# ART: A HIGH LEVEL LAYOUT SPECIFICATION LANGUAGE*

D.W. KNAPP

DIGITAL INTEGRATED SYSTEMS CENTER REPORT

DISC/83-4

DEPARTMENT OF ELECTRICAL ENGINEERING-SYSTEMS
UNIVERSITY OF SOUTHERN CALIFORNIA
LOS ANGELES, CALIFORNIA 90089-0781

MARCH 1983

*This work was supported in part by the National Science Foundation under grant ECS-8005957 and by the Department of the Army, Ft. Monmouth, New Jersey, under contract no. DAAK20-80-C-0278.

Table of Contents

## 1. Abstract

ART is a Pascal-based language that provides a high-level interface between an IC designer and the low-level details of the Caltech Intermediate Form (CIF) IC layout specification language. Most of the CIF primitives have direct counterparts in ART; furthermore, ART implements certain design aids not found in CIF, as well as providing the designer with the capabilities of a general-purpose programming language.

## 2. Introduction

The Caltech Intermediate Form (CIF) IC layout specification language [1]is a widely accepted way to transfer layout information from the designer to the fabrication facility. Unfortunately, CIF is difficult to write and read; ART is a "designer-friendly" interface to CIF.

ART is based on the Pascal programming language [3]. The form it takes is that of a set of predefined procedures and functions that the user can call; the user can also define his/her own procedures and functions. Many, but not all, of the predefined functions of ART are direct counterparts of CIF constructs. The output of ART is a CIF file, which can be plotted and/or sent to a fabrication house.

ART is not an interactive program. Its structure is such that it is most easily used in a batch mode, i.e. the user first modifies the ART skeleton, then compiles and runs it. None of the predefined functions and procedures is designed for interactive use; the user who wishes to have an interactive dialogue with ART will have to define the programming constructs necessary to implement the interaction. While this could be done, the implementation of more than a minimally useful interactive program would require a Pascal interpreter to deal with any of the Pascal programming constructs that were required.

Such layout systems as Caesar [2] would be moderately easy to emulate using a minimally interactive version of ART; however, such systems' major weakness is in their lack of high-level programming facilities, which is where a minimally interactive ART would also be weakest.

There are no plans to extend ART to interactive use at this time. The way in which ART is used, therefore, is to modify the skeleton, compile it, and run it: its output is the CIF file.

The modifications that must be made to produce nontrivial designs consist of adding a program body and such user-defined data structures, procedures, and functions as the user thinks necessary, to the ART skeleton. For example, suppose that the user has used the procedure DEFINE to define a memory cell in the ordinary way (explained in detail later). Having done so, the designer then wants to create an array of memory cells, which he will call a register. An iterative loop is used:

```
define('register');
for  i:=1  to cardinality do
begin
    draw('memcell ',xcoordinate,ycoordinate);
    ycoordinate:=ycoordinate + verticaloffset;
end;
endef;
. . .
```

The result of this is that there is now available to the user a cell called 'register', which consists of a column of memory cells. Another loop could then be used to construct a two-dimensional array of registers in the same manner.

If, on the other hand, a number of arrays of registers were desired, each different in size from the others, it would probably be best to implement the register definition as a procedure, with a parameter corresponding to the number of registers in the array, and another parameter corresponding to the number of bits in a register:

```
procedure register_array(words, bitwidth: integer);
```

```
var i,j: integer;
begin
   for i:=1 to words do
      for j:=1 to bitwidth do
         draw('memcell ',cellwidth*(i-1),cellheight*(j-1));
end;
. . .
```

Such a procedure could be used as follows:

```
define('CPUregs ');
register_array(8,16);
endef;
```

Which would have the effect of defining a cell called 'CPUregs', consisting of eight sixteen-bit registers, which could then be called as part of another definition.

After the design has been completely inserted into the ART program, the program must be compiled and executed. Compilation is done with the aid of a standard Pascal compiler, which will not be discussed here, other than to note that all of the ordinary Pascal conventions and syntax will be enforced in the usual way. No particular dialect of Pascal is enforced by ART; it adheres throughout to the standard defined in [1].

