Appendix D. ICLIC Manual

[section contributed by Maureen Stone, Xerox ASD]

Acknowledgements

Special credit goes to Ron Ayres for his help with the design and implementation of the original ICLIC. I would also like to thank the following people for their contributions: Jim Blinn, Carver Mead, Dave Johannsen, and John Gray.

1. Introduction

ICLIC was developed at Caltech in the spring of 1978. The initial goal was to provide a simple language that could be used by engineers to do IC design on the DECSystem-20. Previously, all design had been done in PAL [Auto 1966]. The original description [Stone 1978] was directed toward the computer novice. Later in the year, ICLIC was expanded and more emphasis placed on user defined functions and programming.

The system was implemented by Ron Ayres as an extension of the language ICL [Ayres 1978]. Therefore, such features as looping and conditionals have always been available, though not described in the original manual. The syntax and much of the underlying structure, such as the strong and varied data types, very much reflect the structure of ICL.

This section is a rewrite and expansion of the ICLIC users manual done specifically for this document. That is, it is no longer intended to be a description of the Caltech system. Many of the details specific to the Caltech implementation have been omitted. The language of this document has been changed slightly to assume an audience that is already somewhat familiar with programming.

The purpose of this writeup is to provide an example of an IC layout language. Most of the language principles described in section 2.2 of the main document are reflected here.

2. Designing Circuits in ICLIC

The design of circuits in ICLIC involves creating symbols (i.e. cells, groups, subpictures) and concatenating them together into larger symbols, up to the chip level. A symbol is described by either a variable or a function call. A number of primitive symbols have been globally defined, which means they can be used by any program running ICLIC (see section 3).

To form a program in ICLIC one defines a number of symbol variables and assigns them values which are strings of function calls and variables. The string, or concatenation operator, is the {} and is described for symbols in section 2.3. Symbols can be concatenated with themselves as well as other variables and function calls to build up more complex symbols. The end result is a symbol which describes the chip.

Variables may be thought of as places to hold values. In ICLIC, each variable name must be given a TYPE that describes what sort of value will be assigned to that name. Declaring and assigning values to variables will be described in sections 2.1-2.3.

A function is a process that is called using the function name and possibly some additional information, called parameters. Section 2.4 describes how to specify functions and their parameters.

2.1 Type Declarations

In order to use a variable to hold a value, the place must be reserved with a declaration of the following form:

```
VAR  <varname> = <type>;

        example:   VAR I,J=INT; R=REAL; MAINCELL=SYMBOL;
```

All variables other than the system defined global variables (section 3.4) must be declared in the user's source file before any value is assigned to them. The following TYPEs are defined:

INT	integer number. i.e. one with no decimal point.
REAL	real (floating point) number.
POINT	describes a point on the layout as coordinate#coordinate. Each coordinate is either an absolute value, expressed as a real number, or a relative value, expressed as ". + real" or ". -

real". The "." signifies the current X or Y coordinate.

```
example:            { 1.5#2.3; 1.5#1.8; 2.0#1.8 }

is equivalent to:   { 1.5#2.3; .#.-5; .+.5#. }
```

The X and Y coordinates of a POINT can be treated separately as REAL numbers by using the notation: POINT.X and POINT.Y.

PATH A list of points such as used to describe the location of the center line of a wire. The following notation is used:

```
example:  { POINT; POINT; ... ; POINT }
```

SYMBOL The name given to a collection of variables and function calls that describe IC mask elements. SYMBOLs can be concatenated to create larger SYMBOLs by surrounding them with brackets as shown below:

```
example:  {SYMBOL ; SYMBOL ; ... ; SYMBOL}
```

SYMBOLS are discussed further in section 2.3.

2.2 Assignment Statements

A variable can have a value assigned to it using the following form:

```
<varname> := <value>;
```

Notice the ":=" for assignment, and the terminating ";". The assignment is not made until the ";" is reached, so nested assignments can be used to build up a the complexity of a symbol. All variable names must be declared by using VAR before the first assignment. Examples of valid assignments to the different type variables follow.

```
I := 1;             "I is an INTeger"

R := 3.146;         "R is a REAL"

ENDPOINT := STARTX+100#55.6;    "ENDPOINT and STARTX are POINTs"

LEFTX := MINPOINT.X;            "LEFTX is REAL, MINPOINT is a POINT"

BUSPATH := { 0#0 ; .+67.3#. ; .#CELLHEIGHT };  "BUSPATH is a PATH"

CELL := { CELL ; ENDCELL ; OUTPAD };           "All variables are SYMBOLs"
```

2.3 Symbols

The basic component of an integrated circuit mask is the SYMBOL. Once a variable has been declared to be type SYMBOL, it may be assigned a value one of three ways:

1. Calling a function that returns a SYMBOL as a value. All the system IC design functions described in section 3 fall into this category.

2. Using a variable that already has a symbol value. This includes variables displaced by the system positioning functions.

