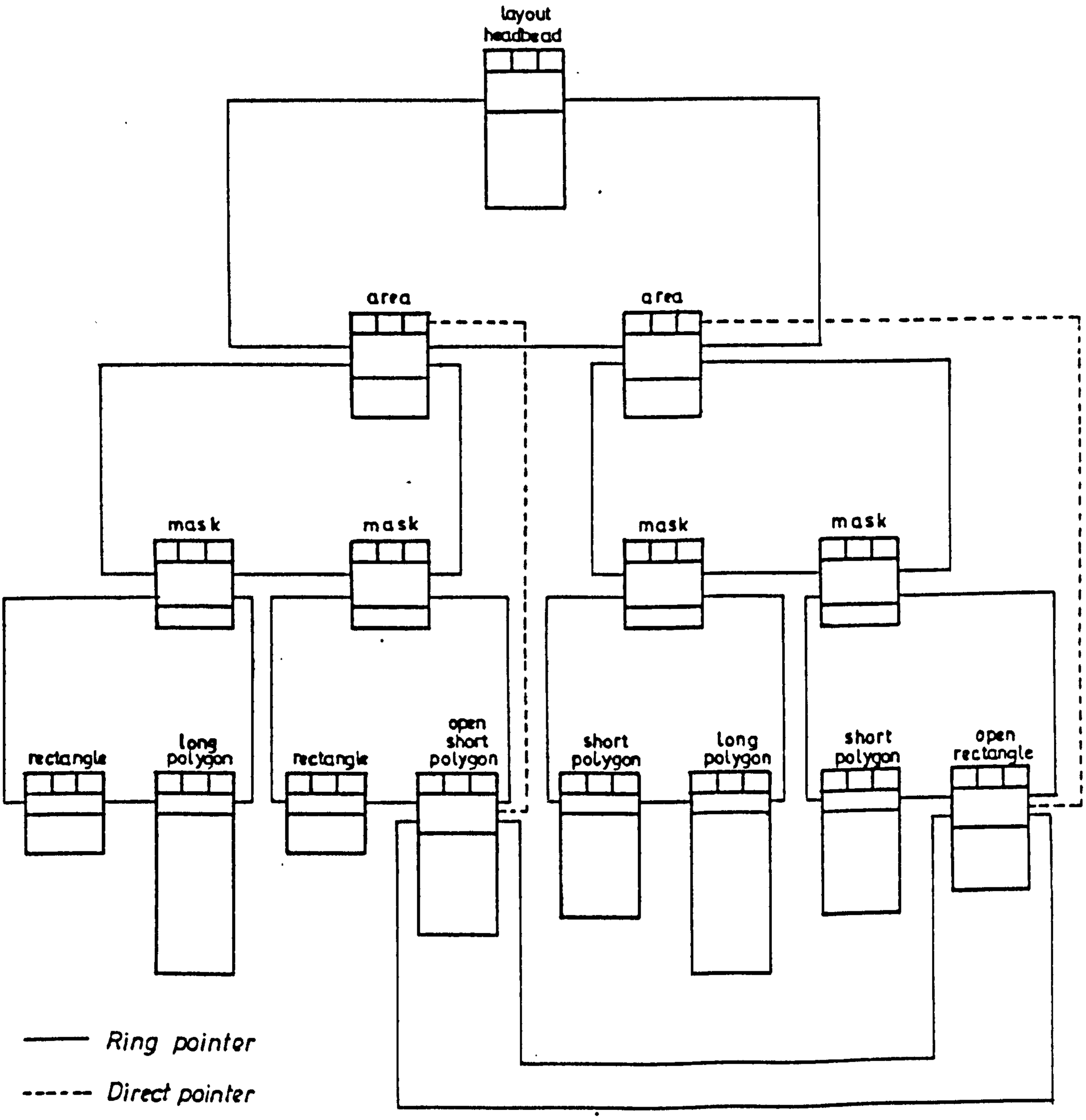


Figure 5.5 Layout ring data structure without group definitions

CHAPTER 6

DRCCAD

6.1 Introduction

DRCCAD (Design Rule Compiler for Computer Aided Design) accepts a 'user readable' description of the design rules required during layout design, and converts this description into a ring data structure readable by CADIC2. Existing design aids generally do not require this type of pre-processor, therefore it is important to justify the use of DRCCAD in the CADIC suite of programs.

The need for DRCCAD stems from the fact that CADIC implements on-line design rule checking. To show this, consider the example of performing a spacing check to ensure that the distance between shapes on mask (1) and the shapes on mask (2) is greater than some specified minimum value.

A typical off-line design rule checker would start by writing all the information about the shapes on mask (1) into a file. Secondly, all the shape information from mask (2) would be written to another file. Note that the information within these files is often arranged into sophisticated data structures so as to allow quicker access to the information.

The checker proceeds by analysing each shape combination to see if any two shapes violate the spacing rule. If yes, then an error message

is printed out to the user, along with details about the violating shape(s). On completion of the check, one or both of the files can be cleared, ready for the next check to be performed.

Identifying each shape combination in this way means that (in theory) every shape in file (1) must be compared against every shape in file (2). The combinatorial explosion thus produced forces sometimes very heavy consumption of CPU time (typically ranging from a few seconds to several minutes).

Performing the same spacing check in an on-line fashion requires a completely different approach. Once a shape is added to, for example mask (1), information about the shape is added to a file. Note that the checker now only considers particular shapes, rather than entire masks.

Next, mask (2) must be searched to identify a shape that is liable to cause a spacing violation with the newly added shape on mask (1). If found, then the mask (2) shape is read into the file. The design rule checker then performs a spacing check between the two identified shapes. If a violation occurs, then the relevant error message is printed out. Once finished with the mask (2) shape, the checker continues to search through mask (2) to find any other shapes that may cause a violation. If a shape is found, then the new shape overwrites the old mask (2) shape, and the above process is repeated. Once mask (2) has been completely searched, the newly added shape on mask (1) can be accepted or rejected, depending on how many violations were identified.

In on-line design rule checking, only <u>one</u> shape is checked against every shape on mask (2) therefore the CPU time required is drastically reduced (typically fractions of a second).

Now consider an off-line design rule checker once again. Usually the description of the design rules is stored in a text file. These files look similar to computer programs in that each line of code defines a (mask) operation for the checker to perform. The checker must therefore read out each line of code, then decode it into a set of instructions before proceeding with the relevant operation.

Using a text file to store the design rules means that access to the information is indirect, and therefore relatively slow. However a faster method of accessing the design rules is not required by the off-line design rule checker. As can be appreciated from the example shown above, the time taken to read and decode from the text file is negligible, compared to the time required to perform each operation.

On the other hand, an on-line design rule checker requires less than one second to perform the checks. Therefore a text file description of the design rules will not be suitable for an on-line design rule checker. There are three main reasons for this :-

1. The time taken to read from, then decode an instruction from a text file could well exceed the time taken to actually perform the operation.

2. The design rule file must be searched each time a shape is added to the layout. If a text file was used, then the time required to rewind then sequentially read the entire file would be too slow for interactive use.

3. Different rules relate to different masks. Therefore when a shape is added to a particular mask, only a small subset of the total number of rules need to be implemented. All other rules can be

ignored. Once again, sequentially reading a text file to find the relevant rule is highly inefficient.

An on-line design rule checker therefore requires a pre-decoded description of the rules to allow quick and easy access to the relevant information. The rules should also be organised into groups, depending on which masks they relate to, so that the time spent reading the design rules is minimised.

Compiling the rules as an independent stage in the design of integrated circuit layouts has two main advantages :-

1. The same set of design rules may be used for several months, on a variety of layouts. It therefore makes sense to compile the rules once, then utilise the compiled version of the rules as required.

2. Compiling the rules in this way allows the designer to identify and correct mistakes (i.e. syntax errors) in the 'user readable' description of the design rules, before actual layout design commences.

6.2 Choice of design rule input language

The ideology behind DRCCAD is that a set of design rules to perform any type of dimension check required can be quickly encoded by the designer. The rules in the manual input language must therefore be easy to build up, free in format, and the commands easy to remember.

Once the full set of design rules has been built up for a specific technology, the same set of rules can be applied to all future designs using this technology. However, a time will come when the designer will

have to alter the set of design rules. This may happen months or years after the rules were originally formed. If the set of design rules consists of a series of numbers representing the commands, mask numbers, and dimensions, it is going to be very difficult to understand. A high degree of 'readability' is therefore required in the language.

Lastly, as described above, the design rule file used by off-line checkers tends to be similar in format to a computer program. The sequence of commands must be generated by the designer, therefore the efficiency of the design rule checker is heavily dependant on the designers implementation of the language used to specify the rules. The onus is also on the designer to create the correct sequence of commands to perform the required set of checks.

Since CADIC2 requires DRCCAD to break down the design rules into a more accessible form, restrictions on the type of design rule language to be used can be lifted. Existing design rule checkers tended to use languages that were a compromise between 'user readable' and 'computer readable'. However, CADIC can now use a language which best suits the user. Decisions on which operations to implement, plus the order in which the operations should be performed can be left to DRCCAD. In this way, the user does not need an in-depth knowledge of how CADIC2 works, plus DRCCAD can re-organise the sequence of operations to obtain maximum efficiency during on-line design rule checking.

Although GAELIC [1] implements off-line design rule checking, it does provide a 'user ergonomic' language to describe the design rules. For example, each rule consists of basically one statement which defines the condition to be checked for. Note that this condition is technology independent, plus is written in almost an identical fashion to the way

it would be described verbally.

The GAELIC language is therefore directly suited to the needs of CADIC. For this reason, it was decided to make the DRCCAD language compatible with the GAELIC language.

Within a design rule file, each rule has the following standard construction :-

```
RULE <rulename>
!Comment
<Var> IS <shape type> MASK <mask number>
FAIL <Error message> IF <failure condition>
END
```

The rulename can be any unique name up to six characters in length (only five characters are significant). Secondly, each rule may contain lines of comment to increase readability. Lastly, the failure condition is the set of commands which define the design condition to be checked for. If the failure condition is satisfied (i.e. a design rule violation has been identified) then the relevant error message can be printed out to the user, so that the nature of the violation can be determined.

Alphanumeric variables may be used in the failure condition to represent particular types of shapes that exist on the desired masks. All such variables must be pre-defined using the 'IS' command, for example :-

```
PD IS RECT,POLY MASK 1
METAL IS RECT,MASK 4
```

PD is defined to be all the shapes on mask one, and METAL is defined to be only the rectangles on mask four.

A list showing the available failure condition commands is shown below :-

OVERLAP ..... Find shapes which overlap

ENCLOSED ..... Find shapes, one enclosed by the other

SEPARATE ..... Find shapes which are separate

ABUTS ..... Find shapes which touch

DISTINCT ..... Find shapes which are distinct

PARTED ..... Find shape, one cut in two by the other

WIDTH ..... Specify minimum width of shape

LENGTH ..... Specify minimum length of shape

INTERLIMB ..... Specify minimum spacing between limbs of shape

XDIM ..... Specify minimum X dimension of shape

YDIM ..... Specify minimum Y dimension of shape

AREA ..... Specify minimum area of shape

BRAREA ..... Specify min. area of shape's bounding rectangle

SPACING ..... Specify minimum spacing between shapes

CLEARANCE ..... Specify minimum clearance between shapes

HORIZONTAL ..... Specify shape to lie in horizontal direction

VERTICAL ..... Specify shape to lie in vertical direction

AND ..... Connecting command

OR ..... Connecting command

NOT ..... Inverting command

UNION (+) ..... Perform logical OR function on shapes

INTERSECTION (*) . Perform logical AND function on shapes

DIFFERENCE (-) ... Perform logical NAND function on shapes

EXCLUSIVE (/) .... Perform XOR function on shapes

INFLATE/DEFLATE .. Inflate/deflate shape

ENDOFFILE ..... End processing

An example of a set of design rules are :-

```
PD IS RECT,POLY MASK 1
PS IS RECT,POLY MASK 2
CW IS RECT,POLY MASK 3
METAL IS RECT,POLY MASK 4
CHAN IS RECT,POLY MASK 5
RULE XMPLA
!Example A
     FAIL 'Minimum width of contact' IF WIDTH (CW) < 6
END
RULE XMPLB
!Example B
     FAIL 'Metal separation' IF SEPARATE (METAL,METAL) &
        AND SPACING (METAL,METAL) < 10
END
RULE XMPLC
!Example C
     FAIL 'Separation of polysilicon outside p-diff to poly' IF &
         OVERLAP (PS,PD) AND INTERLIMB (PS+PD) < 10
END
RULE XMPLD
!Example D
     FAIL 'Minimum spacing contact to poly' IF ENCLOSED (CW,PD) &
         AND OVERLAP (PL,PS) AND SPACING (CW,PS) < 5
END
ENDOFFILE
```

Diagrams showing the checks described by each rule are shown in Figure 6.1.

6.3 Program Operation

After initialisation, DRCCAD asks for the name of the file containing the design rules, then asks for the name of the ring data structure which will store the compiled information. Note that the ring data structure may already exist, in which case the design rules in the input file will simply be compiled, then appended to the existing data structure.

DRCCAD proceeds by reading out each rule from the input file, then performing syntax checking on it. DRCCAD has no way of knowing if the failure condition correctly represents the check required, but can catch

a variety of errors. For example :-

1. Static errors - Misspelt commands, syntax errors, and so on
2. Dynamic errors - Undefined shape variables, illegal rule structures, and so on

Once the manual file has been completely processed, DRCCAD provides the user with three options :-

1. Close the files, and return to monitor level
2. Fetch another input file, which will be added to the existing ring data structure. In this way, library files can be loaded as required.
3. Enter data on-line, through the keyboard. Therefore if only a few rules were rejected by DRCCAD, they can be re-submitted correctly. This saves having to edit the relevant manual file, and start the possibly lengthy and involved 'compilation' from the beginning.

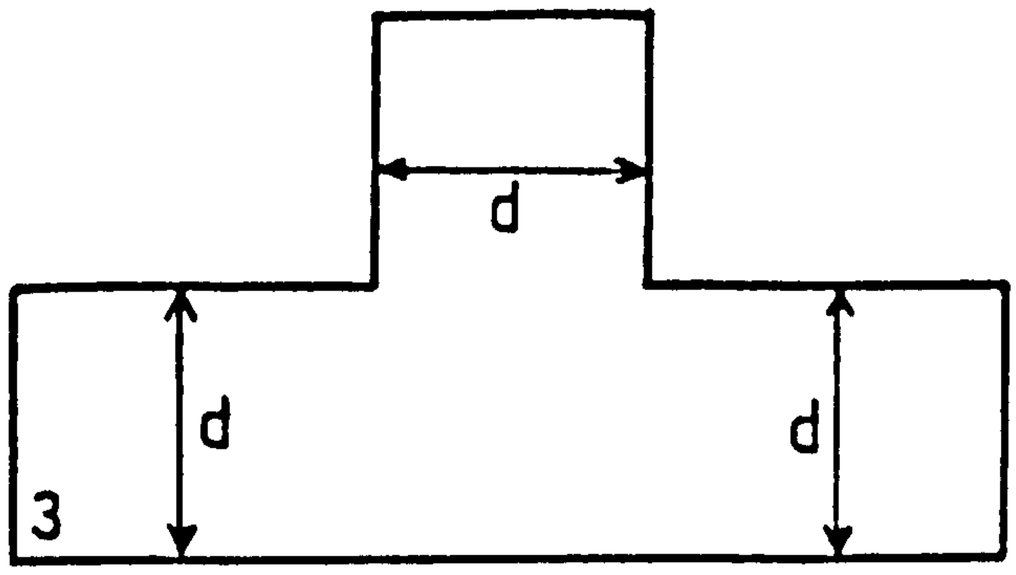
An example of running the manual file (shown in the previous section) through DRCCAD is given below. Note that the manual file deliberately contains an error in one of the shape variables, so as to show how DRCCAD handles a typical error.

```
                    - DRCCAD -

Program to convert DRC input language into a ring data structure

Enter name of DRC input file, or return to finish :- DRC

Does the manual file contain line numbers ? NO

Enter name of existing ring data structure, or return :-

Enter name of the new ring data structure, or return to finish :- DUMP


                    RULE XMPLD
Undefined shapename in OVERLAP
Rule is ignored by program


Enter name of next manual file, or TTY for
keyboard input, or press return to finish :-  TTY

Enter data - without line numbers
RULE XMPLD
     FAIL 'Minimum spacing contact to poly' IF ENCLOSED (CW,PD) &
          AND OVERLAP (PD,PS) AND SPACING (CW,PS) < 5
END .
ENDOFFILE


Enter name of next manual file, or TTY for
keyboard input, or press return to finish :-

END OF EXECUTION
EXIT
```

6.4 Design rule data structure

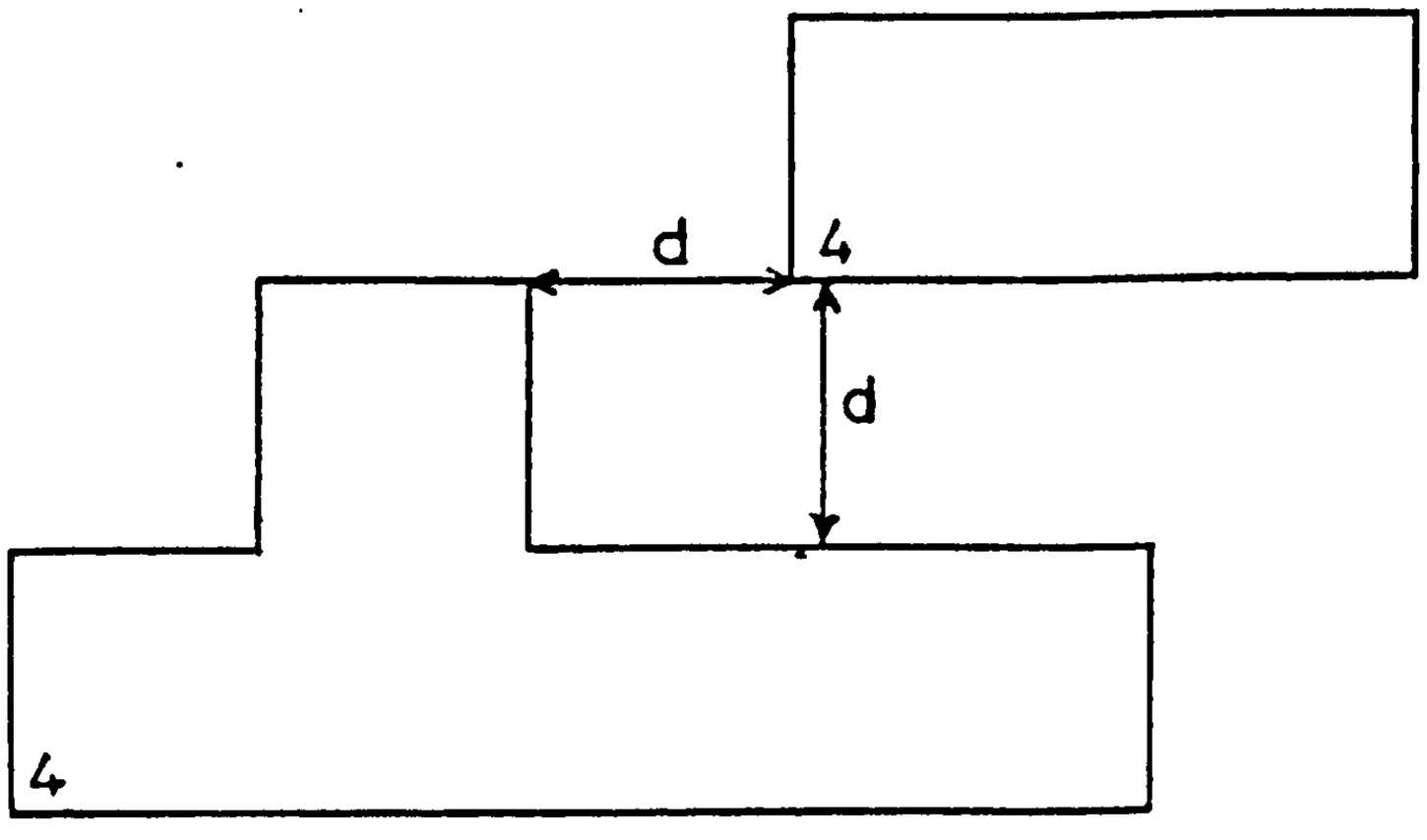
The design rule data structure acts as a control file, which CADIC2 uses to ensure that it performs the minimum number of calculations during design rule checking. This is a different concept to the layout

data structure, which acts in a data storage capacity. More details on the format of the design rule data structure will be given in Chapter seven, once on-line design rule checking and its requirements have been introduced. Only after this can the final format of the data structure be decided upon.



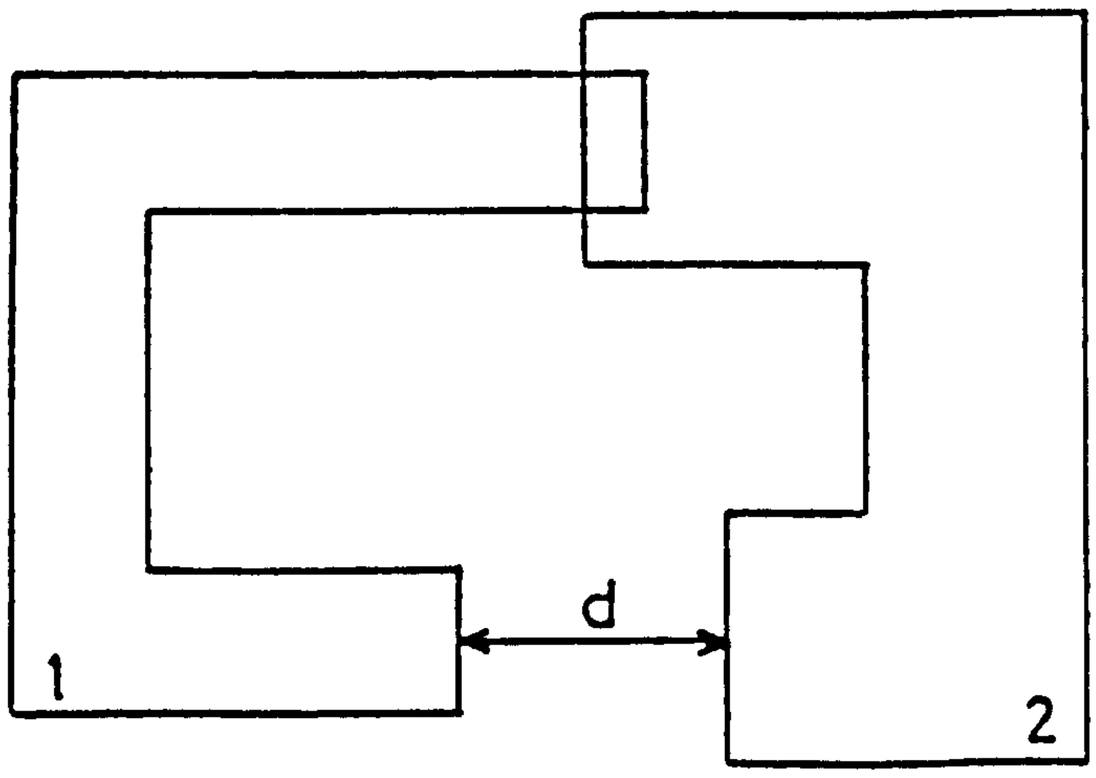
d ≥ 6

Rule XMPLA



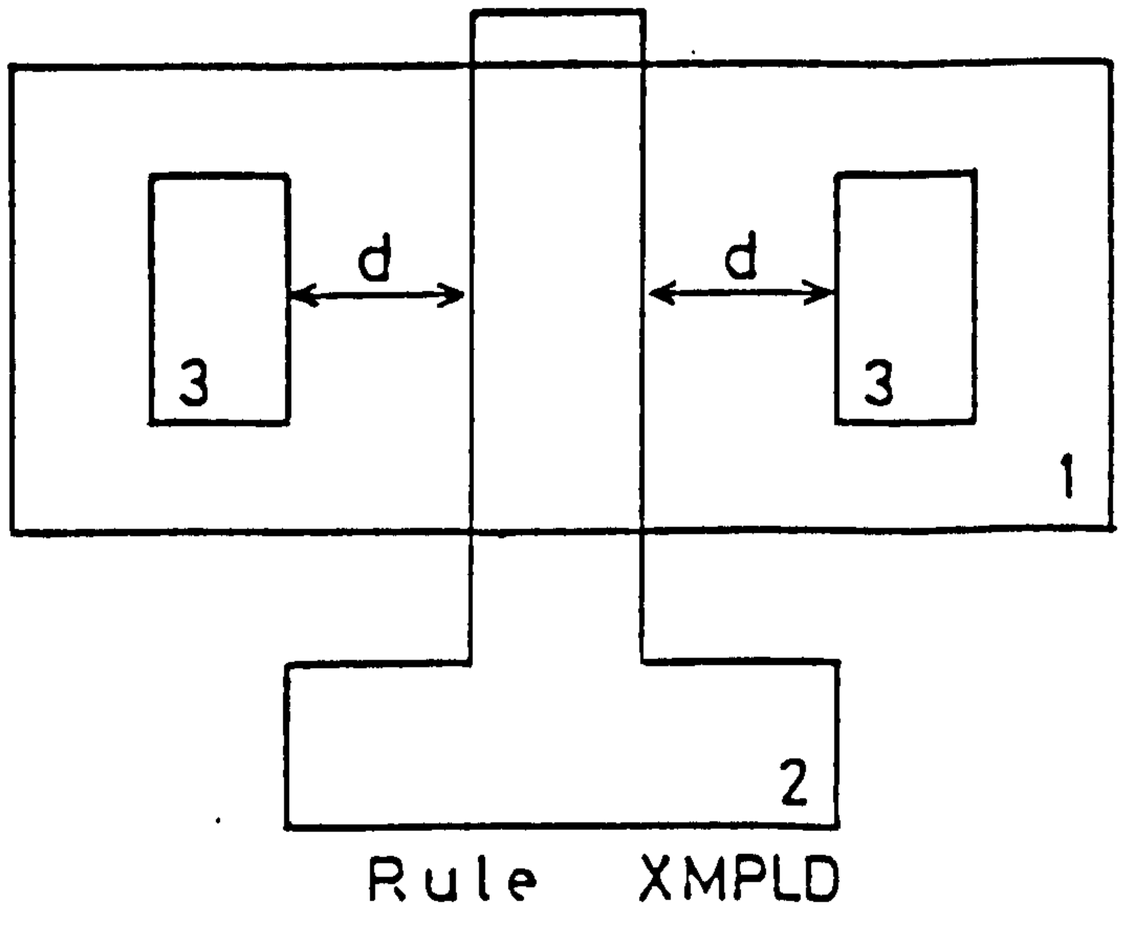
d ≥ 10

Rule XMPLB



d ≥ 10

Rule XMPLC



d ≥ 5

Rule XMPLD

Figure 6.1 Design rule checks

CHAPTER 7

CADIC2 : The on-line design rule checker

7.1 Introduction

CADIC2 is the second phase of CADIC, which only comes into play if on-line design rule checking is required. If a shape or group instance is added to the layout, it is CADIC2's job to check the shape(s) against the existing layout, using the set of design rules specified when the ONLINE option was selected. Because of the importance of on-line design rule checking, the whole of this chapter describes the approach taken by CADIC2.

7.2 Requirements

In order to be regarded as an on-line design rule checker, CADIC2 must be able to do the following :-

1. Check the newly added shape or group instance against the existing layout, using a set of pre-defined design rules.

2. Perform the design rules within the time it takes the user to think of his next move.

The first requirement may seem too obvious to be included as a requirement for on-line design rule checking, but has been included simply to highlight the fact that the checks can be carried out in a variety of ways.

The simplest approach is to build the rules into CADIC2. Using this technique, the designer need not worry about setting up the rules, and without having to reference a file on disc, CADIC2 would run faster, therefore making the second requirement easier to acheive.

Some design rule checking programs do use this technique [69], and work by relating all the rules to the resolution of the fabrication process (λ). Performance is very good, but the programs are limited to only one technology. Changing technology may require a major re-write of the program. Another disadvantage is that the rule dimensions do not scale down linearly with λ, therefore changing λ may again involve editing the program.

With today's fast moving technology, it was decided that CADIC2 should store a description of the design rules in a separate file. The rules can then be built up by the designer to suit any technology, and can be modified as required. Note that the modified design rules only have to be 're-compiled' by DRCCAD, which does not affect the source code for CADIC2.

The second requirement is the 'make-or-break' requirement of any on-line design rule checking program. During a design, the designer will work at a speed which allows his ideas to flow along, and allow a 'design inertia' to be built up. The human brain works most efficiently at this steady pace. If the introduction of on-line design rule checking means that the designer must wait for each shape to be checked, then the design will never 'flow', and may be subsequently impaired as a result. The designer will soon become bored, and may reject the on-line design rule checker in favour of a conventional off-line checker.

Therefore after a shape is added to the layout, the on-line design rule checks must be completed before the designer makes his next move. To satisfy this requirement, the few design aids that incorporate on-line design rule checking [3] limit the checks to the simpler rules. All other rules are checked off-line. The function of the on-line checker is therefore to identify only the obvious errors. This is a fairly sensible approach, as the layout will not have to be edited just to correct the elementary errors, plus the off-line checker will run faster if it can disregard the simpler rules.

CADIC hopes to improve on this by carrying out all the design rule checks once a shape is added to the layout. In this situation, the layout will be correct at all times, thus completely doing away with the need for an off-line checker. The main advantage of this is that the time comsuming two stage design-check cycles will no longer exist, allowing circuit design turnaround time to be greatly reduced.

7.3 Logistics

In the previous Chapter, the design rule input language was introduced, and DRCCAD described. Note that no details were given about the format of the design rule data structure created by DRCCAD. The reason for this is that the format is dependant on the on-line design rule checking requirements and logistics discussed in this Chapter, rather than any requirements of DRCCAD.

So how should the design rules be implemented such that the processing is kept to a minimum ? Consider the following set of rules :-

```
PD IS RECT,POLY MASK 1
PS IS RECT,POLY MASK 2
CW IS RECT,POLY MASK 3
RULE ONE
      FAIL 'Test one' IF WIDTH (CW) < 10
END
RULE TWO
      FAIL 'Test two' IF SEPARATE (CW,CW) AND SPACING (CW,CW) < 6
END
RULE THREE
      FAIL 'Test three' IF OVERLAP (PD,PS) AND AREA (PS) < 25
END
RULE FOUR
      FAIL 'Test four' IF OVERLAP (PD,PS) AND WIDTH (PD*PS) < 4
END
ENDOFFILE
```

The first point to note is that different rules relate to different masks. For example, if a shape is added to mask 'CW', then only rules ONE and TWO need be processed, whereas if a shape is added to masks 'PD' or 'PS', rules THREE and FOUR need to be processed. The first important timesaver therefore is to group rules related to the same mask together in the data structure, so that on adding a shape to the layout, all the relevant rules can be found quickly and easily.

The second timesaving factor relies on the fact that within any one group of rules, two types of rule exists :-

1. Self-rules
2. General rules

Self-rules apply only to the newly added shape, and involve no other shapes. Rule ONE is an example of a self-rule. This independence means that the check can be performed while the shape is being built up. General rules involve other shapes, and can only be processed once the

shape is complete. Rules TWO to FOUR are examples of general rules.

By processing the self-rules while the shape is being built up, CADIC2 will have less checks to perform once the shape is complete, and so CADIC1 will be returned to the cursor command level (see Chapter five) sooner. General rules can become rather complex, so how should they be processed in order to minimise CPU time ? Two main approaches exist.

The first approach is to consider each rule individually, and apply the rule to the whole layout. The data structure would simply contain a series of blocks of information, one block per rule. Each block would basically contain the commands, the masks used, and the error message, making the design rules easy to implement.

Unfortunately, as can be seen in rules THREE and FOUR, a fair degree of redundancy exists in the design rules. To process the rules, CADIC2 would have to perform the same OVERLAP operation twice. The higher the redundancy, the more CPU time is wasted.

The second approach reduces this redundancy to zero by considering the general rules in a more global fashion. The data structure still contains a series of blocks, but now each block defines a single operation, rather than a whole rule. A pointer system is now required to link the operation blocks up in the correct order.

To build up the data structure, the failure condition in each rule is considered, then the necessary operation blocks are added to the data structure in the correct order. Note that existing blocks are used whenever possible. For example, to build up rule THREE, three blocks

would be required ; an OVERLAP block, followed by an AREA block, followed by an ERROR MESSAGE block. To build up rule FOUR, only three blocks are required, instead of four, since the OVERLAP block already exists. The blocks are added into the data structure after the OVERLAP block, and are ; an INTERSECTION block, followed by a WIDTH block, followed by an ERROR MESSAGE block.

Therefore if a shape was added to mask 'PD', CADIC2 would firstly have to find a shape on mask 'PS' that overlapped the newly added shape. If found, the shape on mask 'PS' would be given an AREA check. It would then be used to form a new shape which is the INTERSECTION of the two input shapes, and finally the new shape would be given a WIDTH check. Should either check fail, then the corresponding error message is printed out.

In order to obtain very high efficiency, CADIC2 uses a data structure based on the second approach to implement the general design rule checking.

To sustain the high efficiency, CADIC2 must also minimise the amount of data to be processed during each operation. This is achieved by implementing two main concepts :-

1. Influence bumper
2. Segment type identification

Defining an influence bumper round a shape or segment is a new concept, and is based on the idea that CADIC2 does not have to consider all the shapes in the layout when performing the design rule checks. In fact, as is shown below, only the shapes in the immediate neighbourhood

need be considered.

Consider adding a shape to mask 'PD' and applying the set of design rules shown above. In rules THREE and FOUR, the OVERLAP selector is only concerned with shapes which enter the newly added shape's bounding rectangle. OVERLAP will therefore typically select at the most only the few neighbouring shapes out of the thousands of shapes possibly available on mask 'PS'. The OVERLAP routine therefore implicity defines its own influence bumper of zero increments, since possible shapes must enter the newly added shape's bounding rectangle before they can possibly cause an overlap condition.

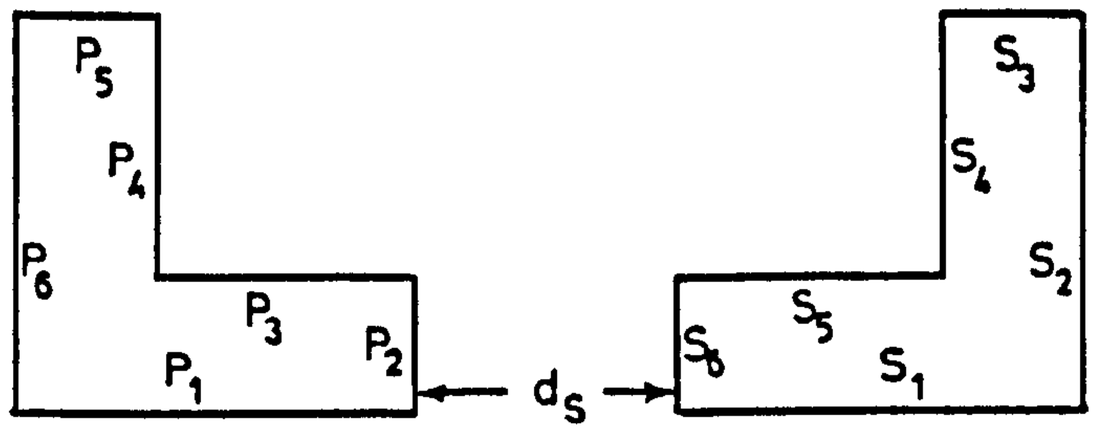
Now consider adding a shape to mask 'CW' and applying rule TWO. In this case, thousands of shapes will satisfy the SEPARATE condition, and so will have to be checked for a possible SPACING violation. On considering the rule in a more global fashion, it is obvious that if a shape's bounding rectangle is separate from the newly added shape's bounding rectangle and further away than six increments, the shape cannot possibly violate the rule.

To filter out these unwanted shapes, DRCCAD calculates an influence bumper for each mask, as it is compiling the rules into the design rule data structure. As described with the OVERLAP example, the bumper width is normally zero, but in rules using the SEPARATE and SPACING commands, the bumper width is set to the spacing dimension.

Whenever a shape is added to the layout, CADIC2 finds the width of the influence bumper to be associated with the shape, then surrounds the shape's bounding rectangle with this bumper. During the design rule checks, all shapes outside the bumper can therefore be ignored

immediately.

The concept of an influence bumper also helps minimise the time taken to perform dimensional checks such as WIDTH, SPACING and so on. Consider the following shapes, ready to be checked for a SPACING violation :-



key

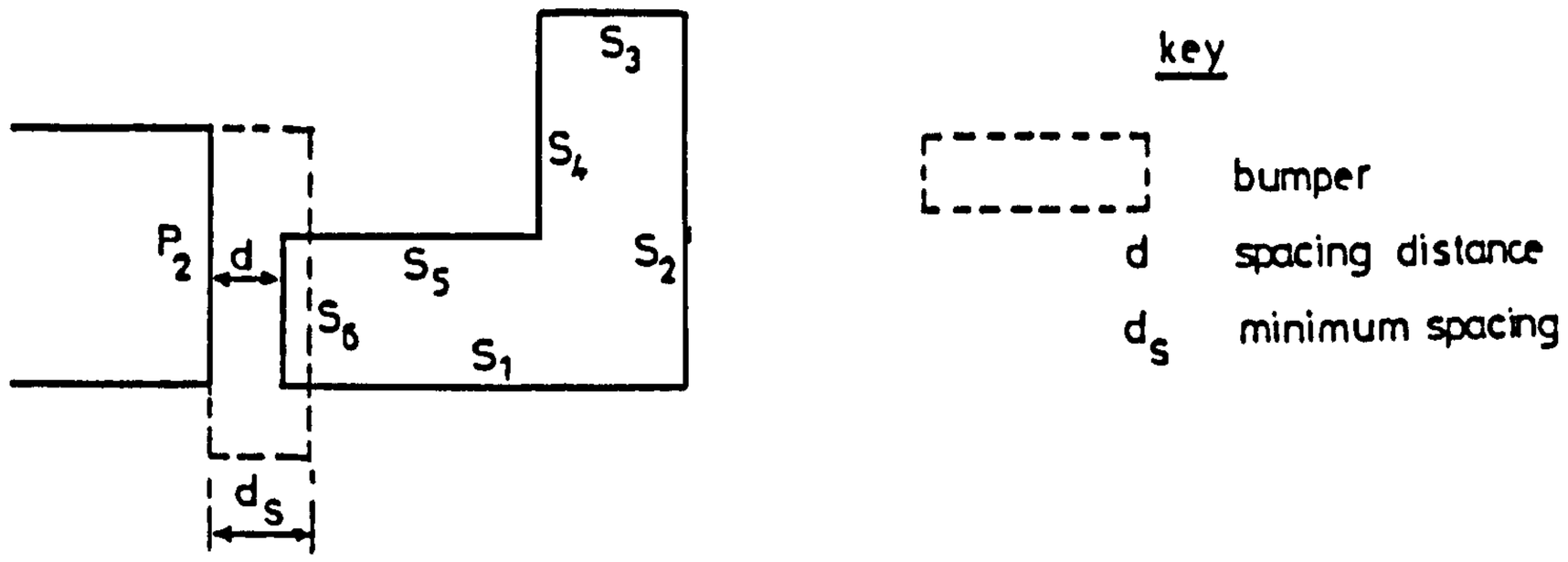
d_S = minimum spacing

The usual approach is to consider each segment combination in turn, calculate the minimum distance between the two segments, then compare this distance with the specified minimum spacing distance. If the calculated distance is less than the specified distance, then a SPACING violation has occurred. The problem with this technique is that the distance computation between two general segments is not trivial, and so is expensive in terms of CPU time to perform.

In an attempt to reduce this time, CADIC2 relies on the fact that in the dimensional check, the actual distance between two segments is never needed. All that is important is that the distance is less than or greater than the specified minimum distance.

In the example of the SPACING check, CADIC2 considers each primary segment in turn, and creates an influence bumper round the outside edge of the segment. The width of the bumper is set to the specified minimum spacing distance. Using highly efficient 'clipping' routines (as used

in computer graphics), all the secondary segments can quickly be checked to see if they enter the influence bumper. If yes, then a violation must have occurred, for example :-



Defining the influence bumper is more expensive to perform than a distance computation, but the operation is performed only once per primary segment. This expense is more that compensated by the fact that checking if a secondary segment enters the bumper is much cheaper than a distance computation. The more secondary segments, then the more CPU time is saved. In this way, CADIC2 can perform dimensional checks much faster than by the normal approach.

The second way in which CADIC2 minimises the data to be processed during design rule checking is through the concept of segment type identification. The concept is not new, but does help reduce the amount of data to be processed quite significantly, especially when dealing with orthogonal geometry.

CADIC2 simply ignores a segment pair if the segment are at right angles to one another. Consider the diagram above in the case when primary segment 'P2' is being compared against the secondary shape, and the distance 'd' is less than the specified minimum.

Without the segment type identification, all six secondary segments must be considered, out of which segments 'S1,S5,S6' will violate the spacing check. With segment type identification, only three segments 'S2,S4,S6' are considered, and only segment 'S6' violates the check. The latter approach is obviously much faster. It also minimises the number of violations detected, without degrading the thoroughness of the spacing check.

7.4 Design rule data structure

In off-line design rule checking, the rules are very often stored in a simple sequential text file. One reason for this is that all the rules must be carried out at some time during the check, therefore the order in which then checks are carried out is not important. Note that the file is only searched once.

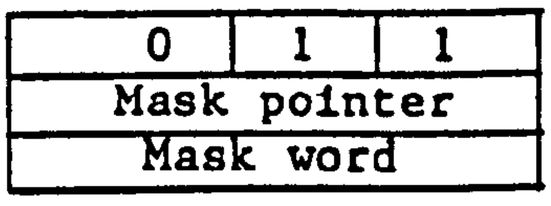
The second reason is that each entry in the list defines a mask operation (i.e. applies to all the shapes on a mask), therefore the time to read each file entry is negligible compared to the time required to implement the instruction.

On-line design rule checking requires a much more sophisticated method of storing the design rules, since the file will be searched every time a shape is added to the layout. The file must therefore possess :-

1. Good selection properties, so that all the rules relating to a particular mask may be identified quickly and easily.
2. High efficiency, so that then time spent accessing the file is kept to a minimum.

Ring data structures (as described in Section 5.5) are ideally suited to the above requirements. Therefore it was decided to use a ring data structure to store all the design rule information. An example of a typical design rule ring data structure is shown in Figure 7.1. The figure shows in fact the data structure that would be produced if the set of design rules described in Section 7.3 were compiled by DRCCAD. Each bead type possible in the design rule data structure will now be described in detail.

Design rule headbead : This bead is the first bead in the data structure, and simply provides a ring to which mask beads can be added. It's form is as follows :-



As with the beads used in the layout data structure implemented by CADIC1, the first byte in a bead provides information about the bead itself. This byte is split into three fields, and the fields are defined as bead identification, number of pointers, and number of data bytes respectively. The mask ring will hold all the mask beads required (see later). Lastly, the mask word is considered as 16 bits, 1 bit per mask. These bits are set to 1 if the relevant mask is used in a design rule, and 0 if not. By reading the word, CADIC2 can quickly find out whether the newly added shape will be involved in any design rules. The same information could be found by searching through the mask beads on the mask ring, but the frequency at which this information is required warrants a compacted storage format for quick reference.