When the compiled code is loaded for execution, three file names will be required. One, called 'TMPFIL' inside the source program, is strictly for the use of the program, and should be deleted by the programmer after execution of the program, because it is of no interest to the user. The other files used by ART are the file OUTPUT, which is used to communicate with the user during program execution, and the file called CIF within the program, which is where the CIF representation of the design will be recorded by ART.

## 3. Primitive Commands: Layer, Box, and Wire

The layout language ART, as distinct from the program that implements it, has a number of commands built into it. These commands are procedures and functions in the implementing program; e.g. the command 'define' mentioned above is implemented as a procedure 'define' in the program. The commands available to the user will now be discussed.

The primitive commands are used to define the elementary objects that compose a callable cell. These commands may not be used outside of a cell definition; if they are, an error message will be sent to the user via file OUTPUT, and the command will be aborted. This is not as restrictive as it seems: everything on a chip is part of a global cell called 'chip', and therefore if an object does not fit in any other cell, it will at least fit into that one.

The command LAYER is used to specify a layer, in which primitive objects will be implemented until next time the command LAYER is called. For example, the command sequence

```
LAYER(POLY); { sets current layer=polysilicon }
{ primitive objects }
LAYER(DIFFUSION); { sets layer=diffusion }
{ other primitive objects }
        .  .  .  .
```

will first set the current layer to be poly; the primitive objects following this command ( which might be wires, boxes and so on ) will all be implemented in the polysilicon layer. When the next LAYER command is encountered, the layer is changed to diffusion; thenceforth the primitive objects will all be implemented in the diffusion layer.

The allowable arguments for the command LAYER are: Note that most of the allowable names have alternates as an aid to the programmer.

Another primitive command is the BOX command. It causes a rectangular box

| ARGUMENT | LAYER SPECIFIED |
|---|---|
| POLY, RED | polysilicon |
| DIFFUSION, GREEN | diffusion |
| METAL, BLUE | metal |
| CUTS, BLACK | contact cuts |
| GLASS | overglass |

to be drawn in the current layer. The size and position of the box is determined by the arguments to the command:

```
BOX(x1,y1,x2,y2);
```

specifies a box with two diagonally opposed corners at the points (x1,y1) and (x2,y2). The relative positions of the two corners specified is immaterial; i.e. x1 may be either greater or less than x2, and similarly with y1 and y2. Note that the box always has two sides parallel to the x-axis and two sides parallel to the y-axis; if the user wants an oblique box some other command must be used ( There are two good ways to create such oblique boxes ).

The four arguments to the BOX command may be either real or integer valued numbers.

The third primitive command is WIRE. This command initializes a wire in the current layer. After initialization, the wire may be made to follow a path by means of the commands X, Y, DX, DY, XY, and DXDY. At least one of these commands must be called after the WIRE command or an error will result; furthermore, the command after the WIRE command must cause a wire of nonzero length to be drawn.

The command WIRE has the following format:

```
WIRE( width, x1, y1);
```

where the width parameter is the width of the wire to be drawn, and x1 and y1 are the coordinates of the initial point of the wire. This initial point is at the center of the end of the wire; e.g. a wire of width three, centered on the origin, would be called by

```
WIRE(3,0,0);
```

and would then have to be drawn along a path by means of the other wire commands.

The commands X and Y extend the wire along a path parallel to the x-axis and the y-axis respectively. The length of the extension is determined by the difference between the original coordinate and the argument of the command, e.g.

```
WIRE(3,0,0); { initiate a wire of width 3 }
   X(20); { draw to the point (20,0) }
```

will cause a wire of width three, originating at the origin, to be drawn along the x-axis to the point (20,0). This wire is exactly equivalent to the wire drawn by

```
WIRE(3,20,0); { originates at (20,0) }
X(0); { draw along x-axis to origin }
```

The procedure Y is analogous; for example, the command sequence

```
WIRE(3,0,0) { initiate wire at the origin }
  Y(10); { draw along y-axis to (0,10) }
```

causes a vertical wire of length 10 to be drawn.

All of the wire-drawing commands may be concatenated to draw wires of complicated shape; such concatenation may involve any number of points. Thus

```
WIRE(3,0,0);
   X(20);
   Y(20);
```

will cause a wire to be drawn that looks a little like a reversed capital "L". Note that each command updates the position of the point that corresponds to the "end" of the wire and draws a straight segment between the old "end" and the new "end".