3. Concatenating together SYMBOLs using the { } notation. This is the way large cells can be built up out of smaller components. The concatenation syntax is:

{SYMBOL; SYMBOL; ... ; SYMBOL}

example: CELL:={BUS; HALFCELL; HALFCELL\MIRX};

Any number of symbols, separated by semi-colons, and surrounded by "{...}" can be concatenated together into a list of symbols and assigned to a variable of TYPE SYMBOL.

2.4 Function Calls

To use a function, you need to know its name and what parameters, if any, are needed. The TYPE of the parameters is important, as well as the TYPE of the value returned by the function. If there is more than one parameter, the order in which they are listed is also important. There are two notations in ICLIC for calling functions which have exactly one or two parameters.

function name	parameters	notation 1	notation 2
F	X	F(X)	X\F
F	X,Y	F(X,Y)	X\F Y

If the function has more than two parameters, notation 1 must be used.

ICLIC system functions all have 2 or fewer parameters, so they can be called using either notation. One convention that is suggested for readability is to call the functions which describe SYMBOLs using notation 1, and the functions which do positioning using notation 2. This convention is followed in the examples in this document.

A function call returns a value. Therefore, function calls can be nested as long as each function returns the correct TYPE. Note that in notation 2 everything before the "\" is the first parameter, so function calls are executed from left to right.

example: RB(-1#-1;1#1)\SIZE 5 \ROT 45

The example makes a red box which is 10 lambda on a side, rotated 45 degrees counterclockwise around its center point.

3. Predefined Functions

This section lists the ICLIC system defined functions useful for the description of IC layouts. They include boxes, contacts, and wires, some predefined variables for design rule distances, and the transformation functions. All these functions return a variable of type SYMBOL. Many names have colors abbreviated in them, using the following conventions:

R = red	(for poly)
G = green	(for diffusion)
B = blue	(for metal)
K = black	(for contact cut)
Y = yellow	(for implant)
W = brown	(for glass)

The definitions given below will use a "c" to indicate that any of these colors can be used.

3.1 Boxes

cB(< low > , < high >)

parameters: low,high = POINT

example: TRANS := {RB(-2#-1,2#1); GB(-1#-2,1#2)};

The low and high POINTs define the lower left and upper right corners of the rectangle. The box is defined oriented parallel to the axes. That is, X is horizontal, and Y is vertical.

3.2 Contacts

Standard contact definitions are built in as symbols. Each symbol includes a box on each layer specified, plus a contact cut (see [Mead 1978] for a complete description of contact cuts).

RTOB	red to blue square contact, 4 lambda/side
GTOB	green to blue square contact, 4 lambda/side
RTOG	red to green butting contact, 6 by 4 lambda oriented horizontally, with red on the left. The 6 by 4 metal layer is included.

3.3 Wires

A wire is described by a PATH, a WIDTH and a LAYER.

LAYER	The layer is a description of the layer the wire runs in.
WIDTH	The width is width of the wired measured in lambdas. Default wires widths have been defined for each layer.
PATH	The PATH describes the location of the center line of the wire.

3.3.1 Single-Layer Wires

To express a wire on a single layer, the following set of functions have been defined in ICLIC.

```
cW( < width > , < path > )

parameters:   path=PATH,  width=REAL  (optional)

        example:  RW({0#0;  .+1.3#2.7;  .+.7#.});

                  BUS:=BW(6,{-1#-1;  10#10});
```

The path defines the center line of the wire. The actual wire will be expanded a half-width all around the path, even at the endpoints. The global variable for wire width on that layer will be used if the width parameter is omitted. If present, the width parameter applies only for the given wire.

3.3.2 Multi-Layer Wires

In many cases, a wire as an electrical path is not defined on a single layer. Therefore, the following syntax has been defined to describe such wires.

{ LAYER; POINT; POINT; ... ; LAYER; POINT; ... ; POINT }

example: { RED; 0#0; 10#0; BLUE; 12#0; RED; 14#0 }

POINT is either an absolute or relative point, as described in section 2.1.

LAYER is the word RED or GREEN or BLUE, subject to the restrictions presented below.

The overall path in the multi-layer wire is the sequence of points with the layer definitions omitted. The width of each section of wire is the standard width for that layer. Each new layer definition inserts a contact at the last point specified before the layer mnemonic. This point is called the feed-through point. For various reasons, only transitions involving the metal layer have been implemented. An illegal transition generates an error message at execution time.

At each feed through point, one of the system defined square contacts, RTOB or GTOB is generated. The center of the contact is aligned with the feedthrough point.

3.3.3 Cables

To facilitate design using multiple wires following roughly parallel paths, the CABLE function has been defined. The syntax is:

CABLE(< layer > , < #wires > , < next start > , < next end > , < sample path >)

parameters:

layer = LAYER specification, RED or BLUE or GREEN.