Mask bead : A mask bead is used to collect together all the rules that relate to a particular mask. For example, to encode the following rules :-

```
PD IS RECT MASK 1
PS IS RECT,POLY MASK 2
CW IS RECT,POLY MASK 3
RULE ONE
      FAIL 'RULE 1' IF SEPARATE (PD,PS) AND SPACING (PD,PS) <10
END
RULE TWO
      FAIL 'RULE 2' IF ENCLOSED (PD,CW) AND CLEARANCE (PD,CW) <6
END
ENDOFFILE
```

Mask bead (1) would require a copy of rules ONE and TWO, whereas mask bead (2) would only require rule ONE, and mask bead (3) would require rule TWO. See Figure 7.1 for another example of this grouping. The form of the mask bead is as follows :-

63	3	2
Mask pointer		
General rule pointer		
Self-rule pointer		
Mask number		
Enlargement factor		

The mask pointer simply points to the next mask bead on the mask ring. The general rule and self-rule pointers however need more explanation. As described earlier, CADIC2 handles design rules in two ways :-

1. Self-rules, which are processed as the shape is built up
2. General rules, which are processed once the shape is complete

The general rules are never required when CADIC2 is performing self-checks, and the self-rules are never required when performing general checks. To keep these rules separate, the mask bead provides

two rings, one for the general rules, and one for the self-rules. The general and self-rule pointers therefore point to the next bead on the respective rings.

The next byte in the mask bead is the mask number. This number simply defines which mask the rules relate to. Lastly, the enlargement factor tells CADIC2 what size of influence bumper to use round a shape or group instance added to that mask.

Self-rule beads : Self-rules apply directly to the shape being added, and involve no other shapes. The form of the self-rule bead is as follows :-

i/d	2	3
Self-rule pointer		
Error pointer		
Rulename		
Sh. No.	Sh. type	
Dimension		

Type	i/d
WIDTH	9
LENGTH	10
INTERLIMB	11
XDIM	12
YDIM	13
AREA	14
BRAREA	15

The self-rule pointer points to the next bead on the self-rule ring. In the event of a rule violation, CADIC2 jumps down to the error ring to find the error bead (see later).

In the ring data structure, every rule bead contains a rulename. The reason for this is that CADIC2 provides the user with the option to switch off/on design rules if required. Because DRCCAD merges the independent rules into one data structure, the rulename is the only clue to the bead's origin.

The next byte in the bead stores the shape number and the shape type. Shapes associated with masks 1 to 15 have a shape number 1 to 15 respectively, and temporary shapes generated from logical operations have shape numbers from 50 upwards. Using this number, the shape information can be picked out from the shape list (see later). Only a certain type of shape may be required from a mask, so a shape type identification is also stored in the rule bead, so that unwanted shapes can be ignored. The shape types are defined as :-

1 - Rectangles

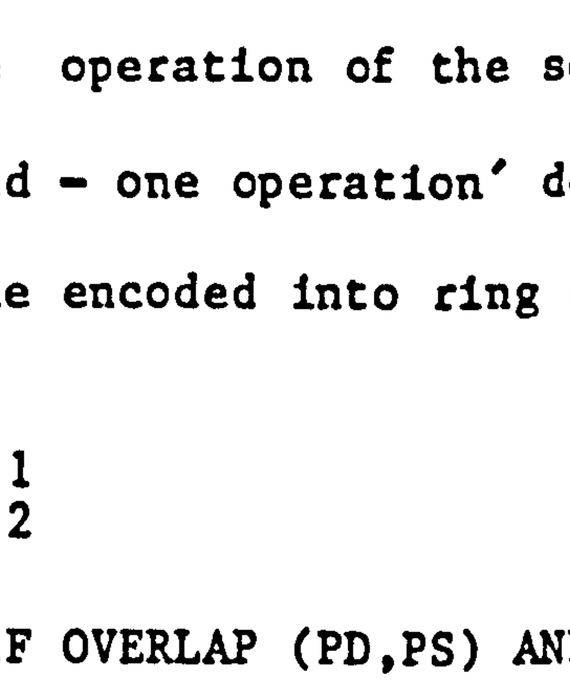
2 - Polygons

3 - Rectangles, and polygons

Lastly, the rule dimension is stored so that shape dimensions can be compared against it.

Selection bead : These beads are used by CADIC2 to select shapes for further processing. The operation of the selector bead is a bit more involved than the 'one bead - one operation' definition of other beads. Consider the following rule encoded into ring data structure format :-

```
PD IS RECT,POLY MASK 1
PS IS RECT,POLY MASK 2
RULE EXMPLE
      FAIL 'RULE 1' IF OVERLAP (PD,PS) AND WIDTH (PS) < 6
END
ENDOFFILE
```



After a shape is added to mask (1), CADIC2 will reach the overlap bead. From this bead, it will find the mask it must search (Shape number 1 to 15) and the shape type to be considered. CADIC2 then searches through the secondary mask (in this example, mask (2)) until a shape is found which overlaps the newly added shape on mask (1). If found, CADIC2 stores information about the secondary shape in the shape list (See later), then jumps down to the secondary ring. On this ring, CADIC2 encounters the width bead, so a width check is carried out on the secondary shape. If okay, CADIC2 will immediately return to the overlap bead, otherwise the error message will be printed out, before returning to the overlap bead.

Continuing from where it left off, CADIC2 now continues to search through mask (2) to see if any more shapes can be found which overlap the primary shape on mask (1). If yes, the above process is repeated. If no, (i.e. all the shapes on mask (2) have been processed), CADIC2 returns to the mask bead, ready for another shape to be added.

The form of the selector bead is as follows :-

i/d	2	3
Primary pointer		
Secondary pointer		
Rulename		
Sh. No. 1	Sh. type 1	
Sh. No. 2	Sh. type 2	

Type	i/d
OVERLAP	1
ENCLOSED	2
SEPARATE	3
ABUTS	4
DISTINCT	5
PARTED	6
* ENCLOSES	7
* PARTS	8

* Generated internally by DRCCAD

The primary pointer points to the next bead on the primary ring, and the rulename specifies the bead's origin. Shape (1) information defines the primary shape attributes, and shape (2) information defines

the attributes of the (secondary) shape to be searched for. If a secondary shape is identified, then CADIC2 jumps down to the secondary ring, to process the information further.

Topological bead - Type 1 : These beads perform a check on a shape against a specified minimum dimension.

The form of the bead is identical to the self-rule bead. The only difference is that their place in the ring data structure means that they can be used to check any defined shape, rather that just the newly added shape.

Topological bead - Type 2 : These beads are used to perform a dimension check between two defined shapes. The shapes may or may not be on the same mask. The form of the bead is as follows :-

i/d	2	4
Primary pointer		
Error pointer		
Rulename		
Sh. No. 1	Sh. type 1	
Sh. No. 2	Sh. type 2	
Dimension		

Type	i/d
SPACING	16
CLEARANCE	17

Logical bead : These beads take two shapes, and performs a logical operation on them, to produce an output shape or shapes. The form of the bead is as follows :-

i/d	1	4
Primary pointer		
Rulename		
Sh. No. 1	Sh. type 1	
Sh. No. 2	Sh. type 2	
Sh. No. 3	Sh. type 3	

Type		i/d
UNION	(+)	25
INTERSECTION	(*)	26
DIFFERENCE	(-)	27
EXCLUSIVE	(/)	28

The first point to note is that the bead contains only one pointer. This is because the bead is required to produce new shapes, and not make decisions. As described in other beads, the rulename defines the bead's origin.

Shape information (1) and (2) define the attributes of the input shapes, and shape information (3) defines the attributes of the output shape. Since the output shape is a temporary shape, it will always have :-

1. A shape number greater than 50, so that it is not confused with shapes found from the layout.
2. A shape type of (3), since the form of the output shape is not known.

<u>Inflate/deflate bead</u> : This bead accepts a shape, then inflates or deflates the shape by the specified amount, and stores the new shape as the output shape. The form of the bead is as follows :-

30	1	4
Primary pointer		
Rulename		
Sh. No. 1	Sh. type 1	
Sh. No. 2	Sh. type 2	
Dimension		

As with the logical bead, this bead has only one pointer, which points to the next bead on the ring. The input shape is defined by shape information (1). The inflate/deflate factor is stored in the dimension, and the attributes of the output shape is defined by shape information (2). As with the logical bead, the output shape number will be greater than 50, and the shape type will be set to (3).

Orientation bead : This bead accepts a shape, decides on the direction in which the shape points, then compares this direction against the required orientation. The form of the bead is as follows :-

1/d	2	2
Primary pointer		
Secondary pointer		
Rulename		
Sh. No.	Sh. type	

Type	1/d
HORIZONTAL	21
VERTICAL	22

The shape information defines the attributes of the shape to be checked. If the shape satisfies the desired orientation (defined by the bead 1/d) then CADIC2 follows the secondary pointer, otherwise CADIC2 follows the primary pointer.

Error bead : The error bead contains the error message to be printed out. This bead is only encountered when a violation has occurred. The form of the bead is as follows :-

31	1	n
Primary pointer		
Rulename		
Error message		

The error message may contain up to 64 characters, and is stored in the bead as two characters per byte.

7.5 Program operation

CADIC2 is written in FORTRAN, and consists of a library of routines, one routine for each failure condition command in the input language. The order in which the routines are processed is controlled by referring to the design rule ring data structure.

To save on development time, it was decided not to code up all the failure condition commands available in the manual input language. Instead only the most common commands were implemented. Because of the modular nature of CADIC2, new commands can be added without requiring changes to the existing software. The available commands are briefly described below :-

OVERLAP ..... Find shapes which overlap

ENCLOSED ..... Find shapes, one enclosed by the other

SEPARATE ..... Find shapes which are separate

WIDTH ..... Specify minimum width of shape

INTERLIMB ..... Specify minimum spacing between limbs of shape

AREA ..... Specify minimum area of shape

SPACING ..... Specify minimum spacing between shapes

CLEARANCE ..... Specify minimum clearance between shapes

AND ..... Connecting command

OR ..... Connecting command

UNION (*) ..... Perform logical OR function on shapes

INTERSECTION (+). Perform logical AND function on shapes

DIFFERENCE (-) .. Perform logical NAND function on shapes

EXCLUSIVE (/) ... Perform logical XOR function on shapes

A detailed description of each routine is given in Appendix (B).

Checking shapes : On-line design rule checking starts as soon as a shape is initiated (i.e. 'R','P','T'). At this stage, the parameters used to perform the self-tests - WIDTH, INTERLIMB, AREA are reset. As each segment is added to the shape, the following proceedures are carried out :-

1. The newly added segment is checked against the existing segments to see if a WIDTH or INTERLIMB violation exists. The details of these

algorithms will be discussed in Appendix (B).

2. The incremental area under the segment is calculated, then added to the summing total. Therefore, on finishing the shape, the total are will already be known, and can be checked for an AREA violation. See Appendix (B) for details.

If any of the self-tests fail, the shape as it presently exists, is drawn out in dashed lines, and the accompanying error message is printed on the alphanumeric screen. CADIC2 then gives the user the chance to accept or reject the violation. On receiving an answer, CADIC2 removes the shape from the screen, then proceeds in one of two ways :-

1. If the violation is accepted, then the shape is 'killed' from memory, and CADIC1 is returned to the cursor command level.
2. If the violation is rejected, then CADIC2 continues as if no violation had been identified.

Once the shape is complete, CADIC2 performs all the required general design rules using the design rule data structure to control its sequence of operations. Whenever a violation is found, the associated error message is printed out on the alphanumeric screen. After all the checks have been completed, CADIC2 proceeds in one of two ways :-

1. If violations existed, then the shape is drawn out in dashed lines, and the user given the chance to accept or reject the violations, just as with the self-test violation.
2. If no violations existed, then the shape is drawn out in solid lines, then added to the layout ring data structure. CADIC1 is then returned to the cursor command level.

Checking group instances : If the designer adds a group instance to the layout, CADIC2 must check all the shapes in the group instance against all the shapes in the layout. Note that shapes within the group instance do not need to be checked against each other, as they will already have been checked when then group definition was originally defined.

The combinatorial problem involved in checking the group instances is large, and wasteful of CPU time since usually only the shapes on the outside edge of the group instance are possible violation candidates. Therefore how can the number of check be minimised ? The check can be carried out in one of two ways :-

1. Each shape in the group instance is checked against all the shapes in the layout.
2. Each shape in the layout is checked against all the shapes in the group instance

At first, it may seem that both methods require the same number of checks. This is true if all shapes are treated as possible violation candidates, but as with adding a shape to the layout, an influence bumper exists round the group instance. This means that the majority of the shapes in the layout can be ignored. To show how this affects then number of checks required, consider the following example.

A group instance contains 60 shapes. The layout contains 5000 shapes, 50 of which enter the group instance's influence bumper. Assuming that a shape outside the bumper can be checked 20 times faster than a shape inside the bumper, a rough estimate of the number of checks required is :-

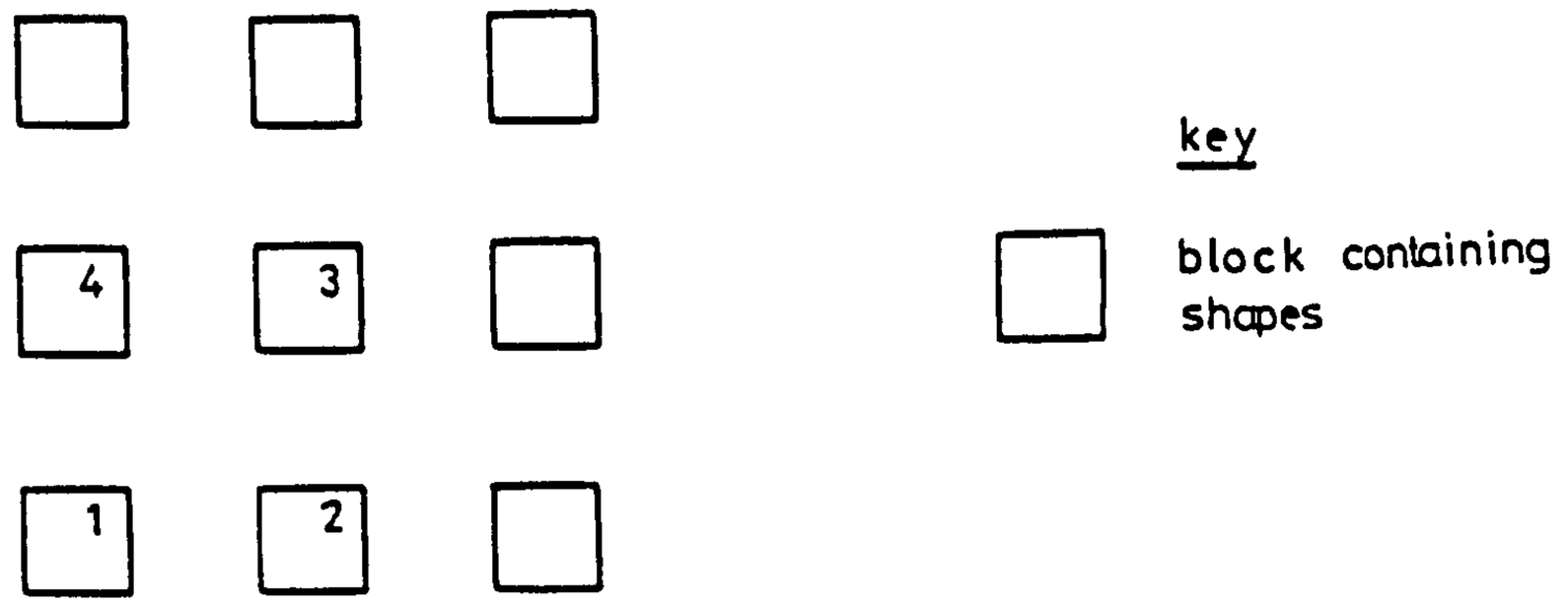
1. (GRIN v LAYOUT) = 60*(50 +4950/20) = 17350 Checks
2. (LAYOUT v GRIN) = 4950/20 +50*60 = 2979 Checks

Method (2), (that is checking the shapes in the layout against the shapes in the group instance) is therefore the method to adopt. To check a newly added group instance, CADIC2 proceeds as follows :-

1. Find the bounding rectangle of the group instance
2. Surround the group instance with the <u>largest</u> influence bumper associated with the masks used in the group instance.
3. Find the next shape in the layout that enters the influence bumper : [if finished RETURN]
4. Consider the group instance to now be the layout, and the shape identified in step (3) to be a shape newly added to the layout. The shape can then be checked against the group instance using the algorithm described in the previous section. Note that in this case, only the general rules are performed.
5. goto (3)

As with checking shapes, CADIC2 reports all violations to the designer. If no violations are found, then the group instance is added to the layout ring data structure, and CADIC1 is returned to the cursor command level. If violations exist, the user is given the chance to accept or reject the group instance.

Checking array instances : If an array instance is added to the layout, CADIC2 proceeds in exactly the same way as if it had been a group instance. The only difference now is the fact that the shapes inside the array may affect each other, and cause design rule violations. Therefore, before the array is checked against the layout, an 'array-test' must be performed. Consider the following array :-



Each 'block' in an array is simply a group instance, so the shapes within each block need not be checked against other shapes in the same block, since this will have been performed when the group definition was defined.

Shapes associated with different blocks will have to be checked against each other, but due to the symmetry of the array, the array-test can be performed using a maximum of 4 blocks, regardless of the size of the array.

CADIC2 performs the array-test by considering blocks (2,3,4) as the layout, and block (1) as a newly added group instance. The algorithm used to check group instances can then be used to perform the array-test.

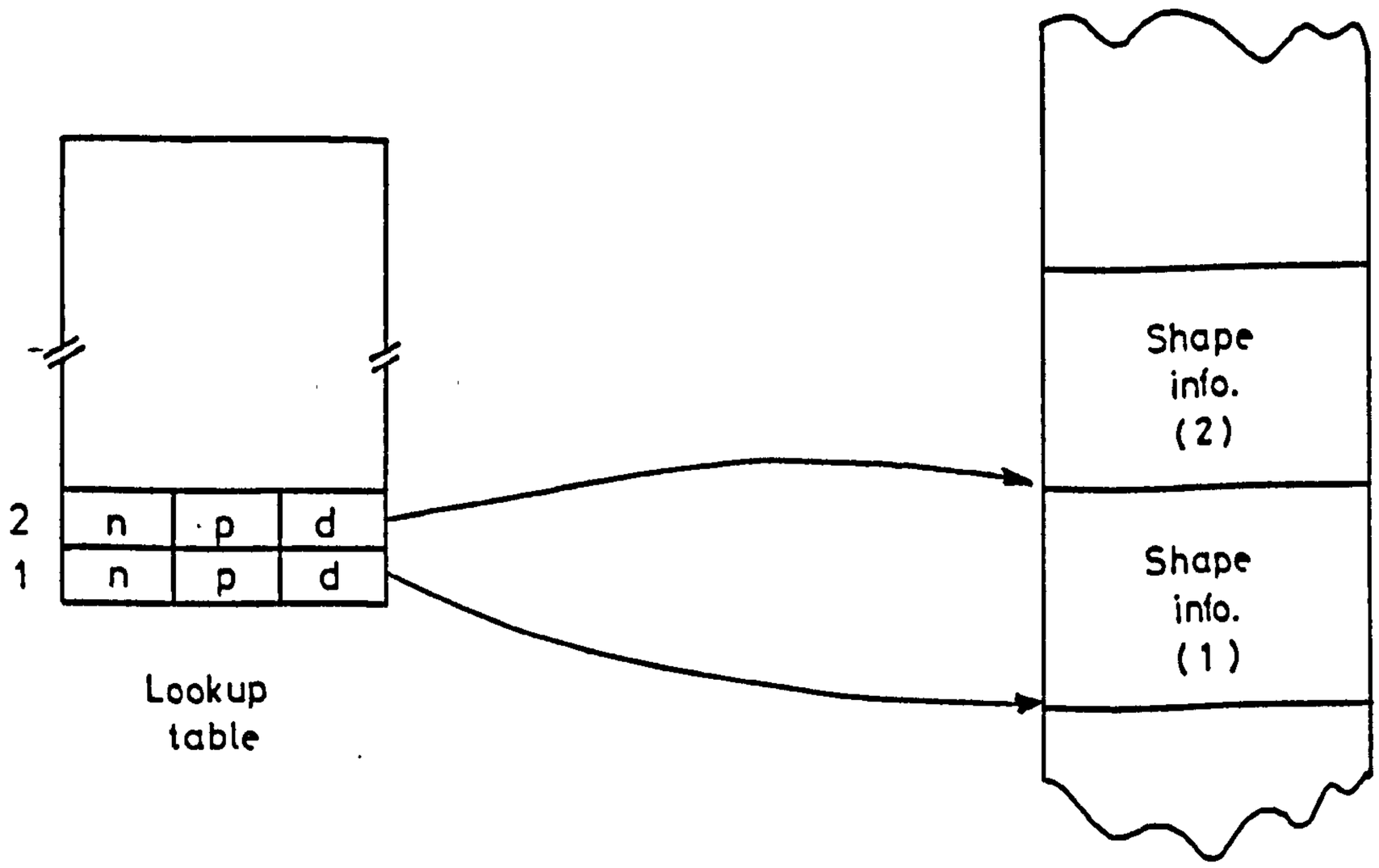
During the array-test, all violations are reported to the user, but are preceded by a note to warn the user of the violation's origin. Therefore a spacing error between horizontal blocks which should have

been reported nine times for the array shown above, is only reported once, along with an array-test warning.

7.6 Shape list

During design rule checking, information about shapes selected or created must be stored for future reference. To handle this, CADIC2 uses the concept of a shape list. The list is resident in the same temporary file used by CADIC1, and is initialised on adding a shape to the layout. In this way the first shape in the shape list is always the newly added shape.

The shape list operates in a last-in-first-out fashion, therefore as shapes are selected or created, they are added to the end of the shape list. Similarly, once the shape has served its purpose, it is removed from the end of the list. Note that there is no restriction on the size of the shapes. The format of the shape list is as follows :-



Lookup table : The lookup table consists of a matrix stored in core, so as to allow fast access to its information. Each entry in the table consists of three values :-

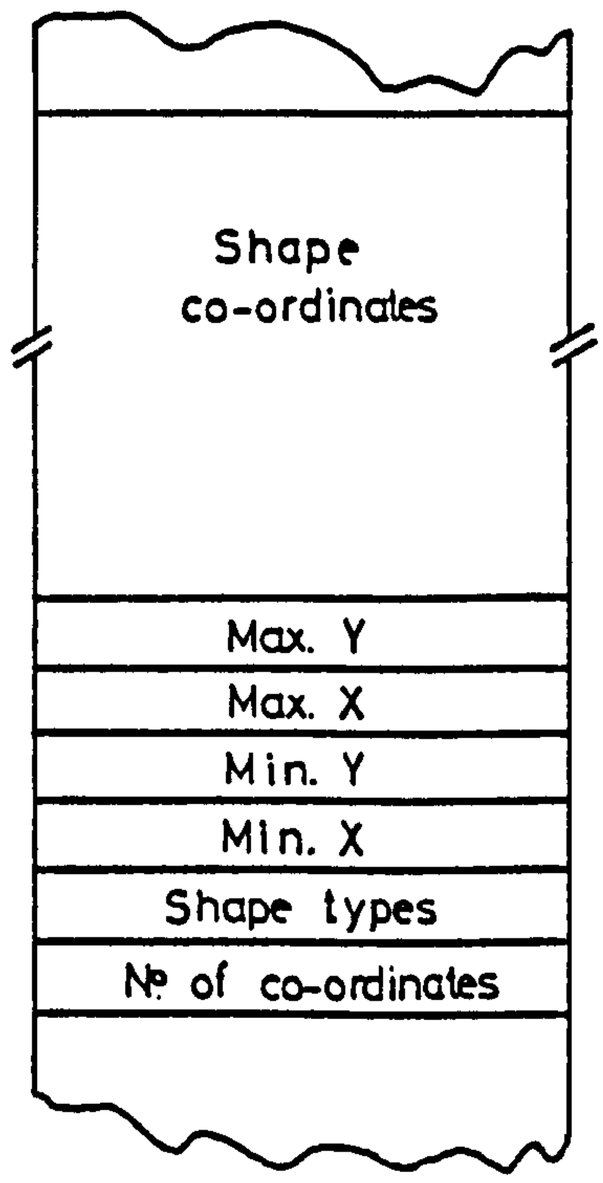
1. Shape number
2. Shape pointer
3. Shape depth

All shapes handled by CADIC2 can be identified by their shape number. Storing this number in the lookup table provides a quick reference facility for routines looking for particular shapes. If more information about a shape is required, then the shape pointer is used to locate the shape in the temporary file.

Lastly the shape depth is associated with which ring CADIC2 was processing in the design rule ring data structure when the shape was created. The newly added shape has a depth of one, all the shapes generated on the first ring are given a depth of two, and so on.

Once CADIC2 has completed a ring, all the shapes generated in that ring, have served their purpose, and can now be removed from the shape list. Therefore if CADIC2 had just completed the third ring, and was jumping back up to continue processing the second ring, all shape with depths of four and above can be deleted from the shape list.

Temporary file : Each block in the file contains information about the shape stored. The contents of each block is as follows :-



The first byte in the block defines the number of coordinates in the shape. The second byte gives information about the type of shape stored, and is defined as follows :-

3. Closed rectangle
4. Open rectangle
5. Closed short format polygon
6. Open short format polygon
7. Closed long format polygon
8. Open long format polygon

The design rule checking routines often use the shape's bounding rectangle to try and minimise the amount of processing required. For this reason, the shape's bounding rectangle is also stored in the block.

The remainder of the block contains the shape coordinates. Note that the coordinates are always stored in long format. The compact storage forms for rectangles and short format polygons could have been used to save space, but space is not a problem, plus decoding the coordinates every time a shape was used made CADIC2 inefficient.

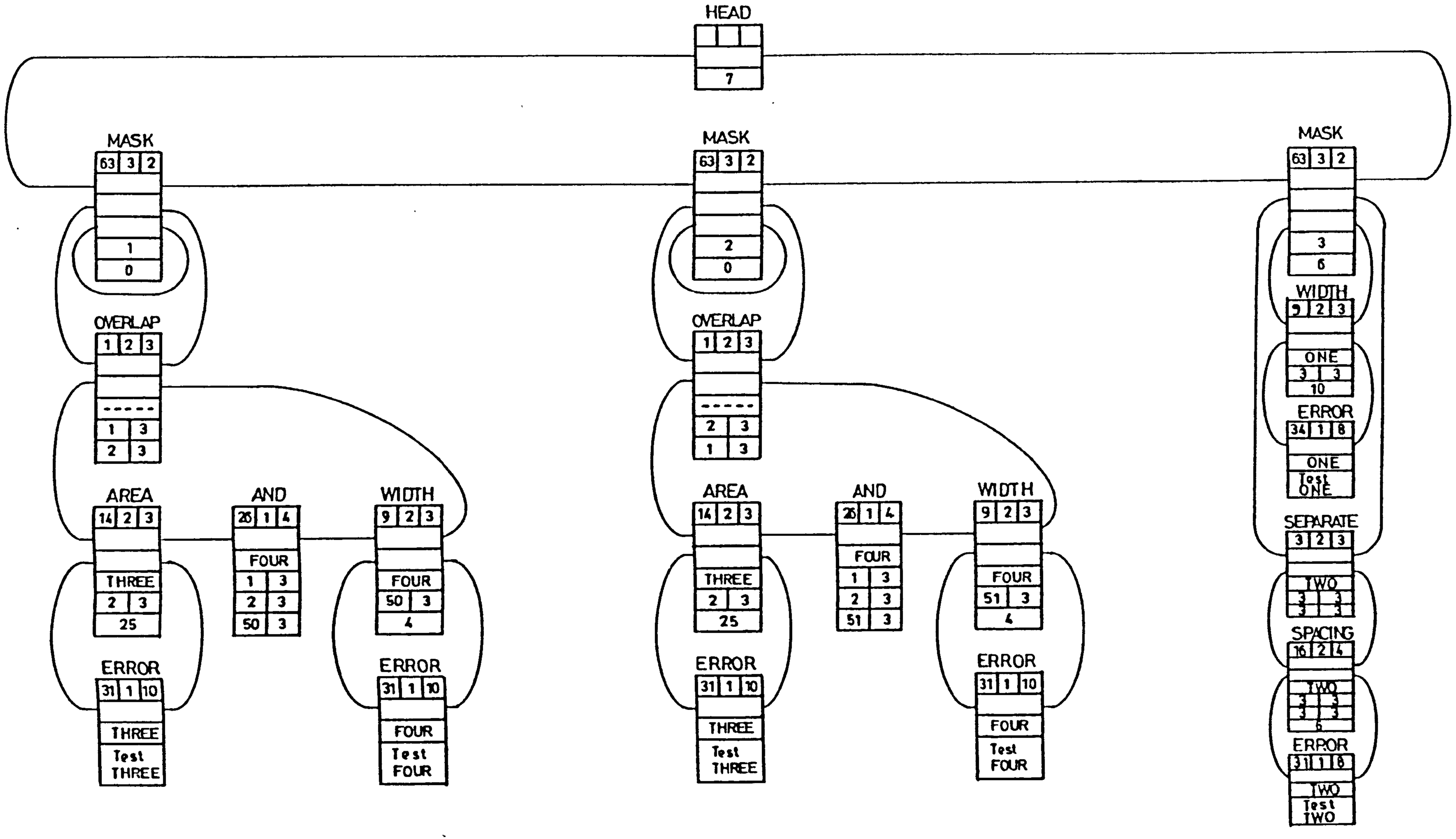


Figure 7.1 Design rule ring data structure

CHAPTER 8

Performance

8.1 Introduction

This chapter is devoted to evaluating the performance of the CADIC suite of programs. Each program will be discussed in terms of its main features, but emphasis is placed upon CADIC1 and CADIC2, the most important programs in the suite.

8.2 CADIC1

CADIC1 is an interactive design aid which allows the user to design integrated circuits at the geometric level. In Chapter five, three main techniques to improve the efficiency of processing disc-based layout data were proposed. These were :-

1. Area segmentation
2. Cleaning the layout ring data structure
3. Organised group processing

This efficiency is evaluated in terms of plotting efficiency, since plotting is probably the most common operation implemented in a design aid, plus one which can easily be related to, or compared against other design aids. Two measurements of plotting efficiency are observed :-

1. CPU time
2. Number of page swaps

CPU time is self explanatory. The number of page swaps is however less obvious as an alternative measurement of efficiency.

Only a few pages of the layout data structure are stored in core at any one time. Therefore if data is required from a page which is not in core, a page must be removed from core to allow the required page to enter. This process is termed page swapping, and is to be avoided if at all possible. The reason for this is that a page swap involves mechanical movement of the disc head, first to locate the page, then to read it out, and may take milliseconds (real-time) to complete. Page swapping also consumes a fair amount of CPU time, so much so that the number of page swaps and CPU time are generally directly related.

If CPU time and number of page swaps are directly related, why the need for both measurements ? The reason is that the computer used for this project is very powerful, therefore the CPU measurements do not show up small inefficiencies to any great extent. The number of page swaps required during any specific operation is not affected by computing power, and so provides a clearer, more sensitive measurement of efficiency.

8.2.1 Area segmentation

In CADIC, the layout is considered as split up into a series of areas, and all shapes in the layout are 'polygon clipped' such that each shape, or sub-shape is associated with only one area. The size of the area is under program control therefore tests must be carried out to find, if possible, the optimal setting. Emphasis will be placed on plotting efficiency, but other factors such as memory requirements, and 'finding' efficiency will also be considered.

Plotting : The circuit chosen for this test is shown in Figure 8.1. This test circuit contains no group calls, so that the effects of area segmentation can be isolated. Note that the test circuit is in no way meant to represent a real circuit. It is the existence of shapes that is important in this test rather than their topology.

To analyse plotting efficiency, CADIC1 was tested using two sizes of window ; large and small. The large window contains the whole layout, whereas the small window shows only a small section of the layout, so that the shapes can be seen in enough detail to be edited.

It is important to note that greater emphasis is placed on maximising plotting efficiency for the case of the small window. There are three main reasons for this :-

1. Around 90% of the design work is carried out using a small window.

2. The designer will expect immediate program response, since he only has to consider the small section of layout visible on the screen, unlike the computer which must always consider the whole layout.

3. During design rule checking, CADIC2 will require information about shapes local to the newly added shape to be found very quickly. This is similar to the case of plotting out a very small window which contains only a few shapes. Therefore optimising the performance of CADIC1 for the case of the small window will enhance the performance of CADIC2.

The results of the plotting test are shown in Figure 8.2. On analysis, there are three points to note :-

1. The layout dimensions are 2040 x 1640 increments. Therefore, at area size 2048 x 2048 increments, the whole layout is held within one area. The results for this size of area are therefore those that would be obtained if area segmentation was not used.

2. Area segmentation inhibits efficiency when plotting the whole layout. The reason for this is that under this case, each area is always inside the window, therefore the extra time is now purely due to processing the redundant area beads. As the area size reduces, so the number of area beads increases, which increases wasted processing. Less than optimal processing efficiency is however not so important when plotting out large windows. In general, the time to plot out the layout will always be lenghty, therefore a few seconds extra will not be noticed to any great extent.

3. Area segmentation enhances efficiency when plotting out the small window. Figure 8.2 shows that a global minimum exists at area size 512 increments. The reason for this is that the window size chosen was in the order of 500 increments (which is a typical size chosen in practice). Larger areas will always contain shapes which do not enter the window, regardless of the window position. Time spent processing these redundant shapes is therefore wasted. The effect obviously becomes worse as the area size increases. When using area sizes smaller than the window size, the increase in CPU time is due to processing extra area beads. Once again, as area size decreases, the CPU time increases.

In conclusion to the effect of area segmentation on plotting efficiency, an area size equal to the typical working window size will maximise processing efficiency.

Memory requirements : As the area size is reduced, more shapes are going to cross area boundaries, hence more sub-shapes must be stored in the ring data structure. The effect of area size on memory requirements is shown in Figure 8.3a.

This graph also explains why the small area size increases CPU time in the plot times. Because the file was larger, the required data was 'further' apart, therefore more page swaps were required to retrieve it.

Point finding : CADIC1 will often want to find out information about shapes local to the point of interest, rather than the more global process of plotting out sections of layout. For example, finding the nearest point in the data structure to the cross-hair cursor ('F' and 'N' cursor commands). The effect of varying area size on the find time is shown in Figure 8.3b.

The results are obvious in the fact that the cursor can only be in one area at a time, therefore, the smaller the area, the smaller the number of shapes that CADIC1 has to check.

8.2.2 Cleaning the layout data structure

In section 5.3, it was mentioned that re-organising or 'cleaning' the data structure should reduce the number of page swaps, and enhance processing times. The order of re-organisation implemented was as follows :-

The designer can only work on one group definition at a time. Therefore each layout/group definition is treated as an independent 'block' of data. Each 'block' can then be placed sequentially on the 'clean' file. Within each 'block', the headbead is placed at the beginning, followed by all the area beads. Next is placed the first mask bead, then all the shapes associated with that mask, then the second mask bead, and so on. Lastly, the group instance beads are grouped together at the end of the 'block'. Note that all the beads previously on the garbage ring are not copied onto the 'clean' file, hence reducing memory requirements.

To test the effect of 'cleaning' a ring data structure, the 'WMLU' circuit was 'cleaned' then re-tested as in Section 8.2.1. The results are shown in Figure 8.4. If Figures 8.2 and 8.4 are compared, it can be seen that substantial improvements in plotting efficiency were achieved. Frequent 'cleaning' of a ring data structure is therefore advisable.

8.2.3 Organised group processing

The way in which the layout group hierarchy is processed will dramatically effect program efficiency, therefore a test circuit (Figure 8.5) was developed to highlight inefficiencies in the various processing algorithms. This 'GROUP' circuit is highly structured (up to 6 levels of nesting), with each group definition being small enough to be enclosed within one area. In this way, the 'GROUP' circuit can isolate the characteristics of the group processing algorithms.

Random processing : This technique is probably the most common in existing design aids, and simply involves processing group instances as they are encountered in the data structure. If the data structure is

small enough to be core resident, then no problems occur. However, if the layout must be stored on disc, and paging is required, random processing of the group instances means random reading of the data structure. Such systems are therefore prone to page thrashing.

The results of plotting out the 'GROUP' circuit depending on the percentage of data structure held in core, are shown in Figure 8.6.

Organised processing (Method 1) : Instead of processing instances as they are encountered, CADIC1 tries to obtain a more global knowledge of the layout hierarchy, by storing information about the group instances in a temporary file (see Section 5.3 for details). In this way, CADIC1 efficiently utilizes the group information while it is in core, and so increases program efficiency.

The 'GROUP' circuit was tested using this algorithm, and the results are shown in Figure 8.6. As can be seen, the page swaps required are less than that required by the random approach, proving the correctness of the logistics for Method 1. On the other hand, higher CPU times were required. There are two reasons for this :-

1. The extra time required to build up, and process the temporary file.

2. The form of the data in the temporary file was a simple sequential list. CADIC1 therefore had to search through a lot of redundant data each time to find all the group instances pointing to the group definition in core.

On obtaining the above results, Method 1 was modified into Method 2.

Organised processing (Method 2) : The modifications to Method 1 are as follows :-

1. Instead of setting up the temporary file each time CADIC1 processed the data structure, the group data, once set up, was permanently stored in the temporary file, until the layout group hierarchy was altered in some way. Future processing only had to read from the file, thus saving CPU time. Note that if not required, the pages in the temporary file containing the group data will automatically be paged out onto disc, and remain there until needed. The storage penalty is therefore restricted to relatively cheap disc space.

2. All group instances in the temporary file which point to the same group definition are now linked together on a ring of pointers. The search time required to find find all instances of a similar nature is thus kept to a minimum.

The 'GROUP' circuit was again tested using this new approach, and the results are shown in Figure 8.6. As can be seen, substantial improvements in CPU time were achieved over Method 1, and more importantly, the random approach. CADIC therefore uses Method 2 to process the layout group hierarchy.

8.2.4 CADIC1 v GAELIC

To find out just how efficient CADIC1 is in practice, it was compared against GAELIC [1], a commercially available design aid, known to be very efficient. The circuit chosen for the comparison is a 'real' circuit which was kindly supplied by Compeda, Stevenage. For copyright reasons, only a few masks are shown in Figure 8.7, but to give an idea

of approximate complexity the layout contained around 80,000 shapes.

Both design aids were given the 'PRIME' circuit to plot out at a variety of window sizes, and a graph showing the CPU times for each design aid is given in Figure 8.8. Three points are worth noting :-

1. Early tests with this circuit showed that the paging routines used by CADIC were inefficient. On discovering this fact, CADIC was changed so that it used the same paging routine as GAELIC. This greatly reduced the CPU time required by CADIC, but unfortunately no longer provided information about page swaps. For this reason, only CPU time is shown in all future tests.

2. At large window sizes, CADIC1 is less efficient than GAELIC. This was expected, since CADIC1 carries more overheads in sustaining area segmentation, and organised group processing.

3. As the window size (and therefore the percentage of the layout actually required) decreases, so CADIC1 improves on its performance over GAELIC. Note that for the size of layout used in this test, most of the design work would be carried out with a window size of 15% full layout and smaller, so that the layout could be seen in enough detail to be edited. In this situation, CADIC1 is much more efficient than GAELIC.

8.3 CADIC2

CADIC2 on-line design rule checks a newly added shape or group call against the existing layout, using a pre-defined set of design rules.

Testing CADIC2 under realistic conditions is however a very difficult problem to solve. The first reason for this is that many factors affect the time taken to design rule check any one particular shape, for example :-

1. The mask containing the shape, since different masks usually contain a different number of rules
2. The complexity of the rules
3. The number of shapes in the existing layout
4. The position of the newly added shape in layout
5. The number of segments in the newly added shape

All these variable factors means that it is extremely difficult (if not impossible) to generate a set of representative results, when considering isolated cases. For example, a slight variation of position of two shapes undergoing a spacing check may double the required design rule checking time.

The only way to solve this problem is to consider the results in a more global nature, for example consider the performance over a whole layout design. In this way, local differences can be ignored in favour of the general trends in performance.

The second problem with evaluating CADIC2's performance is how to collate the design rule checking times for a whole layout design. Using CADIC1 to interactively design a circuit is far too slow, especially if large circuits are required.

A better technique is to use MANCAD to simulate the design of a whole layout. Once MANCAD has decoded a line of the manual input language into a set of shape coordinates, it uses CADIC2 to design rule check the shape against the existing layout, just as if the shape had been added interactively using CADIC1.

By noting the design rule checking time required for each shape/group call, and plotting it on a graph, the performance of CADIC2 over a complete layout design can be obtained in only a few seconds (real time). Therefore MANCAD was used to generate all results displayed in this section.

Chapter seven discussed techniques which would hopefully minimise the time required to design rule check a shape. These were :-

1. The design rule data structure compiled by DRCCAD ensures that CADIC2 performs the minimum number of operations.

2. Each routine in CADIC2 is optimised such that the CPU time required to complete each operation is minimised.

3. The concept of area segmentation allows very quick access to shape information local to the newly added shape/group call.

Each technique will now be discussed in more detail. Unfortunately, the first technique cannot be experimentally verified, since major software changes would be required to implement different forms of design rule data structure. It is hoped however that the logistics given in Chapter seven satisfy the claim that CADIC2 performs the minimum number of operations.

8.3.1 Routine performance

The exact details of each design rule algorithm used by CADIC2 is given in Appendix B. For reasons described in Section 8.3, it is very difficult to isolate each operation, and attempt to relate it to a typical time. Performance of the design rule routines is therefore considered in a more global nature. In general, two main concepts within each routine helped minimise design rule checking time. these were :-

1. Influence bumper
2. Segment type identification

Each concept will now be isolated and experimentally tested.

Influence bumper : The influence bumper placed round a newly added shape/group call allows CADIC2 to ignore all shapes outside the bumper, and so minimise redundant processing. The concept is really only implemented when the SEPARATE command is used, since all other selection commands (i.e. OVERLAP, and ENCLOSED) have an implicit influence bumper width of zero. However the frequent use of the SEPARATE command warrants the use of the shape influence bumper. Consider the layout shown in Figure 8.9. The layout contains around 500 shapes on a single mask, and the following design rule was applied :-

```
PD IS RECT, POLY MASK 1
RULE A1
     FAIL 'Spacing test' IF SEPARATE (PD,PD) AND SPACING (PD,PD) < 70
END
ENDOFFILE
```

Note that the layout is a test layout, and does not represent a working circuit. The test layout also contains no design rule violations.

A graph showing the performance of CADIC2 with and without using a shape influence bumper is shown in Figure 8.10. Some points to note about the results are as follows :-

1. No design rule violations were identified, as expected
2. The time taken to design rule check each shape increases approximately linearly with the size of the layout. Note that this is a very important characteristic of CADIC2 which will be discussed in more detail later. However, this linear relationship allows a 'quality factor' to be attached to any particular set of results, so that the effect of changes within a routine can be evaluated. The technique used in these tests is to apply a best-line fit to the results, and so obtain the equation of the line :-

$$y = mx$$

where 'y' is the typical time (in milliseconds) required to design rule check the x'th shape added to the layout. The gradient 'm' therefore acts as the 'quality factor'. The lower the value of 'm', then the more efficient is CADIC2. Note that the best line always passes through the origin, since the first shape in the layout requires no design rule checking.