The procedures DX and DY are similar to X and Y, except that instead of setting the absolute X or Y coordinate, they take as an argument an increment that is added to the current coordinate. Thus

```
WIRE(3,0,0);
  DX(20);
  DY(20);
```

Would cause the same reversed "L" to be drawn as in the previous example.

The procedures XY and DXDY are used to draw slanted wire segments, with XY using its arguments as the values of the new end point, and DXDY adding its arguments to the old end coordinates. For example,

```
WIRE(3,0,20); { originates at (0,20) }
 DX(20);{ horizontal run to (20,20) }
 DXDY(-20,-20); { oblique to origin }
   DX(20); { horizontal to (20,0) }
```

Would result in a wire shaped like a capital "Z", with its lower left corner at the origin. The same wire could be drawn:

```
WIRE(3,0,20); { same initial point }
         X(20);
   XY(0,0) { oblique to origin }
  X(20); { horizontal to (20,0) }
```

It is usually a matter of preference whether or not to use X or Y instead of DX or DY; however, there are times when one or the other can be very useful.

Another wire command that can be used to modify the characteristics of the wire is the procedure W, which terminates the old wire and initiates a new one at the endpoint of the old wire, but with a different width; this is, in effect, a call that changes the width of the wire. Its argument is a real number or an integer, which is the width of the new wire. Thus

```
              WIRE(3,0,0); { initiate a wire at (0,0) }
                  X(10); { horizontally to (10,0) }
             W(5); { width of the wire is henceforth 5 }
                  Y(20); { vertically to (10,20) }
```

Will result in another reversed "L", that will have a much wider upright than its foot.

The procedure Z is used to change layers without explicitly calling a new wire. In effect, what it does is to terminate the old wire, draw an appropriate contact, and then begin a new wire on the new layer. The allowable transitions are red/blue and green/blue, but not red/green, which would be a trap for the unwary.

The procedure STUB is used in much the same manner as WIRE, but with some additional parameters. It is used when a particular point in a cell is to be named for later reference; the data required in the procedure call are kept in a record that is accessible to the programmer. These data are:

1. The name of the point, a packed array of characters exactly 8 characters long;

2. The layer in which the wire is implemented; this is a variable of type LAYERTYPE, and may take any of the values that are acceptable to the LAYER command;

3. A direction, either NORTH, WEST, SOUTH, or EAST, out of which a wire may come to make contact with the stub;

4. Three real-valued numbers, which correspond to the width and to the originating coordinates of the stub.

For example, the command sequence

```
STUB('DUMBSTUB',POLY,NORTH,3,0,0);
                      X(20);
                      Y(20);
```

Would cause the same reversed "L" to be drawn as in the previous examples, and would also create a record of the position, name, layer and direction from which this point could be approached in order to make contact to it. This record is stored as part of the record of the cell currently being defined, and the information about the stub can be accessed by calling the procedure BRISTLE with the name of the stub as a call-by-value parameter, and suitable variable names as call-by-reference result variables. For example, if a stub called 'VDD ' were to be found, and its characteristics put into suitable variable locations called LAIR, DYRECTION, WID, XCOOR, and YCOOR, the call

```
BRISTLE('VDD ',LAIR,DYRECTION,WID,XCOOR,YCOOR);
```

Would be used. Note that since the name of the bristle is a packed array of eight characters, it is necessary to pad the name with blanks; note also that the position of the blanks is important.

## 4. Cell Operations: Define and Draw

In order to define a cell, the command DEFINE must be used. This command writes the 'DS' command onto the CIF file, converts the cell name used by the programmer into an ID number suitable for CIF, and initializes some records having to do with the cell being defined. The cell name used as an argument by DEFINE must be a packed array of exactly eight characters, with single quote marks at either end. If the desired cell name is shorter than eight characters, it must be padded with blanks or other nulls. For example, if the designer wanted the cell to be named 'adder', correct syntax would be:

```
DEFINE('ADDER...');
```

Where the user is using the string '...' as padding. This will not be the same cell as the cell created by the command

```
DEFINE(' ADDER');
```

In which the name of the cell has padding in a different place. At the beginning of the definition a layer must be defined; otherwise the result of any primitive calls will be unpredictable. This is because the current layer is restored after calls and definitions; the definition is a strictly local environment.