#wires = The integer number of wires in the cable

sample path = { STARTP; POINT; ... ; ENDP}. This path describes the center line of one of the end wires. STARTP and ENDP are the starting and ending POINTs for the path.

next start = The starting point for the wire next to the sample path is given as either an absolute or relative POINT.

end start = The ending point for the wire next to the sample path is given as either an absolute or relative POINT.

example: CABLE(RED, 3, .#.-RWSPACE, .#.-10, {0#10; .+5#.; .#0; 10#0})

All wires will be parallel to the sample path at the globally defined minimum spacing except where the start and end points are connected. Default values for the minimum spacing on each layer are given in section 2.4.

No test is made for crossed wires. Whatever wiring pattern is generated by the above algorithm is drawn without comment.

Figure D.1 shows an example of a CABLE.

3.4 Global Variables

Global variables are a means to ensure that commonly used distances remain uniform across the layout. Furthermore, global changes can be made by simply reassigning a variable. Therefore, it is strongly recommended that the user use variables for distances wherever possible.

The following system defined global variables have been built in to reflect the design rules described in [Mead 1978].

LAMBDA	Definition of the basic measurent unit, in microns. The default value is 3 microns.
cWIDTH	width of colored wire
cFRMc	design rule distance between colors
cWSPACE	minimum spacing between wires (R, G, B only)
cWFRMc	minimum spacing between a wire (R, G) and an area that is, the design rule distance plus half the wire width

These variables have the following defaults:

BWIDTH	3 lambda
cWIDTH	2 lambda (all other colors)
RFRMR	2 lambda
RFRMG	1 lambda

```
C1:=CABLE(RED,4,.#.-RWSPACE,.#.-RWSPACE,
{0#70;.+18#.;.#.-20;.+11#.;.+10#.-10;.+20#.});
```

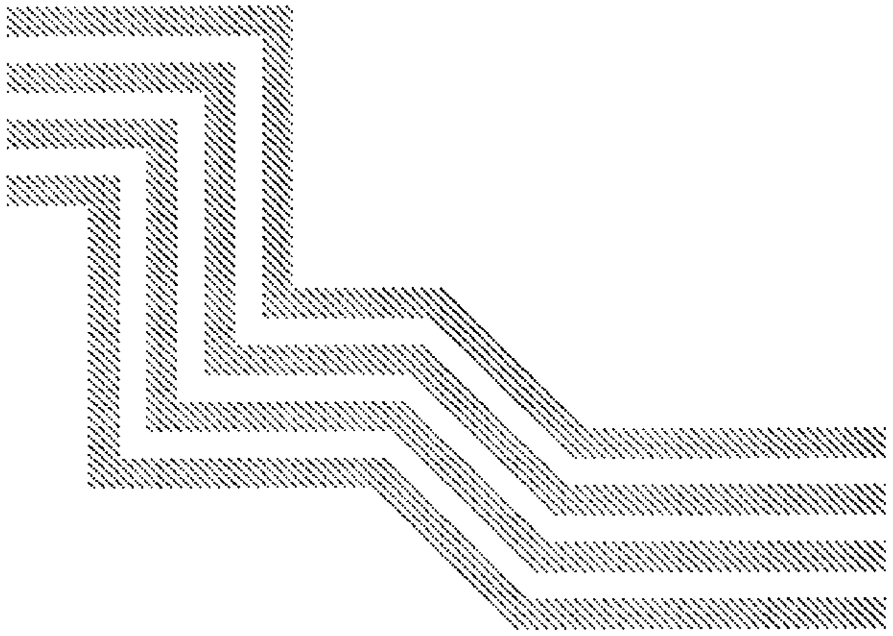


Figure D.1. An Example of a CABLE

GFRMG	3 lambda
RWSPACE	4 lambda
GWSPACE	5 lambda
BWSPACE	6 lambda
RWFRMR	3 lambda
RWFRMG	2 lambda
GWFRMR	2 lambda
GWFRMG	4 lambda

These variables can be assigned new values by using them on the left hand side of an assignment statement, just like any variable in ICLIC.

3.5 Positioning Functions

These functions are used to position symbols, and can be used on any variable of type SYMBOL.

Displacement

```
<symbol>\AT  <point>

<symbol>\AT  {<string of points>}

<symbol>\AT  [IX:<real>  IY:<real>  NX:<int>  NY:<int>]

        example:  INVERT\AT 4#4

                  INVERT\AT {0#0 ; .+5#3.7 ; 0#.}

                  CELL\AT [IX: 20.7 IY: 44.7 NX: 5 NY: 3]
```

AT translates the symbol by the amount point.x, point.y, i.e. it adds these values to all X,Y values in the symbol. If a string of points is given, the symbol is duplicated at each of the locations specified. Note that the relative point notation can be used, just as in wire PATHs.

The third form of the AT command specifies a regularly repeated pattern. The square bracket notation defines an array of positions, with increment (IX, IY) and count (NX, NY) parameters in X and Y. The origin of the lower left cell is positioned over the current X,Y. The example describes a 5 by 3 array of cells, spaced 20.7 lambda apart in X and 44.7 lambda apart in Y.