3. Without a shape influence bumper, CADIC2 is checking many more shapes against the newly added shape than is required. Applying a best-line fit to each graph shown in Figure 8.10 produces the following :-

	Quality factor
Without bumper	5.11
With bumper	0.21

Therefore implementing the concept of a shape influence bumper means that the CPU time required by rules using the SEPARATE command is only 4% of the CPU time required if the bumper was not used.

The layout shown in Figure 8.9 was then re-tested with a design rule check that required a different type of selector, for example :-

```
PD IS RECT, POLY MASK 1
RULE A2
     FAIL 'Overlap test' IF OVERLAP (PD,PD) AND WIDTH (PD) < 60
END
ENDOFFILE
```

Results showing the effect of performing the test with and without influence bumpers is shown below :-

	Quality factor
Without bumper	0.23
With bumper	0.23

As was expected, the bumper has no effect on the OVERLAP selector. Similar results would also be obtained for the ENCLOSED selector. An important point to note is that the influence bumper greatly helps the SEPARATE selector, but not at the expense of the other selectors.

The second use for an influence bumper is during dimensional checks. Chapter seven also proposed that CADIC2 would perform better if it used the concept of an influence bumper round a segment and then checked to see if any segments entered it, rather than use the classical approach to determine the minimum distance between each segment combination. To test this, the layout shown in Figure 8.9 was again tested using RULE A1 shown above. One test used the concept of an

influence bumper, and the other test performed minimum distance calculations. The results of performance obtained is shown below. Note that the shape influence bumper was used in both cases to minimise the shapes identified by the SEPARATE command.

	Quality factor
Bumper approach	0.22
Classical approach	0.24

As can be seen by the quality factors, a routine employing the bumper approach will operate about 10% faster than a routine using the classical approach. CADIC2 therefore incorporates the concept of segment influence bumpers into all dimensional routines.

Segment type identification : Chapter seven finally proposed that only certain combinations of segments need be checked during dimensional checks. By calculating a type (i.e. horizontal, vertical, or angled) for each segment, large reductions in the number of segments to be considered is possible.

To test this proposal, the layout shown in Figure 8.9 was again used, along with RULE A1. Results showing the performance with and without the use of segment type identification are as follows :-

	Quality factor
Without identification	0.22
With identification	0.21

In this case, identification improves performance by 5%. CADIC2 therefore incorporates segment type identification into all dimensional checks.

8.3.2 Area segmentation

During all selection operations (i.e. SEPARATE, OVERLAP, and ENCLOSED), CADIC2 must find information about the shapes close to the newly added shape/group call. Just as was shown with plotting out small windows (see Section 8.2.1) the area size should have an important effect on the number of shapes considered, and so the time taken to complete the design rule checks.

To analyse the effect of area size, CADIC2 was tested against the layout shown in Figure 8.11. This layout (which in no way represents a working circuit) uses six masks, and contains a total of 1840 shapes. Note that no group instances are present, so that the effect of area segmentation can be isolated. A complete set of design rules (shown in Figure 8.12) was used in the design rule checks, to test the layout under realistic conditions. To add to this reality, the layout contains 48 design rule violations.

A graph showing the performance of CADIC2 against various area sizes is shown in Figure 8.13. Note that plotting speeds are also shown for a typical small window, so that the optimal setting between CADIC1 and CADIC2 can be compared. Some points to note about the results are as follows :-

1. The optimal setting for CADIC2 is 128 increments. If the area size is increased, then the quality factor increases since more shapes within any one area must be analysed. In theory, smaller area size should always mean lower quality factor. However as area size is reduced, more shapes are liable to be 'polygon clipped' into sub-shapes. The routines within CADIC2 always reconstruct

sub-shapes into the original shapes to prevent the generation of false violations. This reconstruction process is rather expensive in terms of CPU time, therefore very small area size forces an excessive number of reconstructions which increases the quality factor.

2. The optimal setting for CADIC1 when plotting out the small window is 256 increments. In this case, because the layout was smaller, the typical working window size would be around 150 increments. Note how the optimal area size for CADIC1 changes with window size, so backing up the conclusion stated in Section 8.2.1.

3. A compromise on area size is therefore required between CADIC1 and CADIC2. The variation of CADIC1's optimal area size with the working window size (largely determined by the size of the layout) means that it is very difficult to specify an <u>overall</u> optimal area size for CADIC (CADIC1 and CADIC2).

Priority should be given to optimising CADIC2, since on-line design rule checking must always remain 'transparent' to the user. However, if the difference between CADIC1 and CADIC2's optimal area size is too large, then CADIC1 will perform very inefficiently.

In conclusion, if the layouts to be designed are liable to be small, then CADIC's optimal area size would best be set at 128 increments. However, as the layout size increases, CADIC's optimal area size becomes less well defined. Under these conditions, a final decision on the area size would best be left until the requirements of the user were discussed in more detail.

4. 96 design rule violations were identified. This number is artificially high due to the fact that the layout is made up of a matrix of sixteen identical sections. Within each section, three real violations existed. CADIC2 identified the three violations, plus another three, caused by the implicit over-expansion of the orthogonal influence bumpers at shape corners. CADIC2 will of course identify the same violations in each section of the matrix, so producing the high number of violations quoted above.

Note that most existing design rule checking programs use some sort of orthogonal distance test during dimensional checks, and nearly all programs produce false violations due to over-expansion at the shape corners. CADIC2 is therefore not alone with this problem. The justification for using the orthogonal approach is that it performs the checks very quickly. However, other (slower) techniques which do not generate these false violations are discussed in Section 9.2.

8.3.3 Hierarchical design

It is important to point out that good hierarchical or structured design will significantly improve the performance of CADIC2. This is largely due to the fact that all the shapes within a group definition only have to be checked against each other once. For example, if a transistor is defined as a group definition, and the shapes within the group definition satisfy all the design rules, then all group calls of the transistor must also be correct, and so do not need to be checked. The checks are therefore limited to checking the group call against the existing layout.

Hierarchical design also helps limit the number of violations generated during checking. If the above mentioned group definition contained a violation, then the violation would be generated only once, instead of every time the group call is used. The designer therefore is not swamped with multiple versions of the same violation.

To show how hierarchical design effects performance, consider the layout shown in Figure 8.11 as broken up into one group definition (containing all the shape information for one of the sixteen sections), plus sixteen group calls of the group definition. In this way the 'new' layout appears identical to the original layout, which consisted purely of shapes. This example is rather trivial, but it does serve to show how hierarchical design can help CADIC2. The results for checking the two versions of the same layout are as follows :-

Description	Quality factor	Total time	Number of errors
1. 1840 shapes	0.12	223.7	96
2. 1 def. (115 shapes) + 16 group calls	0.08	5.0	6

The substantial difference in total time comes from the fact that in layout (1), CADIC2 checks 1840 shapes with a quality factor of 0.12, whereas in layout (2), CADIC2 checks only 131 shapes/group calls with a quality factor of 0.08. Good hierarchical design therefore significantly reduces the time spent design rule checking the layout as it built up.

8.3.4 Checking a large layout

Lastly, it is important to observe the performance of CADIC2 as it design rule checks a 'real' circuit, using 'real' rules. Such data was kindly supplied by the Wolfson Microelectronic Unit, Edinburgh.

The design rules are shown in Figure 8.12, and a small section of the circuit is shown in Figure 8.14. Due to copyright requirements only a few masks of the layout are shown, but the whole layout uses eight masks, and contains around 30,000 shapes, incorporated into a hierarchical design, with nesting down to four levels. The graph of CADIC2's performance is shown in Figure 8.15. Some points to note about the results are as follows :-

1. The most important point to note is that the time taken to check each shape/group call increases linearly with the size of the layout. This is a vast improvement over existing off-line design rule checkers, which usually experience parabolic (n*n) performance. The linearity is largely due to three factors :-

   1.1 The area segmentation concept discussed in Chapter five minimises the amount of shape data to be analysed to often just the shapes within the present area, regardless of how many other areas have previously been filled.

   1.2 The use of the influence bumper limits the number of shapes and/or segments to be considered

   1.3 The use of segment type identification limits the number of segment combinations required during the dimensional checks

2. Out of the 2800 shapes/group calls added to the layout, around thirty 'additions' required more than two CPU seconds to perform the checks. In fact, the peaks go higher than ten CPU seconds. The reason for such a variation in performance is that each increment in the x-axis represents a shape/group call being added to the layout. A group call contains possibly hundreds of shapes, so the time to check a newly added group call is obviously going to be much greater than the typical time for a newly added shape. This fact is amplified by the fact that the layout shown in Figure 8.14 uses a very large group call which contains around 20,000 shapes. Any shapes/group calls which must be checked against this large group call is going to require an enormous number of checks. Nothing can be done to improve this. The only consolation is that is this circuit, the situation is limited to around 0.1% of the total number of 'additions'.

To try and iron out the large variations in performance, the method of noting the design rule checking time was modified when considering group calls. Instead of simply noting the time to check the whole group call, the time to check each shape within the group call was recorded, just as if it had been added independently of the group call. The graph showing the modified results is shown in Figure 8.16. Some points to note about the graph are as follows :-

1. The total number of shapes checked was 6712, yet the layout contains around 30,000 shapes. The reason for the difference is that when a group call is added to the layout, CADIC2 minimises the number of checks required by considering only the shapes that <u>have</u> to be checked. Details of how CADIC2 does this is given in Section 7.5.

2. The graph still contains a few large peaks. These peaks are due to shapes which must be checked against the large group call mentioned above. As mentioned above, nothing can be done to improve this.

8.4 DRCCAD

DRCCAD compiles a set of design rules into a ring data structure readable by CADIC2. The performance of this program is not really important, yet a set of results for a single run of the program is presented, just to show that DRCCAD is no better, and no worse than expected.

DRCCAD was tested while it compiled the set of design rules shown in Figure 8.12. Results are as follows :-

1. Number of rules	33
2. CPU time required	1.6 secs.
3. Size of ring data structure	2 pages (512 words/page)

8.5 MANCAD

MANCAD can operate in one of two modes :-

1. Manual input language compiler
2. Off-line design rule checker

The performance of MANCAD in each mode will now be discussed in more detail.

8.5.1 Manual input language compiler

As with DRCCAD, there is nothing exceptional about MANCAD's performance as a compiler. Compilation speeds are typical for the type of processing being undertaken. However, results of MANCAD's performance are given below, largely for completeness. The manual input file used in this test was the one required to produce the layout shown in Figure 8.14.

1. The input file contained 2800 lines (35 pages) representing about 30,000 shapes.

2. CPU time required was 64.5 seconds

3. Memory requirements for layout ring data structure was 79 pages (512 words/page)

8.5.2 Off-line design rule checker

Because MANCAD simply envokes on-line design rule checking techniques to simulate classical off-line design rule checking, it follows that any improvements in the on-line design rule checking performance must also appear in the off-line design rule checking performance.

Re-testing the concepts of influence bumper, segment type, area size, and hierarchical design when applied to off-line design rule checking is therefore not required. What is more important is the actual off-line checking time required to design rule check a 'real' circuit.

The circuit shown in Figure 8.14 was used for the test. As stated earlier, the circuit used eight masks, and contained around 30,000 shapes. Lastly, the set of design rules used is shown in Figure 8.12. Results of the test are as follows :-

Total time = 61 min.

N.B. This time includes 64.5 seconds required to compile the manual input file.

It would have been useful to compare the performance of MANCAD against an existing off-line design rule checker. However, no such access was available. No comments on MANCAD's performance as an off-line design rule checker can therefore be justified. It suffices to say that MANCAD's performance is of secondary importance, since its use within the CADIC is limited to a few special cases (see Section 4.3.2).

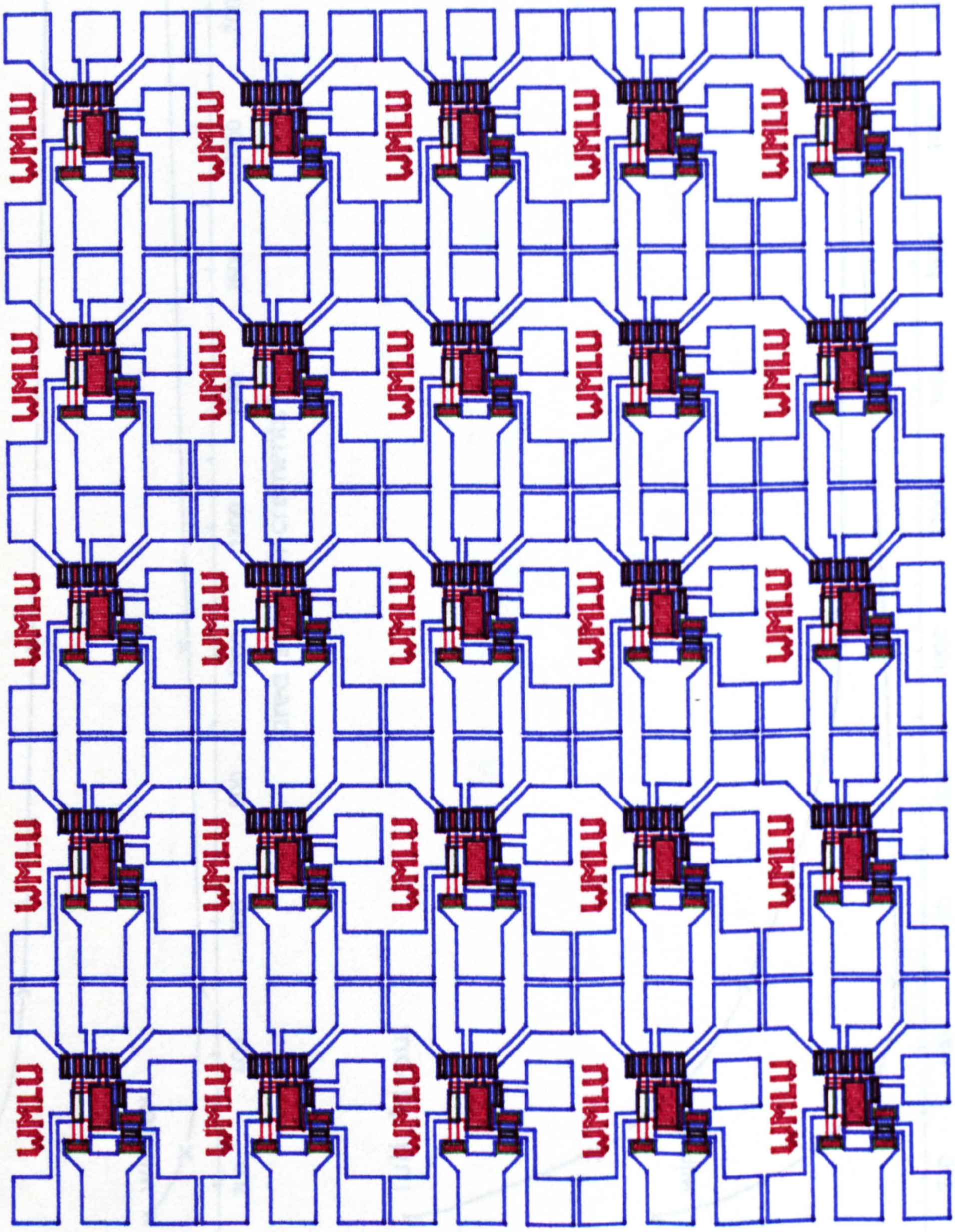
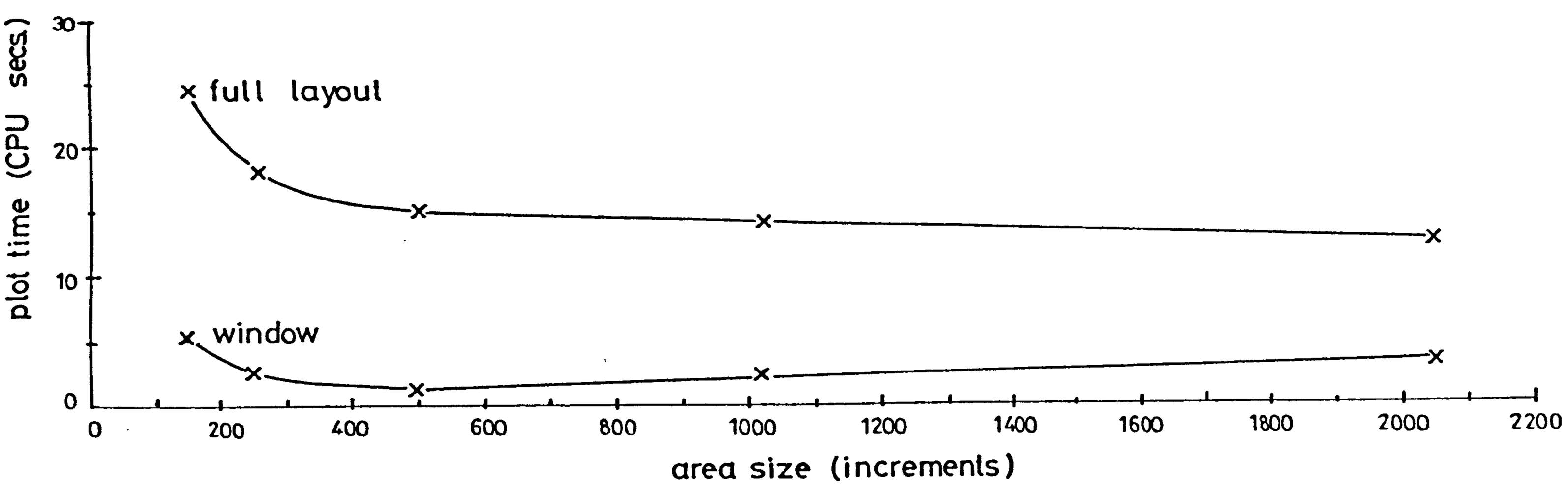


Figure 8.1 'WMLU' layout



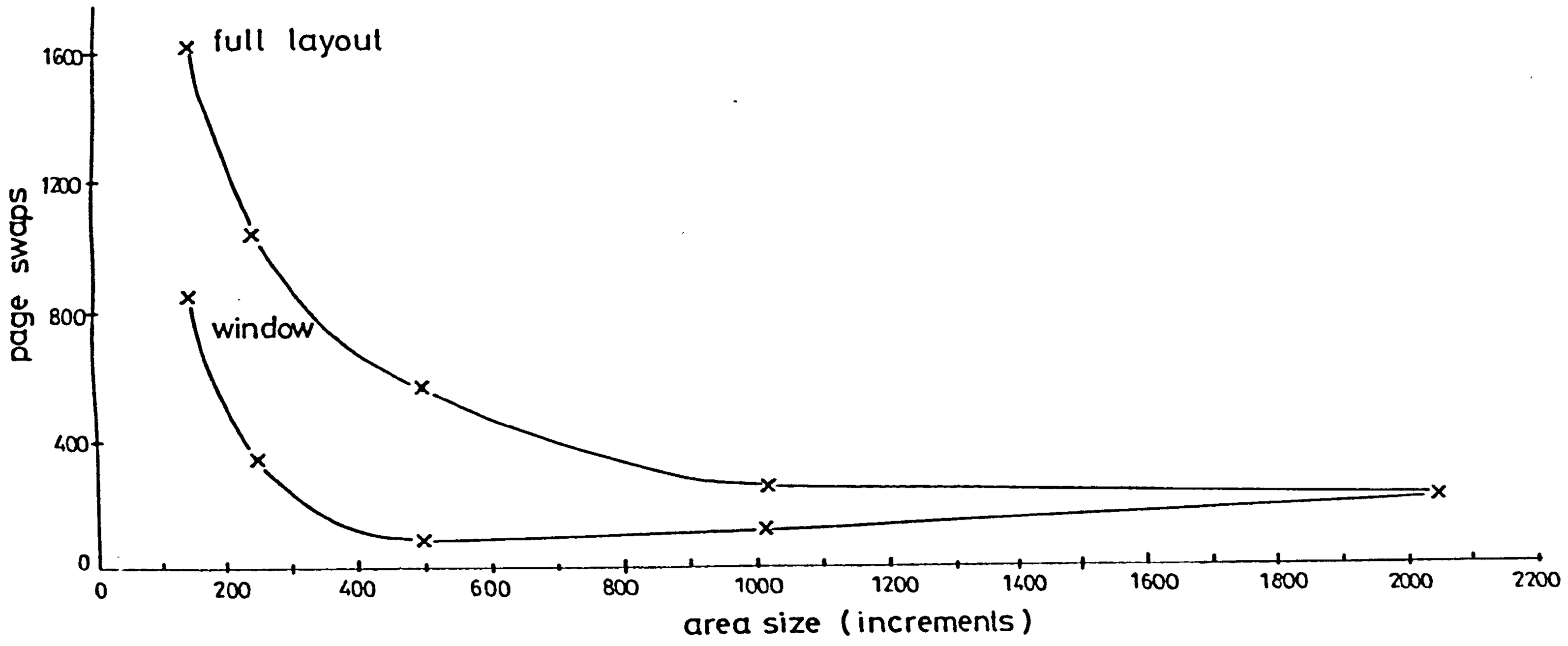


Figure 8.2 The effect of varying area size in "WMLU" circuit

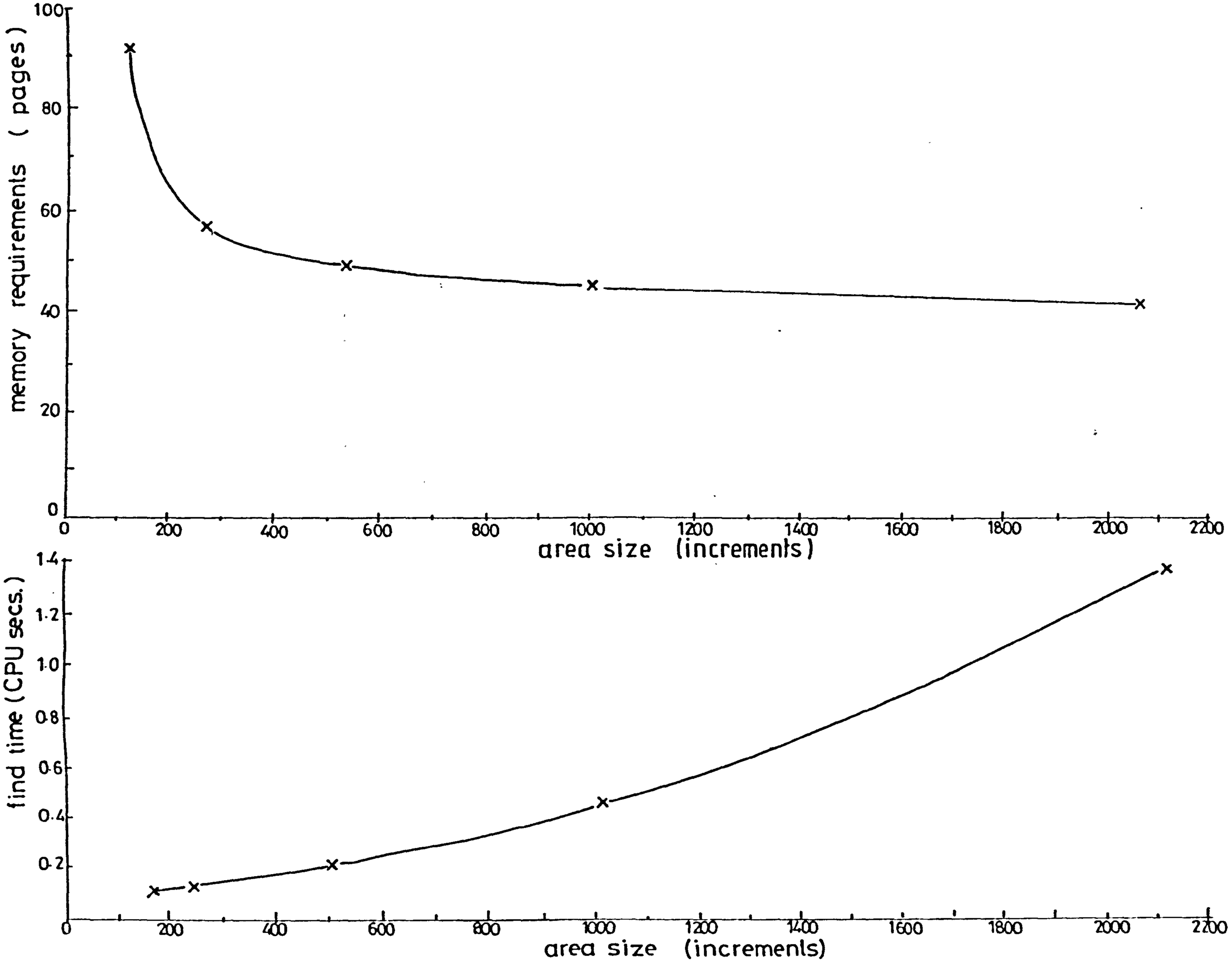

memory requirements (pages)
100
80
60
40
20
0
0
200
400
600
800
1000
1200
1400
1600
1800
2000
2200
area size (increments)
find time (CPU secs.)
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0
200
400
600
800
1000
1200
1400
1600
1800
2000
2200
area size (increments)

Figure 8.3

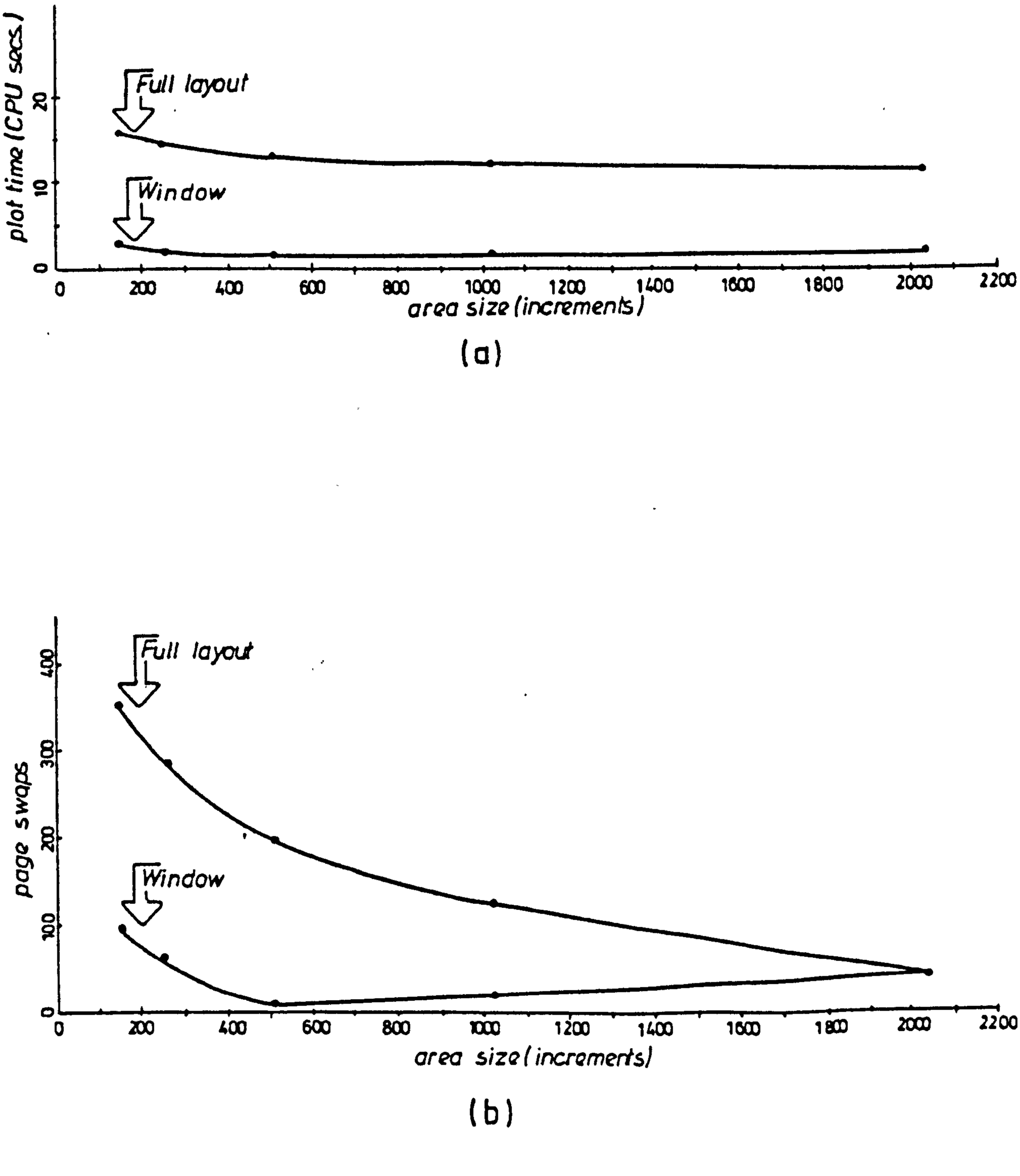


Figure 8.4 The effect of varying area size on clean "WMLU" circuit

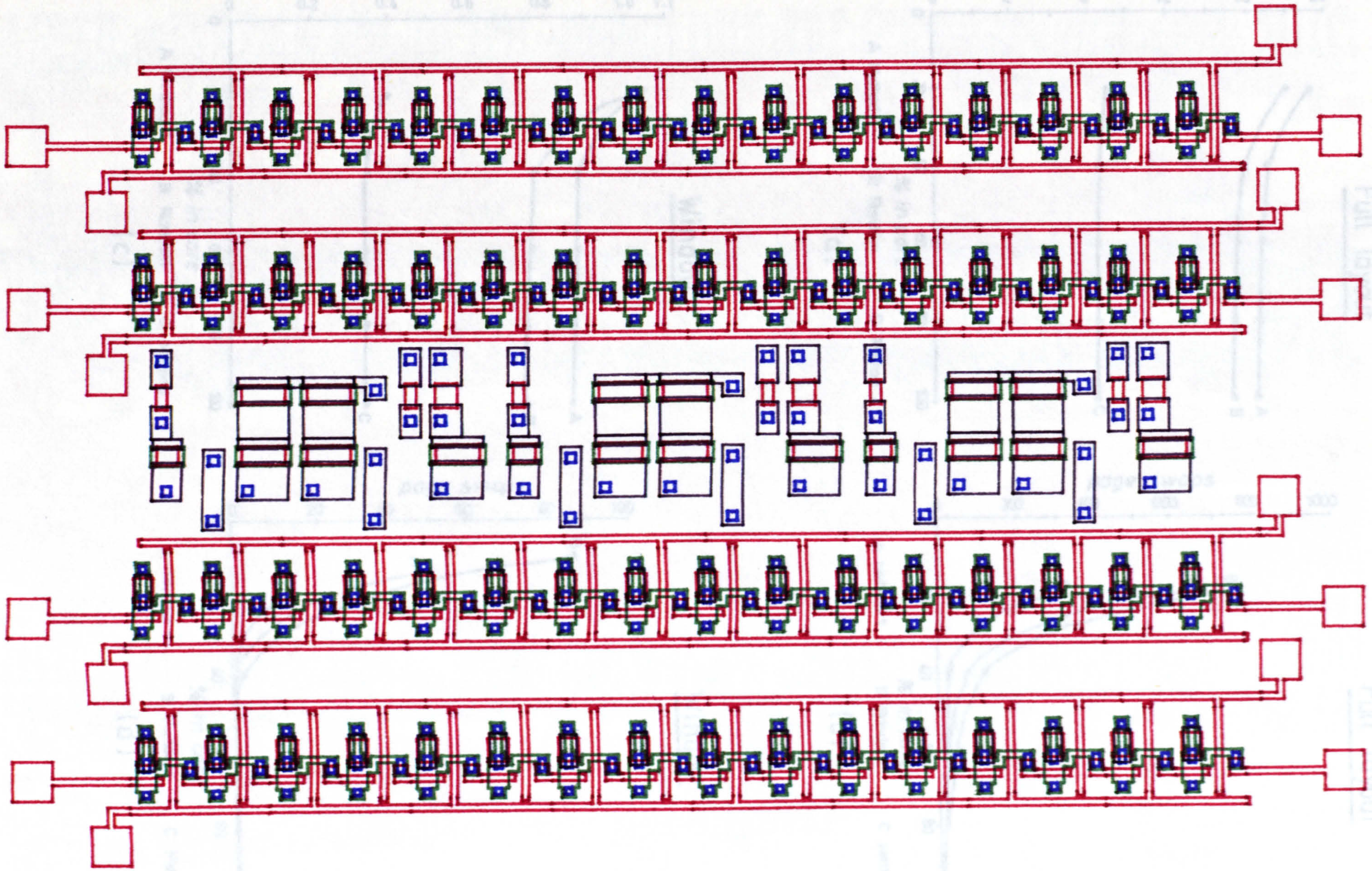


Figure 8.5 'GROUP' layout

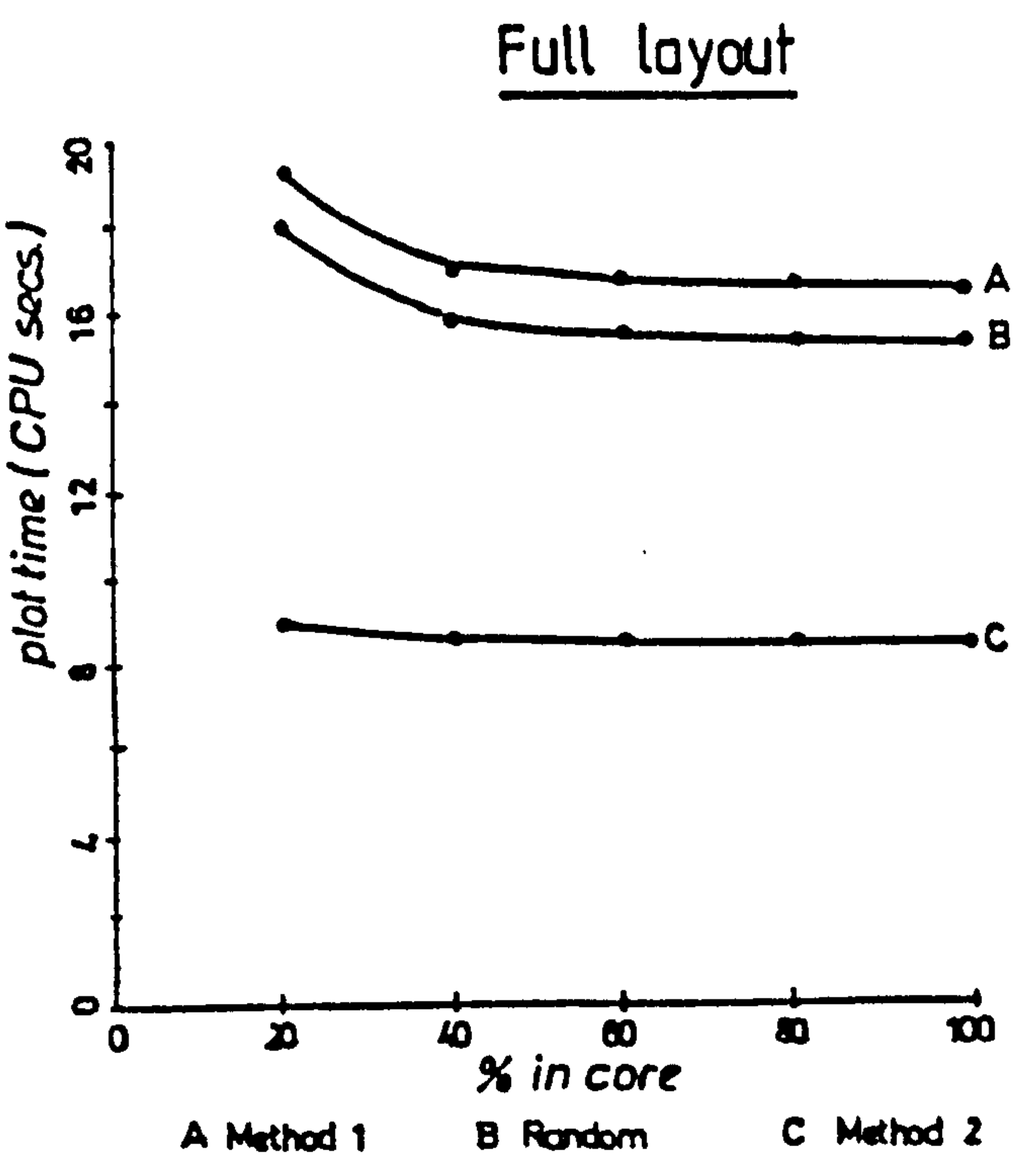


Figure 8.6 The effects of varying the amount of data structure held in core for 'GROUP' circuit