Cells that have already been defined may be called inside a definition, to any level of nesting. However, definitions may not be nested; thus there must be a call to the procedure ENDEF ( which terminates the current definition ) between every pair of DEFINE's.

The command DRAW given in the example above is one of six ways a cell can be called; each of these six applies a different transformation to the coordinates of the called cell. DRAW, the simplest, draws the called cell at a translated location, but does not apply any other transformation to the cell. This is done by adding the horizontal component of the translation to the x-coordinates of all the objects specified in the definition of the called cell, and the vertical component to all of the y-coordinates. Thus if there was a cell called 'RBCELL ', which was a red-blue contact cell of minimum dimension, and which was centered about the origin, the call

```
DRAW('RBCELL ', X, Y);
```

Would cause a red-blue contact to be drawn at the coordinates (X,Y). The parameters X and Y may be either integer or real-valued, and may also be function values. Thus

```
DRAW('RBCELL ', F(ALPHA,G(BETA))
```

Is perfectly legal as long as the functions F and G are either real or integer valued.

The procedures DRAWMX and DRAWMY are similar to DRAW, except that before translation, they are mirrored; in the case of DRAWMX, about the Y-axis, and in the case of DRAWMY, about the X-axis. This is accomplished by multiplying all of the cell's x-coordinates ( in the case of DRAWMX ) by -1, and then adding the translation values. The action of the procedure DRAWMY is quite similar, multiplying the y-coordinates by -1 before adding the translation. For example, suppose the user had defined a cell called 'OFFSET ' which contained a red-blue contact at the coordinates (2,2). The call

```
DRAWMX('OFFSET ',0,0);
```

Would result in the RB contact being drawn at the coordinates (-2,2), while the call

```
DRAWMY('OFFSET ',0,0);
```

Would draw it at (2,-2). The call

```
DRAWMX('OFFSET ',10,20);
```

Would first mirror the cell to (-2,2), then translate it to (8,22).

Two other cell call commands are DRAWTX and DRAWTY, which are rather like DRAWMX and DRAWMY, except that the translation of the cell is done before the mirroring. Thus the same cell, when called by

```
DRAWTX('OFFSET ',10,20);
```

Would appear at the location (-12,22). Note the difference between the result of this call and the DRAWMX call with the same parameters. DRAWTY behaves in an analogous fashion.

The last cell-calling procedure is DRAWROT, which first rotates the cell about its origin, and then translates it. The amount of rotation is specified by two real-valued parameters, which define a vector. The cell is rotated until its x-axis is parallel to the rotation vector; then it is translated.

DRAWROT therefore takes five parameters: the cell name, the two translation coordinates, and the two rotation coordinates. For example, the same cell 'OFFSET ' could be called:

```
DRAWROT('OFFSET ',5,5,1,1);
```

Which would result in the RB contact being drawn rotated 45 degrees, centered at a point close to (5,8). ( Note that the coordinates for such cells can become inconvenient; for this reason the predefined contact cells are centered about the origin, as will be described later.)

## 5. Bounding Boxes

A bounding box is a rectangle associated with each defined cell; it is the smallest rectangle that has two sides parallel to the x-axis, and inside which the entire cell can be contained. It is useful when laying out large designs on slow I/O devices, because only the bounding box need be drawn in many cases; this suppresses the details of the called cell, makes for a cleaner picture, and speeds up the image and CIF processing tasks. These bounding boxes are accessible to the programmer as part of the records describing each defined cell.

There are two ways in which the bounding boxes can be used. The first is as an automatic-layout aid; the programmer can use the box records to determine cell spacings, the size of arrays of cells, and so on. The chief drawback of this use is that the bounding boxes can be quite a lot larger than the cells they surround; therefore, it is possible to waste a good deal of area by using them blindly. For example, suppose that a cell was shaped like a capital "T"; the bounding box for that cell would contain a good deal of unused space at the lower corners, that could be used to advantage by an alert programmer. This problem is usually worst when cells are rotated to oblique angles.