If the parameter is not specified, IX,IY default to 0,0 and NX,NY default to 1. This is useful for making single rows or columns of symbols.

example: CELL\AT [IX: 20.7 NX: 5]

This makes a single row of the previous example.

Note that to position an array at an arbitrary point instead of using the current X,Y it is possible to use nested AT commands.

example: CELL\AT 55.5#25 \AT [IX: 20.7 NX: 5]

This example makes the previous example starting at X=55.5 and Y=25 lambda.

Rotation

< symbol > \ROT < angle >

example: INVERT\ROT 90

Rotate the symbol by the value of the angle. The angle is a real number, given in degrees. Positive angles are counterclockwise.

Scaling

< symbol > \SIZE < real >

< symbol > \SIZE < point >

example: GTOB\SIZE 2

TRANS\SIZE 1#2

Scale (multiply all X,Y values) by the parameter value. If the SIZE parameter is a real number, it is used to scale both the X and Y coordinates. If the parameter is a point, the SYMBOL is scaled by the value of the X coordinate in X, and by the value of the Y coordinate in Y.

Mirroring

< symbol > \MIRX

< symbol > \MIRY

< symbol > \MIRXY

These functions mirror (reflect) the symbol around the X axis, Y axis or both axes respectively.

example: LATCH := { HLATCH ; HLATCH\MIRX };

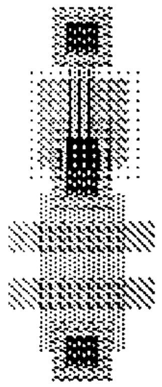
3.6 Simple IC Example

The following is a description of a 2 input NAND gate. The pullup length has been made a variable so the pullup ratio can be easily modified.

```
VAR   LEN=REAL;
      LEN:=5.5;                        "LENGTH OF ACTIVE AREA OF PULLUP"
      PULLUP=SYMBOL;                   "VARIABLE LENGTH PULLUP"
PULLUP:=     {
             RTOG \ROT -90;                    "CONTACT TO PULLDOWN"
             GW( { 0#0 ; .#LEN+4 } );          "ACTIVE AREA"
             YB( -4#-1.5, 4#LEN+2.5 );         "DEPL MODE IMPLANT"
             RB( -3#0, 3#LEN+1 );              "GATE"
             GTOB \AT 0#LEN+4                  "VDD CONTACT"
             };

VAR   NAND2=SYMBOL;                            "TWO INPUT NAND GATE"
NAND2:=      {
             GTOB;                             "CONNECT TO GND AT 0#0"
             RW( { 4#4; -4#.} );               "INPUT 1"
             RW( {-4#8; 4#.} );                "INPUT 2"
             GB( -3#0, 3#11 );
             PULLUP \AT 0#13
             };
```

The checkplot of this example follows.



4. Building Your Own Tools

The functions described in section 3 provide a way of digitizing a layout. A more powerful approach to description is to actually compute the layout. This section describes the tools in ICLIC that are directed toward that goal.

4.1 Loops and Conditionals

The form of the conditional statement in ICLIC is:

```
IF <BOOL> THEN <EXPR> ELSE <EXPR> FI
```

where <BOOL> is some boolean expression, and <EXPR> is one or more assignment statements.

```
example: IF N=3 THEN ARRAY:= { ARRAY; INVERTER; CELL};
         ELSE ARRAY:={ ARRAY ; CELL}; FI
```

The example inserts an extra inverter when N equals 3.

The basic form of the loop control in ICLIC is:

```
DO
      <computations>
FOR <loop counter> FROM <start> TO <end> BY <increment>;
```

The loop counter must be a variable of type INTEGER or REAL, and start, end, and increment must reduce to numerical values of the same type. The BY value is optional. The loop counter is tested at the end of the block, so all loops are executed at least once.

```
example: DO
             NEWROW:=ROW(NBITS,I);          "MAKE A NEW ROW, I IS A FLAG"
             Y:=Y+INCY;
             SRA:={SRA: NEWROW \AT X#Y}; "ADD THE NEW ROW TO SRA"
         FOR I FROM 2 TO NBITS;
```

This example is taken from the barrel shifter example in section 4.4 and builds an array of similar rows as dictated by the flag I.

4.2 Minimum Bounding Box

A function is provided that returns the minimum bounding box of any SYMBOL. The return in in the form of the minimum and maximum X,Y for the symbol. The easiest way to use the function is to assign the return to two separate POINT variables as shown in the example.

```
MBB( < symbol > )

parameter: SYMBOL
        example: LOWPOINT:=MBB(CELL).LOW;
                 HIPOINT:=MBB(CELL).HIGH;
                 LEFTX:=LOWPOINT.X;         "LOWPOINT is POINT, LEFTX is REAL"
                 TOPY:=HIPOINT.Y;           "HIPOINT is POINT, TOPY is REAL"
                 SIZE:=HIPOINT-LOWPOINT;      "SIZE is a POINT"
                 WIDTH:=SIZE.X+1;             "WIDTH is REAL"
```