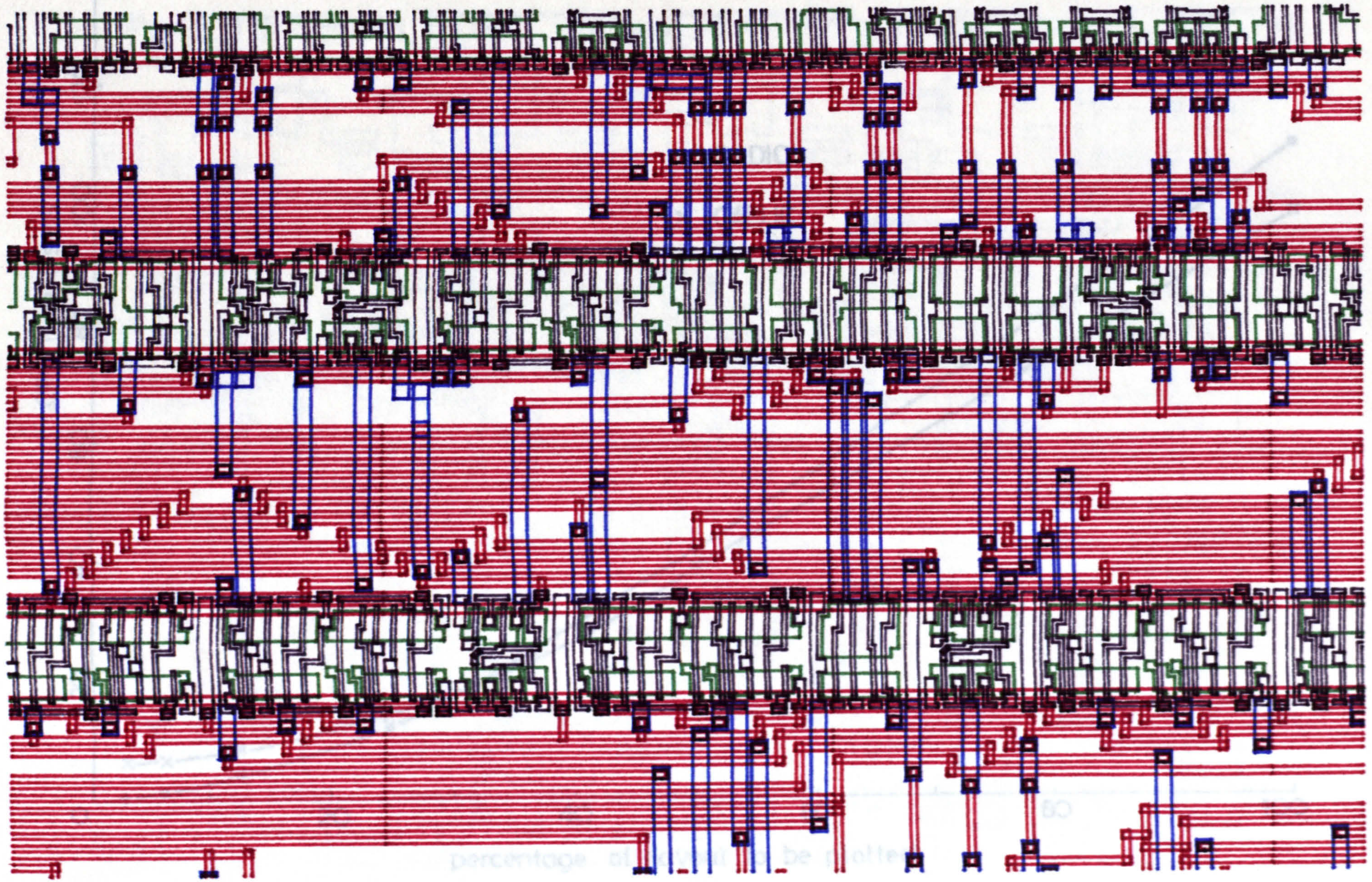


Figure 8.7 'PRIME' layout

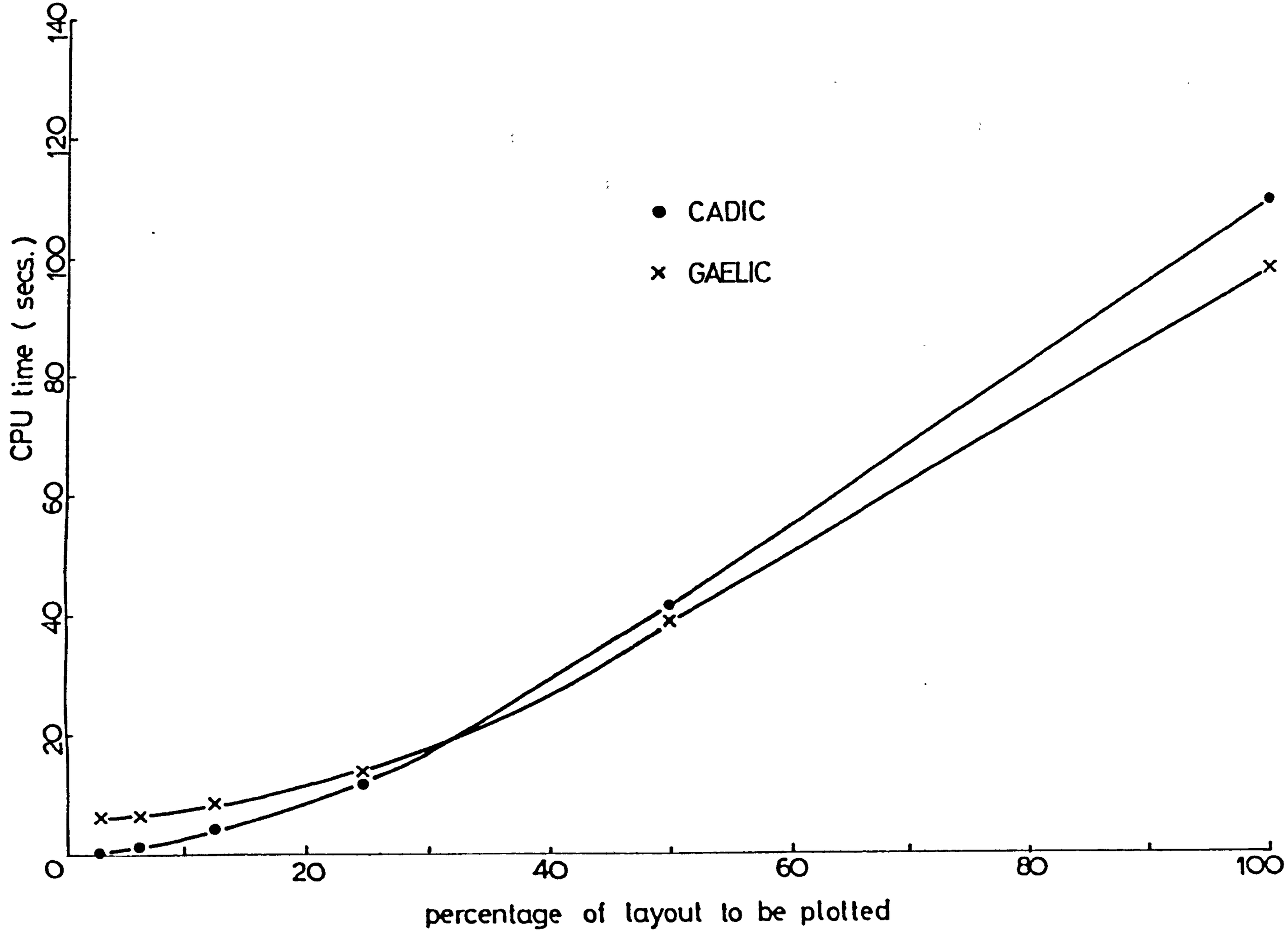


Figure 8.8 CADIC v GAELIC

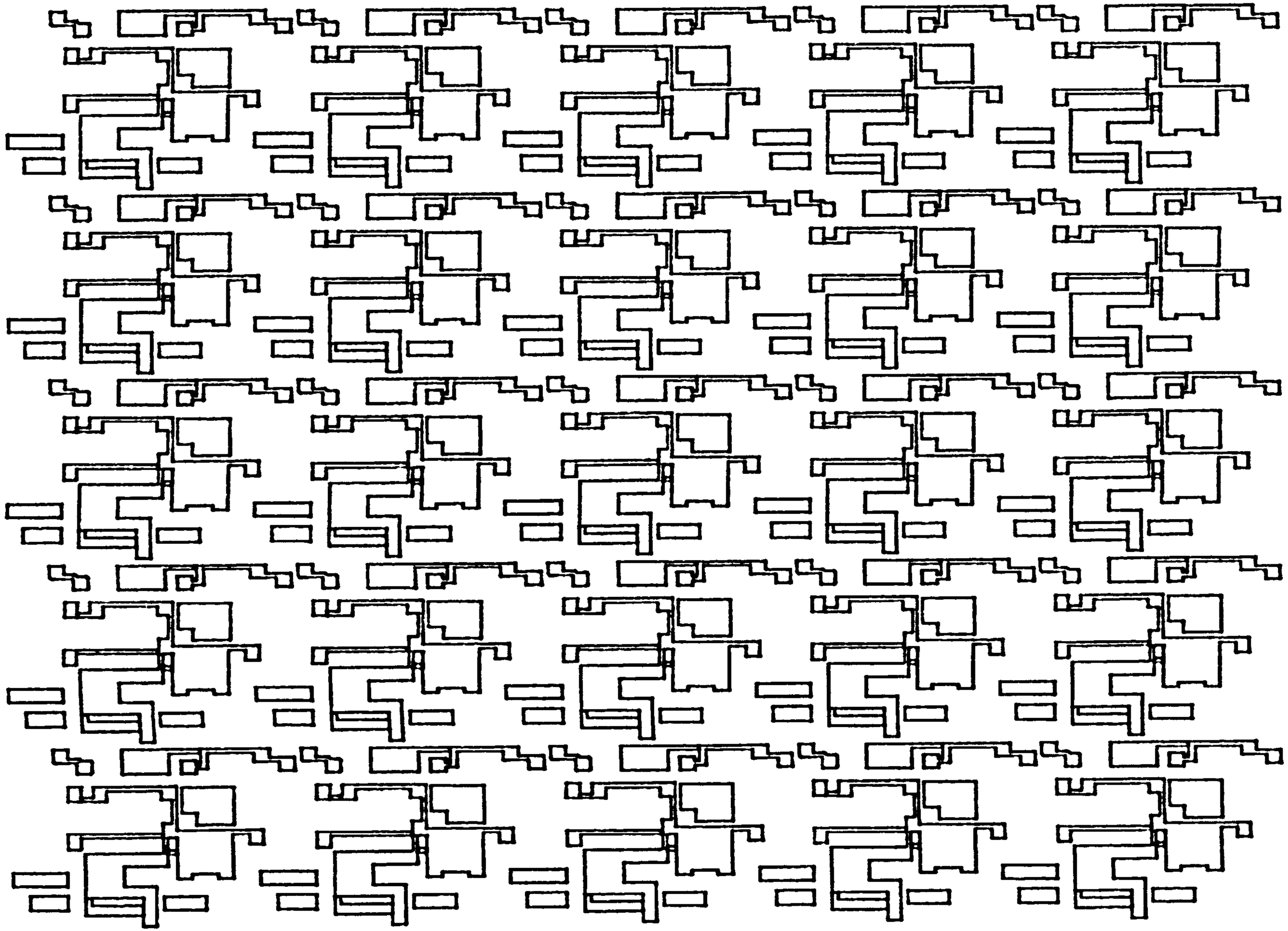


Figure 8.9 Single mask layout

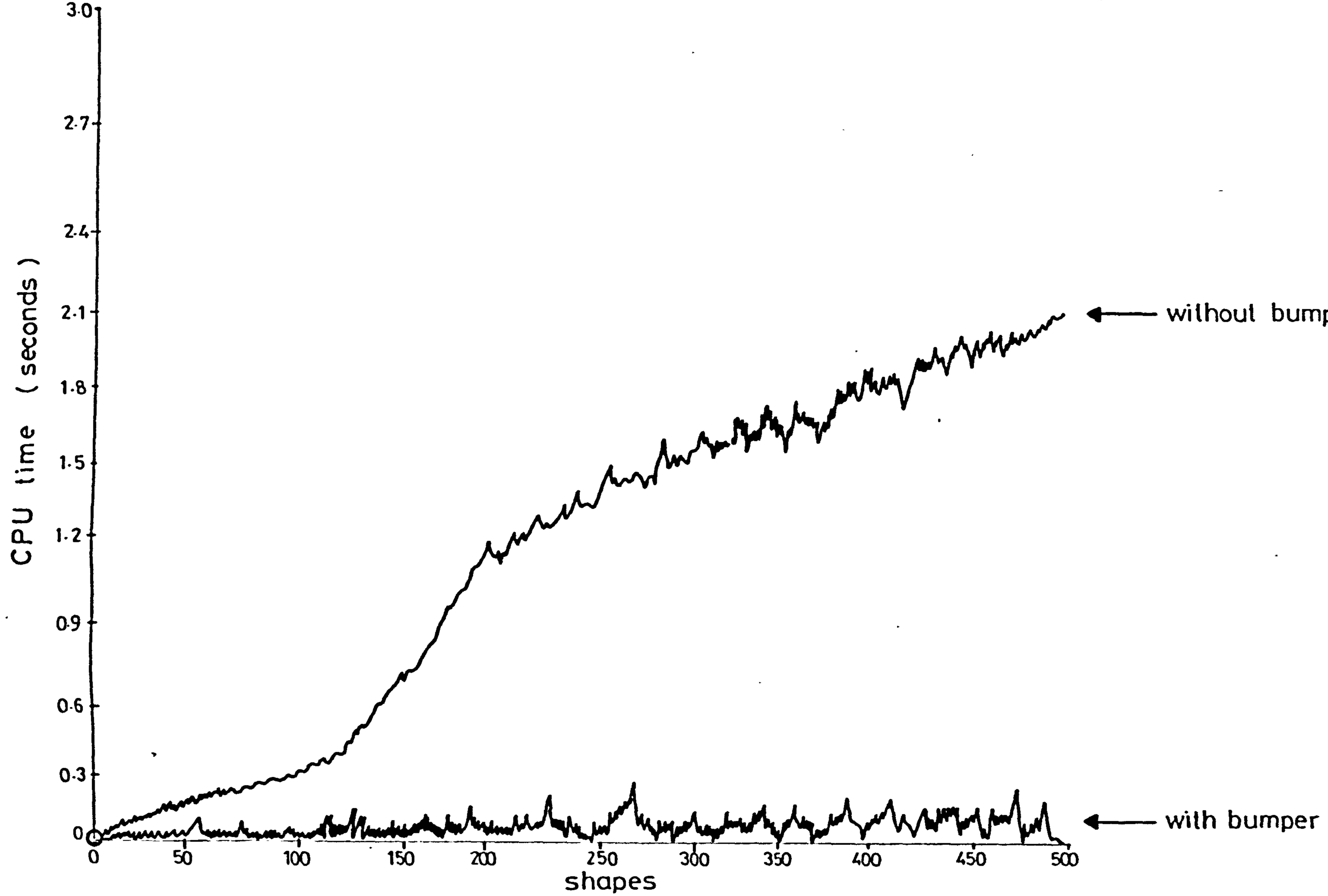


Figure 8·10 Effect of shape influence bumper

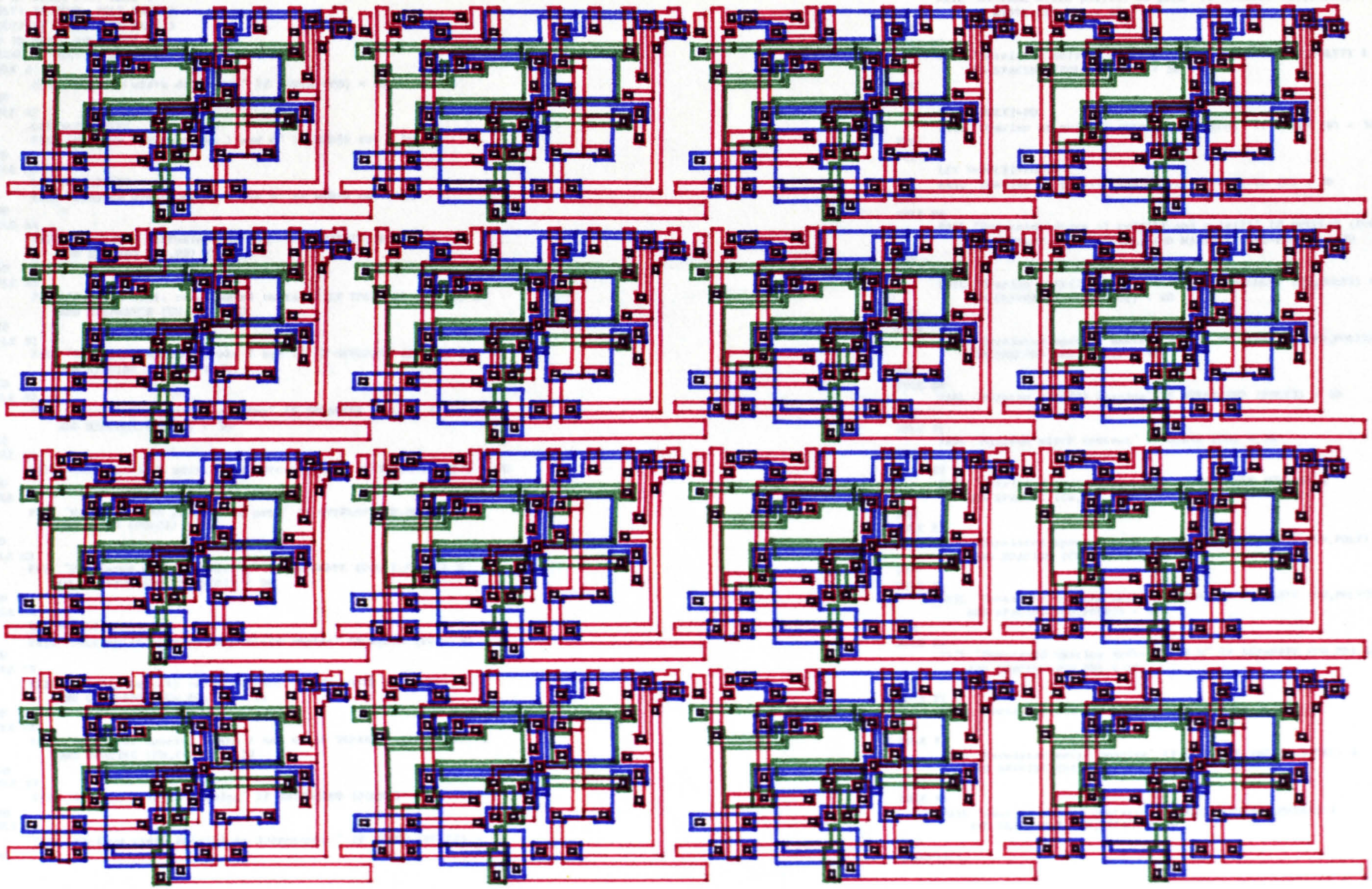


Figure 8.11 Multi-mask layout

```
PD IS RECT, POLY MASK 1
PS IS RECT, POLY MASK 2
POLY1 IS RECT, POLY MASK 4
POLY2 IS RECT, POLY MASK 5
CW IS RECT, POLY MASK 6
METAL IS RECT, POLY MASK 8
RULE A1
      FAIL 'Minimum width diffusion' IF WIDTH (PD) < 60
END
RULE A2
      LET W=PD-POLY1
      FAIL 'Minimum window masks 1 and 4' IF WIDTH (W) < 50
END
RULE A3
      LET W=PD-POLY2
      FAIL 'Minimum window masks 1 and 5' IF WIDTH (W) < 60
END
RULE A4
      FAIL 'Unrelated diffusion spacing' IF SEPARATE (PD,PD) &
          AND SPACING (PD,PD) < 70
END
RULE A5
      FAIL 'Overlap diff. round metal contact' IF ENCLOSED (CW,PD) &
          AND CLEARANCE (CW,PD) < 60
END
RULE B1
      FAIL 'Unrelated spacing masks 1 and 2' IF SEPARATE (PD,PS) &
          AND SPACING (PD,PS) < 30
END
RULE B2
      FAIL 'Unrelated depletion spacing' IF SEPARATE (PS,PS) &
          AND SPACING (PS,PS) < 50
END
RULE C1
      FAIL 'Minimum width poly(1) as interconnect' IF WIDTH (POLY1) < 50
END
RULE C2
      FAIL 'Minimum width poly(1) as gate' IF OVERLAP (PD,POLY1) &
          AND WIDTH (POLY1) < 60
END
RULE C3
      FAIL 'Unrelated poly(1) spacing' IF SEPARATE (POLY1,POLY1) &
          AND SPACING (POLY1,POLY1) < 50
END
RULE C4
      LET W=POLY1-PD
      FAIL 'Overlap of poly(1) onto field region' IF WIDTH (W) < 30
END
RULE C5
      FAIL 'Overlap poly(1) round contact' IF ENCLOSED (CW,POLY1) &
          AND CLEARANCE (CW,POLY1) < 20
END
RULE C6
      FAIL 'Unrelated spacing masks 1 and 4' IF SEPARATE (PD,POLY1) &
          AND SPACING (PD,POLY1) < 30
END
RULE C7
      FAIL 'Related poly(1) spacing' IF INTERLIMB (POLY1) < 30
END
RULE D1
      FAIL 'Minimum width poly(2) as inteconnect' IF WIDTH (POLY2) < 50
END
```

```
RULE D2
      FAIL 'Minimum width poly(2) as gate' IF OVERLAP (PD,POLY2) &
          AND WIDTH (POLY2) < 60
END
RULE D3
      FAIL 'Unrelated poly(2) spacing' IF SEPARATE (POLY2,POLY2) &
          AND SPACING (POLY2,POLY2) < 50
END
RULE D4
      LET W=POLY2-PD
      FAIL 'Overlap of poly(2) onto field region' IF WIDTH (W) < 50
END
RULE D5
      LET W=POLY1*POLY2
      FAIL 'Overlap of poly(1) and poly(2)' IF WIDTH (W) < 20
END
RULE D6
      FAIL 'Non-coincidence of poly(1) and poly(2)' IF OVERLAP (POLY2,POLY1) &
          AND OVERLAP (POLY2,METAL) AND WIDTH (POLY2-POLY1) < 30
END
RULE D7
      FAIL 'Overlap poly(2) round contact' IF ENCLOSED (CW,POLY2) &
          AND CLEARANCE (CW,POLY2) < 40
END
RULE D8
      FAIL 'Unrelated spacing masks 1 and 5' IF SEPARATE (PD,POLY2) &
          AND SPACING (PD,POLY2) < 40
END
RULE D9
      FAIL 'Related poly(2) spacing' IF INTERLIMB (POLY2) < 40
END
RULE E1
      FAIL 'Minimum width contact' IF WIDTH (CW) < 50
END
RULE E2
      FAIL 'Unrelated contact spacing' IF SEPARATE (CW,CW) &
          AND SPACING (CW,CW) < 50
END
RULE E3
      FAIL 'Unrelated spacing masks 4 and 6' IF SEPARATE (CW,POLY1) &
          AND SPACING (CW,POLY1) < 40
END
RULE E4
      FAIL 'Unrelated spacing masks 5 and 6' IF SEPARATE (CW,POLY2) &
          AND SPACING (CW,POLY2) < 60
END
RULE E5
      FAIL 'Unrelated spacing masks 1 and 6' IF SEPARATE (CW,PD) &
          AND SPACING (CW,PD) < 60
END
RULE F1
      FAIL 'Minimum metal width' IF WIDTH (METAL) < 70
END
RULE F3
      FAIL 'Unrelated metal spacing' IF SEPARATE (METAL,METAL) &
          AND SPACING (METAL,METAL) < 60
END
RULE F4
      FAIL 'Overlap around contacts' IF ENCLOSED (CW,METAL) &
          AND CLEARANCE (CW,METAL) < 20
END
ENDOFFILE
```

Figure 8.12 Design rules

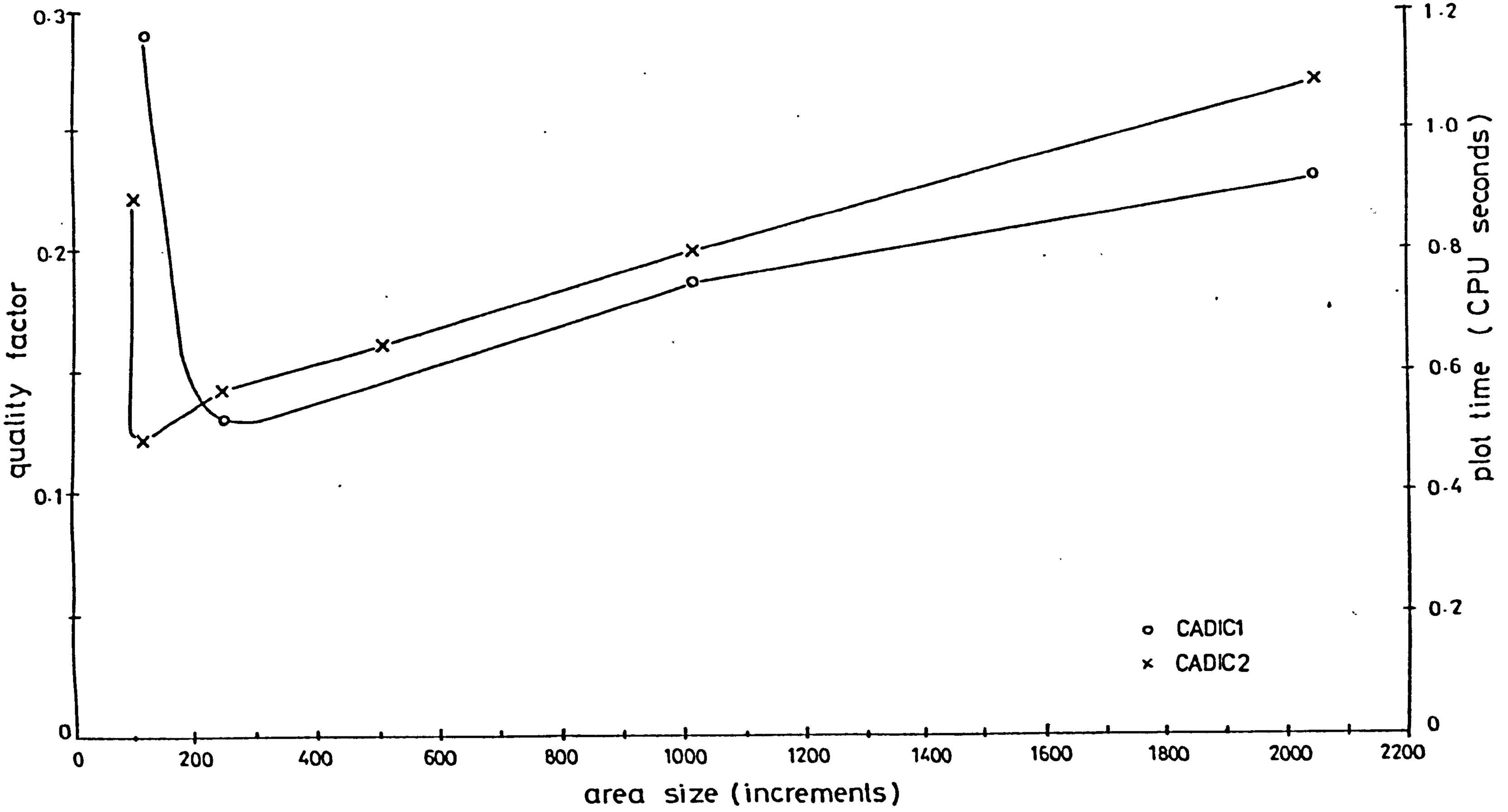


Figure 8·13 The effect of area size on CADIC 1 and CADIC 2

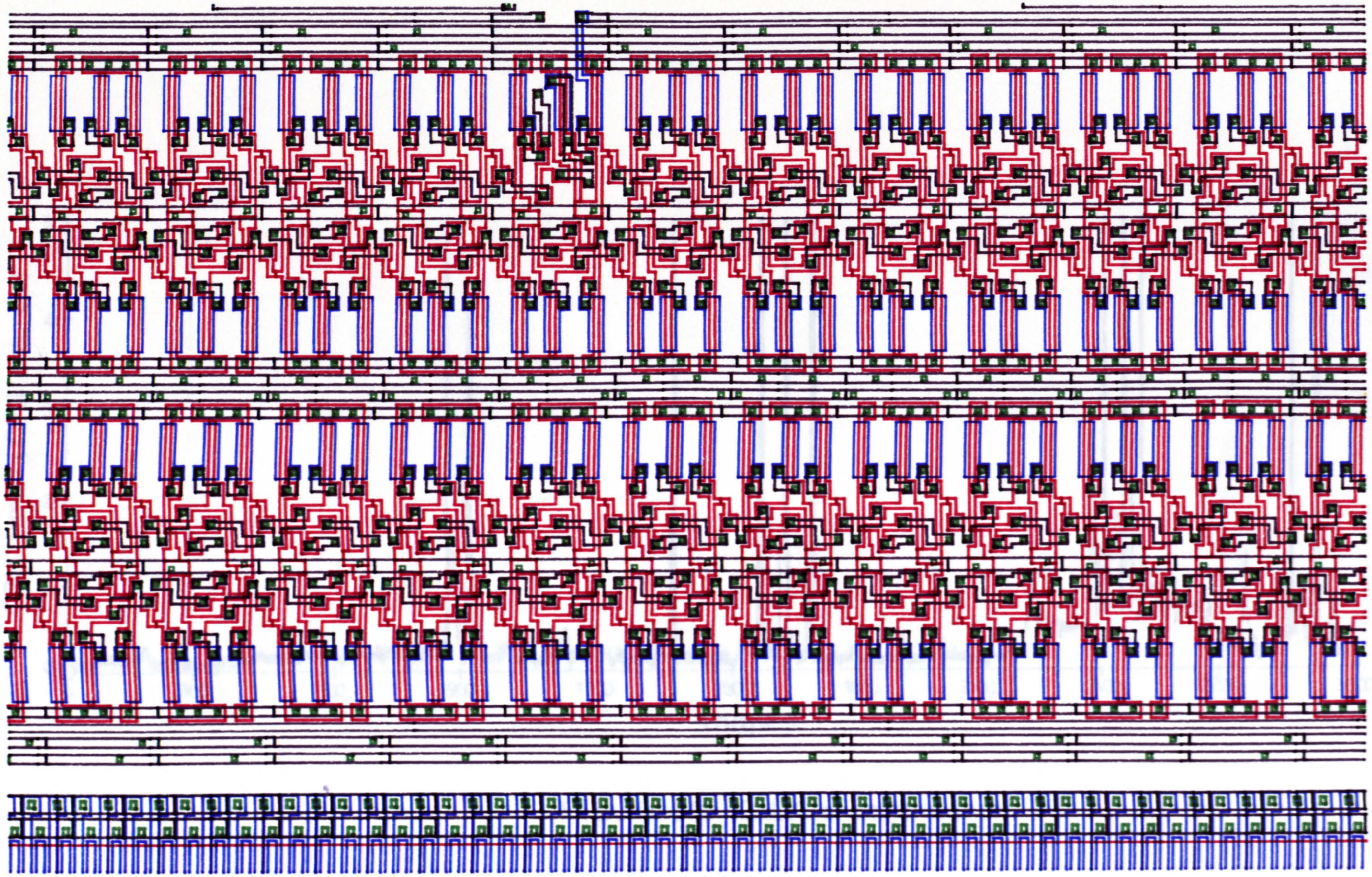


Figure 8.14 'WOLF' layout

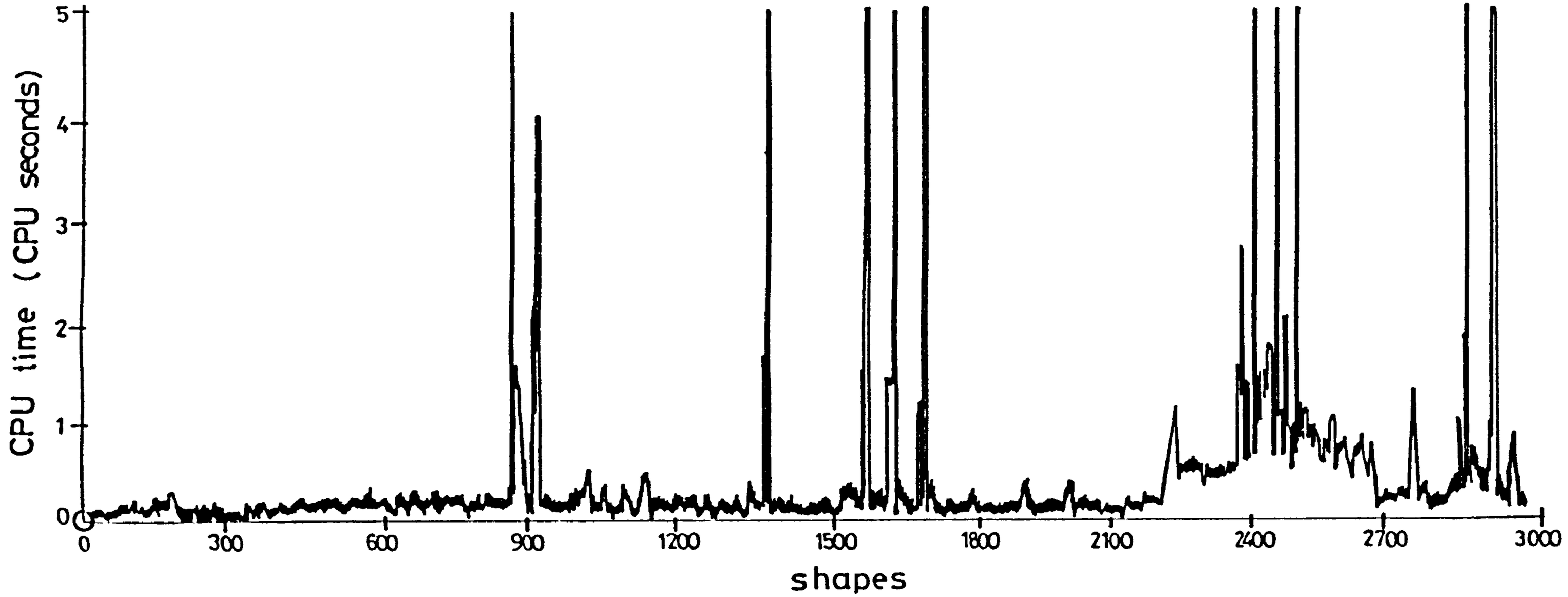


Figure 8.15 CADIC's performance

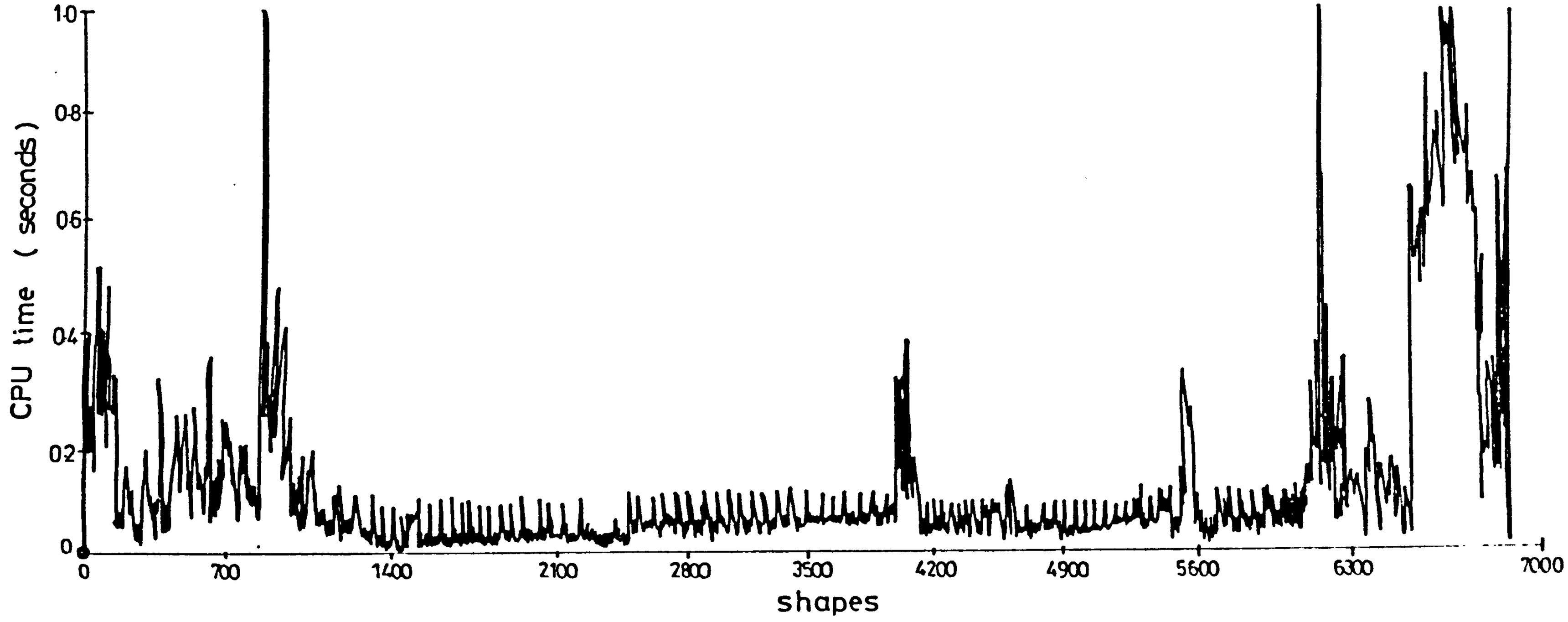


Figure 8.16 CADIC's modified performance

CHAPTER 9

Conclusions and Future Work

9.1 Overview of project

This thesis began with a brief description of the various stages required in integrated circuit production. In this way Chapter one highlights some of the problems faced by the integrated circuit manufacturer.

Chapter two discussed in detail existing computer aids developed to solve some of these problems. The advantages and disadvantages of each computer aid were considered, with the view towards developing CADIC (Computer Aided Design of Integrated Circuits), a suite of computer programs which aid integrated circuit design. The review showed that manual design aids still play a vital role in integrated circuit design. There are two main reasons for this :-

1. Manual aids are capable of producing the most compact layouts
2. Manual aids are required to produce the cells used in the automatic approach

The design turnaround time associated with manual aids is however comparatively long. New techniques to reduce this time are therefore required.

Once a layout is designed, it must be design rule checked, so that tolerance errors in the fabrication process do not affect the final circuit. At present, layouts are checked off-line. This approach is expensive in terms of time and money, due to the repetative design-check stages. Chapter two argues that on-line design rule checking would break this 'bottleneck', and allow substantial reductions in design turnaround time.

Lastly, the proposals for CADIC are justified in light of the review. The CADIC suite is split into four programs :-

1. MANCAD : Manual input language compiler
2. CADIC1 : Interactive graphic design aid
3. DRCCAD : Design rule language compiler
4. CADIC2 : On-line design rule checker

The hardware associated with a design aid can greatly affect the performance, reliability, and useability of a design system. For this reason, Chapter three gave a critical review of available hardware, and evaluated their performance when applied to integrated circuit design. As a result of the review, CADIC uses a DEC2050 time-shared mainframe computer as host, and a SIGMA 5000 micro-processor based colour raster scan terminal as a workstation.

Chapter four discussed MANCAD (MANual Computer Aided Design), a pre-processor which accepts a manual description of an integrated circuit layout, and converts this description into a data structure readable by CADIC1 and CADIC2. MANCAD can operate in two modes :-

1. Compiler
2. Off-line design rule checker

The compiler is very useful when the SIGMA workstation is not readily available. Using the manual input language, layouts, or sections of layouts can be quickly 'coded-up' on sheets of paper, then entered into MANCAD using a standard alphanumeric terminal. The SIGMA is therefore only required to view and/or edit the layout.

Occasionaly, some circuits must be design rule checked off-line. MANCAD uses the on-line design rule checking techniques developed in CADIC2 to check each shape/group call as it is compiled into the layout data structure. In this way MANCAD provides a highly efficient batch mode or off-line design rule checking facility.

Chapter five discussed CADIC1, an interactive graphic design aid which allows the user to design integrated circuit layouts at the geometric level. The most important feature of CADIC1 is its high efficiency in processing the disc-based layout data. This was made possible by implementing two new techniques :-

1. Area segmentation
2. Organised group processing

The first technique required a new form of data structure to store the layout information. CADIC1 considers the layout as divided up into a series of areas, and associates each shape with an area. Shapes which enter two or more areas are 'polygon clipped' into sub-shapes, such that each shape, or sub-shape is associated with one area. Due to a system of pointers, all shapes associated with a particular area can be found quickly, so when the designer is say, plotting out a small section of the layout, only the shapes associated with the areas inside the plotting window need be considered. This high degree of selection

greatly reduces redundant searching, and thus increases program efficiency.

The second technique involves considering the layout group hierarchy in a more global nature. If a layout is to be plotted out, all the shapes in the layout are plotted, then information about the group instances called from the layout are stored in a temporary file. Note that the group instances are not plotted out at this stage. CADIC1 then goes to the top of the temporary file, identifies the first group instance, then brings the related group definition in core. All the shapes within the group definition are then plotted out, and any group instances called from the group definition are added to the end of the temporary file.

The temporary file is then searched to see if any other instances of the group definition (presently in core) exist. If yes, then it is plotted out, and all group instances added to the temporary file. If no, then CADIC1 goes to the top of the file, and identifies a new group instance. The above process is then repeated until all group instances are plotted out. In this way, CADIC1 fully utilizes the group definitions while it is in core, and so increases program efficiency.

Chapter six goes on to discuss DRCCAD (Design Rule Compiler for Computer Aided Design) a pre-processor which accepts a 'user readable' description of the design rules, and converts this description into a 'low-level' ring data structure readable by CADIC2.

Time spent on-line design rule checking a newly added shape is critical, therefore this 'low-level' description of the design rules acts as a control file which CADIC2 can quickly access for information

on how to perform the checks. In this way, a minimum amount of time is spent accessing and decoding the rules, which leaves more time to perform the checks.

Chapter seven discussed CADIC2, the on-line design rule checker. Whenever a shape or group call is added to the layout (either using CADIC1 or MANCAD), it is CADIC2's function to design rule check the shape(s) against the existing layout. The main feature of CADIC2 is the speed in which it can complete these checks. Three factors have made this possible :-

1. The design rule data structure set up by DRCCAD always ensures that CADIC2 will perform the minimum number of operations during design rule checking.
2. The layout ring data structure is very efficient in finding information about shapes local to the newly added shape
3. Each routine in CADIC2 has been optimised such that the CPU time required to complete each operation is kept to a minimum

Finally, Chapter eight discussed the performance of each program in the CADIC suite, with emphasis on CADIC1 and CADIC2, the most important programs in the suite. Logistics previously suggested for each program were experimentally tested, and optimal working conditions identified. The results of the tests confirmed three main points :-

1. MANCAD and DRCCAD performed as expected for the type of processing being carried out.
2. CADIC1 is very efficient at data processing, especially when small sections of layout are considered.
3. CADIC2 can perform complete on-line design rule checking within the time it takes the designer to start adding the next shape.

9.2 Possible improvements

In general, the CADIC suite of programs have performed very well in achieving all the original aims of this project. However, with the benefit of hinesight, certain weakpoints in CADIC have been identified. The purpose of this section is therefore to discuss these weakpoints, and suggest possible improvements.

MANCAD : Two main areas in this program could be improved. These are :-

1. Manual input language
2. Off-line design rule checking violation details

For reasons described in Section 4.2, the MANCAD manual input language was made compatible with the GAELIC manual input language, except for the commands; "LINE", "CIRCLE", and "TEXT". The first improvement to the .MANCAD language would be to update MANCAD so as to accept these un-used commands, even though CADIC does not truly support them. This could be achieved as follows :-

1. Accept the "LINE" command, then automatically add a terminating dark segment, so that the line becomes a closed polygon. CADIC can then handle the polygon, even though it still appears as a line in the layout.
2. A circle defined by the "CIRCLE" command could be automatically transformed into a multi-segment polygon, which would approximate to the circle.
3. The SIGMA workstation has the facility to plot out text on the screen, but CADIC has no way of storing the information in the layout data structure. Therefore the best MANCAD could do with the "TEXT" command is accept it, but do nothing with it.

In the future, there is no reason why the MANCAD language has to remain identical to the GAELIC language. Indeed any new language would do well to copy many of the features and constructs available in the GAELIC language, yet certain modifications could be incorporated into the new language to enhance language ergonomics and minimise the amount of data to be entered. Consider an example of a typical section of a GAELIC manual input file :-

```
"NEWGR" GRP1;
    "RECT" (1) 1250,4520:180,740;
    "POLY" (1) L,840,3500:140,0,0,690,100,0,0,80,-240,0,0,-770;
    "POLY" (1) S,5310,100:870,-100,150,120,-80,50,-940,-70;
    "POLY" (1) S,5310,310:940,50,80,120,-150,-100,-870,-70;
    "RECT" (1) 1670,3040:60,120;
    "RECT" (1) 910,2660:180,560;
    "RECT" (2) 1870,3210:120,1120;
    "POLY" (2) S,2930,3100:270,870,100,150,-120,-90,-50,-930;
    "TRAC" (2) 60,L,3440,270:-330,0,-50,50,-300,0,-80,80,-140,0,-40,40;
    "TRAC" (2) 60,L,2670,750:350,0,30,-30,270,0,80,-80,180,0,30,-30;
    "TRAC" (2) 60,S,2930,3100:270,870,100,150;
"ENDGR";
"FINISH";
```

The most fundamental modification that can be made is the removal of the double quotes round each command word. In the original specification for GAELIC, the designer was allowed to attach labels to specific shapes possibly for use by a future functional verification program. The label was entered after the command word, and was separated by an oblique, for example :-

```
"RECT/INPUT1" (1) 1250,4520:180,740;
```

The label could vary in length, therefore quotes were required to delimit the label. Unfortunately, a use for the label information never materialized, therefore GAELIC no longer supports the label option. For this reason, the quotes are redundant in the manual input language.

When building up a layout using the manual input language, the designer tends to enter the information a mask at a time. This is different to interactive design where the designer is likely to swap frequently between masks, as the various elements are added to the layout. The authors of GAELIC did not foresee this difference, and so included the mask information into each shape command, so as to facilitate frequent mask changes. As can be seen by the example above, rather than include the mask information each time a shape is defined, it would be better to remove the mask information, and define a new command :-

MASK <masknum>

which would set the mask number to 'masknum'. All shapes that follow the MASK command would then be placed on mask 'masknum' until another MASK command is identified, for example :-

```
MASK 1;
    RECT  1250,4520:180,740;
    RECT  1670,3040:60,120;
MASK 2;
    POLY  S,2930,3100:270,870,100,150,-120,-90,-50,-930;
FINISH;
```

In a similar way, the track width information could be removed from the TRACK command, and a new command ; WIDTH <trackwidth> defined, for example :-

```
MASK 1;
WIDTH 60;
      TRAC  L,3440,270:-330,0,-50,50,-300,0,-80,80,-140,0,-40,40;
      TRAC  L,2670,750:350,0,30,-30,270,0,80,-80,180,0,30,-30;
FINISH;
```

In this way, the amount of information to be entered can once again be reduced.

Standardisation of the shape command construction is a very important step towards making a language easier to use. At present the RECTANGLE, POLYGON and TRACK commands all have different constructs within the GAELIC language. Defining the WIDTH command changes the TRACK construction to be the same as the POLYGON construction, which helps the aim of standardisation. The remaining difference is the format specification for the POLYGON command which may be 'L' (Long format) or 'S' (Short format). It is also very easy to forget to enter the format letter into the command, therefore it would be better if the manual input language defined unique commands to handle long and short format shapes, for example :-

```
MASK 1;
WIDTH 10;
      RECT    1250,4520:180,740;
      LPOLY   840,3500:140,0,0,690,100,0,0,80,-240,0,0,-770;
      SPOLY   5310,100:870,-100,150,120,-80,50,-940,-70;
      LTRAC   3440,270:-330,0,-50,50,-300,0,-80,80,-140,0,-40,40;
      STRAC   2930,3100:270,870,100,150;
FINISH;
```

Note that the construction for each shape command is now identical :-

<COMMAND> <origin>:<incremental coordinates>;

If the above mentioned modifications were incorporated into the MANCAD input language, the original example would be entered as :-

```
NEWGR GRP1;
  MASK 1;
      RECT    1250,4520:180,740;
      LPOLY   840,3500:140,0,0,690,100,0,0,80,-240,0,0,-770;
      SPOLY   5310,100:870,-100,150,120,-80,50,-940,-70;
      SPOLY   5310,310:940,50,80,120,-150,-100,-870,-70;
      RECT    1670,3040:60,120;
      RECT    910,2660:180,560;
  MASK 2;
      RECT    1870,3210:120,1120;
      SPOLY   2930,3100:270,870,100,150,-120,-90,-50,-930;
  WIDTH 60;
      LTRAC   3440,270:-330,0,-50,50,-300,0,-80,80,-140,0,-40,40;
      LTRAC   2670,750:350,0,30,-30,270,0,80,-80,180,0,30,-30;
      STRAC   2930,3100:270,870,100,150;
ENDGR;
FINISH;
```

The second area in which MANCAD could be improved is concerned with its handling of off-line design rule checking violations. At present, if a shape in the manual input file violates any design rules, the violation message is printed out, followed by the line in the manual input file corresponding to the failed shape, along with the name of the group definition containing the shape (See Section 4.3.2). In this way, the designer can use the list of error messages to identify the shapes in the layout, then use CADIC1 to edit them as required.

Even though this approach works well, it would be better if the shapes which cause a violation were also stored in a plot file, along with the other shape(s) involved in the violation. On plotting out this file, the designer will find it easier to locate the erroneous shapes. This approach was not implemented by MANCAD for two reasons :-

1. The plot is not essential, it only makes the erroneous shapes easier to find.
2. The development time was not available.