The second way in which the bounding boxes can be used is as a design and layout aid. A global boolean variable called BOUNDBOXES can be set or reset at any time. The effect of setting this boolean "true" is that from that point on, any called cells will appear only as boxes. Setting the boolean "false" means that all called cells will be drawn in their full detail. Since this can be done at any time, arrays of cells can be only partly drawn, all cells called within the definition of a cell can be shown only as boxes, and so on. For example, the sequence

```
DEFINE('THISCELL'); { initiate definition }
    .    .    .    .
  BOUNDBOXES:=TRUE; { draw only boxes }
  DRAW('ADDER234',X1,Y1); { call the adder }
  DRAW('MULTIPLY',X2,Y2); { call multiply }
  BOUNDBOXES:=FALSE; { draw details }
  DRAW('SUBTRACT',X3,Y3); { call the subtract }
    .    .    .    .
ENDEF;
```

Will cause the cell THISCELL to be defined with only empty boxes where ADDER234 and MULTIPLY will be, but will show all the details of the cell SUBTRACT. Note that this is not at all the same mechanism for bounding box implementation that is in the CIF20 programs; that is completely different and can be used or not used irregardless of ART.

## 6. Programmable Logic Arrays

The PLA implemented in ART is a standard nor-nor PLA. It implements a standard sum-of-products form. It is constructed by calling a procedure called "pladefine", which defines a new PLA each time it is called. The procedure must be passed the following parameters:

1. The name of the pla to be defined. This name may be any legal cell name.

2. The number of inputs, an integer.

3. The number of product terms, an integer.

4. The number of output terms, an integer.

5. A boolean representing whether or not the PLA has a built-in two-phase clock.

Power may be connected at any point to the positive power wire that runs along three sides of the PLA, viz. the top, the left side, or the bottom (All of those orientations assume that the inputs and outputs face south, with the outputs east of the inputs). Ground must be connected to the lower corner of the vertical ground wire that runs along the right side; if it is connected anywhere else the power supply voltage drop along that wire may be excessive.

The inputs of the PLA are found as follows. First, one must calculate the width of the left-hand vertical power-supply wire; this width is called Wpb, and is given by

```
        Wpb = max{ 4,
    2(Lpa+Lpb)(inputs+outputs+terms)*unitload*sheetresistance/Vdrop}
```

where

- Lpa is given by (inputs*14 + outputs*7 + 50)
- Lpb is given by (inputs*14 + outputs*7 + terms*7 + 70)
- inputs is the number of input bits to the PLA
- outputs is the number of output bits
- terms is the number of product terms
- unitload is the current drawn, in amperes, of a single pullup[1]
- sheetresistance is the sheet resistance of metal, in ohms/square
- Vdrop is the permissible voltage drop along the power supply wire

Once this width has been calculated, the leftmost input can be found by

$$x = Wpb + 7$$

$$y = 2.$$

The succeeding inputs being found on 14 lambda centerlines to its right. All

[1]The unit load, sheet resistance, voltage drop, and termsperground are constants in the procedure "pladefine"'s header; the values given there should be used.

inputs are on diffusion wires, and connections should be made from the south.

The jth output is found by

```
    x = (j + j div termsperground)*7 + (ins + 1)*14 + 8 + Wpb + Wga

                       y = Wgd - Wgc - 4
```

where

- Wpb is as given above

```
       Wga = max{ 4,
    (terms * 7 + 30) * terms * unitload * sheetresistance / Vdrop}

       Wgd = max{ 11,
    (inputs*14)*(terms+2*inputs)*unitload*sheetresistance/Vdrop}

       Wgc = max{ 4,
         (outputs*7+20)*(2*(inputs+outputs)+terms)
                        *unitload*sheetresistance/Vdrop}
```

Programming PLAs is done by means of a PLA data file, into which the "programs" for all of the PLAs in an ART file are put; it is possible to have the programs for many PLAs in one file.

The PLA data file consists of one or more PLA programs, separated by one or more empty lines. The programs must be in the order in which the PLAs are defined, or errors will result.

The PLA program consists of n lines, where n is the number of product terms in the PLA; each line corresponds to a term. The lines consist of unbroken strings of characters; the only characters permitted are

T: this stands for "true", F: this stands for "false", X: this stands for "don't care", B: this stands for "blank".