4.3 User Defined Functions

The most basic type of function takes a parameter list and returns a value. While this value could be of any defined TYPE, to produce SYMBOLs it must be of TYPE SYMBOL. Parameters to functions are passed by value.

```
DEFINE <name>(<parm list>)= <return type>:

<body>

ENDDEFN
```

Note the colon after the return type, and the lack of semi-colon after the ENDDEFN. The < name > is the name used in the function call (section 2.4). The < parm list > is the parameter list definition, and must contain the dummy parameters and their TYPEs in the order they will be called. The < return type > is the TYPE the function returns, usually SYMBOL. Some examples of the first line, or header, of a function definition follow.

```
        DEFINE RB(LOW,HIGH:POINT)=SYMBOL:

        DEFINE ROT(SYMB:SYMBOL R:REAL)=SYMBOL:
```

It is possible to define more than one function with the same name as long as the parameter lists are different. That is, the parameters must be of different TYPEs, or in a different order. The value of this is that it is then possible to define several functions of the same name that do different things, depending on what TYPE of parameters are passed to them. For example:

```
DEFINE SIZE(SYMB:SYMBOL  R:REAL)=SYMBOL:

DEFINE SIZE(SYMB:SYMBOL  P:POINT)=SYMBOL:
```

The <body> of the function contains ICL statements just as in the main program. In the simplest case, this will be just a symbol definition. Notice that there is no semi-colon after the symbol description.

```
"VARIABLE LENGTH PULLUP"
"LENGTH IS IN LAMBDAS"
DEFINE VPULLUP(LEN:REAL)=SYMBOL:
      {
      RTOG \ROT -90;                    "CONTACT TO PULLDOWN"
      GW({ 0#0 ; .#LEN +4 });           "ACTIVE AREA"
      YB( -4#-1.5, 4#LEN+2.5 );         "DEPL MODE IMPLANT"
      RB( -3# 0  , 3#LEN+1  );          "GATE"
      GTOB \AT  0#LEN+4                 "VDD CONTACT"
      }
ENDDEFN
```

If local variables are required, the following form is used.

```
DEFINE <name>(<parm list>) = <return type>:
  BEGIN
  VAR    <variable declarations>;
     DO
            <computations>
     GIVE   <return value>
  END
ENDDEFN
```

Notice the lack of semi-colons after the DO, BEGIN, GIVE, and END statements. Following is an example of this form of function definition.

```
"GIVEN THE WIDTH OF THE PULLDOWN, MAKE A 2 INPUT NAND"
"USE MINIMUM WIDTH RED WIRES FOR INPUT GATES"
"CALLS VPULLUP, WHICH IS DEFINED ABOVE"
"USES SYSTEM DEFINED GLOBAL VARIABLES RWIDTH AND GWIDTH"
```

```
    DEFINE  NAND2(  PDWIDTH:REAL  )=SYMBOL;
       BEGIN
       VAR NAND=SYMBOL;     LEN,W1=REAL;
            DO
            LEN:=(8*RWIDTH*GWIDTH)/PDWIDTH;        "COMPUTE  8:1  RATIO"
            W1:=1+PDWIDTH/2;                       "HALF  WIDTH  OF  INPUT  GATE"
            NAND:=     {
                       GTOB;                       "CONNECT  TO  GND  AT  0#0"
                       RW({  W1#4;  -W1#. });      "INPUT  1"
                       RW({ -W1#8;   W1#. });      "INPUT  2"
                       GB(  -PDWIDTH/2#0,  PDWIDTH/2#11  );
                       VPULLUP(LEN)  \AT  0#13
                       };
          GIVE  NAND
       END
    ENDDEFN
```

(This NAND gate is the same as the one in section 3.6, for PDWIDTH=6.)

4.4 More Circuit Examples

The code for two larger circuits is shown here. The first circuit is the 4 bit inverter shift register shown in figure D.2. The inverter shift register is a single function, SR(NBITS), that takes as its parameters the number of bits desired. The array form of the AT function (see section 3.4) is used to duplicate the elements. Placement of wires is achieved by using pre-defined size constants such as cell width.

```
"INVERTER SHIFT REGISTER. PARAMETER IS NUMBER OF BITS"
"NUMBER OF CELLS = 2*NUMBER OF BITS"

DEFINE SR(NBITS:INT)=SYMBOL;
BEGIN
 VAR   PULLUP2, CELL, ENDCELL, CELL1=SYMBOL;
       INX, CINY, COUTX, COUTY  =REAL;
       CELL, SRA=SYMBOL;
       1, D2, D3, SRWIDE, CWIDE, CHIGH, DFRMBT, DFRMPAD=REAL;

 DO
       PULLUP2:=       {
                       RTOG\AT 0#0\ROT -90;
                       RB(-10#0, 4#7);
                       GTOB\AT -7#10;
                       YW(5, {1#0; .#3; -7#.; .#8});
                       GW({1#0; .#3; -7#.; .#8})
                       };
       CINX:=-10;   CINY:=9;
       COUTX:=-CINX;   COUTY:=CINY;
       CELL1:=         {
                       PULLUP2\AT 3#11;
                       GB(-1#0, 5#10);
                       GW({5#9; COUTX#COUTY});
                       RW({-2#6; 6#6})
                       };
       CELL:=          {
                       RTOG\ROT 180\AT -5#7;
```