CADIC1 : At present, it is felt that CADIC1 has a consise range of commands which covers certainely the most common requirements in layout design. However, as with any design aid, someone will want it to perform a function that is not available. In the future, more commands can therefore be added to CADIC1, limited only by the number of keys on the keyboard.

The second improvement to CADIC1 requires a more sophisticated method of paging the layout data in and out of computer memory. The main problem faced by a paging routine is which one of the pages presently in core must be swapped out, to allow a new page to enter.

Obviously, the aim is to keep the important pages in memory for as long as possible, so as to prevent page thrashing. CADIC1 uses a paging routine which removes the oldest page in the computer's memory. This technique suited CADIC1 very well, but took no account of any special features in the format of the layout ring data structure.

During processing it is important for CADIC1 to keep the area beads in memory as much as possible, as they act as the first 'filter' in the task of selecting relevant data and are frequently accessed. On the other hand, a shape bead usually is required only once (i.e. while being plotted out) yet when using the paging routine, an area bead has as much chance of staying in core as the shape bead. It would be better if the paging routine could sub-divide its memory allocation of six pages into say two pages for area and mask beads, and four pages for shape and group call beads, then page each sub-division independently of each other. In this way, the useful area and mask information would not be paged out just because CADIC1 had to process a large number of shapes.

To stand any chance of competing with the present paging routine, the new routine must also be written in machine code. The author has no expertise in this area, therefore the concept was never pursued.

DRCCAD : The only possible improvements to DRCCAD are concerned with the input language used to describe the design rules. Consider an example of the manual input language :-

```
PD IS RECT, POLY MASK 1
PS IS RECT, POLY MASK 2
POLY1 IS RECT, POLY MASK 4
POLY2 IS RECT, POLY MASK 5
CW IS RECT, POLY MASK 6
METAL IS RECT, POLY MASK 8
RULE A1
      FAIL 'Minimum width diffusion' IF WIDTH (PD) < 60
END
RULE A2
      FAIL 'Unrelated spacing masks 1 and 2' IF SEPARATE (PD,PS) &
          AND SPACING (PD,PS) < 30
END
RULE A3
      FAIL 'Overlap poly(1) round contact' IF ENCLOSED (CW,POLY1) &
          AND CLEARANCE (CW,POLY1) < 20
END
RULE A4
      FAIL 'Non-coincidence of poly1/poly2' IF OVERLAP (POLY2,POLY1) &
          AND OVERLAP (POLY2,METAL) AND WIDTH (POLY2-POLY1) < 30
END
ENDOFFILE
```

The first point to note is that the RULE and END commands serve no useful purpose, and only increase the amount of data to be entered by the user. If a violation occurs, the violation message gives ample information about which design rule failed.

Another improvement that could be made is concerned with the SPACING and CLEARANCE commands. At present, these commands must be preceded by the commands SEPARATE and ENCLOSED respectively. It is obvious however that the SPACING stipulation has no meaning if the shapes are not separate. Similarly the CLEARANCE stipulation loses relevance if one shape does not enclose another. The SEPARATE and ENCLOSED commands should therefore be implicitly accepted if the SPACING and CLEARANCE commands are used. Implementing these modifications would mean that the example shown above could be entered as :-

```
PD IS RECT, POLY MASK 1
PS IS RECT, POLY MASK 2
POLY1 IS RECT, POLY MASK 4
POLY2 IS RECT, POLY MASK 5
CW IS RECT, POLY MASK 6
METAL IS RECT, POLY MASK 8

     FAIL 'Minimum width diffusion' IF WIDTH (PD) < 60

     FAIL 'Unrelated spacing masks 1 and 2' IF SPACING (PD,PS) < 30

     FAIL 'Overlap poly(1) round contact' IF CLEARANCE (CW,POLY1) < 20

     FAIL 'Non-coincidence of poly1/poly2' IF OVERLAP (POLY2,POLY1) &
          AND OVERLAP (POLY2,METAL) AND WIDTH (POLY2-POLY1) < 30

ENDOFFILE
```

CADIC2 : Now that CADIC2 has been shown to work well for the subset of commands presently available in the design rule language, the first improvement to CADIC2 would be to update the program to handle the new commands. Note that CADIC2 is written in a highly modular fashion, therefore adding routines to perform each new operation in no way affects the existing software. The commands not yet handled are as follows :-

ABUTS ..... Find shapes which touch

DISTINCT ..... Find shapes which are distinct

PARTED ..... Find shapes, one cut in two by the other

LENGTH ..... Specify minimum length of shape

XDIM ..... Specify minimum X dimension of shape

YDIM ..... Specify minimum Y dimension of shape

BRAREA ..... Specify min. area of shape's bounding rectangle

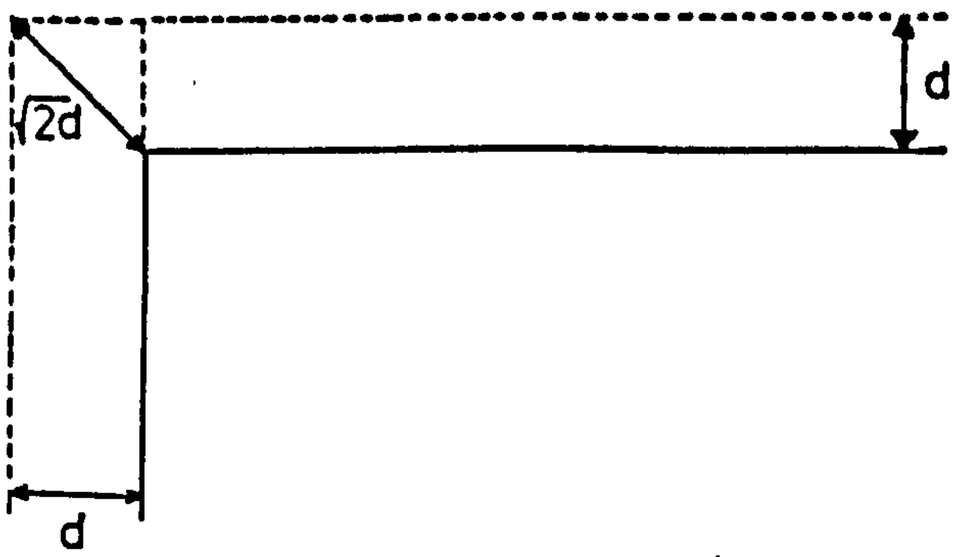
HORIZONTAL ..... Specify shape to lie in horizontal direction

VERTICAL ..... Specify shape to lie in vertical direction

NOT ..... Inverting command

INFLATE/DEFLATE .. Inflate/deflate shape

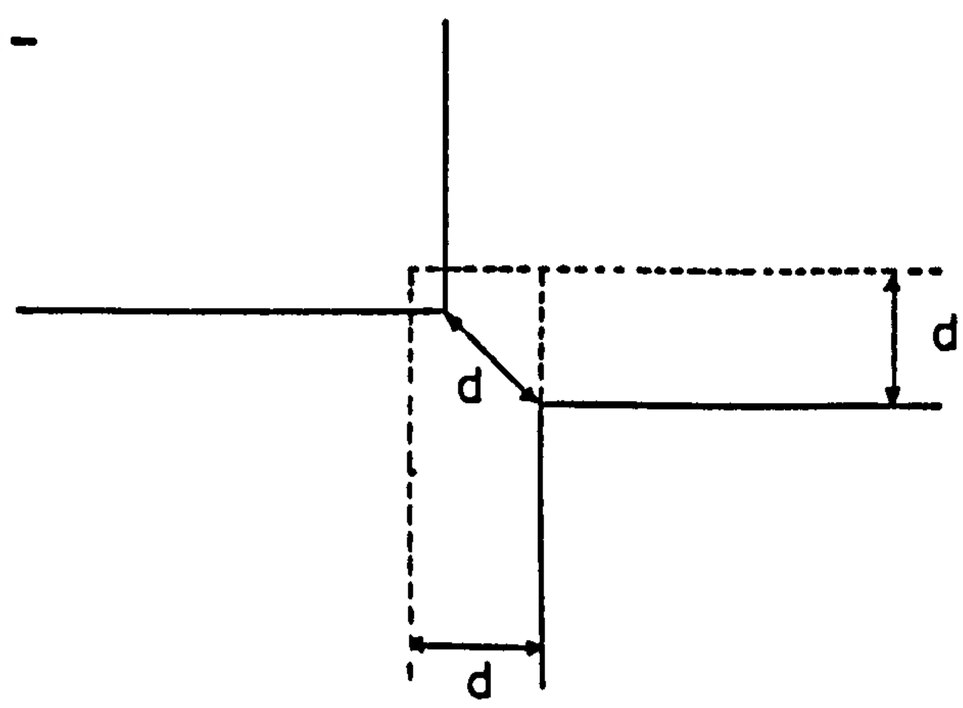
The second improvement that can be made to CADIC2 is to use euclidian bumpers during dimensional checks. At present all bumpers are orthogonal, for example :-



key

d = minimum spacing

Correct width of bumpers is observed at all points except at the corners of shapes, where the width may reach a maximum of '$\sqrt{2}$d' units. Under certain conditions, this over-expansion can cause false violations to be generated :-

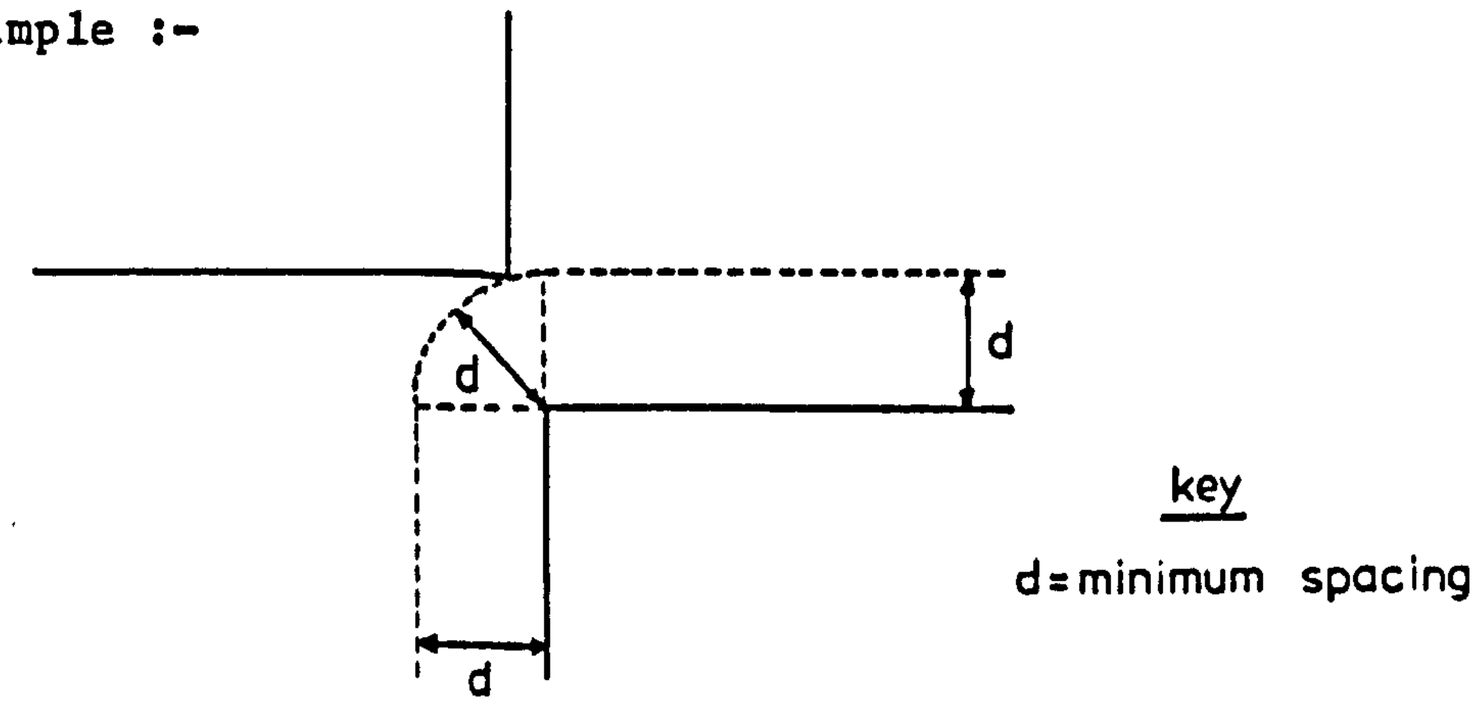


key

d = minimum spacing

Note that most existing design rule checking programs use some sort of orthogonal distance test during dimensional check, and nearly all produce false errors for the above mentioned reason. The justification for using the orthogonal approach is that it is very easy, and fast to implement.

The only way to remove the false violations is to use euclidian bumpers. In this approach, the area of influence round a corner is described by the arc of a circle with radius 'd', and centre at the corner point, for example :-



Now the bumper is guaranteed to be of width 'd' for all conditions. The euclidian approach however requires much more CPU time to implement. Other design rule checking programs must represent the arcs as a series of straight segments, therefore the large increase in the number of segments per shape forces about an order increase in the CPU time required to perform the design rule checks.

Although more expensive than using orthogonal bumpers, CADIC2 could use euclidian bumpers without forcing such a large increase in CPU time. This is because the reasoning behind the use of bumpers applies to any type of bumper, whether it be rectangular or circular. If euclidian bumpers were used, CADIC2 only has to decide on which type of bumper to create, then use the relevant 'clipping' algorithm to check if any segments enter it.

9.3 Future work

CADIC has been discussed, and possible improvements to the existing design aid proposed. There are however many other programs that could be incorporated into CADIC to enhance its use in integrated circuit design. The most important of these are discussed below.

Automatic design : In the future, as the complexity of integrated circuits increases from VLSI to WSI (Wafer Scale Integration), it is envisaged that manual aids will slowly be phased out of whole layout design, in favour of automatic design aids. This has the problem of creating layouts that will be larger than necessary, but it is felt that manual design will prove to be too expensive to implement at the circuit level. Note that manual aids will however still be used to design the cell library used by automatic aids.

The CADIC suite would therefore benefit from a program which could automatically place and route a cell layout. In this way, the designer could switch between manual and automatic aids, to acheive the optimal layout design.

In many respects, the importance of manual design aids will not be greatly affected by this swing from manual to automatic design aid. Since the manual design aid will only be working with comparatively small sections of layout, new features can be incorporated into the interactive design aid, which would otherwise not be feasible, due to excessive CPU time requirements. These features such as automatic layout adjustment and on-line functional verification are discussed below.

Automatic layout adjustment : This would be a useful extension to the on-line design rule checking facility available in CADIC. Instead of simply warning the designer that a shape has violated the design rules, it would be better if CADIC could automatically shift the violating shape, such that the design rules were in fact satisfied.

The problem of course may not be limited to moving just the newly added shape. More than likely, the adjustment will cause a 'knock-on' effect which may cause a combinatorial explosion within the layout. Therefore this technique will almost definitely have to be restricted to small sections of layout, before it can be feasibly considered for use in an interactive environment.

On-line functional verification : Design rule checking ensures that the layout is geometrically correct, but will not ensure that the circuit will operate correctly. This is the job of the functional tester. At present, the functional checks are performed off-line, and so create a 'bottleneck' in the design process, just as described for off-line design rule checking (See Chapter two).

The problem with on-line functional verification is how to include it into an interactive environment. With design rule checking, the rules were specified at the beginning of the design, and applied to all group definitions produced thereafter. Functional verification is not so simple, since a different functional description would have to be supplied for each group definition.

A useful technique would be to constrain the function of each group definition to be any one of a pre-defined library of elements. On entering a group definition, the user would specify the type of element

he was going to develop. On completion, CADIC could check the group definition to ensure that the correct element was in fact produced. In this way the information to be entered by the user is limited to at most a few words, and the computer need only check for a particular type of element, rather than having to guess the function of the element by itself.

REFERENCES

1. Eades J D
"The design of an interactive computer system for microelectronic mask making"
Phd Thesis 1976

2. Smith T F, and Woods B J
"Poligon : An interactive graphics design tool"
Computer Aided Design Conference 1980 pp 31 - 37

3. Infante B, Bracken D, McCalla B, Yamakoshi S, and Cohen E
"An interactive graphics system for the design of integrated circuits"
15th Design Automation Conference 1981 pp 182 - 187

4. Fairbairn D G
"ICARUS : An interactive integrated circuit layout program"
15th Design Automation Conference 1981 pp 188 - 192

5. Franco D, and Reed L
"The cell design system"
18th Design Automation Conference 1981 pp 240 - 247

6. Carmody P, Barone A, Morrell J, Weiner A, and Hennesy J
"An interactive graphics system for custom design"
17th Design Automation Conference 1980 pp 430 - 439

7. Williams J D
"STICKS : A graphical compiler for high level LSI design"
National Computer Conference 1978 pp 289 - 295

8. Clary D, Kirk R, and Sapiro S
"SIDS : An interactive colour graphics system for the symbolic layout and checking of MOS IC's"
Enro-Graphics Conference 1979 pp 317 - 328

9. Richardson F K et al.
"An interactive graphical system for the design of photomasks"
N.E. Electronic Research and Engineering 1970 pp 182 - 183

10. Dunlop A E
"SLIP : Symbolic layout of integrated circuits with compaction"
CAD publication Vol 10 No. 6 1978 pp 387 - 391

11. Hsueh M Y
"Symbolic layout compaction"
NATO Advanced Study on Computer Design Aids
for VLSI circuits 1980

12. Weste N
"Virtual grid symbolic layout"
18th Design Automation Conference 1981 pp 225 - 233

13. Cho Y E, Korenjak A J, and Stockton D E
"Floss - An approach to automated layout for high-volume designs"
14th Design Automation Conference 1977 pp 138 - 141

14. Kozawa T, Horino H, Ishiga T, Sakemi J, and Sato S
"Advanced LILAC - An automated layout generation system for MOS/LSI's"
11th Design Automation Conference 1974 pp 26 - 46

15. Beke H, and Sansen W
"CALMOS : A portable software system for the automatic and interactive layout for MOS/LSI"
16th Design Automation Conference 1976 pp 102 - 108

16. Persky G, Deutsch D N, and Schweikert D G
"LTX - A system for the directed automatic design of LSI circuits"
13th Design Automation Conference 1976 pp 399 - 408

17. Persky G
"PRO : An automatic string placement program for polycell layouts"
13th Design Automation Conference 1976 pp 417 - 424

18. Schweikert D G
"A 2-dimensional placement algorithm for the layout of electrical circuits"
13th Design Automation Conference 1976 pp 408 - 416

19. Shiraishi H, and Hirose F
"Efficient placement and routing techniques for Master Slice LSI"
17th Design Automation Conference 1980 pp 458 - 464

20. Cox G W
"The standard transistor array (STAR) Part 2"
17th Design Automation Conference 1980 pp 451 - 457

21. Feller A
"Automatic layout of low-cost quick-turnaround random-access custom LSI devices"
13th Design Automation Conference 1976 pp 79 - 85

22. Preas B T, and vanCleemput W M
"Placement algorithms for arbitrarily shaped blocks"
16th Design Automation Conference 1979 pp 474 - 480

23. Preas B T, and Gwyn C W
"Methods for hierarchical automatic layout of custom LSI circuit masks"
15th Design Automation Conference 1978 pp 206 - 212

24. Loosemore K J
"I.C. Layout - The automatic approach"
5th ESSIRC 1979 pp 48 - 50

25. Preas B T, and Gwyn C W
"Architecture for contemporary computer aids to generate I.C. mask layouts"
11th Annual Asilomar Conference 1977 pp 353 - 361

26. Johannsen D
"Bristle Blocks : A silicon compiler"
Caltech conference on VLSI 1979 pp 303 - 310

27. Werner J
"The silicon compiler : Panacea, wishful thinking, or old hat ?"
VLSI design publication Vol 3 No. 5 1982 pp 46 - 52

28. Baird H S
"A survey of computer aids for I.C. mask artwork verification"
IEEE Symposium on circuits and systems 1977 pp 441 - 445

29. Chao S, Huang Y, and Yam L
"A hierarchical approach for layout versus circuit consistency check"
17th Design Automation Conference 1980 pp 270 - 276

30. Losleben P, and Thompson K
"Topological analysis for VLSI circuits"
16th Design Automation Conference 1979 pp 461 - 473

31. Corbin L V
"Custom VLSI electrical rule checking in an intelligent terminal"
18th Design Automation Conference 1981 pp 696 - 701

32. Williams L
"Automatic VLSI layout verification"
18th Design Automation Conference 1981 pp 726 - 732

33. Allgair R M, and Evans D S
"A comprehensive approach to a connectivity audit or a fruitful comparison of apples and oranges"
14th Design Automation Conference 1977 pp 312 - 321

34. Baird H S, and Cho Y E
"An artwork design verification system"
12th Design Automation Conference 1975 pp 414 - 420

35. Hollander Y
"Using an RTL simulator to simplify VLSI design"
VLSI publication Vol 4 No 5 1983

36. Flake P, Musgrave G, and White I J
"HILO - A logic system simulator"
IEE conference on Computer Aided Design 1974 pp 130 - 136

37. Denneau M, Kranstadt E, and Pfister G
"Design and implementation of a software simulation engine"
CAD publication Vol 15 No. 3 1983 pp 123 - 130

38. Reynolds J S
"A conversational logic simulator for use with a time-sharing computer"
IEE conference on CAD 1969 pp 608 - 615

39. Weeks W, et. al.
"Algorithms for ASTAP - A network-analysis program"
IEEE trans. on circuits and systems Vol CT-20 1973 pp 628 - 634

40. Nagel L W
"SPICE2 : A computer program to simulate semiconductor circuits"
University of California, Berkeley

41. Tanabe N, Nakamura H, and Kawakita K
"MOSTAP : An MOS circuit simulator for LSI"
Int. symposium on circuits and systems 1980 pp 1035 - 1038

42. Chanta B R, Gummel H K, and Kozak P
"MOTIS - An MOS timing simulator"
IEEE trans. on cir. and sys. Vol CAS-22 1975 pp 301 - 310

43. Newton A R, and Pederson D O
"Simulation program with large scale integrated circuit emphasis"
International symposium of circuits and systems 1978 pp 1 - 4

44. Hill D D, and VanCleemput W M
"SABLE : Multi-level simulation for hierarchical design"
Int. symposium on circuits and systems 1980 pp 431 - 434

45. Whitney T
"A hierarchical design-rule checking algorithm"
VLSI design publication (formerly Lambda) Vol 2 No. 1 1981

46. Lindsay B W, and Preas B T
"Design rule checking and analysis of IC mask designs"
13th Design Automation Conference 1976 pp 301 - 308

47. Kozawa T, Tsuki A, Sakemi J, Muira C, and Ishii T
"A concurrent pattern algorithm for VLSI mask design"
18th Design Automation Conference 1981 pp 563 - 570

48. McCaw C R
"Unified shapes checker : A checking tool for VLSI"
16th Design Automation Conference 1979 pp 81 - 87

49. Baird H S
"Fast algorithms for LSI artwork analysis"
14th Design Automation Conference 1977 pp 303 - 311

50. Lambert D R
"Graphics language One"
18th Design Automation Conference 1981 pp 713 - 719

51. McGrath E
"Design integrity and immunity checking"
17th Design Automation Conference 1980 pp 263 - 268

52. Edmonston T H, and Jennings R M
"A low cost hierarchical system for VLSI layout verification"
18th Design Automation Conference 1981 pp 505 - 510

53. Treble D P
"Dimensional checking of MOS LSI circuits"
International conference on CAD 1972 pp 7 -11

54. Franqui B, and Culliney J N
"Computer aided verification of MOS LSI layouts"
Wescon Electronics Show and Convention 1980

55. Alexander D
"A VLSI design rule checker"
Computer Aided Design conference 1980 pp 57 - 60

56. Yoshida K, Mitsuhasi T, Nakada Y, Chiba T, Ogita K, and Nakatsuka S
"A layout checking system for large scale integrated circuits"
14th Design Automation Conference 1977 pp 322 - 330

57. Yamin M
"XYTOLR : A computer program for integrated circuit mask design checkout"
Bell system technical journal Vol 51 No. 7
1972 pp 1581 - 1593

58. Rosenberg L M, and Benbassat C
"CRITIC : An integrated circuit design rule checking program"
11th Design Automation Conference 1974 pp 14 - 18

59. Stratford R I
"Computer aided checking of integrated circuit layout constraints"
International Conference on Computer Aided Design
1972 pp 45 - 50

60. Wilcox P, Rombeek H, and Caughey D M
"Design rule verification based on one dimensional scans"
15th Design Automation Conference 1978 pp 285 - 289

61. Mitchell C L
"MAP : A user-controlled automated mask analysis program"
11th Design Automation Conference 1974 pp 107 - 118

62. Lewis D
"CIFSYM : Layout design on an alphanumeric terminal"
VLSI design publication (formerly Lambda) Vol 2 No. 3 1981

63. "ICMASK"
University of Florida

64. Weste N
"A color graphics system for I.C. mask design and analysis"
15th Design Automation Conference 1978 pp 199 - 205

65. Ackland B D, and Weste N
"Colour display terminals for VLSI : Another perspective"
VLSI publication Vol 3 No. 1 1982

66. "User manual for 5500 series"
4082.075.41 1978 Issue A

67. Eades J D
"GAELIC user's manual"
Wolfson Microelectronic Laison Unit 1974

68. "KICK"
Personal communication
University of California Berkeley

69. Lyon R E
"Simplified design rules for VLSI layouts"
VLSI publication (formerly Lambda) Vol 2 No. 1 1981

BIBLIOGRAPHY

1. Mead and Conway
"Introduction to VLSI systems"
Addison Wesley

2. Sansen W, Heyns W, and Beke H
"Layout automation based on placement and routing algorithms"
NATO Advanced Study Institute on Computer Design Aids for VLSI circuits 1980

3. VLSI publication (formerly Lambda)
CMP publications Ltd.

4. Pavlidis T
"Algorithms for graphics and image processing"
Springer-Verlag Berlin-Heidelberg New York

5. Foley J D, and Van Dam A
"Fundamentals of Interactive computer graphics"
Addison Wesley

6. Camezind Hans
"Electronic Integrated systems design"
Van Nostrand Reinhold Company New york

7. Mavor J
"M.O.S.T. integrated circuit engineering"
Peter Peregrinus Ltd. London

8. Breuer M
"Design automation of digital systems ; theory + techniques"
Prentice-Hall

9. Ayres R F
"VLSI : Silicon compilation and the art of automatic microchip design"
Prentice-Hall

APPENDIX A

CADIC

Computer Aided Design of Integrated Circuits

USER MANUAL

CONTENTS

Section 3 Cursor level commands

3.1 SPACE....Return to main command level
3.2 -.....Remove mask from plot list
3.3 0 => 9...Add masks to plot list
3.4 ?.....Available cursor commands
3.5 C.....Add collection or array of group instances
3.6 F.....Find nearest point in the layout
3.7 G.....Add group instances
3.8 I.....Identify point in shape to be moved
3.9 J.....Jump back to full layout
3.10 K.....Kill shapes
3.11 L.....Last window
3.12 M.....Change mask
3.13 P.....Add polygons
3.14 Q.....Query distance
3.15 R.....Add rectangles
3.16 T.....Add tracks
3.17 U.....Undefined Zoom
3.18 V.....Verify cursor coordinates
3.19 W.....Redraw window
3.20 Z.....Zoom by a factor of 2
3.21 [.....Kill group instances and arrays
3.22 a.....Plot axis once
3.23 n.....Plot net once
3.24 s.....Show how shape is segmented
3.25 w.....Define window size

SECTION 1

GETTING STARTED WITH CADIC

1.1 BASIC INFORMATION ABOUT THE SIGMA 5000 AND CADIC

The SIGMA 5000 is a microprocessor-based system consisting of a GOC (Graphics Option Controller), colour raster scan terminal, and an alphanumeric terminal. The input device used in conjunction with the graphics terminal is a cross-hair cursor, and is controlled by a hand-held control box, containing five buttons - movement left, right, up, down, and 'hit'.

At various times during the use of CADIC, the cross-hair cursor will be displayed on the graphics screen. If a single alphanumeric key is pressed while the c/h cursor is visible, the ASCII equivalent of the key pressed, and the coordinates of the cursor are sent to the computer. CADIC accepts this key as a command and uses the coordinates accordingly.

During the execution of CADIC, all graphic work is displayed on the graphic screen, with all alphanumeric input/output being carried out on the alphanumeric screen.

The framed area of the screen shows the virtual window, and any part of the artwork contained in this window will be displayed on the screen. Therefore if the window is larger that the size of the layout, the whole layout will be displayed, but if the window is smaller, then only part of the layout will be seen. The position and size of the virtual window is under user control, so the user can use a large magnification (small window) to check spacing widths etc. or use a small magnification (large window) for global checks (Figure A1.1).

Above the virtual window frame, on the left hand side, is written the name of the layout, or group definition that the user is presently working on. Above the frame and to the right is displayed the virtual window dimensions Xmin,Ymin,Xdim,Ydim. Below the window area, and to the left is a plot list which shows the mask numbers presently displayed on the screen. Each mask number is enclosed in a box of the relevant colour, so as to make identification simpler.

1.2 PROGRAM INITIALISATION

On running CADIC, the alphanumeric screen will clear and the following message will appear :-

- CADIC -

PROGRAM TO GRAPHICALLY MODIFY AN INTEGRATED CIRCUIT LAYOUT

ENTER NAME OF EXISTING RING DATA STRUCTURE OR RETURN :-

The program will then wait for a filename. If a new ring data structure (i.e. new layout) is required, press carriage return. If the user wants to look at, or modify an existing layout, just type in the name of the data structure followed by carriage return. (Note - do not include the filename extension .RNG)

The program will then check to see if the filename does in fact exist. If yes, then the graphic screen will be set up and the program placed at the main command level (See Section 2). If the filename does not exist in the user's directory, the program will return with :-

FILE <filename> DOES NOT EXIST - PLEASE TRY AGAIN :-

At this point, the user replies as described above. Had the user pressed carriage return for a new data structure, the program puts up the following message :-

ENTER NAME FOR THE NEW RING DATA STRUCTURE OR RETURN :-

Pressing carriage return will abort the program and return the user to the monitor level. Any filename entered is again checked by the program. If the filename is unique the program will set up the graphic screen, ask for a title (see below), and then enter the main command level. If the filename is not unique, the program warns :-

FILE EXISTS - DO YOU WANT TO OVERWRITE ?

The answer to this is YES or NO. YES will cause the file to be overwritten, wheras NO gives the user a chance to cover up his mistake, with the program again asking for the name of the new data structure as before.

To help the user identify different layouts, the program always asks for a layout title when dealing with new data structures. A title up to 30 characters can be entered. Note - all group definitions in the layout require identity names, so the user can always tell where he is situated (i.e. in the main layout, or in a group definition) simply by looking at the title written at the top left of the screen. Therefore, to save confusion, it is advisable to use a layout title different from those likely to be chosen for group titles.

After supplying a title, the program enters the main command level (See below) and waits for further instruction.

SECTION 2

MAIN LEVEL COMMANDS

Whenever these commands are available to the user, the program will print :

WHAT NOW :

All main level commands are described below. Note that only the first two letters of each command need be typed to uniquely identify the command.

2.1 ADJUST

The mask colours are automatically defined during the initialisation stage of the program, but the colours can be changed dynamically by the user with the use of the ADJUST command.

On receiving the command, the graphics screen clears, then 15 boxes (one per mask) are drawn on the screen, each with an identifying number, and drawn in the relevant colour. The program then asks :-

ENTER MASK REQUIRED OR PRESS RETURN TO FINISH :-

If a number between 1 and 15 is entered, the program asks :-

PRESENT SETTINGS FOR MASK <num> ARE :-
RED = <num1>, GREEN = <num2>, BLUE = <num3>

ENTER NEW AMOUNTS OF R,G,B OR RETURN TO FINISH :-

Note that amounts of R,G,B can vary from 0 to 15. On entering the three integers, the respective mask colour is updated on the graphic screen, so that the user can see what it looks like. The above question is again asked, so that the user can try several combinations of R,G,B to acheive the correct colour.

When finished testing the colour, press carriage return. The program then asks :-

DO YOU WANT TO ACCEPT THE NEW SETTINGS ?

If it has been decided that the original colour was better, type NO, otherwise type YES, after which the new settings will be used by CADIC. On answering this question, the program again asks :-

ENTER MASK REQUIRED OR PRESS RETURN TO FINISH :-

Now another mask can be processed. If no more masks are required, press carriage return, and the program will return to the main command level.

2.2 AXIS

This command complements a flag in the program. By default the flag is off, but if set, the program draws scaled axes whenever the screen is redrawn.

The layout is actually quantized to a grid of allowable points (See later), and the ticks on each axis correspond to the grid lines. Should the user be using a large window, too many ticks would be required to show every grid line, therefore the program ticks, for example, only every third grid line. The user is made aware of this by a note positioned above the virtual window frame, which for this example would show :

AXIS GRID X 3

The AXIS command is cancelled by typing a second AXIS when at the main command level.

2.3 CHANGE

This option allows the user to change the name of a group definition. A more subtle option is available, in which the user can change the group that group instances previously referred to. So if the user has a revised group to take the place of the old definition, this command saves the user from having to manually delete then re-insert all the affected group instances.

On typing CHANGE at the main command level, the alphanumeric screen clears and the following question is asked :-

```
- CHANGE GROUP NAME -
DO YOU WANT TO CHANGE GROUP DEFINITION OR INSTANCE NAME ?
```

The user must type 'DE' or 'IN' as required, followed by carriage return. If 'DE' was typed, the program asks :

```
ENTER NAME OF GROUP DEFINITION TO BE CHANGED OR RETURN TO FINISH :
```

To return to the main command level, just press carriage return, otherwise supply the necessary group name. If the name does not exist, the program returns with :

```
GROUP DEFINITION NAME <groupname> DOES NOT EXIST.
PLEASE TRY AGAIN OR RETURN TO FINISH :-
```

Should the name exist, the program will ask :

```
ENTER NEW GROUP DEFINITION NAME OR RETURN :-
```

Pressing carriage return will cancel the command and return the user to the main command level. Otherwise, a unique group name must be supplied. If not unique, the program will return with :

```
GROUP DEFINITION NAME <groupname> ALREADY EXISTS
PLEASE TRY AGAIN OR RETURN TO FINISH :-
```

The change of an instance is as described for a definition, except that the words GROUP INSTANCE are used to replace GROUP DEFINITION. Note - group instances do not have names, they only refer to groups with names, so a GROUP INSTANCE NAME really means the GROUP DEFINITION NAME that a group instance refers to.

2.4 CLEAN

In general, the user will not add shapes to the layout in a sequence that will ensure efficient build up of the data structure. Usually beads of similar type are scattered throughout the file, instead of being stored on the same or adjacent pages.

On typing 'CLEAN' at the main command level, the program will re-arrange the data structure such that the information is stored in a more efficient manner. The layout is in no way affected, but faster plotting times, and better response times are possible with a 'clean' data structure.

2.5 CURSOR

In most cases, the building up of I.C. artwork is aided by the use of a grid. For this reason, the cursor coords are rounded to the nearest grid point, as set by the user.

The default setting is XOFF,YOFF,GRID = 0,0,10 so the cursor coordinates will always be a multiple of 10. Note that if the settings were XOFF,YOFF,GRID = 3,2,10 then the x-coords will progress as 3,13,23,,, and the y-coords will progress 2,12,22,,,.

On typing CURSOR at the main command level, the alphanumeric screen clears and the program asks :

```
        - CURSOR GRID UPDATE -

PRESENT CURSOR GRID SETTINGS ARE :  <num1>,<num2>,<num3>
ENTER NEW SETTINGS OR RETURN :
```

The new values are then entered as integers, separated by spaces. Note that only the minimum amount of information need be entered. If only the XOFF setting was to be changed, the carriage return could be pressed after entering it's new value, and the YOFF and GRID values will remain as before. If both offsets needed updating, then the carriage return can be pressed after the second entry, and so on.

2.6 DEPTH

On typing DEPTH at the main command level, the alphanumeric screen clears and the program asks :-

```
                - NESTING DEPTH UPDATE -

PRESENT GROUP NESTING DEPTH IS :  <num>

ENTER NEW VALUE OR RETURN :
```

This command allows the user to specify what depth of group nesting is to be drawn out. By default, DEPTH is set to 10, but when plotting out a layout, to save time, DEPTH could be set to 1, which would cause the program to draw out only the groups in the highest level of the group hierarchy.

2.7 EXIT

This command puts the user up one program/system level. Therefore if one was presently dealing with the whole layout, EXIT will close all the files used by CADIC, then return the user to the system monitor level. If the user is modifying a group definition, typing EXIT will return him to the whole layout. Again note that a quick look at the name displayed at the top left of the graphic screen will tell the user whether he is presently at layout, or group level.

2.8 FILL

This command complements a flag in the program. By default the flag is off, but if set, the program fills shapes whenever they are plotted on the screen.

When the flag is off, the shapes are plotted out in outline. Therefore the user can watch the layout be built up. When the flag is on, the mask is first plotted on an 'invisible' plane. Only when complete will the SIGMA fill the shapes and copy the whole mask layout onto the screen. During the plotting period, the user will see no activity, and so may cause confusion to the first time user.

The FILL command is cancelled by typing a second FILL when at the main command level.

2.9 GROUP

This command allows the user to set up, or modify a group definition. After typing GROUP, the program clears the screen then asks :

- GROUP MODIFICATION -

ENTER NAME OF EXISTING GROUP DEFINITION OR RETURN :-

If modification of an existing group definition is required, type in the name followed by carriage return. If the name does not exist, the program will warn :

GROUP NAME <groupname> DOES NOT EXIST - PLEASE TRY AGAIN :-

and the user can make another attempt. If a new group definition is required, press carriage return, after which the program will ask :

ENTER NAME OF THE NEW GROUP DEFINITION OR RETURN TO FINISH :-

Pressing carriage return allows the user to cancel the command and return to the main command level, otherwise a unique name (up to 6 characters) must be supplied. If the name is not unique, the following message will appear :

GROUP NAME <groupname> ALREADY EXISTS - DO YOU WANT TO OVERWRITE :

IF YES is typed, the contents of the group definition are removed, and the group opened as if it had been newly set up. Typing NO forces the program to return to the previous question, so that the user can try a different groupname.

Note that at all times while using CADIC, group names must be unique, and also not the same as the first 6 letters of the layout title.

On accepting the group name, the program sets up the screen and then reaches its main command level just like that described for the whole layout.

CADIC is set up to handle only one group definition at a time. Should the GROUP command be attempted while the user is still modifying a group definition, the warning :

STILL IN <groupname>

will be displayed in the menu area. If another group is required for modification/inspection, the user must exit from the present group, and enter the required group.

IMPORTANT - At present, CADIC has no facility for deciding on which level in the hierarchy a group is on. Therefore the lowest level must be added first, and the hierarchy built up accordingly (i.e. If A calls B, group B must already exist).

2.10 HELP

This command simply clears the alphanumeric screen and prints out a list of all the possible main level commands, plus a brief description of their use.

2.11 INFORM

This command clears the alphanumeric screen and prints out the status of various parameters in the program, then returns to the main command level. The output looks like :-

```
                - SYSTEM INFORMATION -

YOU ARE PRESENTLY DEALING WITH THE MAIN LAYOUT/GROUP groupname


SWITCH SETTINGS ARE :-
     AXIS    = <stat1>
     NET     = <stat2>
     FILL    = <stat3>
     ONLINE = <stat4>


PARAMETER SETTINGS ARE :-
     CURSOR (XOFF,YOFF,GRID) :  <num1>,<num2>,<num3>
     DEPTH :  <num4>
     TRACK (DELTA) :  <num5>
     WINDOW (XOFF,YOFF,XDIM,YDIM) :  <num6>,<num7>,<num8>,<num9>
```

2.12 LIST

This command clears the alphanumeric screen and prints out a list of all the existing group definitions. If there are more that 20 names, the program will stop and print :

PRESS RETURN FOR MORE :-

To continue, press carriage return, after which the screen will clear, and the list will continue from where it left off.

2.13 MODIFY

This command allows the user to modify whatever layout/group definition he is presently in. On receiving this command, the following question appears in the alphanumeric screen :-

MASK REQUIRED :

All layouts can have up to 15 masks, and this question defines which mask is to be dealt with. Note that only one mask can be operated on at any one time, but commands do exist for jumping between masks, without having to return to the main command level (See Cursor Commands).

After the user types in a valid mask number, the program checks to see if the mask has already been plotted out on the graphic screen. If not, the mask will be plotted out if a space in the plot list exists. Remember that a maximum of four masks can be displayed at any one time.

If the mask number is accepted, the cross-hair cursor is displayed. At this point in time, the user is at the Cursor Command Level. All possible commands at this level will be described in Section 3.

2.14 NET

This command complements a flag in the program. By default the flag is off, but if set, the program draws a net of points whenever the layout is redrawn. These points show where the grid points lie, and exactly line up with the axis grid (Section 2.2), but lets the user position the cursor accurately, without having to keep referring to the edges of the screen.

Once the net is drawn, the note similar to that given with the AXIS command is shown at the top of the screen :-

NET GRID x <num>

The NET command is cancelled by typing a second NET when at the main command level.

2.15 ONLINE

This command complements a flag in the program. By default the flag is off, but if set, the program will design rule check each shape or group call added to the layout, against a set of supplied design rules. After typing ONLINE, if the flag is being switched on, the program clears the alphanumeric screen, then asks :

- ONLINE DESIGN RULE CHECKING -

ENTER NAME OF DATA STRUCTURE CONTAINING THE RULES, OR RETURN :-

Pressing carriage return cancels the command, and returns the program to the main command level. If a filename is specified, the program checks to see if it exists. If not, the program will warn :

FILE <filename> DOES NOT EXIST - PLEASE TRY AGAIN :-

and the user can make another attempt. If the file does exist, it is copied into CADIC's temporary file, then the program return to the main command level.

The ONLINE command is cancelled by typing a second ONLINE when at the main command level.

2.16 ORIGIN

This command shows the user where all the group origins are situated. The points are shown using isosceles triangles, with the 'top' of the triangle lying on the origin point.

2.17 PLOT

This command is used to plot out specified masks on the graphic screen. The SIGMA contains 4 display planes, so allowing a maximum of 16 colours to be plotted out simultaneously. The intuitive approach is to allow all fiveteen masks to be shown at once if required. In this situation, shapes on later masks will overwrite previous shapes if they overlap. For example, if the aluminium mask is plotted out after the contact mask, then all the contact holes would be overwritten.

Another approach uses the fact that the SIGMA can mask the writing of data to the display planes. By putting a mask on each plane, the overlap conditions produce unique colour numbers, and so specific colours can be assigned to the overlap. In the case of plotting the aluminium and contact masks, the contact holes would still be seen under the aluminium, and correctly coloured to show it up against contact holes that were not covered. This approach is obviously better, and was the approach adopted by CADIC. Therefore, when plotting out masks, CADIC limits the number of masks to a maximum of four at any one time.

2.18 SAVE

To prevent the ring data structure becoming corrupt due to, say a system failure, CADIC takes a working copy of the data structure during initialization. All future work is performed using the working copy. Typing 'SAVE' at the main command level will copy the working copy into the actual data structure, thus producing a protected version of the layout.