The meanings of these characters are slightly different in the and and the or planes. The program does not distinguish between upper and lower cases in the PLA data file, but leading and embedded blanks or other characters will cause the PLA programming to be abandoned. The framework of the PLA will be constructed nevertheless.

It is important to note that the first line of the PLA program corresponds to the lowermost term of the unrotated PLA, and the last to the uppermost; this is somewhat counterintuitive and may cause trouble if the user is not careful. The leftmost character in a line corresponds to the leftmost input, and the rightmost character corresponds to the rightmost output.

The semantics of the PLA data file are as follows:

1. In the and-plane, a true value associated with an input means that that input will participate in the term in its uninverted form. For example, if the term a'bcd'e was one of the terms of the and-plane, then the second, third, and fifth characters on that line would be T's.

2. In the and-plane, a false value associated with an input term means that its complement will participate. for the example above, the first and fourth characters would be F's.

3. In the and-plane, either an X or a B means that the input in question does not participate in the given term; i.e. it is a don't care input with respect to that term.

4. In the or-plane, a product either participates in a sum or it does not. It will participate in the sum if its corresponding character is a T; otherwise it does not participate.

As an example of a PLA program, consider the following:

```
tfbxfftt
fxtbfbtf
```

This is a program that describes a PLA with two terms, four inputs, and four outputs. The first line describes the lower term: if we call the inputs a, b, c, and d, then that term represents the boolean product ab'; the other two inputs do not participate. The upper term (lower line) represents the product a'c; the other inputs do not participate.

In the or-plane, the two product terms are combined as follows. The first sum having two f's, it is the sum of no products, and therefore is always false. The second sum is also always false. The third sum is of both of the products, i.e. its equation is ab'+a'c; and the fourth sum is of the upper line's product only, i.e. ab'.

The PLA data file can contain the programs for more than one PLA, as mentioned above. These programs should be separated by blank lines, but otherwise will not interfere with one another.

## 7. Errors and Warnings

ART attempts to detect certain errors that might not be detected by the CIF-processing software. These errors cause messages to be sent to the user via the file OUTPUT, which in most cases will be the user's terminal itself. These errors fall into two classes: errors and warnings. An Error, as distinct from a warning, is a serious problem, that in most cases will abort the procedure in question. A message is sent to the user, which tells him what the error is, that it is an error, and that the program ART is responsible for the message. All errors should be corrected by the user, as they are almost certainly fatal.

Warnings, on the other hand, are not necessarily fatal; they do not abort the procedure, and they do not interfere with the CIF written to the output file. However, they do represent circumstances that are felt to be unusual enough that they should be brought to the programmer's attention. ART prints out the warning message, a note that it is a warning, and a note to the effect that the warning originates with ART. A list of the error and warning conditions is given in an appendix of this manual.

## 8. Predefined Cells

There are some predefined cells implemented in ART to make the task of design less tedious. These are the red-blue contact, the green-blue contact, and the red-green-blue butt joint. They should not be called directly; instead there are procedures that draw the contact where it is wanted.

| Procedure | Result |
|---|---|
| RB(x1,y1) | Red-blue contact centered at (x1,y1) |
| GB(x1,y1) | Green-blue contact centered at (x1,y1) |
| BN(x1,y1) | Butt joint, poly to "north",at (x1,y1) |
| BW(x1,y1) | Butt joint, poly to "west", at (x1,y1) |
| BS(x1,y1) | Butt joint, poly to "south", at (x1,y1) |
| BE(x1,y1) | Butt joint, poly to "east", at (x1,y1) |

Other predefined cells are available (e.g. pads, certain functions), but they are not part of the basic ART program, and must be either appended to ART in the form of ART definitions, or else the CIF for the predefined cells can be appended to the CIF file created by ART. In appending to the CIF file, some caution must be exercised: ART has no way of knowing either the name or the identifying number of the predefined cells in question, and the identifying numbers may well be nonunique, which will cause the CIF processing software to choke.

## I. An Example of ART

This appendix contains an example of an ART program. It does not contain many of the Pascal programming constructs; instead it is a definition of a typical cell. The syntax is correct. Some of the procedures that are available to the ART user are not used, and a good deal of the original cell definition has been removed in order to compress the example; this is not a working cell.