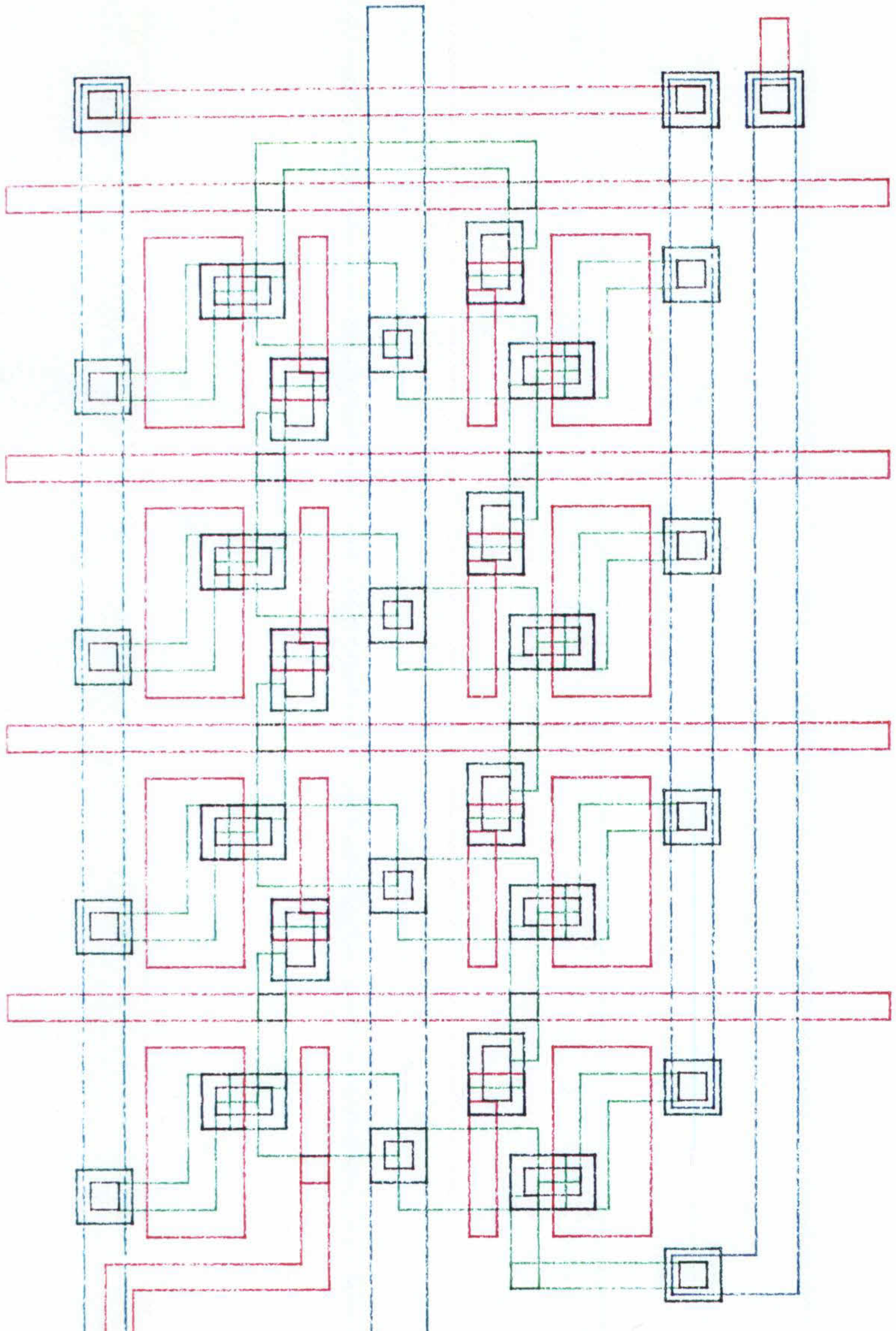


Figure D.2 Four Bit Inverter Shift Register

```
                    GW({CINX#CINY; .+3#.});
                    CELL1
                    };
      DCELL:=       {
                    CELL\ROT 180;
                    CELL;
                    GTOB\AT -1#0
                    };

      ENDCELL:=     {
                    CELL\ROT 180;
                    CELL1;
                    GTOB\AT -1#0
                    };
      CWIDE:=20;    CHIGH:=21;

      "MAKE THE ARRAY PART"
      SRA:=         {
                    ENDCELL\AT CWIDE/2#0;
                    DCELL\AT [NX:NBITS-1 IX:CWIDE NY:1 IY:0]
                    \AT 1.5*CWIDE#0
                    };

      "PUT IN THE WIRES"
      DFRMBT:=5;    "DISTANCE FROM BUS TERMINATOR"
      DFRMPAD:=12;      "DISTANCE FROM PAD"
      D1:=7;  "FROMLAST CELL TO VDD CONN"
      D2:=3;   "FRM LAST CELL TO FEED AROUND WIRE"
      SRWIDE:=CWIDE*NBITS;   "DISTANCE TO END OF SR ARRAY"

      SRA:=         {
                    SRA;
                    BW(4, {-DFRMBT#0;  (SRWIDE+DFRMPAD)#0});   "GND"

      " VDD...2 BLUE WIRES CONNECTED BY A RED ONE"
                    {BLUE;  14#-CHIGH; SRWIDE+D1#.; RED;  .#CHIGH; BLUE;
                    - DFRMBT#.};

       "CONNECT UPPER TO LOWER AND SHORT ONE OUT"
                    GW({(SRWIDE)#COUTY; .+D2#.; .#-COUTY; .-D2#.});
                    YB(SRWIDE+D2-5#COUTY-1.5, SRWIDE+D2-.5#COUTY+1.5);

      "OUTPUT WIRE"
                    {GREEN;  0#-COUTY;  .#-CHIGH;  BLUE;  .#.-BWSPACE;
                    SRWIDE+D1#.;  RED;  .+(DFRMPAD-D1)#.};

      "INPUT WIRE"
                    RW({-DFRMBT#CHIGH-1; 0#.; .#6; 8#.});

      "NOW PUT IN CLOCK LINES.  NEED NBITS OF THEM"
                    RW({0#-CHIGH-13; 0#CHIGH+6})\AT CWIDE#0
                    \AT [IX:CWIDE IY:0 NX:BITS NY:1]
                    };
      GIVE SRA
 END
ENDDEFN
```

The second circuit is a barrel shifter. The layout is shown in figure D.3. The barrel shifter is structured in similar, but not identical rows. The rows differ in the placement of one special cell. The barrel shifter function, SHIFTR(NBITS), takes the number of bits as its parameter, and calls a subfunction once for each row. The subfunction, ROW(NBITS,NPOS), takes the number of bits and an integer indicating which position the special cell sits in the row and generates one row of