The 'SAVE' command should be used frequently if the user is building up and/or editing the layout, so that a system failure loses only the work up to the most recent 'SAVE' command, instead of the whole day's work. Note that an automatic save of the working copy is made when exiting from CADIC.

2.19 SWITCH

During on-line design rule checking, the user may want to temporarily switch off certain rules, either because the rule is not relevant, or because the rule is taking too long to implement. The task off switching off/on rules can be achieved using the SWITCH command. On receiving the command, the alphanumeric screen clears, and the program asks :-

ENTER RULE NAME TO BE CHANGED OR PRESS RETURN TO FINISH :-

If an existing rule name is entered, the program asks :-

RULE <rulename> IS PRESENTLY <status> - do you want to change it ?

If the rule is already in the correct status, type NO, otherwise typing YES will invert the status (i.e. OFF -> ON, ON -> OFF). Once complete, the program again asks :-

ENTER RULE NAME TO BE CHANGED OR PRESS RETURN TO FINISH :-

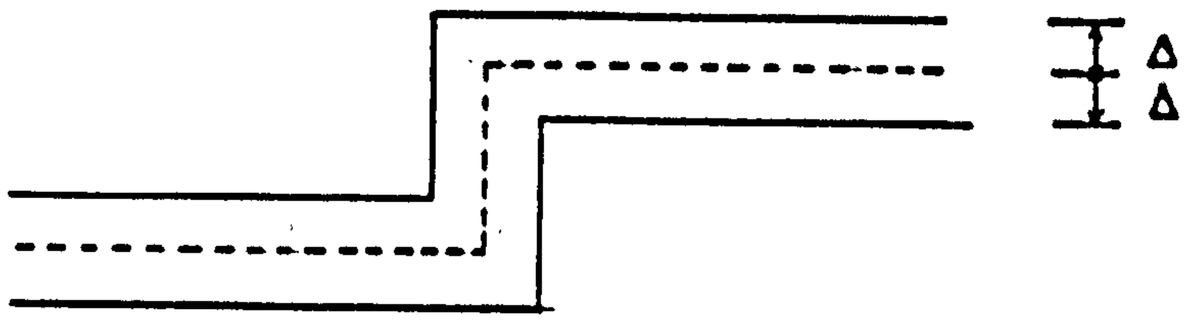
In this way, several rules can be changed at the same time. If no more changes are required, press carriage return, and the program will return to the main command level. Note that the SWITCH command only has relevance if the ONLINE flag is set. If not, the program gives the warning :-

ON-LINE DESIGN RULE CHECKING NOT YET IMPLEMENTED

then returns to the main command level.

2.20 TRACK

This command allows the user to change the width of a track. By default Δ is set to 10 units, where Δ is defined as :



On typing TRACK, the alphanumeric screen clears, and the program asks :

```
- TRACK UPDATE -

PRESENT TRACK DIMENSION IS :  DELTA = <num>

ENTER NEW VALUE OR RETURN :
```

Pressing return leaves the value of Δ as before, otherwise Δ is updated as required.

2.21 WINDOW

This command allows the user to specify the virtual window size. The program clears the alphanumeric screen and asks :

```
- WINDOW UPDATE -

PRESENT WINDOW SIZE IS :  <num1>,<num2>,<num3>,<num4>

ENTER NEW VALUES OR RETURN :-
```

As in the CURSOR command, only the minimum number of values need be entered.

SECTION 3

CURSOR COMMAND LEVEL

Whenever the cross hair cursor is visible, the user is at the cursor command level. At this level, the user can alter the artwork, or simply inspect it using the range of windowing functions available. The commands are as follows :-

3.1 SPACE.....RETURN TO MAIN COMMAND LEVEL

Pressing the space bar will return the user to the main command level.

3.2 -.....REMOVE MASK FROM PLOT LIST

On typing '-', the program asks :-

MASK REQUIRED :

Enter the mask number to be removed. If the mask is not displayed on the screen, the program will reply with :-

MASK <num> IS NOT IN THE PLOT LIST

If the mask number entered is displayed on the screen, then it will be immediately removed, and the plot list at the bottom left hand corner of the screen will be updated accordingly. If the mask the user was working on is to be removed, then the program will remove it, but ask :-

THE MASK YOU WERE WORKING ON HAS BEEN REMOVED

MASK REQUIRED :

On completion of this command, the program returns to the cursor command level.

3.3 ?....AVAILABLE CURSOR COMMANDS

At any time while the cross hair cursor is visible, the user can type '?' to find out which cursor commands are presently available. For example, if the user is at the cursor command level, the full list of commands will be given. If the user is in middle of adding a polygon, only those commands relevant to the addition of the polygon will be given.

The list of commands appears on the alphanumeric screen, along with a brief description of their use, and will provide useful information for both the inexperienced and experienced user.

3.4 0 => 9.....ADD MASKS TO PLOT LIST

Pressing keys 1 => 9, plots out the corresponding mask number if a space in the plot list exists. If there are already four masks displayed on the screen, the program will warn :-

THERE IS NO MORE ROOM IN THE PLOT LIST

and will return to the cursor command level. If the mask already exists, the program will warn :-

MASK <num> IS ALREADY IN THE PLOT LIST

then will return to the cursor command level.

The graphic screen does not clear, so the plot will superimpose itself onto any existing artwork. This facility allows the user to check alignment between shapes on different masks etc.

If the 0 key is pressed, the question :

MASK REQUIRED :

appears in the menu area, allowing the user to choose a mask number over the whole range 0 => 15.

3.5 C - ADD COLLECTION OR ARRAY OF GROUP INSTANCES

The user can insert an array of group instances using this command. When typed, the program asks :

ENTER GROUPNAME OR RETURN TO FINISH :-

To cancel the command, press carriage return, otherwise enter the name of the group required in the array, followed by carriage return. If the groupname does not exist, the program replies with:

GROUPNAME <groupname> DOES NOT EXIST
PLEASE TRY AGAIN OR RETURN TO FINISH :-

If the name does exist, the program asks :

ORIENTATION :-

By this it means the orientation of the group instances in the array, as the array cannot be orientated. The orientation is a 3-digit decimal number of the form 'abc' where a = reflection in X-axis, b = reflection in Y-axis, c = rotation of +90 degrees. The letters a to c are given the value 1 or 0 depending on whether the transformation is, or is not required. Note that if an orientation involves a rotation, the rotation is always implemented first.

As an example, if the user wants the group reflected in the X-axis, the code would be 100. If rotation followed by reflection in the Y-axis is required, the code would be 011.

If the user presses carriage return without entering an orientation code, a default value of 000 will be assumed. On accepting an orientation code, the program proceeds by asking :

X NUMBER AND SPACING

which means the number of group instances required in the X direction of the array, plus the spacing between instances. Pressing only carriage return will assume the X number as 1.

Once answered, a similar question will be asked in reference to the Y direction :

Y NUMBER AND SPACING

On completion of the data input, a point on the screen will show where the origin of the bottom left-hand group instance is situated. Note - if the group instance has been rotated, then this point may not be the bottom left-hand corner of the whole array (See Figure A1.2)

Once placed, the cross hair cursor is returned, as the user has the ability to adjust the position of the array if not correct.

Typing an 'S' will just substitute the new cross hair cursor coordinates, whereas typing a '#' will ask for the coords to be entered at the keyboard, allowing the user to specify the point exactly. Should the array be incorrect, the user can remove it from the artwork by typing a 'K'. Note - to be effective, this command must be used before any other command (other than 'S' and '#') is implemented.

Once the array is in the correct position, the user can draw it out by using the 'D' command.

If the ONLINE flag is set, then the array must be design rule checked before it is drawn out and added to the ring data structure. Once the checks have been applied, CADIC proceeds in one of two ways.

If no violations exist, the array is drawn out in solid lines, and added to the data structure. If violations do exist, the error messages are printed out on the alphanumeric screen. The array is then drawn out in dashed lines, and the following question asked :

DO YOU WANT TO OVER-RULE THE ERRORS ?

Answer YES or NO. If the answer is YES, then the array is drawn out in solid lines, and is added to the ring data structure. If the answer is NO, then the array is removed from the screen, and 'killed' from memory.

3.6 F - FIND NEAREST POINT IN THE LAYOUT (INCLUDING GROUPS)

On typing an 'F', the program searches the data structure for the point that is closest to the cross hair cursor. Note that this search includes all group instances and arrays. If a point is found, the program replies with :

NEAREST POINT TO THE CURSOR IS :-
X = <num1>, Y = <num2>

If the point cannot be found, for example if the user is on the wrong mask, the program warns :

NO SHAPES ON MASK <num>
CLOSE TO THE CURSOR

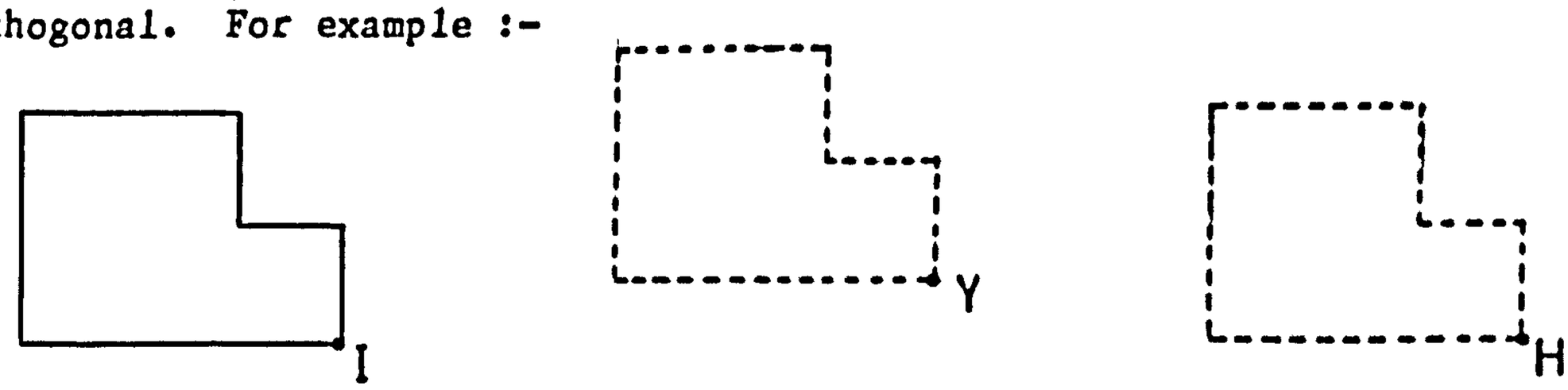
3.7 G - ADD GROUP INSTANCE

This command follows exactly as the 'C' command, except of course, the X and Y numbers and spacing are not asked for. As with the arrays, the group instance can be moved, drawn out, and aborted once it is inserted.

3.8 I - IDENTIFY POINT IN A SHAPE TO BE MOVED

On typing an 'I', the program searches the data structure for the point that is closest to the cross-hair cursor. Note that the search does not include group instances and arrays.

If a point is found, the user can move the whole shape if required. Typing a 'Y' at the new position for the point allows the shape to be moved at an angle. Typing an 'H' will force the program to calculate the nearest point to the new cursor position, such that the movement is orthogonal. For example :-



In either case, a point will be drawn at the new location. This point can be adjusted using any of the commands 'S','#','K','N'. Once happy with the position of the point, typing a 'D' will delete the original shape, and draw it in its new location. Note that an automatic delete and draw will take place (if not already done so) before commencing with a new cursor level command.

3.9 J - JUMP BACK TO FULL LAYOUT

The bounding rectangle of the layout is dynamically updated whenever new shapes or group calls are added to the layout/group definition. The program stores these dimensions and so typing a 'J' at the cursor command level forces the program to redraw the layout, such that the whole layout fits neatly into the window area. The position of the cursor is not important.

3.10 K - KILL SHAPES

To implement this command, the user must place the cross hair cursor over the shape that is to be deleted. If the program cannot find the shape, the message :

THERE ARE NO SHAPES ON MASK <num>
CLOSE TO THE CURSOR

If this is the case, the cursor must be repositioned and the 'K' command tried again. If the shape is found, it will be immediately removed from the layout.

3.11 L - LAST WINDOW

This command redraws the layout using the previous window dimensions. Such a command is useful, for example, if the user is presently using a large window. He can zoom in to make a detailed check, then when finished, can type 'L' to return to the original window size that he was using.

3.12 M - CHANGE MASK

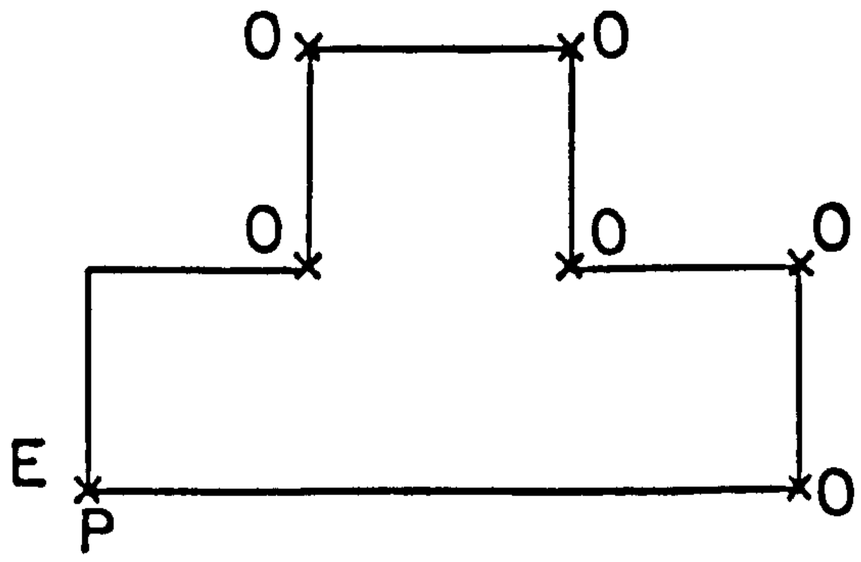
When in the MODIFY mode, the user can operate on only one mask at a time. Should modification be required on another mask, using the 'M' command will force the program to ask :

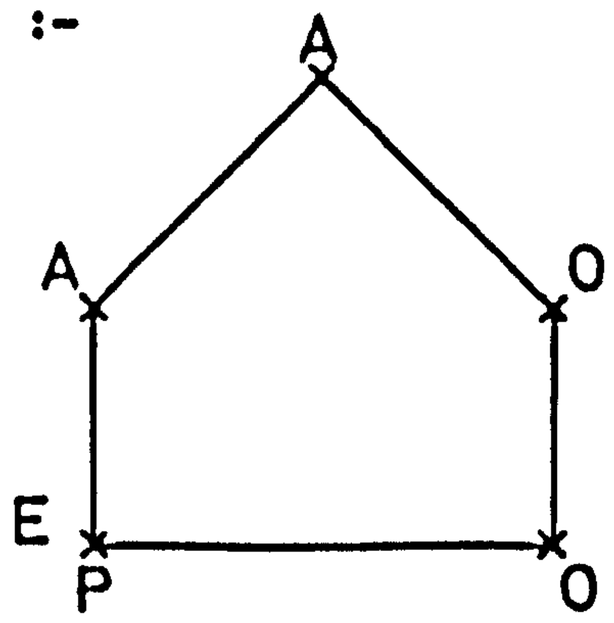
MASK REQUIRED :

The new mask number can be entered, so continuing the modification, but now on the new mask number.

3.13 P - ADD POLYGONS

This command will initiate the adding of a polygon to the data structure. Note - this point must be the bottom left-hand corner of the polygon. In CADIC there are two classifications for polygons. Short format polygons, and long format polygons. Short format polygons contain only orthogonal segments (Manhattan geometry), wheras long format polygons may contain angled segments :-





After shape initialisation, the user can set about adding the other points. To do this, he has the choice of eight commands : '0','A','E','X','o','a','e','x'.

The '0' key will calculate the nearest point to the cursor, such that the segment between the new point and the last point is orthogonal. (i.e. horizontal or vertical). To finish a polygon orthogonally, the user must type an 'E'. Note - the position of the cursor is not important when finishing polygons, as the last point must be equal to the first point to satisfy the closed shape constraint.

On the other hand, if the 'A' command is used, the cursor coordinates will be accepted, allowing angled segments to be added. To finish a polygon with an angled segment, type an 'X'. Again the position of the cursor for this finishing command is not important.

Note that if a light segment is required, use the upper case commands, and if a dark segment is required, used the lower case commands.

If the user adds a point which he realises to be in the wrong place, he can move it using one of the following options : 'S','#','N'. Note - these commands only apply to the point newly added.

'S' will simply recalculate the coordinates of the point, using the new cursor coordinates. '#' is identical to the 'S' command except that the coords are entered at the keyboard. 'N' will search through the data structure, and find the point nearest to the cursor position. If found, this point replaces the incorrect one, so allowing the user to 'tag' shapes onto existing anchor points.

At any point during the formation of the polygon, the user can abort the 'P' command, by typing a 'K' for kill.

Once the polygon is complete, the user can draw out the shape by using the 'D' command. Note that if not already drawn out, through using the 'D' command, the polygon will be drawn out automatically before commencing any new cursor level command.

If the ONLINE flag is set, then the polygon must be design rule checked before it is drawn out and added to the ring data structure. In the event of a violation, the program will proceed as described in Section 3.5.

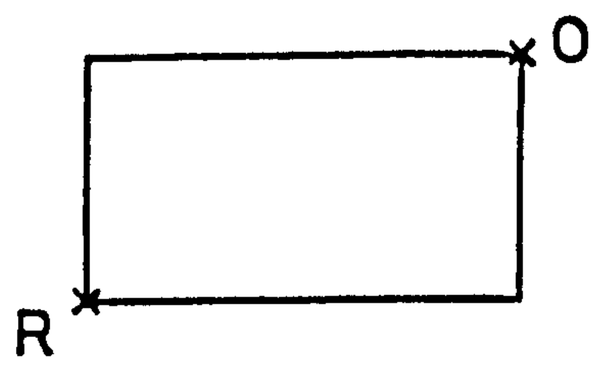
3.14 Q - QUERY DISTANCE

This command is used to check distances between two points on the graphic screen. On pressing 'Q', the present cursor position will be shown as a point on the screen. If the user then moves the cursor, and types a second 'Q', the new cursor position will be shown by a point, and the incremental distance between the two points will be diplayed in the alphanumeric screen :

INCREMENTAL DISTANCE BETWEEN THE TWO POINTS IS :-
X-INC = <num1>, Y-INC = <num2>

3.15 R - ADD RECTANGLES

This command initialises the program to accept a rectangle into the data structure, and will show the present cursor position by a point on the screen. Note - this point must be the bottom left-hand corner of the rectangle. To complete the rectangle, type an 'O' at the position of the top right-hand corner of the rectangle :-



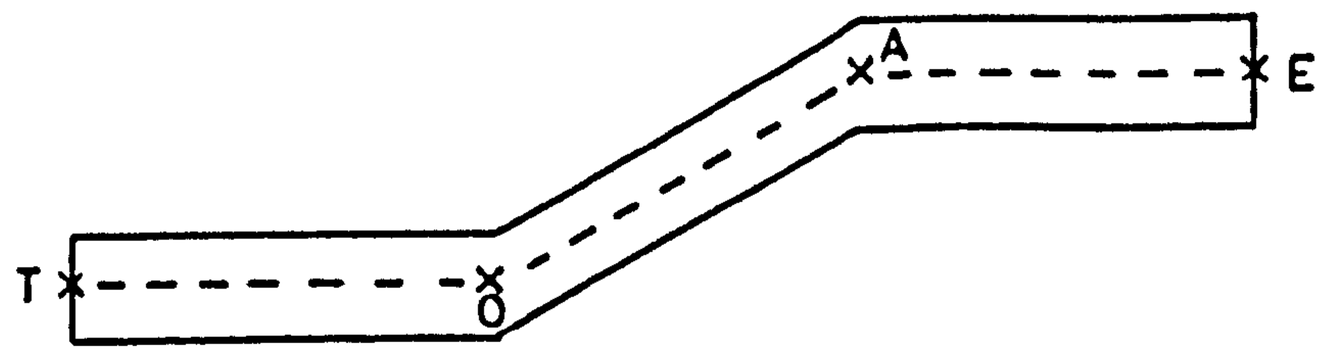
As with the polygon command, any of the options : 'S','#','N', 'K' can be used in conjunction with the 'R' and 'O' commands, with the 'D' command also being available to draw out the rectangle, when satisfied that the rectangle is correct. Again, the program will automatically draw out the shape (if not already done so) before commencing any new cursor level command.

If the ONLINE flag is set, then the rectangle must be design rule checked before it is drawn out and added to the ring data structure. In the event of a violation, the program will proceed as described in Section 3.5.

3.16 T - ADD TRACKS

This command initialises the program ready for insertion of a track, and shows the present cursor position as a point on the screen. Note that this point must be the bottom left hand point of the track centre line.

To add a track, the user specifies the centre line, with the width of the track being defined by DELTA. To change this value the user must return to the main command level.



After typing a 'T' to initialize the track, the commands 'O' and 'A' can be used to add orthogonal and angled track centre-line points. As with the polygon and rectangle, the points can be moved by commands : 'S','#','N','K'.

To finish the track, type an 'E' at the required point if the last segment is to be orthogonal, or type an 'X' if the segment is to be angled. To draw out the track, type 'D', but remember that an automatic draw will take place when commencing a new cursor level command, if not already done so.

If the ONLINE flag is set, then the track must be design rule checked before it is drawn out and added to the ring data structure. In the event of a violation, the program will proceed as described in Section 3.5.

3.17 U - UNDEFINED ZOOM

This command redraws the layout, but with the facility of allowing the user to 'zoom' in or out of the artwork. On typing a 'U', the program will ask :

ZOOM IN FACTOR :

If a positive integer is entered, the window dimensions will decrease by the zoom-in-factor (effectively increasing the artwork by the same factor), and the centre of the new virtual window will correspond to the cursor position at the time when the 'U' key was pressed.

If a negative integer was entered as the zoom-in-factor, then the window size will increase, giving the effect of moving away, or zooming out, from the layout.

3.18 V - VERIFY CURSOR COORDINATES

This command simply tells the user the present coordinates of the cross hair cursor. These are given in the menu area as :

CURSOR POSITION IS :-
X = <num1>, Y = <num2>

These coordinates will also be represented as a point on the screen.

3.19 W - REDRAW WINDOW

This command lets the user 'slide' the virtual window around, so that he can look at different areas of the artwork, without changing the window size (The window offsets will of course change). The direction moved is dependent on the cursor position, so the screen can be considered to be cut up into 9 sections as shown in Figure A1.3a.

Assume for example, that the cursor is in area 6 when the 'W' command is implemented.The program then asks :

ENTER DISPLACEMENT FACTOR :

By this, it means the number of half windows that the user wants to move along. The effect of different displacement factors for area 6 are shown in Figure A1.3b.

Similarily, if the cursor is in area 4, the window will move left by the specified amount, and areas 8 and 2 will cause the window to move up or down respectively. If areas 1,3,7,9 are chosen, the window will move in a diagonal fashion, with the direction being dependent on which area is chosen. Note that if the cursor is in area 5 the displacement factor will not be asked for, and the artwork will be redrawn using the same window settings as before.

3.20 Z - ZOOM IN BY A FACTOR OF 2

This command acts in exactly the same way as the 'U' command, except the the zoom-in-factor is set automatically to +2.

3.21 [- (Shift K) - KILL GROUP INSTANCES AND ARRAYS

To delete a group instance, place the cross hair cursor over the instance and type '['. If the program finds the group instance, it will instantaneously delete it.

To delete an array, the procedure is exactly as above, except that the cross hair cursor must be placed over the bottom left instance in the array to be effective.

3.22 a - PLOT AXIS ONCE

This command is useful when at the cursor command level, and the axes are required for a quick check on a shape's position. The axes are plotted once, and the axis flag is not set.

3.23 n - PLOT NET ONCE

This command is useful at the cursor command level,when the net of points is required for a quick check on a shape's position. The net is plotted once, and the net flag is not set.

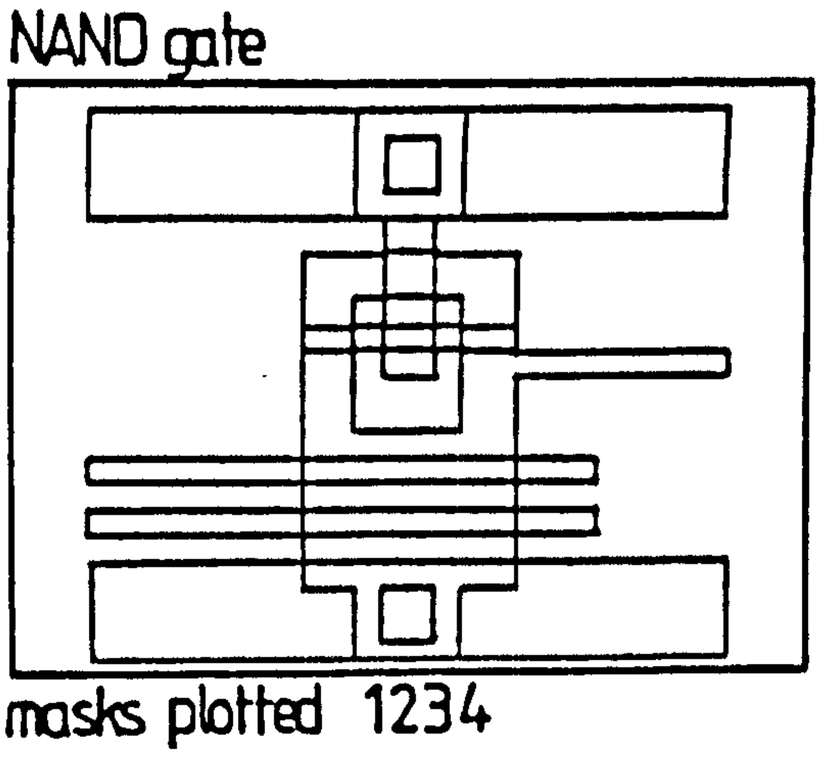
3.24 s - SHOW HOW A SHAPE IS SEGMENTED

Future work in CADIC may allow the individual sub-polygons to be processed, rather than the polygon as a whole. For example moving only a segment of a track. In such situations it may be useful to see just how a polygon is cut up (if at all).

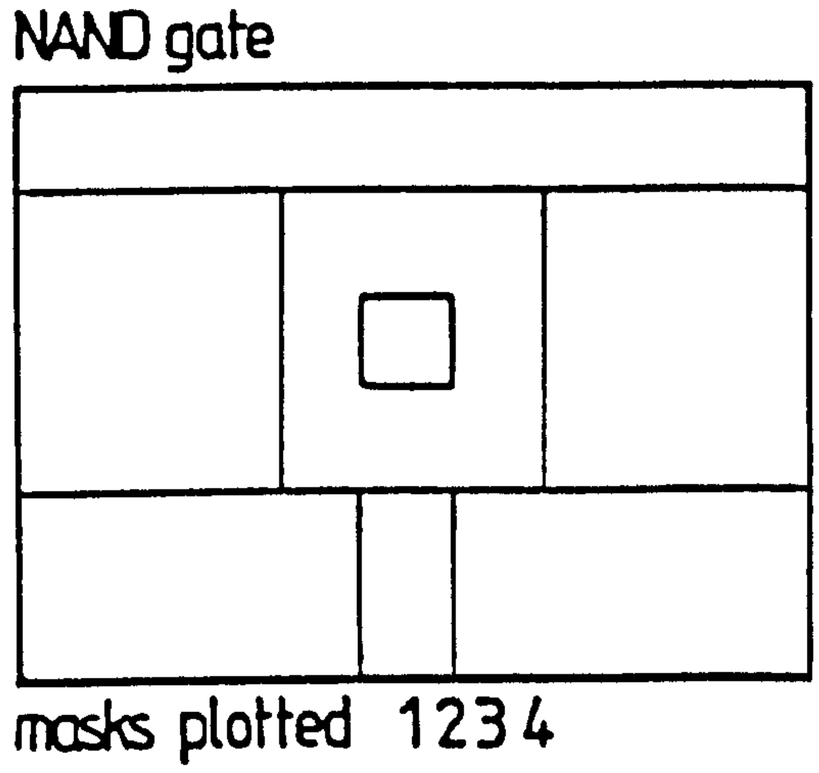
Positioning the cursor over the shape under question, and typing 's' will show up all the sub-polygons if they exist.

3.25 w - DEFINE WINDOW SIZE

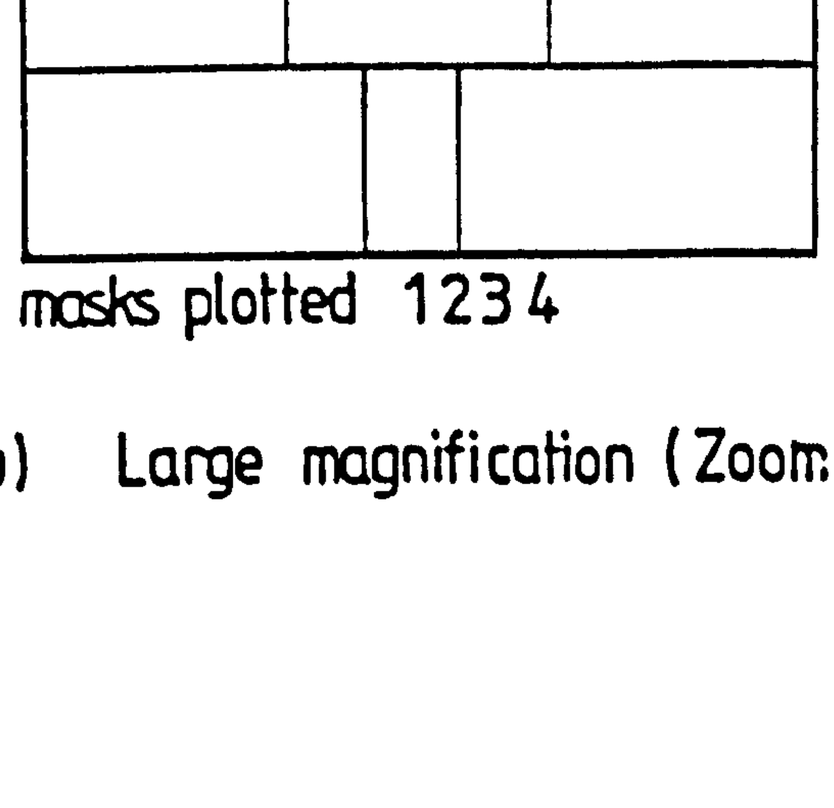
This command allows the user to choose an area of the layout to be redrawn. The position of the cursor is taken as the bottom left hand corner of the area. The cursor is then repositioned at the required top right hand corner of the area, and a second 'w' is typed. The program then redraws the layout, such that the chosen area fills the screen.



(a) Full layout



(b) Large magnification (Zoom in)



(c) Small magnification (Zoom out)

Figure A1.1

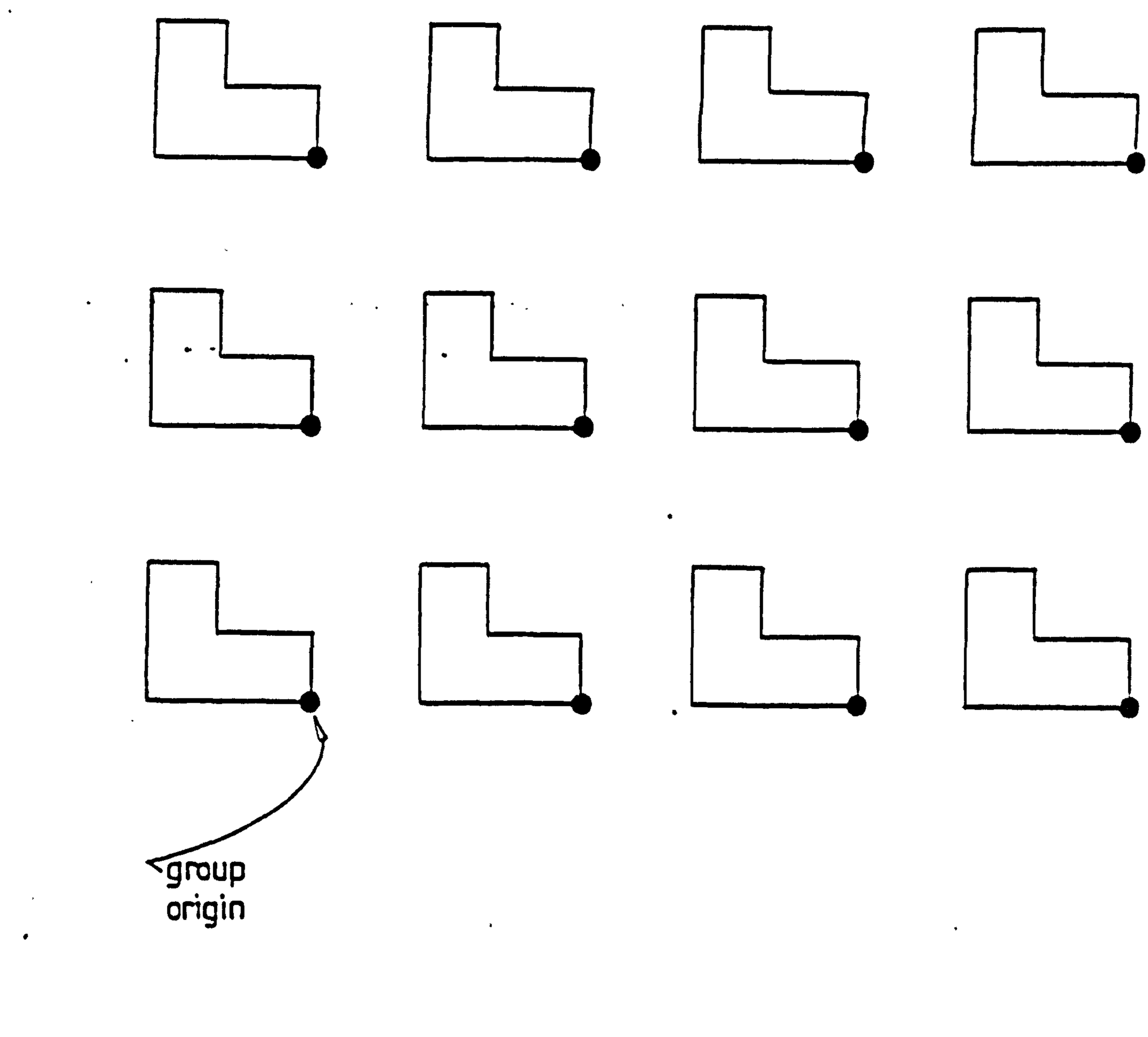
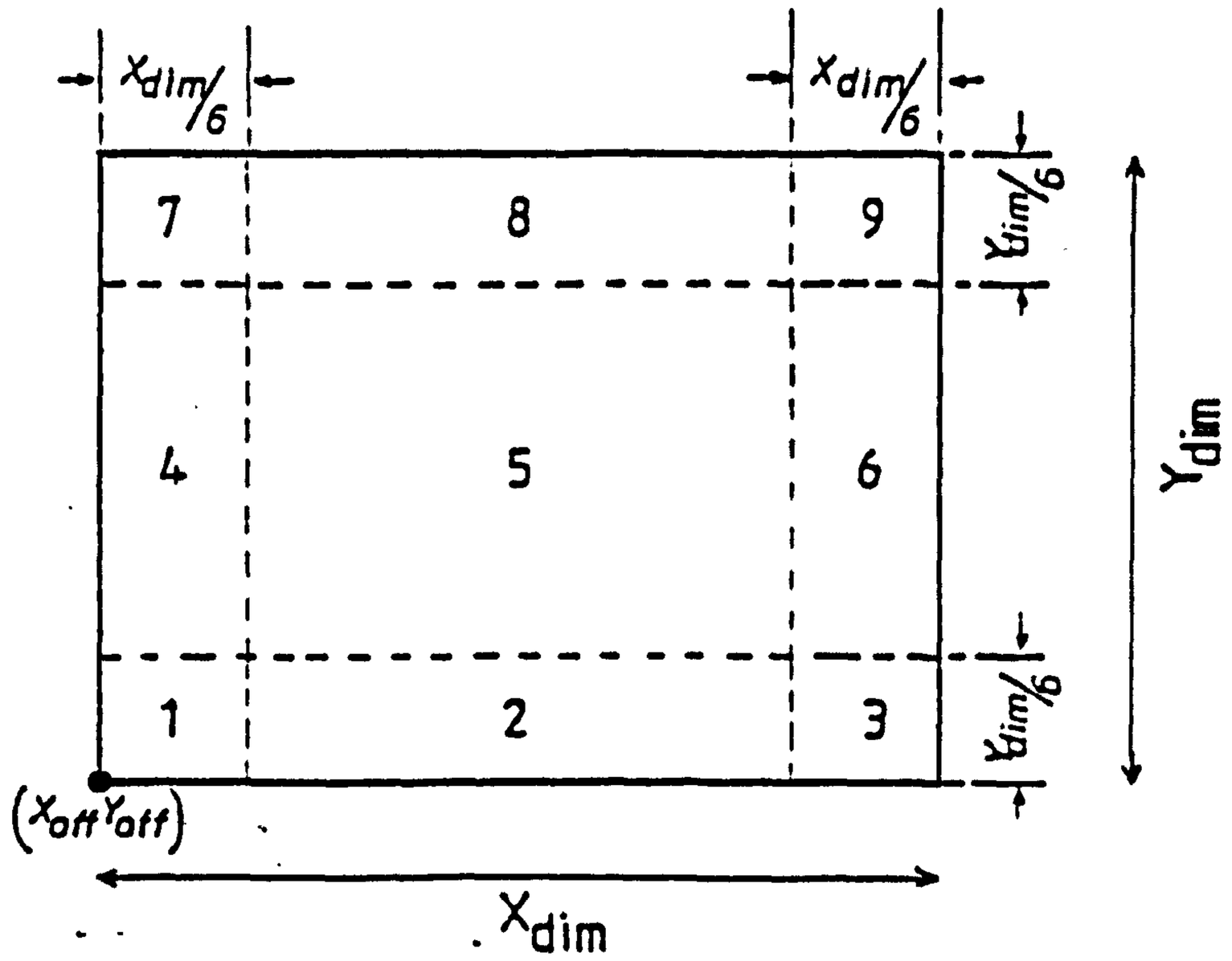
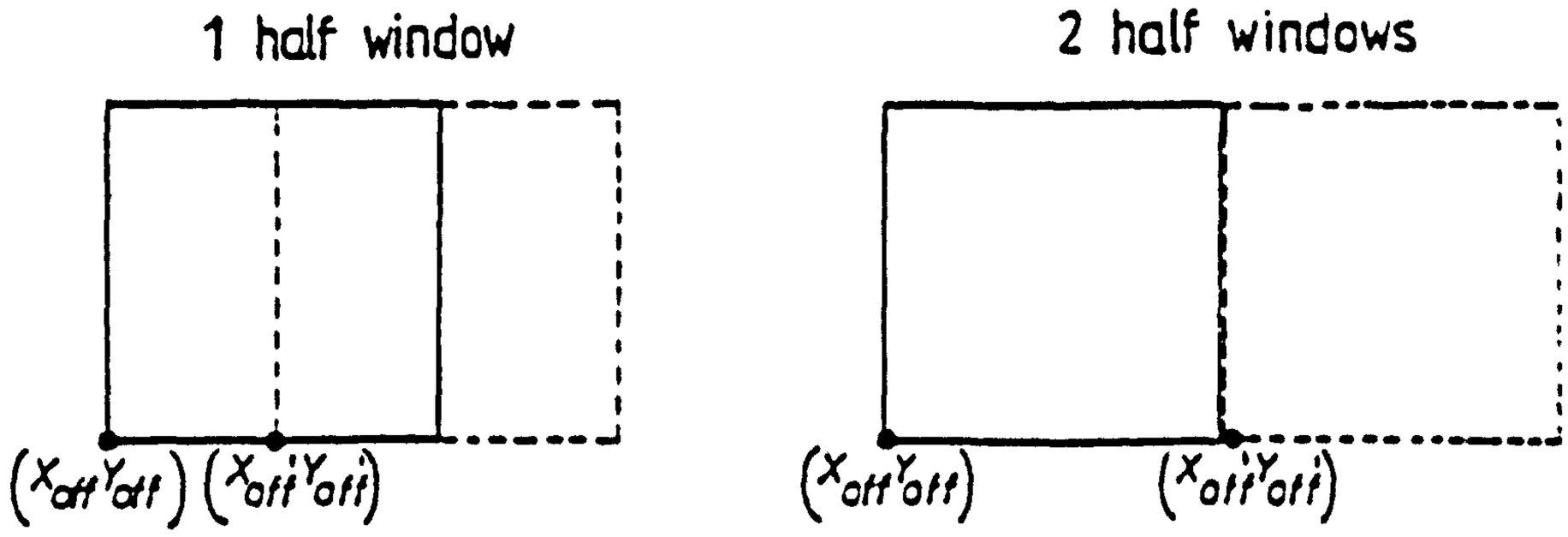


Figure A1.2 The array command



(a) Segmentation of screen window when using "W" command



(b) The effect of the displacement factor

Figure A1.3

APPENDIX B

On-line design rule checking algorithms

CONTENTS

SECTION 1

AVAILABLE COMMANDS

In this section, the algorithms used to perform the design rule checks will be described. Before doing this, it is useful to describe some of the more general definitions that will be used.

Firstly, a shape can be defined as primary or secondary. A primary shape is one which has already been stored in the shape list. A secondary shape is the shape to be found. For example, the dimension checks (WIDTH, AREA etc) only use primary shapes, selectors (OVERLAP,SEPARATE etc) use a primary shape to find a secondary shape. Note that the definition only exists within any one algorithm, since a secondary shape found by an OVERLAP test will become a primary shape if it is tested in WIDTH, and so on.