This example program would be appended to the ART skeleton when it was completed, right after the first statement in the main program body, which initializes ART. The program would then be compiled and executed in the normal fashion.

```
{ this is a general logic function block. See h42doc for details. }
define('h42paon ');
        rb(31,27);rb(37,6);rb(63,18);rb(69,8);
        gb(2,7);gb(2,26);gb(6,17);gb(24,7);gb(24,32);
        gb(44,18);gb(52,17);gb(57,25);gb(57,37);
        bn(9,3);bn(16,25);bn(44,31);
        bs(9,36);bs(17,16);bn(45,10);
   layer(metal);
        wire(4,2,2);y(42); { gnd 1 }
        box(0,9,11,30); { " }
        wire(4,24,2);y(42); { vdd 1 }
        box(16,0,26,10); { vdd 1 }
        box(21,22,26,32); { vdd 1 }
        box(16,31,26,43); { vdd 1 }
        wire(3,30.5,1.5);y(42); { and }
        wire(3,37.5,1.5); y(42); { or }
        wire(3,44.5,20);y(28); { out of inverter }
        wire(3,50.5,1.5);y(2.5);
            x(52);w(4);y(17.5);w(3);dx(-2);y(42);
        wire(3,56.5,27);y(35); { a sinkline }
        wire(3,63.5,1.5);x(62.5);y(18.5);x(63.5);y(42); { not a }
        wire(3,69.5,1.5);y(42); { not b }
      layer(diffusion);
        wire(4,5,9);x(6); { gate of nor }
        wire(6,6,26);x(10); { gates of nand }
        wire(4,6,17);x(17); { gate of nor }
        wire(2,-2,1);x(73); { b }
        wire(2,-2,38);x(73); { a }
        wire(2,9,8); { p.u.of nor }
           x(13);y(16);x(18);y(6);x(22);
        wire(2,19,18); { out of nor }
           x(38);y(10);x(41);y(8);x(46);y(2);
        wire(2,73,31); { p.u.'s of nand and inv }
           x(50);y(29);x(43);y(33);x(17);y(23);x(38);y(19);
        wire(2,38,24);x(55); { in of inverter }
        wire(4,44,18);x(49); { gate of invrtr }
```

```
layer(poly);
        wire(2,7,5);x(6); { gates of nand and nor }
           y(13);x(2);y(21);x(6);y(30);
        wire(2,10,14);y(33); { a gates of nand ands nor }
        wire(6,18,11);dy(2); { p.u. of nor }
        wire(6,17,28);y(33);x(18); { p.u. of nand }
        wire(2,30,21);y(25); { enable and }
        wire(2,35,7);x(34);y(20); { enable or }
        wire(6,38,33);x(39); { p.u. of inv }
        box(35,31,46,36); { p.u. of inv }
        wire(2,47,12);x(48);y(21); { gate of inv }
        wire(2,44,5);x(68);y(6); { not b }
        wire(2,53,26);y(21);x(62);y(20); { not a }
layer(implant);
        box(15.5,6.5,20.5,17.5);
        box(14.5,23.5,22.5,35.5); { p.u. pf nand }
        box(33.5,30.5,47.5,35.5); { p.u. of inverter }
endef; { end of defn of h42pandornot }
draw('h42paon ',10,10);
```

## II. ART Keywords and Functions

```
- BE   x,y:real        BE(3.3,6.2)

- BN   x,y:real        BN(2.6,3.3)

- BS   x,y:real        BS(3.6,2.3)

- BW   x,y:real        BW(3.2,3.6)

- BOX          x1,y1,x2,y2:real        BOX(2,3,3,6)

- BRISTLE      name:packed array[1..8] of  char
                var  layer:layertype
               var   direction:direction
                 var   w,x,y:real
                            BRISTLE('GROUND  ',LAER,DIR,W,X,Y)

- DEFINE       name:packed array[1..8]of char
                               DEFINE('CELL1234')

- DRAW      name:packed  array[1..8]of char
                   x,y:real
                          DRAW('CELL1234',3.14,6.28)

- DRAWMX           name:packed array[1..8]of char
                          x,y:real
                          DRAWMX('CELL1234',3.14,6.28)

- DRAWMY           name:packed array[1..8]of char
                   x,y:real
                          DRAWMY('CELL1234',3.14,6.28)

- DRAWROT       name:packed array[1..8]of char
               x,y:real {translation}
                   xr,yr:real{rotation vector}
                       DRAWROT('CELL1234',3.14,6.28,10,0);
- DRAWTX           name:packed  array[1..8]of char
                   x,y:real
                          DRAWTX('CELL1234',3.14,6.28)

- DRAWTY           name:packed array[1..8]of char
                   x,y:real
                          DRAWTY('CELL1234',3.14,6.28)

- DX          x:real   DX(.693)

- DXDY         x,y:real        DXDY(.693,1.73)

- DY           y:real  DY(.707)

- ENDEF        no argument     ENDEF

- GB           x,y:real        GB(3,4)
```