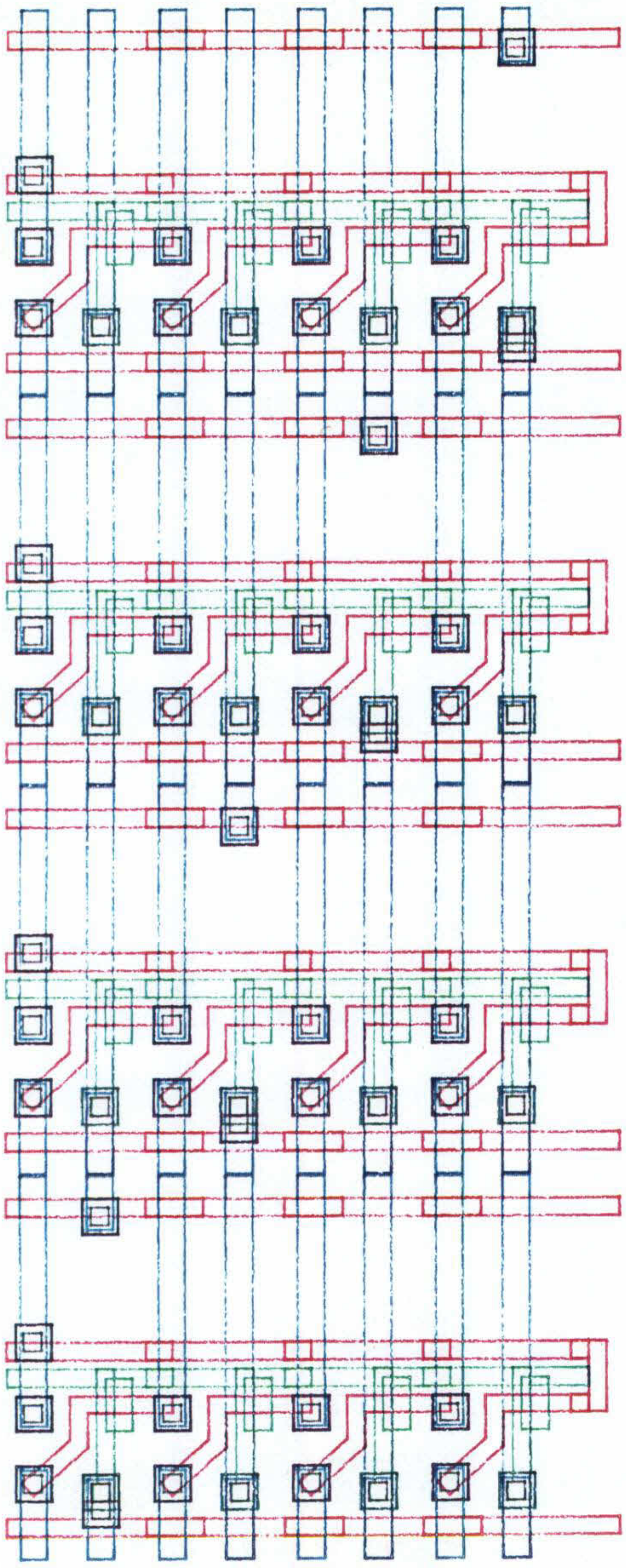


Figure D.3 Four Bit Barrel Shifter

the layout. The outer function computes the minimum bounding box for the row to determine the displacement in y.

```
"THIS IS THE CODE FOR THE BARREL SHIFTER"
"UPPER LEVEL FUNCTION IS SHIFTR(NBITS)"
DEFINE SHIFTR(NBITS:INT) = SYMBOL:
BEGIN
 VAR   NEWROW, SRA=SYMBOL;
       X, Y, INCY=REAL;
       P1, P2=POINT;
       I=INT;

 DO
       X:=0; Y:=0;
       SRA:=ROW(NBITS, 1);
 "COMPUTE THE Y INCREMENT FROM THE MINIMUM"
 "BOUNDING BOX OF ROW"
       P1:=MBB(SRA).LOW;    P2:=MBB(SRA).HIGH;
       INCY:=(P2-P1).Y-1;
       DO
         NEWROW:=ROW(NBITS, I);
         Y:=Y+INCY;
         SRA:=         {
                       SRA;
                       NEWROW\AT X#Y
                       };
         FOR I FROM 2 TO NBITS;
         GIVE SRA
 END
ENDDEFN

"DEFINES ONE ROW OF THE SHIFT REGISTER"
DEFINE ROW(NBITS:INT POS:INT)=SYMBOL:
BEGIN
 VAR   SUBCELL, CELL, CELL1, CELLN, ONEROW=SYMBOL;
       BUSA, BUSB, OUT, SHIFT, RCON1, RCON2, GATES,
       GCON, X1, X2, X3, CWIDTH, CTOP=REAL;
       IX=REAL;

 DO
       "SET UP THE DEFINED COORDINATES
        IF WE DO THIS RIGHT, THE REST IS EASY"
       BUSB:=2;     SHIFT:=18;
       OUT:=SHIFT+RFRMG+RWIDTH/2+GWIDTH/2;
       GATES:=OUT+RFRMG+RWIDTH/2+GWIDTH/2;
       RCON1:=GATES+1; RCON2:=RCON1+8;
       GCON:=RCON2+1;
       BUSA:=GCON+4;
       CTOP:=BUSA+2;
       X1:=3.5;     X2:=X1+7;     X3:=X2+7.5;
       CWIDTH:=X3+2;

       "BASIC CELL"
       SUBCELL:=     {
                     RW({0#BUSB; CWIDTH#BUSB});
                     RW({X1#SHIFT; CWIDTH#SHIFT});
                     GW({X1#OUT; CWIDTH#OUT});
                     RW({0#BUSA; CWIDTH#BUSA});
                     {RED; X1#GATES; X2+2.5#.; .#.+4.5;
                       X3#RCON2; BLUE; .#CTOP}
                     {GREEN; X2-.5#OUT; .#GCON; .+.5#.; BLUE; .#CTOP};