Secondly, much use is made of the segment type to try a cut down the number of segments to be processed. The segment type for each segment is calculated when required, and is defined as follows :-

0 - angled segment in

1 - horizontal in

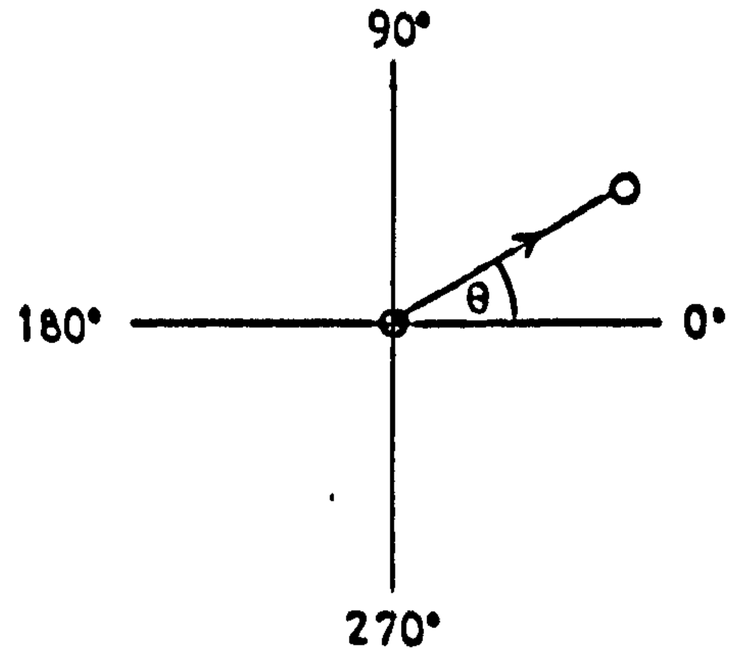
2 - vertical in

3 - angled segment out

4 - horizontal out

5 - vertical out

The direction of a segment is from the starting coordinates to the finishing coordinates, and is deemed to be travelling outwards or inwards using the following rule :-



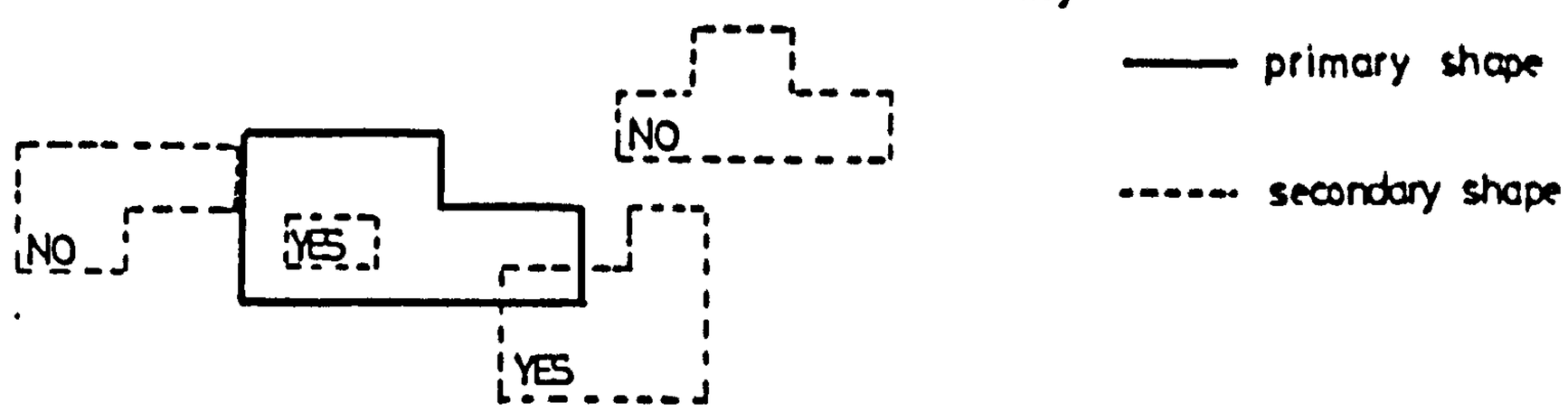
outwards if $-90° < \theta < 90°$

inwards if $180° < \theta < 270°$

The design rule algorithms are described below.

1.1 OVERLAP

This routine finds a secondary shape which overlaps the pre-defined primary shape. An overlap between two shapes exists if the shapes share a common area, for example :-



The algorithm used to find overlapping shapes is as follows :-

1. Set overlap flag to .FALSE.

2. Find next primary shape from shape list : [if finished RETURN]

A 3. Find next secondary shape from data structure : [if finished goto (2)]

4. Do primary and secondary bounding rectangles overlap ?

B YES - goto (5) {overlap still possible, so carry out more detailed analysis}

NO - goto (3) {if the bounding rectangles are separate, then the shapes cannot possibly overlap}

C 5. Find next primary segment in primary shape : [if finished goto (3)]

6. Does primary segment enter secondary bounding rectangle ?

D YES - goto (7) {overlap still possible, therefore carry out more detailed analysis}

NO - goto (5) {if outside bounding rectangle, the primary segment cannot possibly intersect secondary segments}

E 7. Does the primary segment intersect any secondary segments in secondary shape ?

YES - goto (8)

NO - goto (5)

8. Here for overlap
Set overlap flag to .TRUE.
Store secondary shape in shape list
RETURN

Some points to note about the algorithm are as follows :-

At step (A), if the secondary shape identified is a sub-polygon, then the whole polygon must be reconstructed before continuing the test. The OVERLAP test will not fail if only the sub-polygon is considered, but if a future routine tests, for example, the area of the overlapping shape, a false error may be generated due to the fact that only the sub-polygon was stored, and not the whole polygon.

At step (B), if both the primary and secondary shapes are rectangles then the rest of the check can be ignored, since the OVERLAP condition is automatically satisfied. The reason for this is that the coordinates of the rectangle's bounding rectangle are identical to the rectangle's coordinates. Similarly, at step (D), if any primary segment enters the secondary bounding rectangle, and the secondary shape is a rectangle, the OVERLAP condition is automatically satisfied.

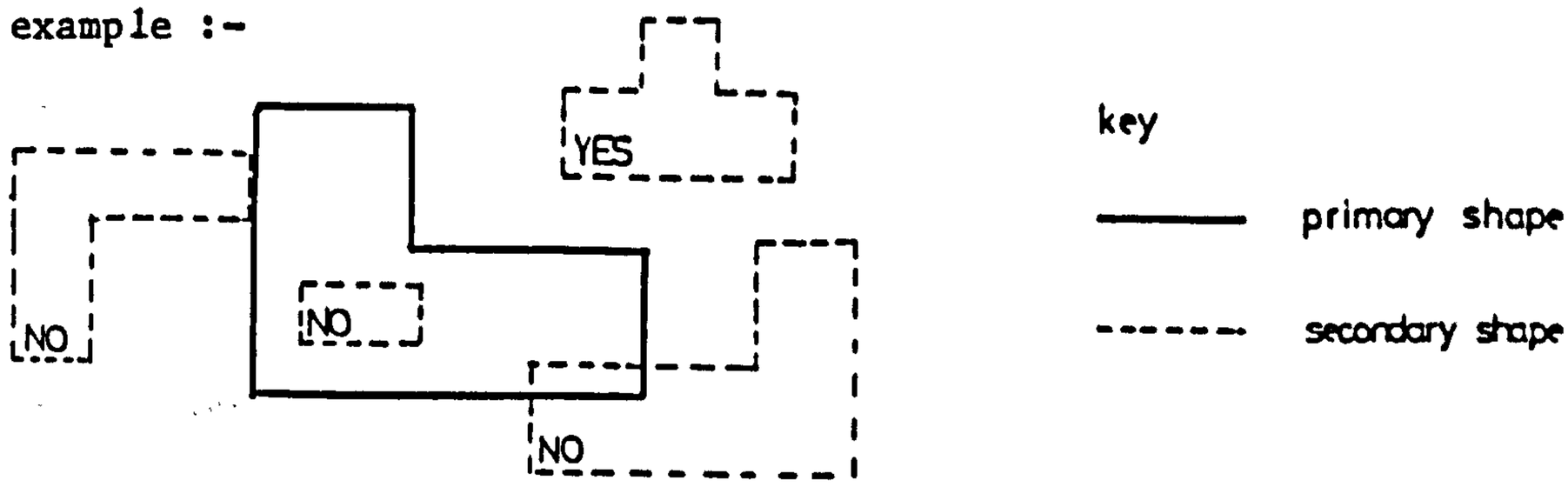
If the secondary shape is totally inside the primary shape, then the OVERLAP condition should be satisfied, but will not be, because no segment crossovers occurred. To catch this special case, a test is performed at step (C) which checks if the bottom left hand corner of the secondary shape is inside (i.e. to the left) of the primary segment. Therefore if no segment intersections were found, and the above mentioned corner was always inside the primary segments, the OVERLAP condition is over-ruled at step (3), and CADIC2 re-directed to step (8).

At step (E), only certain combinations of primary versus secondary segments need be considered. These combinations are as follows :-

Primary segment	Secondary segment
Horizontal	Vertical, Angled
Vertical	Horizontal, Angled
Angled	Horizontal, Vertical, Angled

1.2 SEPARATE

This routine finds a secondary shape which is separate from the pre-defined primary shape. Two shapes are separate if the shapes do not share any common area, for example :-



The algorithm used to find separate shapes is as follows :-

1. Set separate flag to .FALSE.

2. Find next primary shape from shape list : [if finished RETURN]

A 3. Find next secondary shape from data structure : [if finished goto (2)]

4. Do primary and secondary bounding rectangles overlap ?

B YES - goto (5) {possible separation, therefore perform more detailed analysis}

NO - goto (8) {if bounding rectangles are separate, shapes must be separate}

C 5. Find next primary segment in primary shape : [if finished goto (8)]

6. Does primary segment enter secondary bounding rectangle ?

D YES - goto (7) {possible separation, therefore perform more detailed analysis}

NO - goto (5) {if outside bounding rectangle, the primary segment cannot intersect any secondary segments}

E 7. Does the primary segment intersect any secondary segments in secondary shape ?

YES - goto (3) {shapes cannot be separate}

NO - goto (5)

8. Here for separate shapes
Set separate flag to .TRUE.
Store secondary shape in shape list
RETURN

Some points to note about the algorithm are as follows :-

At step (A), if the secondary shape identified is a sub-polygon, then the whole polygon must be re-constructed in case it is needed in other routines. The problem now is that if the polygon is separate, then each sub-polygon will satisfy the SEPARATE condition, and multiple versions of the same polygon will be stored in the shape list. This redundancy will cause excessive processing in future routines, so the approach taken by CADIC2 is to only consider a sub-polygon if it contains the bottom left hand corner of the original polygon. All other sub-polygons are ignored.

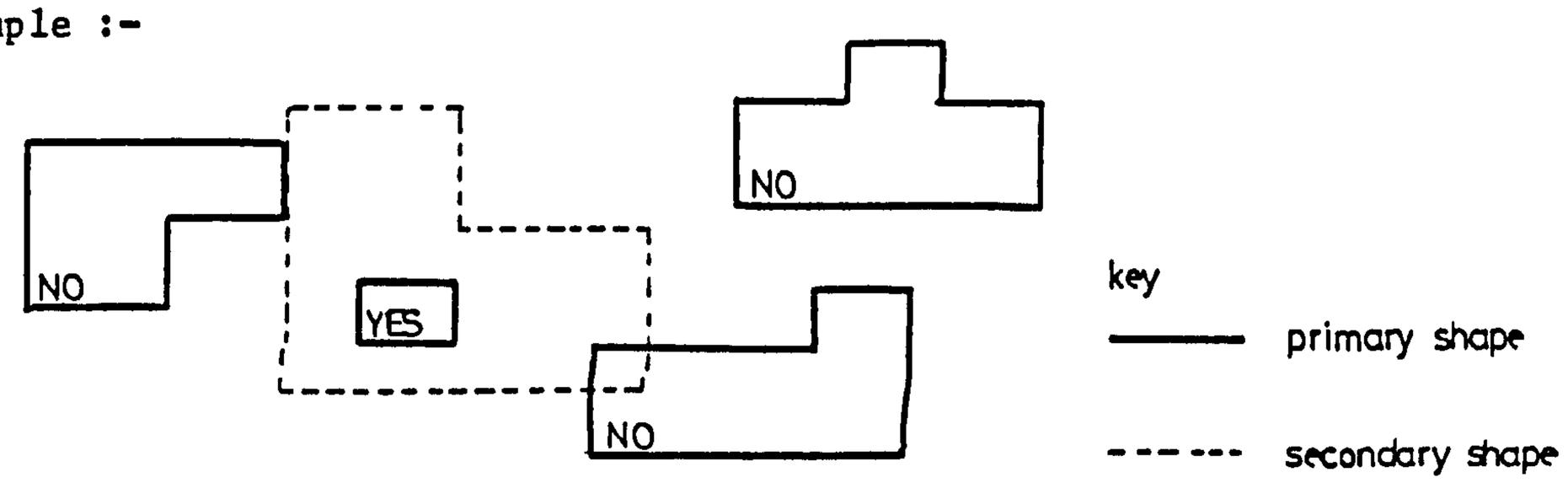
At step (B), the SEPARATE condition automatically fails if the primary and secondary shapes are both rectangles. The reason for this is that the coordinates of a rectangle's bounding rectangle are identical to the coordinates of the rectangle. Similarly, at step (D), any primary segment that enters the secondary bounding rectangle, when the secondary shape is a rectangle means that the SEPARATE condition automatically fails.

If the secondary shape is totally inside the primary shape, then the SEPARATE condition should fail, but will not, because no segment intersections occured. To catch this special case, a test is performed at step (C), which checks if the bottom left hand corner of the secondary shape is inside (i.e. to the left) of the primary segment. Therefore if no segment intersections were found, and the above mentioned corner was always inside the primary segments, the SEPARATE condition is over-ruled at step (8), and CADIC2 re-directed to step (3).

At step (E), only certain combinations of horizontal versus secondary segments need be considered. These combinations are as described in the OVERLAP algorithm.

1.3 ENCLOSED

This routine finds a secondary shape which encloses the pre-defined primary shape. A shape is enclosed when none of its area is outside the enclosing shape, for example :-



The algorithm used is as follows :-

1. Set enclosed flag to .FALSE.

2. Find next primary shape from shape list : [if finished RETURN]

A 3. Find next secondary shape from data structure : [if finished goto (2)]

4. Does the secondary bounding rectangle enclose primary bounding rectangle ?

B YES - goto (5) {possible enclosure, therefore perform more detailed analysis}

NO - goto (3) {if the primary bounding rectangle is not enclosed, then the primary shape cannot possibly be enclosed}

C 5. Find next secondary segment in secondary shape : [if finished goto (8)]

6. Does secondary segment enter primary bounding rectangle ?

D YES - goto (7) {possible enclosure violation, therefore perform more detailed analysis}

NO - goto (5) {if outside bounding rectangle, the secondary segment cannot intersect any primary segments}

E 7. Does the secondary segment intersect any primary segments in primary shape ?

YES - goto (3) {primary shape cannot be enclosed by secondary shape}

NO - goto (5)

8. Here for enclosure
Set enclosure flag to .TRUE.
Store secondary shape in shape list
RETURN

Some points to note about the algorithm are as follows :-

At step (A), if the secondary shape identified is a sub-polygon, then the whole polygon must be re-constructed before the ENCLOSED test can continue. The reason is that the sub-polygon may not enclose the primary shape, whereas the whole polygon does enclose the primary shape. Re-constructing the polygon for every sub-polygon would create multiple copies of the same shape in the shape list. Therefore, the above process is only carried out when the sub-polygon containing the bottom left hand corner of the original polygon is found. All other sub-polygons are ignored.

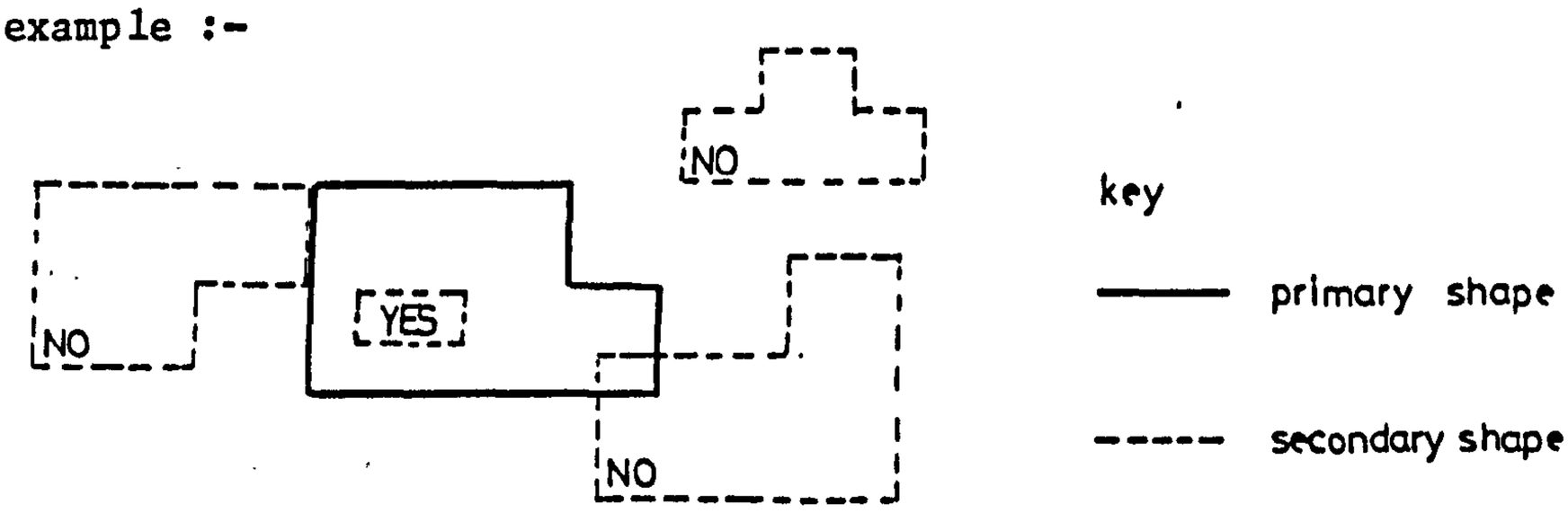
At step (B), the ENCLOSED condition is automatically satisfied if the primary and secondary shape are both rectangles. Conversely, at step (D), if any secondary segment enters the primary bounding rectangle when the primary shape is a rectangle, the ENCLOSED condition automatically fails.

If the primary shape is totally outside the secondary shape, but the primary bounding rectangle is enclosed by the secondary bounding rectangle, then the ENCLOSED condition should fail, but will not, because no segment intersections were found. To catch this special case, a test is performed at step (C), which checks if the bottom left hand corner of the secondary shape is outside (i.e. to the right) of the primary segment. Therefore if no segment intersections were found, and the above mentioned corner was always outside the primary segments, the ENCLOSED condition is over-ruled at step (8), and CADIC2 re-directed to step (3).

At step (E), only certain combinations of primary versus secondary segments need be considered. These combinations are as described in the OVERLAP algorithm.

1.4 ENCLOSES

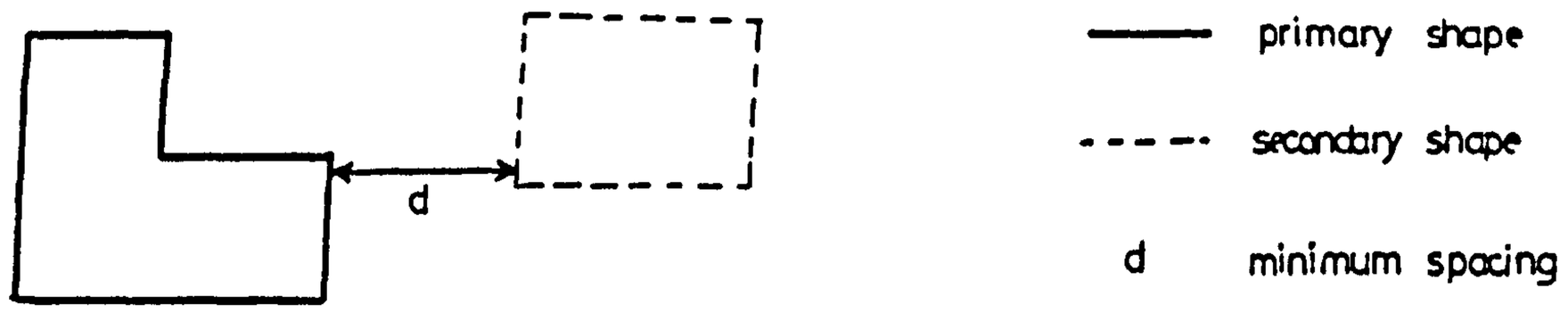
This routine finds a secondary shape which is enclosed by the pre-defined primary shape. A shape is enclosed when none of its area is outside the enclosing shape for example :-



The algorithm used is identical to the ENCLOSED algorithm, except that the roles of the primary and secondary shapes are reversed. Note that this command is generated internally by DRCCAD, and is not available in the manual input language.

1.5 SPACING

This routine takes two separate shapes, and carries out a check to see if the spacing between the shapes is less than a specified minimum, for example :-



The algorithm used proceeds as described below. Note that the terms primary and secondary shape is now used to isolate the two groups of shapes involved. For example, if the spacing test was between the shapes on mask (1) and mask (2), then

the primary shape(s) would relate to those shape(s) in the shape list that were on mask (1), and the secondary shape(s) would relate to the shape(s) in the shape list that were on mask (2).

1. Set spacing violation flag to .FALSE.

2. Find next primary shape from shape list : [if finished RETURN]

A 3. Expand primary bounding rectangle by spacing factor

4. Find next secondary shape from shape list : [if finished goto (2)]

5. Do the primary and secondary bounding rectangles overlap ?

B YES - goto (6) {possible spacing violation, therefore perform more detailed analysis}

NO - goto (4) {if the secondary shape is outside the expanded primary bounding rectangle, then the spacing test is automatically satisfied}

6. Find next secondary segment from secondary shape : [if finished goto (4)]

7. Does secondary segment enter expanded primary bounding rectangle ?

C YES - goto (8) {possible spacing violation, therefore perform more detailed analysis}

NO - goto (6) {if outside bounding rectangle, the secondary segment cannot possibly violate test}

8. Form bumper along outside edge of the secondary segment

9. Do any primary segments from primary shape enter bumper ?

YES - goto (10)

NO - goto (6)

10. Here for violation
Set spacing violation flag to .TRUE.
RETURN

Some notes about the algorithm are as follows :-

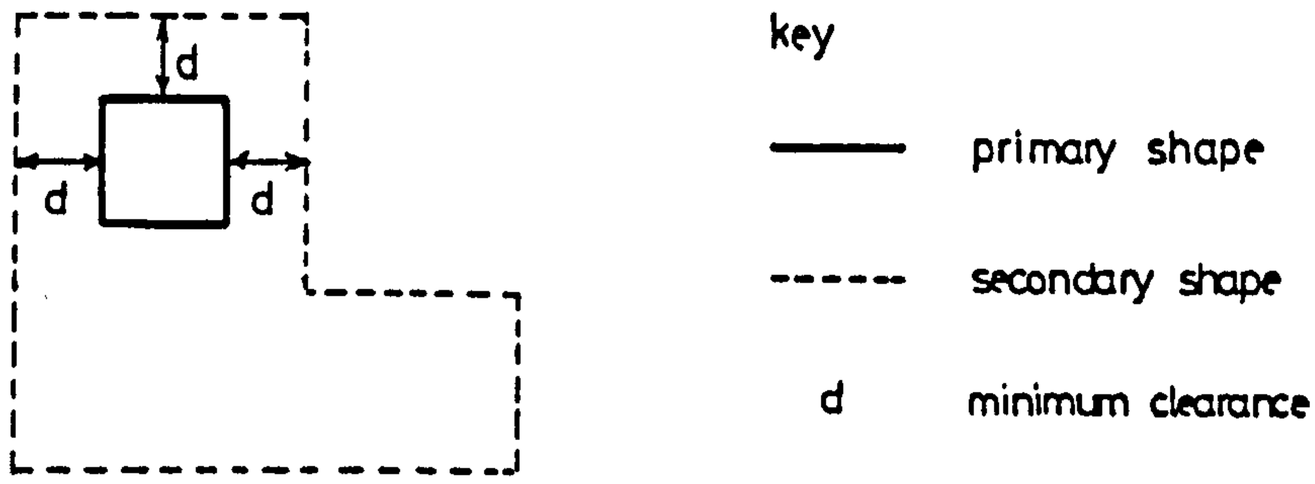
In step (A), the primary bounding rectangle is expanded by the minimum spacing distance 'd'. One of the reasons for doing this is to allow the outcome of the SPACING test to be decided using only the bounding rectangle information. On entering the routine, the secondary shape is known to be separate from the primary shape. If the secondary bounding rectangle is also separate from the expanded primary bounding rectangle, then the spacing distance must be greater than 'd'.

The second reason is that CADIC2 knows that any secondary segments that do not enter the expanded bounding rectangle must be further away than the minimum spacing distance. All such segments can therefore be ignored.

In step (B), if both the primary shape and the secondary shape are rectangles, then the spacing condition must be violated. Similarly, in step (C), if a secondary segment enters the expanded bounding rectangle, and the primary shape is a rectangle, the SPACING condition must be violated.

1.6 CLEARANCE

This routine takes two shapes, the primary enclosed by the secondary, and performs a check to see if the distance between the two shapes is less than the specified minimum, for example :-



The algorithm proceeds as follows. As described in the SPACING algorithm, the terms primary and secondary shape isolate the two groups of shapes involved.

1. Set clearance violation flag to .FALSE.

2. Find next primary shape from shape list : [if finished RETURN]

A 3. Expand primary bounding rectangle by clearance factor

4. Find next secondary shape from shape list : [if finished goto (2)]

5. Does the secondary bounding rectangle enclose expanded primary bounding rectangle ?

B YES - goto (6) {correct clearance possible, therefore perform more detailed analysis}

NO - goto (11) {the secondary shape cannot possibly enclose primary shape with minimum of clearance all round}

6. Find next secondary segment from secondary shape : [if finished goto (4)]

7. Does secondary segment enter expanded primary bounding rectangle ?

C YES - goto (8) {clearance violation possible, therefore perform more detailed analysis}

NO - goto (6) {if outside bounding rectangle, the secondary segment cannot possibly violate rule}

8. Form bumper along inside edge of the secondary segment

9. Do any primary segments from primary shape enter bumper ?

YES - goto (10)

NO - goto (6)

10. Here for violation
Set clearance violation flag to .TRUE.
RETURN

Some points to note about the algorithm are as follows :-

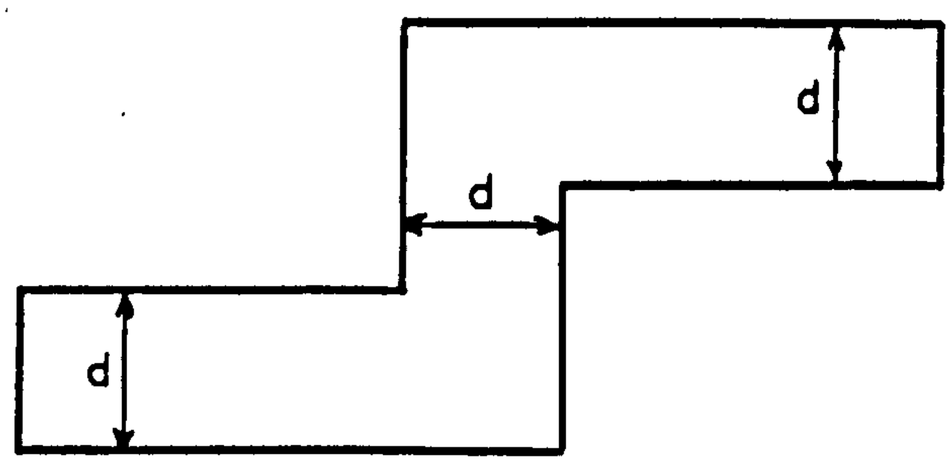
In step (A), the primary bounding rectangle is expanded by the minimum clearance distance 'd'. One of the reasons for doing this is to allow the outcome of the CLEARANCE test to be decided using only the bounding rectangle information. On entering this routine, the primary shape is known to be enclosed by the secondary shape. If the expanded bounding rectangle is now not enclosed by the secondary bounding rectangle, then the clearance between the shapes must have been less than 'd'.

The second reason is that CADIC2 knows that any secondary segments that do not enter the expanded primary bounding rectangle must be further away that the minimum clearance. All segments can therefore be ignored.

In step (B), if the expanded bounding rectangle is enclosed, and the two shapes are rectangles, then the CLEARANCE condition must be satisfied. Conversely, in step (C), if a secondary segment enters the expanded bounding rectangle, and the primary shape is a rectangle, then the CLEARANCE must be violated.

1.7 WIDTH

This routine checks the width of a shape, *against a specified* minimum distance 'd', for example :-



The algorithm is described below. Note that the terms primary segment and secondary segment are now used to isolate the segments within the primary shape. The segment presently being checked is the primary segment, and all the segments between the primary segment, and the start of the shape, are the secondary segments.

1. Set width violation flag to .FALSE.
2. Find next primary shape from shape list : [if finished RETURN]
3. Is shape a rectangle ?

   YES - goto (4)

   NO - goto (6)
4. Check width using the bounding rectangle dimensions
5. Is there a violation ?

   YES - goto (10)

   NO - goto (2)
6. Find next primary segment from primary shape : [if finished goto (2)]
7. Is primary segment travelling outwards ?

   YES - goto (6) {width violation can only be caused by segments travelling inwards}

A NO - goto (8)
8. Form bumper along inside edge of primary segment

9. Do any secondary segments from primary shape enter bumper ?

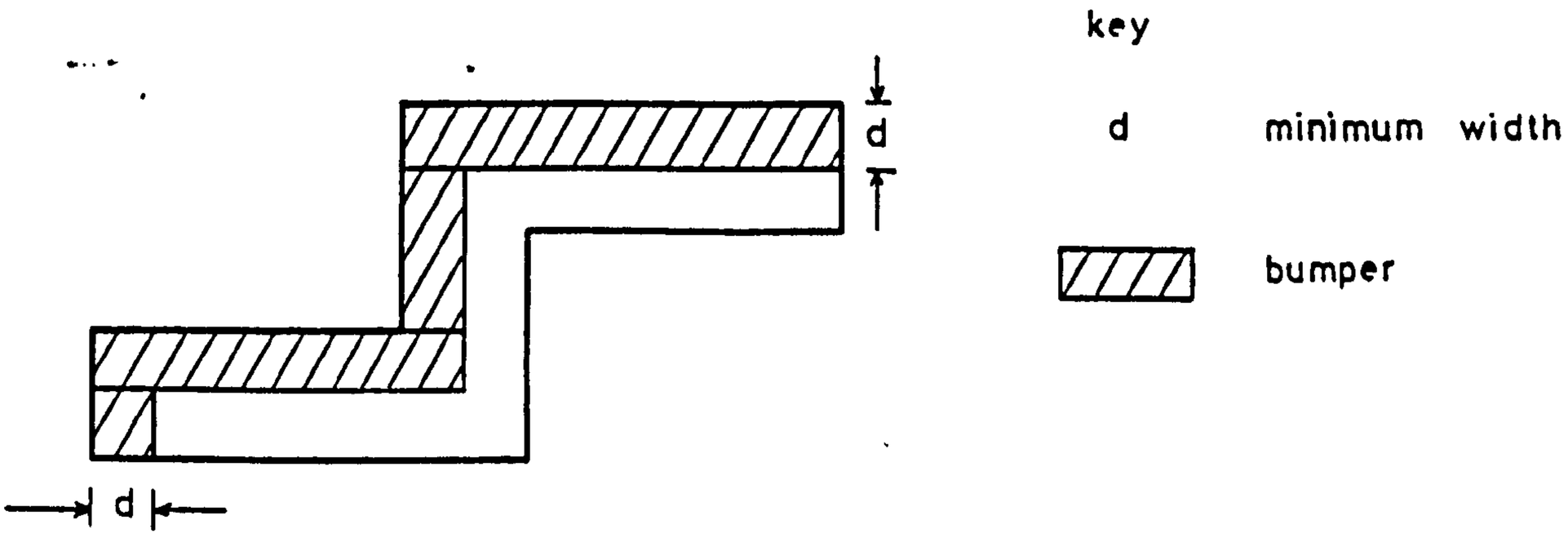
   YES - goto (10)

   NO - goto (6)

10. Here for violation
    Set width violation flag to .TRUE.
    RETURN

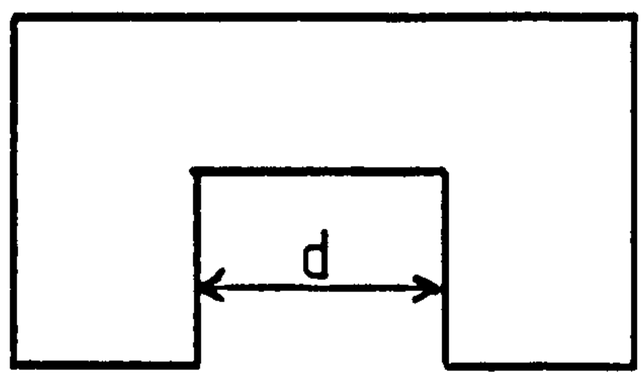
A point to note about the algorithm is as follows :-

In step (A) only in-going segments are checked. The reason for this is that by forming bumpers round the inside of in-going segments, the processing is cut by half, yet all the dimensions are checked. To show this, consider the following shape :-



1.8 INTERLIMB

This routine checks the spacing between limbs of a shape against a specified minimum distance 'd', for example :-



The algorithm proceeds as follows :-

1. Set interlimb violation flag to .FALSE.
2. Find next primary shape from shape list : [if finished RETURN]
3. Is shape a rectangle ?

   YES - goto (2) {interlimb check does not apply to rectangles}

   NO - goto (4)

4. Find next primary segment from primary shape : [if finished goto (2)]
5. Is primary segment travelling inwards ?

   YES - goto (4) {interlimb violation can only be caused by segments travelling outwards}

A NO - goto (6)

6. Form bumper along outside edge of the primary segment
7. Do any secondary segments from primary shape enter bumper ?

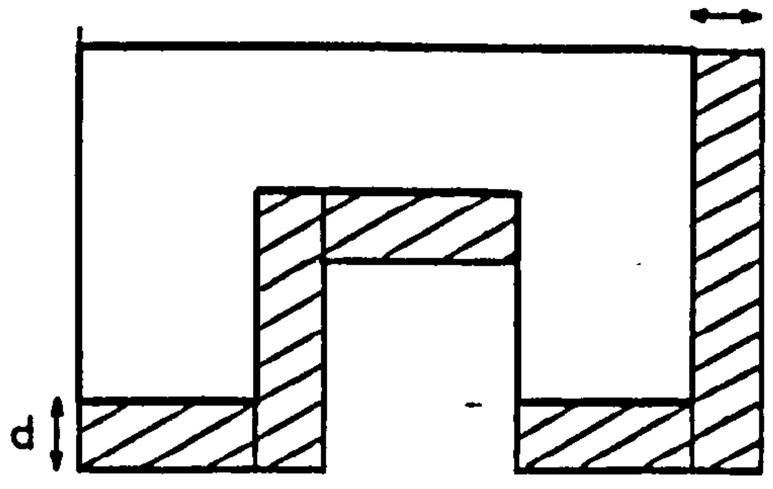
   YES - goto (8)

   NO - goto (4)

8. Here for violation
   Set interlimb violation flag to .TRUE.
   RETURN

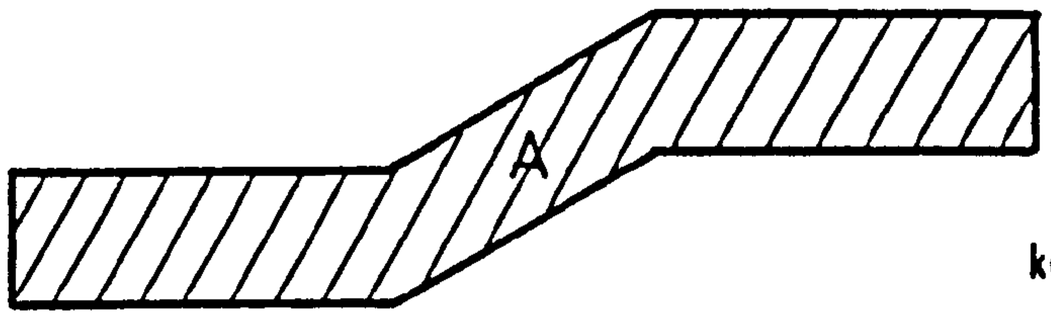
A point to note about the algorithm is as follows :-

In step (A), only out-going segments are checked. The reason for this is that by forming bumpers round the outside of each out-going segments, the processing is cut by half, yet all the dimensions are checked. To show this, consider the following shape :-



1.9 AREA

This routine checks the area of a shape against a specified minimum area, for example :-



key

A = Area

The algorithm is as follows :-

1. Set area violation flag to .FALSE.

2. Find next primary shape from shape list : [if finished RETURN]
   Set area total to zero

3. Is shape a rectangle ?

   YES - goto (4)

   NO - goto (6)

4. Check area using the bounding rectangle dimensions

5. Is area greater than limit ?

   YES - goto (2)

   NO - goto (9)

6. Find next segment from primary shape : *[if finished goto (8)]*

A 7. Calculate incremental area under segment
   Add area to total
   goto (6)

8. Is total area greater than limit ?

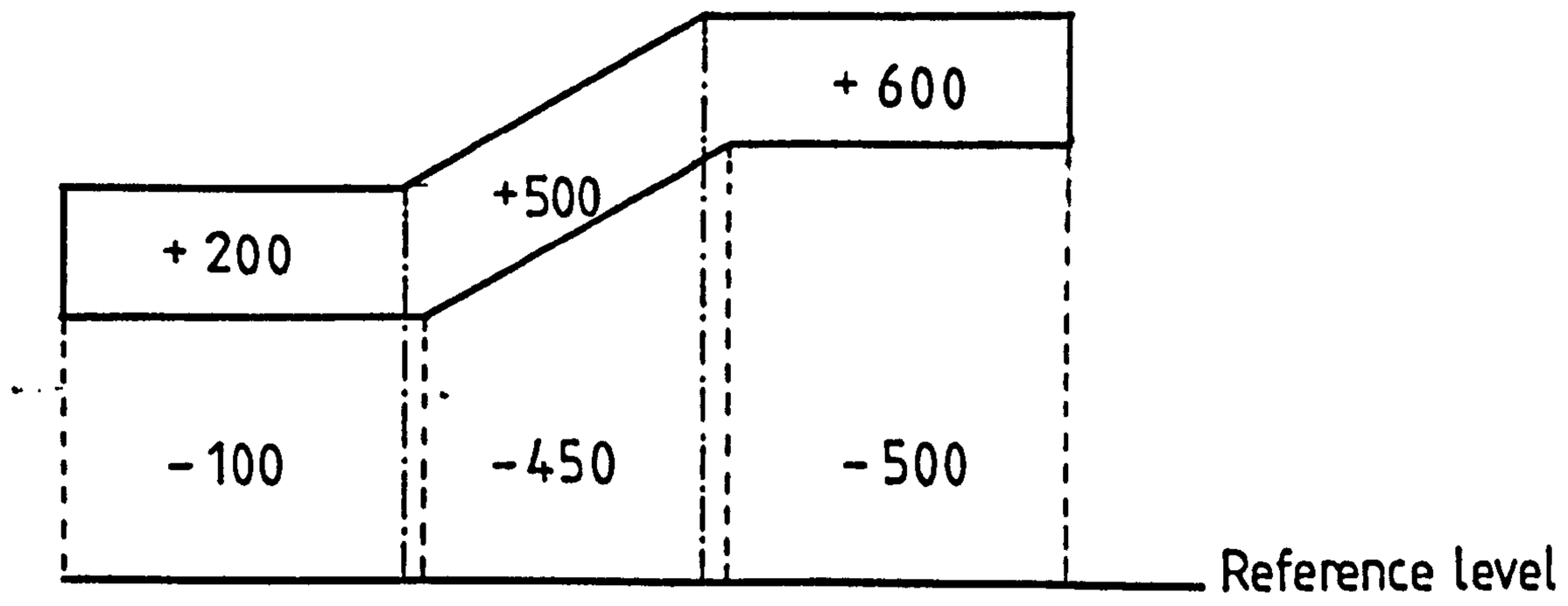
   YES - goto (2)

   NO - goto (9)

9. Here for violation
   Set area violation flag to .TRUE.
   RETURN

A point to note about the algorithm is as follows :-

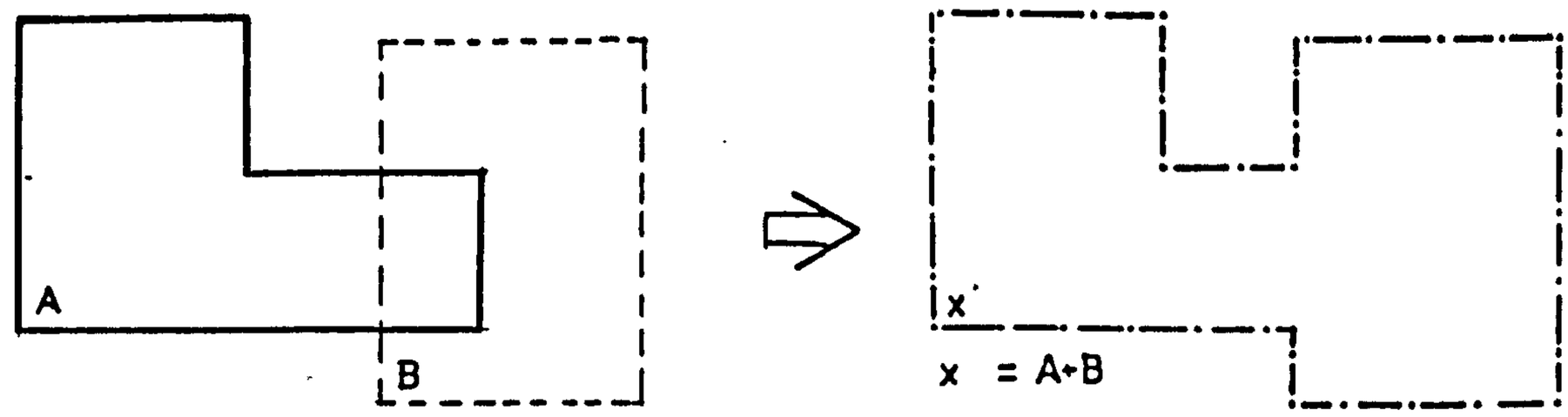
In step (A), we can easily calculate the area under a segment. If a negative area is attached to out-going segments, and a positive area attached to in-going segments, then by calculating the area for each segment, and summing it to a total, the area of the shape can be found. For example :-



Area = -100-450 -500 +600 +500 +200 = 250 units^2

1.10 UNION

This routine forms a new shape which is the logical OR of the two input shapes, for example :-



The algorithms is as follows :-

key
—— primary shape
---- secondary shape
·—·— output shape

1. Find next primary shape in shape list : [if finished RETURN]

2. Find next secondary shape in shape list : [if finished goto (1)]

3. Set output shape information to zero

4. Find next primary segment in primary shape : [if finished goto start of shape]

5. Add primary segment's starting coordinates to the output shape coordinates.

6. Is the output shape closed ?

   YES - goto (10) {output shape now complete}

   NO - goto (7) {continue building up shape}

A 7. Does the primary segment intersect any of the secondary segments travelling out from the primary shape ?

   YES - goto (8)

   NO - goto (4)

8. Re-define the secondary segment that caused the intersection to now start at the intersection point.

B 9. Swap the shape information such that the secondary shape now acts as the primary shape, and vice-versa. Note that the secondary segment re-defined in step (8) will now become the present primary segment.
goto (5)

10. Store output shape in shape list.
goto (2)

Some points to note about the algorithm are as follows :-

The output shape is built up by following the primary shape in an anticlockwise direction until an intersection point is found. The routine must then turn outwards, and follow the secondary shape in an anticlockwise direction until an intersection point is found. The above process is then repeated until the output shape is complete.

In step (A), because the routine always turns outwards at an intersection point, only the secondary segments travelling out from (as opposed to into) the primary shape need be considered. Not only does this rule half the number of checks required, but it automatically keeps the routine moving in the correct direction. Note that the intersection check is carried out in exactly the same way as detailed in earlier routines.