| | | |
|---|---|---|
| - LAYER | L:layertype | LAYER(RED) |
| - RB | x,y:real | RB(4,3) |
| - STUB | name:packed array[1..8]of char<br>layer:layertype<br>dir:direction<br>w,x,y:real | STUB('PRECHARG',METAL,3,10,20) |
| - W | w:real | W(4.2) |
| - WIRE | w,x,y:real | WIRE(3,0.5,-45.2) |
| - X | x:real | X(40) |
| - XY | x,y:real | XY(20,40) |
| - Y | y:real | Y(34) |
| - Z | layer:layertype | Z(BLUE) |

## III. Error and Warning Messages

### III.1. Errors

1. Nested Definition: Definitions must be terminated before new definitions can begin

2. Extra Endef: Cannot terminate a definition that was never initiated

3. Cell Undefined: Attempt was made to call a nonexistent cell, or cell name is wrong

4. Bristle;nonexistent cell: Bristle cannot find the cell to which the bristle is attached

5. Bristle;nonexistent bristle: Bristle cannot find the bristle in question

6. Primitive outside: May not call a primitive command outside a cell definition

7. Procedure Z: Procedure Z can only make RB and GB contacts

8. Bad Wire Call: Either X,W,Z,Y,DX,DY,XY,or DXDY was called without first calling WIRE or STUB

9. Drawrot 0,0: 0,0 rotation vector is meaningless

10. PLA End-of-File : the PLA procedure ran into an unexpected EOF. This is not completely fatal, but the PLA will not be properly programmed.

11. PLA Unexpected character: same effect as PLA EOF above.

### III.2. Nonfatal Warnings

1. Bound Box: A bound box was called outside a definition. While ART doesn't mind, the other CIF processing software may recognize an illegal primitive- outside-definition condition

2. Bad Wire: WIRE checks to see if wires obey the Mead&Conway design rules.

3. Bad Box: BOX checks to see if boxes are too small to meet Mead&Conway design rules

4. Size of Cell: If an empty cell or a cell which cannot contain legal structures because it is too small is called, this warning will be sent to the user.

## IV. Input and Output Files

When an ART program is executed, the loader will usually expect to get the names of four files: OUTPUT, TMPFIL, PLA, and CIF. OUTPUT is where the warnings and errors will be sent: usually the user should designate the TTY or terminal at which he is working. TMPFIL is the private temporary file used by the program, and can be given such a name as DELETE.ME; it should be deleted after the program has executed.

PLA is the name of the file from which the program will try to read PLA programming information if any PLAs are instantiated in the program. If there are no such instantiations, the data file can be empty or nonexistent; different operating systems will treat this problem differently. The safest way to deal with the issue is to have a dummy file for PLA data, unless of course there really is a PLA in the design.

CIF is the file into which the results of running ART will be written; it should be given a name that will reflect its significance, e.g. CHIP.CIF or PROJECT.CIF. This is the file that will be given to the other CIF-processing software to run a plotter, graphics unit, or mask machine.

## REFERENCES

1. Carver Mead and Lynn Conway. Introduction to VLSI Design. Addison-Wesley, 1980.

2. J. Ousterhout. Editing VLSI Circuits With Caesar. Computer Science Division, EECS, University of California Berkeley, California.

3. N. Wirth and K. Jensen. Pascal User Manual and Report. Springer-Verlag, 1972.