                     {BLUE; X3#0; .#RCON1; RED};
                     GB(X1+3.5#OUT, X2-.5#RCON1+2)
                     };
        "SPECIAL CELL"
        CELL1:=         {
                        SUBCELL;
                        RTOG\ROT-90\AT X2#GCON+1;
                        {BLUE;  X2#0;  .#BUSB+1;  RED}
                        };

        CELL:=          {
                        SUBCELL;
                        BW({X2#0;  X2#CTOP})
                        };

        "COMPUTE THE SPACING IN X"
        IX:=X3-X1+1;  "X3 ALIGNS WITH X1"

        "POS IS THE POSITION OF CELL1"
        ONEROW:=        {
                        CELL\AT [IX:IX IY:0 NX:POS-1 NY:1];
                        RW({X1#GATES; .-2#.; .#SHIFT; .+2#.});
                        CELL1\AT (POS-1)*IX#0;
                        CELL\AT  POS*IX#0
                          \AT [IX:IX IY:0 NX:NBITS-POS NY:1];
                        RTOB\AT NBITS*IX+2.5#SHIFT-1
                        };

        GIVE ONEROW
  END
ENDDEFN
```

5. Operating Procedures

This section should contain information specific to the system and facilities available. Important features are listed under each topic.

5.1 Plotting

The following functions are needed:

Mapping from the layer specification to the plotter (colors, stipples, whatever).

Specifying what layers and what circuit elements will be included in the plot.

Specifying circuit size and location relative to the plotting device.

If facilities dictate, how to pre-process plot files off-line from the plotter to optimize plotter usage.

How to abort plots.

5.2 Getting In and Getting Out

The areas that need to be discussed are:

File creation and storage.

Operating the compiler.

Errors and their explanation.

Conversion of finished circuit to CIF or other intermediate description form.