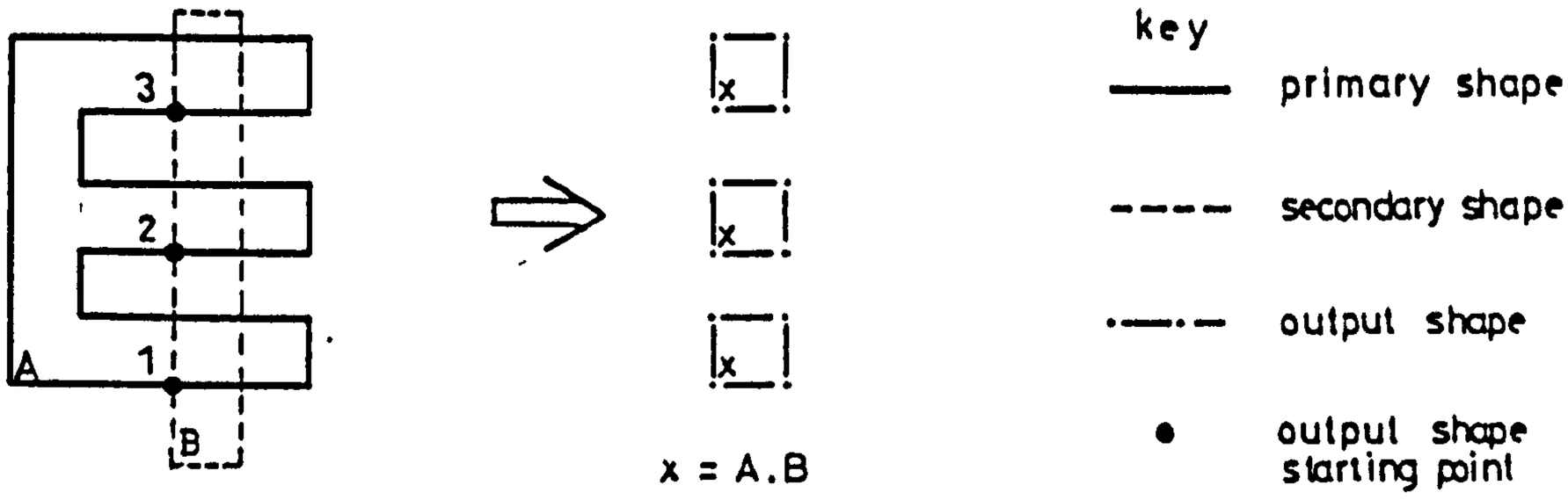
Once an intersection point is found, the secondary shape takes over the role as primary shape and vice-versa. Writing an algorithm to perform this can take two forms :-

1. Produce a two-stage routine, one for when shape (A) is the primary shape, and one for when shape (B) is the primary shape. The algorithm then jumps between stages as the intersection points are encountered.
2. Produce a single-stage routine, but swap the primary and secondary shape information after each intersection point.

CADIC2 uses the latter approach at step (B), because the single-stage routine reduces the software required by half, and the construction of the lookup table in the shape list means that only the two relevant addresses in the lookup table need to be interchanged to effectively swap the shape information.

1.11 INTERSECTION

This routine performs a logical AND function on the two input shapes to produce new output shape(s), for example :-



Note that more than one shape may be produced. The algorithm proceeds as follows :-

1. Find next primary shape in shape list : [if finished RETURN]

2. Find next secondary shape in shape list : [if finished goto (11)]

A 3. Find next starting point of output shape : [if finished goto (2)]

4. Re-define the primary segment to start at intersection point. Initialise output shape by storing its starting point

5. Find next primary segment in primary shape : [if finished goto start of shape]

6. Add primary segment's starting coordinates to the output shape coordinates

7. Is the output shape closed ?

   YES - goto (11)

   NO - goto (8)

B 8. Does primary segment intersect any secondary segments travelling into the primary shape ?

   YES - goto (9)

   NO - goto (5)

9. Re-define the secondary segment that caused the intersection point to now start at the intersection point

C 10. Swap the shape information . Note that the secondary segment re-defined in step (9) will now become the present primary segment. goto (6)

11. Store the output shape in the shape list. goto (3)

Some points to note about the algorithm are as follows :-

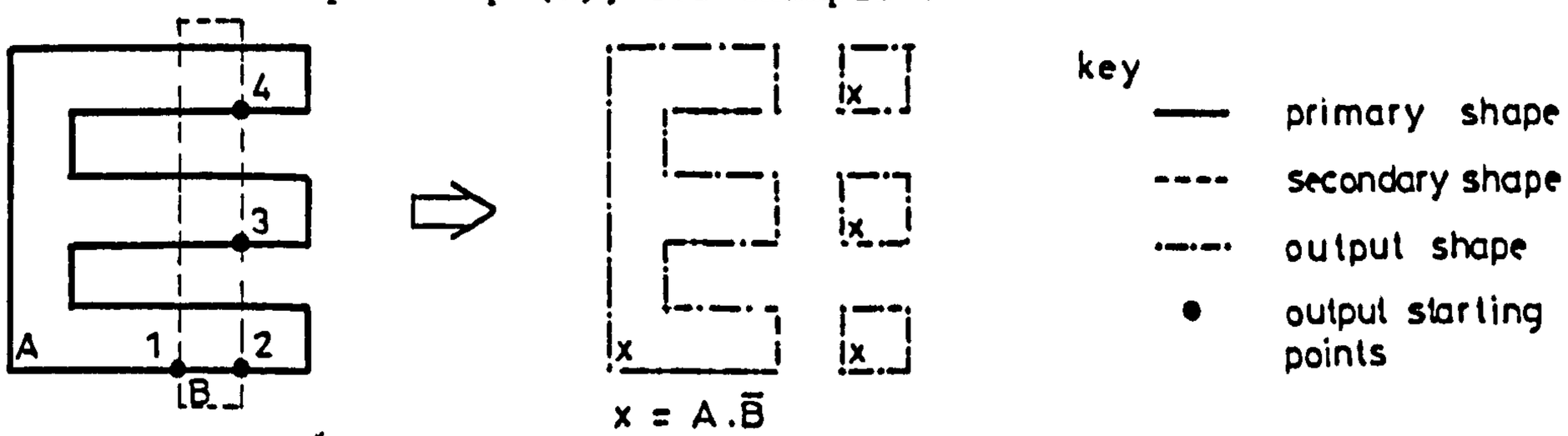
In step (A), finding the starting point of the output shape involves finding the first intersection point that has not already been used to form a previous output shape. These points are shown in the diagram above.

On finding a starting point, the output shape is built up as follows. The routine proceeds along the edge of shape (A) in an anticlockwise direction, until an intersection point is found. The routine then turns inwards, and proceeds along shape (B) in an anticlockwise direction until an intersection point is found. The above process is then repeated until the output shape is complete. In step (B), because the routine always turns inwards at the intersection point, only secondary segment travelling into (as opposed to segments travelling out from) the primary shape need be considered.

At step (C), the routine swaps the shape information for the same reasons described in the UNION algorithm.

1.12 DIFFERENCE

This routine performs a logical NAND operation on the two input shapes, to produce the new output shape(s), for example :-



Note that more than one shape may be produced. The algorithm proceeds as follows :-

1. Find next primary shape in shape list : [if finished RETURN]

2. Find next secondary shape in shape list : [if finished goto (1)]

3. Set 'INC' to clockwise

A 4. Find next starting point of output shape : [if finished goto (2)]
Goto (10)

5. Travelling in the 'INC' direction, find next primary segment in primary shape : [if finished goto start of shape]

6. Add primary segment's starting coordinates to the output shape coordinates

7. Is the output shape closed ?

YES - goto (12)

NO - goto (8)

B 8. Does primary segment intersect any secondary segments travelling into the primary shape ?

YES - goto (9)

NO - goto (5)

9. Re-define the secondary segment that caused the intersection point to now start at the intersection point

C 10. Swap the shape information

11. Reverse the direction of 'INC'.
Goto (6)

12. Store the output shape in the shape list.
Goto (3)

Some points to note are as follows :-

In step (A), finding the starting point of the output shape involves finding the first intersection point that has not already been used to form a previous output shape. These points are shown in the diagram above.

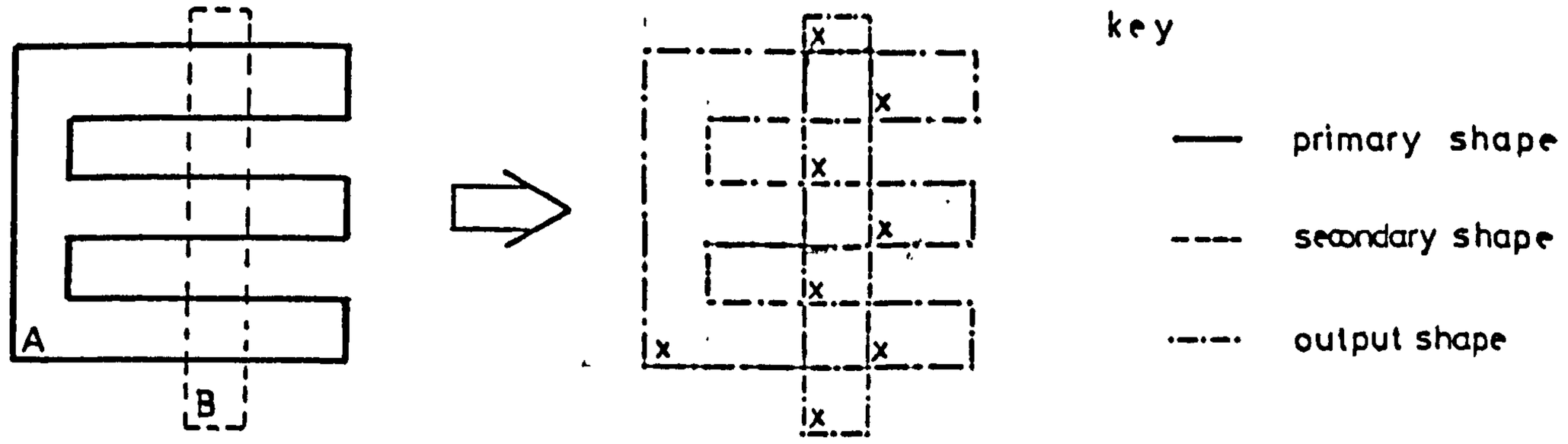
On finding a starting point, the output shape is built up as follows. The routine proceeds along the edge of shape (B) in a clockwise direction until an intersection point is found. The routine then turns inwards, and proceeds along shape (A) in an anticlockwise direction until an intersection point is found. The above process is then repeated until the output shape is complete. Because of this continual reversal of direction, the secondary segments in step (B) are processed in the <u>opposite</u> direction to the direction of the primary segment. This rule automatically ensures that the routine moves in the correct direction.

As described in the INTERSECTION algorithm, because the routine always turns inwards at an intersection point, only secondary segments travelling into the primary shape need be considered.

At step (C), the routine swaps the shape information for the same reasons described in the UNION algorithm.

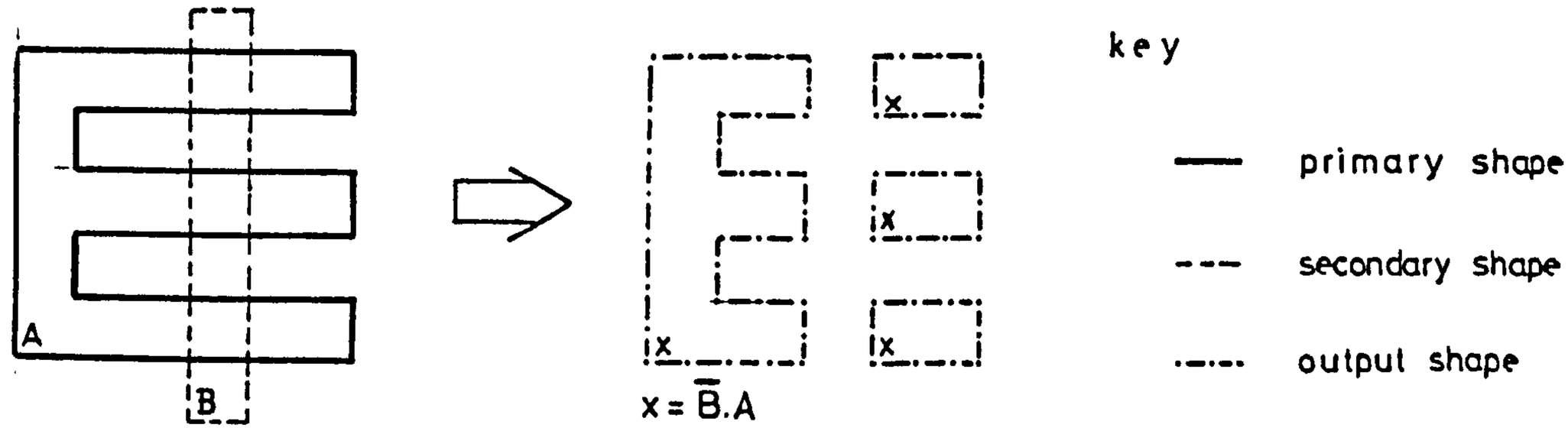
1.13 EXCLUSIVE

This routine performs a logical XOR function on the two input shape to produce new output shape(s), for example :-

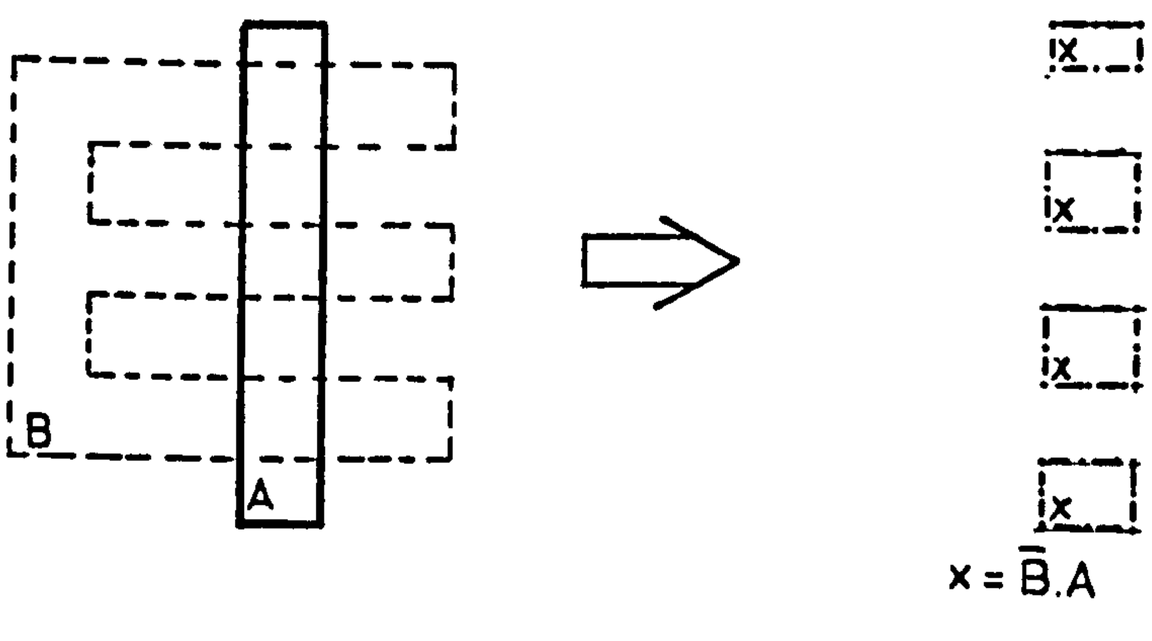


$x = \bar{A}.B + \bar{B}.A$

Note that the EXCLUSIVE operation is very similar to the DIFFERENCE operation. In fact all the output shapes can easily be produced by using the DIFFERENCE routine twice with shapes (A) and (B) defined as below :-



$x = \bar{B}.A$



$x = \bar{B}.A$

PUBLICATIONS

1. Swan G B
"Computer Aids for the Design of Large Scale Integrated Circuits"
Published in Royal Television Society Journal May 1982

2. Swan G B, and Eades J D
"CADIC : An efficient integrated circuit design aid"
Accepted for CAD84 conference, Brighton March 1984

Computer aids for the design of large scale integrated circuits

A report on the 1981 Royal Television Society/John Logie Baird Travelling Scholarship *(sponsored by Radio Rentals Ltd)*

George B Swan

Robert Gordon's Institute of Technology, Aberdeen

Introduction

The economics of IC (integrated circuit) technology advocate the use of high density circuitry. As a result, in the past, IC density has been doubling every two years. Having to repeat the very expensive mask making and fabrication process because the circuit contained errors is obviously unwanted, but increasing the density does increase this possibility. Stringent tests must therefore be carried out on the mask layouts (which are used to control the fabrication process) to ensure that the correct circuit will be produced.

Present day VLSI (Very Large Scale Integration) can now produce silicon chips containing up to 450,000 gates. Even for an average sized layout, a computer may require several hours to complete just the design rule checks (see later). Computer time is not cheap, for example a single run of a design rule checker will typically cost £10,000 to £25,000. The use of such programs must therefore be kept to a minimum.

This article is split into three main sections. Firstly there is a brief description of how a silicon chip is formed, and so hopefully help the reader understand why design rule checks are required. This leads on a summary of some existing computer techniques for the design and checking IC's. Lastly there is a description of the approach taken at Robert Gordon's Institute of Technology, Aberdeen, plus a comparison against other techniques.

John Logie Baird Travelling Scholarship

The Scholarship allowed me to travel to the University of Arizona, which aided my research in two ways. Firstly, the excellent on-campus facilities for IC fabrication allowed me to obtain hands-on experience in IC design and production. Secondly, I was able to carry out a detailed survey of ICMASK, the computer design aid used at the University.

The Scholarship also allowed me to travel to California and visit other Universities, and IC companies in the Silicon Valley region. These visits involved demonstrations of popular design aids, talks to the authors, plus discussions with the users.

The first stage of my research project involves a critique of existing computer design aids for which the John Logie Baird Travelling Scholarship has proved invaluable. Exposure to other design aids has made me more aware of desirable features, which will undoubtably improve the quality of all future work.

IC Fabrication

Integrated circuit fabrication allows hundreds of identical circuits to be produced simultaneously on a single wafer of silicon. By adding various chemical elements to pre-defined areas on the wafer (called *doping* the silicon), it is possible to make transistors, diodes, resistors, and capacitors which form the circuit.

Prior to circuit fabrication, the designer must produce photographic *masks* for each stage of the process (up to 30 masks may be required). A mask consists of patterns of transparent and opaque areas, which will ultimately define which areas of the wafer will be doped, and which areas will not.

The first step in the fabrication process is to protect the silicon by growing a thin layer of silicon oxide over the surface of the wafer. Next, a layer of photographically active material (*photo-resist*) is spread over the oxide, and the mask plate laid on the photo-resist. This sandwich is then exposed to a strong source of ultra-violet light. The radiation causes molecular change in the exposed photo-resist, allowing the unexposed photo-resist to be washed away easily.

Acid is then used to remove the unprotected oxide, leaving the bare silicon once again (termed etching). Note that the photo-resist and silicon are unaffected by the acid. Now the pattern on the mask has been directly transferred on to the silicon.

If the wafer is then placed into a temperature controlled furnace, and fed with for example, Boron gas, the exposed areas of silicon will start to absorb the Boron molecules. Controlling the density of the gas, and the temperature of the furnace, allows very accurate levels of doping to be achieved.

The above process is now repeated using different masks and different chemical elements to produce the individual components. By depositing metal over the entire wafer, and then selectively etching, the components can be connected to form the complete circuit.

The problem with the fabrication process is that in practice, the elements are absorbed into the silicon as fast transversely (along the wafer) as they are

longitudinally (through the wafer). This means that the previously well defined doped areas now contain curved 'walls' which travel underneath the oxide protection layer.

Should two areas be too close together, they may be seen to be separate on the mask, but in fact be joined together in the silicon, so leading to circuit failure. The designer must therefore produce the mask with regard to a set of *design rules*.

In the geometric sense, these rules set a minimum spacing between areas on any one mask, minimum spacing between areas on different masks, the amount of overlap required to ensure connectivity between areas, and so on.

Present Design Aids

As a result of being awarded the John Logie Baird Scholarship, a survey of existing computer design aids was carried out. These aids ranged from fully automatic programs, to digitizing a layout drawn out by hand. For low-volume custom designed chips, the automatic approach is ideal, since the designer simply chooses modules or *cells* from a standard library, and specifies which cells are connected to which. The program then places all the cells and routes the tracks to the best of its ability.

When first introduced, this approach was considered to be the answer to all the designer's problems. Unfortunately, due to the computer's implicit inability to recognize shapes, and lack of ingenuity, the layouts computed (even after using substantial amounts of computer time) consume more silicon area than necessary, and problems in trying to route all the wires are often encountered.

It was soon realized that some human intervention must be included. This led to several approaches in which the designer uses his intelligence to do the design, and leaves the computer to handle all the calculations and tedious work.

One of the off-shoots from this ideology was the symbolic approach, in which the geometric definitions are represented as lines and/or boxes. *Stick diagrams* as they are known have attracted much attention recently, as the simplified diagrams help ease the designer's job. Routines exist to compact these diagrams, and convert them into geometric layouts, but even the best routines have difficulty when processing large layouts. The resulting non-efficient use of silicon area is undesirable, therefore many companies are returning to the geometric approach, to achieve the required density.

At the geometric design level, the designer is manipulating the actual shapes that will apear in the mask layout. The design time is longer, but very compact layouts are possible, which is a necessary stipulation if large-volume production of the circuit is required.

With any design technique, which involves human intervention, the complete layout must be checked using a set of design rules. In general, the design rule checking of a layout is done *off-line* ie as a separate process. Therefore the designer generates the layout, which is passed to the design rule checker, along with a set of design rules. The checker prints out a list of all the violations, and the designer then returns to the design stage and modifies the layout. Correction of one error may require the repositioning of part of the layout, which could introduce new errors. Therefore in practice, this two-stage cycle must be repeated about three to four times before an acceptable layout is achieved.

It would obviously be much better if the design rule checks could be carried out as the shapes were being added (ie *on-line*) so that any violations could immediately be spotted. The problem that has stopped this approach being carried out before is how to complete the checks fast enough, because a user who has to wait for each shape to be accepted will soon become discontented, and hence prone to even more mistakes.

The approach taken at Robert Gordon's Institute of Technology

The aim of research at Robert Gordon's Institute of Technology is to produce a program which provides the user with a full range of facilities to build up and/or modify a mask layout, at the geometric level. Through using a novel data-structure, the designer will have a set of pre-defined design rule checks carried out for each shape added, within the time it takes him to think of his next action.

To tackle the problem of storing the huge amount of data produced in designing a layout, consider the layout as a collection of much smaller areas. All the information in the data structure is connected by a system of pointers, so by knowing the area a shape is in, and the mask it has been assigned to, the shape co-ordinates can be found very quickly.

Area assignment for shapes which lie in more than one area is treated using a new approach. This should drastically cut the number of shapes that must be checked against each other, when carrying out the design rule checks. With such a data-structure, it is hoped to be able to carry out design rule checks two to four times faster than the most efficient techniques around.

At present, the graphics package required to design the layouts has been completed. Through the use of simple commands, the designer can add/modify shapes, move them about, or delete them. Should a collection of shapes be repeated often in the layout, there exist facilities to define the collection as a group definition. Instances of this group can then be added to the layout again through the use of a simple command. All these commands plus many more take the burden off the designer, and leave him to do what he can do best – defining shapes and fitting them together.

Future work will involve adding on-line design rule checking to the existing package. Some time will also be spent carrying out tests so that computer time and memory requirements will be minimised.

Acknowledgements

I would like to thank Radio Rentals for providing the John Logie Baird Travelling Scholarship. I am also indebted to the Royal Television Society for handling all the arrangements.

CADIC : AN EFFICIENT INTEGRATED CIRCUIT DESIGN AID

G. B. Swan and J. D. Eades

Robert Gordon's Institute of Technology, Aberdeen

The CADIC suite of programs to aid integrated circuit design is presented. The most important features of this design aid are high efficiency in data-processing, and on-line design rule checking. CADIC can therefore substantially reduce the design turnaround time normally associated with manual design aids.

Hardware and software details will be given. Emphasis however, is placed on how CADIC's main features were obtained. Experimental results highlighting the performance of CADIC are also presented.

Key-words : integrated circuit design, high efficiency, on-line design rule checking, research in progress

INTRODUCTION

The CADIC (Computer Aided Design of Integrated Circuits) suite of programs allows the user to design manually integrated circuits. This was one of the first types of design aid available, yet it is still capable of producing the most compact layouts. The design turn-around time associated with manual design aids is comparatively long, therefore new techniques to reduce this time are required.

Integrated circuit layouts must be designed with respect to a set of design rules, so that tolerance errors in the fabrication process do not affect the final circuit. In general, layouts are checked after the layout has been designed (i.e. off-line). The combinatorial explosion caused by checking all the shapes against one another means that these design rule checks are very expensive to carry out. Once complete, the layout must be edited to correct the errors, then re-checked. Typically this design-check cycle is repeated three or four times before an acceptable layout is acheived.

Checking the layout as it is being designed (i.e. on-line) would be much cheaper, since a new shape need only be checked against existing shapes. In addition, the layout is correct at all times, therefore doing away with re-runs of the checker. Ideally, the design rule checks should be performed within the time it takes the designer to start adding the next shape. Previous attempts at on-line design rule checking have never achieved this, unless limited to very simple checks [1,2].

This paper describes new techniques to increase program efficiency, such that complete on-line design rule checking can be incorporated into CADIC as a design option.

HARDWARE

A photograph of the SIGMA 5000 'intelligent' workstation used by CADIC is shown in Figure 1. The microprocessor-based GOC (Graphic Option Controller) forms the basis of the system, by monitoring all the data sent to and from the host computer (DEC 2050). Data received from the host is dealt with in one of two ways. Alphanumeric data is routed to the alphanumeric monitor, whereas graphic data is mapped into the GOC's pixel store (4 x 512 x 512 bits), to be displayed on the high quality colour raster-scan monitor.

Similarly, the GOC receives alphanumeric and/or graphic data from the downstream monitors, and sends this data to the host. In this way, each monitor appears to be directly connected to the host, and thus can operate independently of each other.

The SIGMA does have the disadvantage of having only four bit planes to store graphic data. CADIC is therefore restricted to plotting out a maximum of four masks at any one time [3]. However, more modern hardware is now available which would overcome this problem.

SOFTWARE

The CADIC suite consists of four programs :-

1. MANCAD - Manual input language compiler
2. CADIC1 - Interactive design aid
3. DRCCAD - Design rule language compiler
4. CADIC2 - On-line design rule checker

MANCAD, CADIC1, and DRCCAD operate as independent programs. However, MANCAD and CADIC1 must include CADIC2 in the link-list if design rule checking is required.

The CADIC software is written entirely in FORTRAN, except for two machine code routines which handle disc I/O operations.

Because the host computer is time-shared, it was decided to limit the amount of data in memory in the hope that the computer's operating system would favour CADIC. For this reason, CADIC keeps only six pages of disc-based data in memory at any one time, and uses a paging routine implicit in the disc I/O routines to swap data in and out of the memory as required.

Each program in the CADIC suite will now be discussed in more detail.

(a) MANCAD

MANCAD (MANual Computer Aided Design) accepts a manual description of an integrated circuit layout, and converts this description into a data structure readable by CADIC. Note that the data structure may already exist, in which case the new shapes are added to the existing layout.

This type of program is very useful when the SIGMA workstation is not readily available. Layouts, or sections of layouts can be 'coded-up' on paper, then quickly entered into MANCAD using a standard alphanumeric terminal. The workstation is therefore only required to view and/or edit the final artwork.

By on-line design rule checking each shape as it is compiled, MANCAD ensures that all sections of layout added to the data structure will satisfy the predefined set of design rules, just as if the shapes had been added interactively using CADICl.

(b) CADICl

CADICl is an interactive design aid which allows the user to design integrated circuit layouts at the geometric level. CADICl provides around 50 commands, all of which are easy to use and easy to remember.

The most important feature of CADICl is its high efficiency in processing the disc-based layout data. This was made possible by using two new techniques :-

1. Area segmentation
2. Organised group processing

The first technique requires a new form of data structure to store the layout information. CADICl considers the layout as divided up into a series of areas, and associates each shape with an area. Shapes which enter two or more areas are 'polygon clipped' into sub-shapes, such that each sub-shape is associated with only one area. For an example of a 'polygon clipped' shape, see Figure 2.

Therefore if the designer wants to plot out a small section of the layout, CADICl need only consider the shapes associated with the areas inside the plotting window. By tracing through a system of pointers in the data structure, CADICl can quickly find all the shapes associated with a particular area. This high degree of selection greatly reduces redundant searching, which increases program efficiency.

The second technique involves considering the layout group hierarchy in a more global nature, in an attempt to fully utilize the group information while it is in computer memory.

If a layout is to be plotted out, all the shapes in the layout are plotted, then information about the group instances called from the layout are stored in a temporary file. Note that the group instances are not plotted out at this stage. CADICl then goes to the top of the temporary file, identifies the first group instance, then brings the related group definition into memory. All the shapes within the group definition are then plotted out, and any group instances called from the group definition are added to the temporary file.

The temporary file is then searched to see if any other instances of the group definition (presently in memory) exist. If yes, then it is plotted out, and all the group instances added to the file. If no, then CADICl goes to the top of the file, and identifies a new group instance. The above process is then repeated until all group instances in the file are plotted out. In this way, much less page swapping is required, and so program efficiency is improved.

To find out just how efficient CADIC1 is in practice, it was compared against GAELIC [4], a commercially available design aid, known to be efficient. Both design aids were given the same layout to plot out, and results showing the CPU times for each design aid, at variety of window sizes is shown in Figure 3.

Two points are worth noting :-

1. At large window sizes, CADIC1 is less efficient than GAELIC. This is to be expected since CADIC1 carries more overheads in sustaining area segmentation and organised group processing.

2. As the window size (and therefore the percentage of the layout actually required) decreases, so CADIC1 improves its performance over GAELIC. Note that for the size of layout used in the test, most of the design work would be carried out at 15% full layout and smaller, so that the layout could be seen in enough detail. In this situation, CADIC1 is much more efficient than GAELIC.

(c) DRCCAD

DRCCAD (Design Rule Compiler for Computer Aided Design) accepts a description of the design rules required, and converts this description into a data structure readable by CADIC2.

Note that CADIC2 is simply a library of design rule routines. All the information about the design tolerances and how CADIC2 should carry out the checks is stored in this design rule data structure. Therefore after compilation, DRCCAD re-arranges the information in the data structure, so that CADIC2 will have to perform the minimum amount of work to design rule check a newly added shape.

(d) CADIC2

Whenever a shape or group call is added to the layout, it is CADIC2's function to design rule check the shape(s) against the existing layout, within the time it takes the designer to think of his next action. Three main factors have made this possible :-

1. The design rule data structure always ensures that CADIC2 performs the minimum number of operations.

2. The layout data structure is very efficient in finding information about shapes local to the newly added shape.

3. Each routine in CADIC2 has been optimised such that the CPU time required to complete the relevant operation is kept to a minimum.

To test CADIC2, a layout containing around 2000 shapes was designed, and the time taken to design rule check each shape was recorded. Note that a full set of design rules was applied. Too many factors affect the design rule checking time to be able to give an accurate prediction of how long any particular shape will take to be checked, therefore it is better to consider the performance of CADIC2 in a more global nature.

Consider figure 4 which plots out the performance of CADIC2 as the above mentioned circuit is created. There are two points to note :-

1. The time taken by CADIC2 to design rule check a shape increases linearly with the size of the layout. This is a vast improvement over existing off-line design rule checkers, which usually experience parabolic (n^2) performance

2. As can be seen by the graph, CADIC2 seldom required more than 0.5 CPU seconds per shape to complete the checks. More typically, CADIC2 required only around 0.2 CPU seconds per shape. Therefore, CADIC2 can perform on-line design rule checking well within the time it takes the user to start adding a new shape.

Future tests with CADIC2 will involve much larger circuits, but it is expected that the time to design rule check a newly added shape/group call will rise only slightly above the previously mentioned results. This is largely due to the fact that by using area segmentation in the layout data structure, only the shapes in the present area need be considered, regardless of how many other areas have previously been filled.

CONCLUSION

The CADIC suite of programs to aid integrated circuit design has been presented. The most important features of CADIC are high efficiency, and on-line design rule checking.

Logistics, backed up with experimental results are also presented, confirming two points :-

1. CADIC is very efficient at data processing, especially when small sections of layout are considered.

2. CADIC can perform complete on-line design rule checking within the time it takes the designer to start adding the next shape.

Future work will involve continual assessment of CADIC's efficiency, plus application of on-line design rule checking to much larger circuits.

ACKNOWLEDGEMENTS

The authors wish to acknowledge the support of the Science and Engineering Research Council in this work. In addition, the assistance of Mr. B. Davidson, RGIT in the preparation of the diagrams is acknowledged.

REFERENCES

1. Carmody P, Barone A, Morrell J, Weiner A, and Hennesy J
"An intractive graphics system for custom design"
17th Design Automation Conference 1980 pp 430 - 439

2. Smith T F, and Woods B J
"Poligon - An interactive graphics design tool"
Computer Aided Design conference 1980 pp 31 - 37

3. Eades J D
"The use of color graphics in integrated circuit layout design"
European conf. on Electronic design automation 1981 pp 98 - 101

4. Eades J D
"The design of an interactive computer system
for microelectronic mask making"
Ph D Thesis 1976

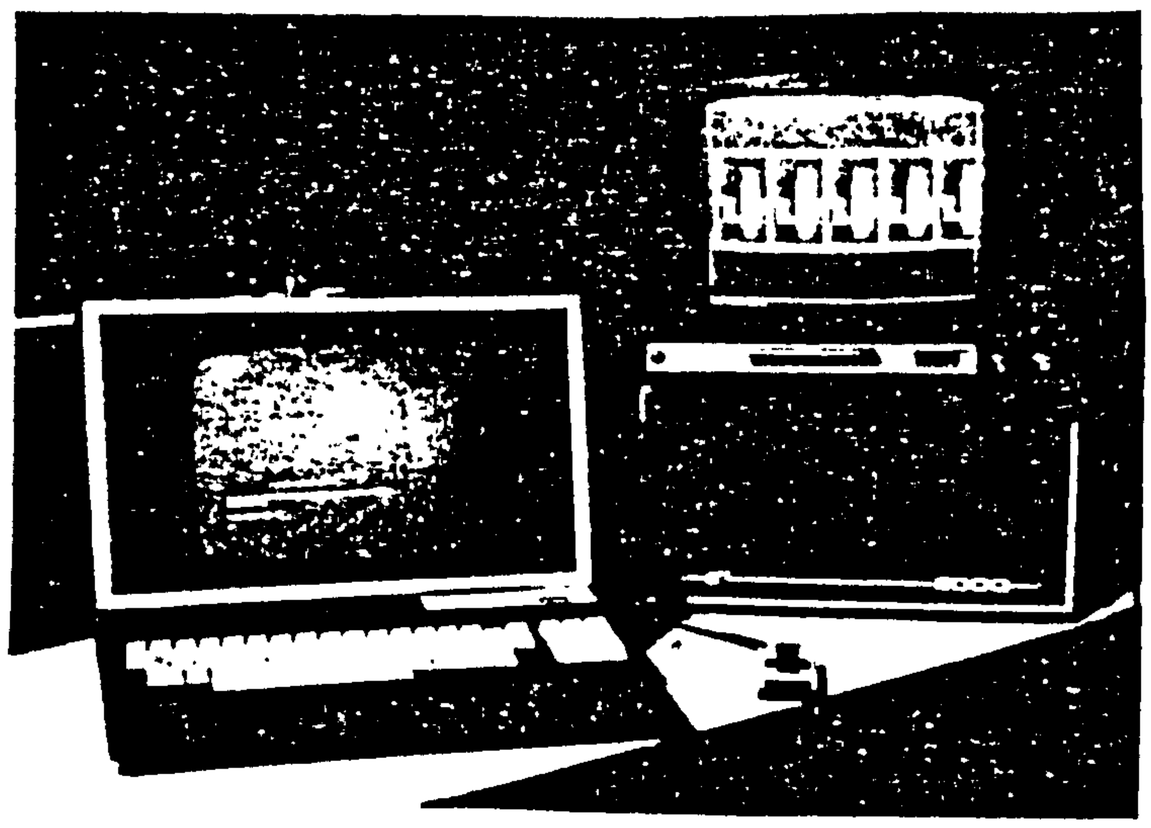


Figure 1 The SIGMA Work Station

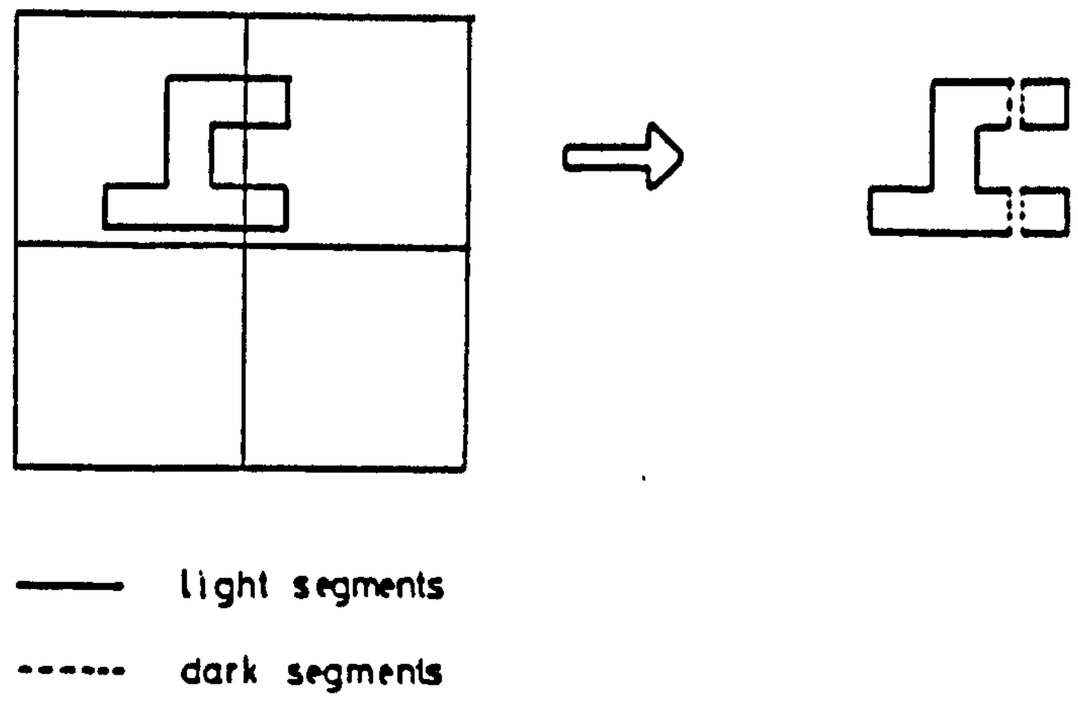


Figure 2 Polygon Clipping

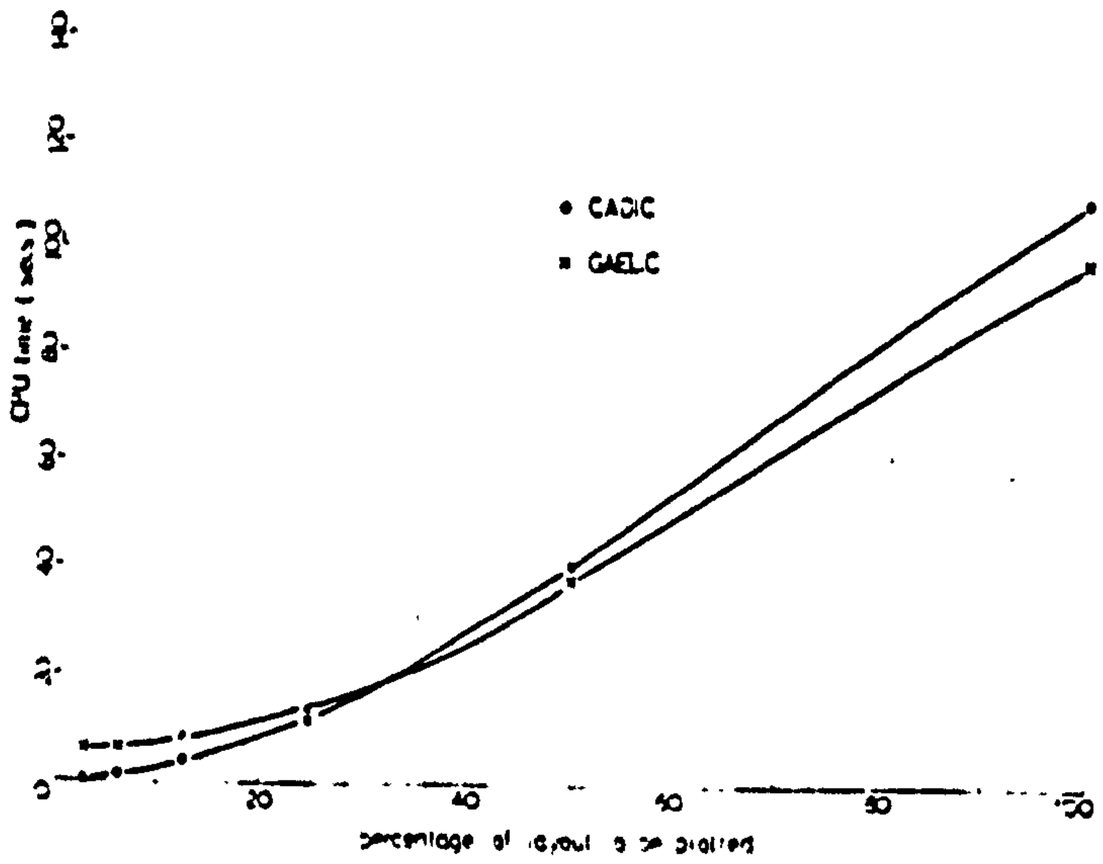


Figure 3 CADIC1 V GAELIC

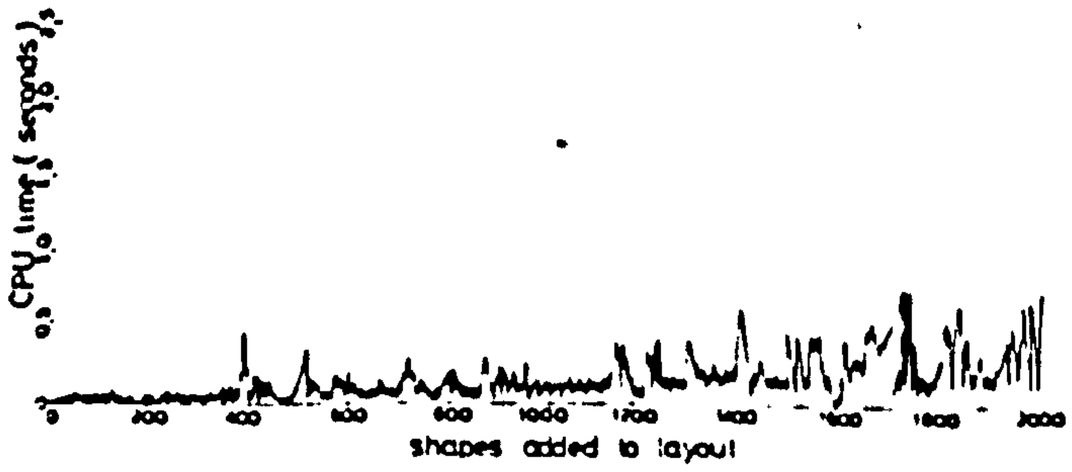


Figure 4 Performance of CADIC2