CHAPTER 7

A CIF PRIMER

Caltech Intermediate Form (CIF) is a low-level graphics language for specifying the geometry of integrated circuits. Its purpose is to serve as a standard machine-readable definition of chip geometry that is an essential input to a chip fabrication facility. CIF is intended to serve as an unambiguous definition of geometries for designs ranging from a small student project up to the extremely complex VLSI chips anticipated in five years' time.

The most important feature of CIF is that it strives to be a *standard.* It defines geometry in a way that is not tied to specific pattern-generation equipment, to specific computer-aided design or artwork systems, or to specific fabrication technologies. Thus, a designer using *any* design technique may communicate, via a CIF file, with a fabrication facility using *any* pattern-generation equipment, an arrangement illustrated in Figure 7.1. This figure emphasizes that CIF is not a design data base, but rather an interchange format to couple the designer to the fabricator. Even though design systems and pattern-generation techniques may change, CIF can remain the standard way to specify a chip's geometry.

In addition to the benefits of CIF as a standard, it offers the following advantages:

1. It is easily generated and processed.
2. It has a clearly defined, unambiguous syntax.
3. It is compact due to its hierarchical symbol structure.
4. It is a text format, and is therefore machine-independent and transportable.
5. It is easily read by people as well as computers.
6. It provides a common denominator for designs, encouraging the sharing of cell libraries, etc.
7. It is independent of the process used to fabricate the wafer.

CIF 2.0 is already in use by a number of universities and industrial laboratories. Designs specified in CIF have been transported electronically between Xerox-PARC, Caltech, Carnegie-Mellon, Stanford, MIT, University of California at Berkeley, and the University of Washington using the ARPANET and other data links. As more universities begin to offer courses on VLSI design, we anticipate the use of CIF will grow substantially.

This chapter begins with a self-contained definition of CIF, including both detailed syntax and semantics of CIF files and some less formal conventions. The second part of the chapter contains some examples of how CIF can be generated and interpreted.

Portions of this chapter were written by R. F. Sproull and R. F. Lyon; sections 7.1 and 7.3 contain some material that is virtually identical to that found on pages 115-127 (section 4.5) of *Introduction to VLSI Systems* [Mead & Conway 1980], copyright 1980 by Addison-Wesley Publishing Company, Inc. and are reprinted by permission.

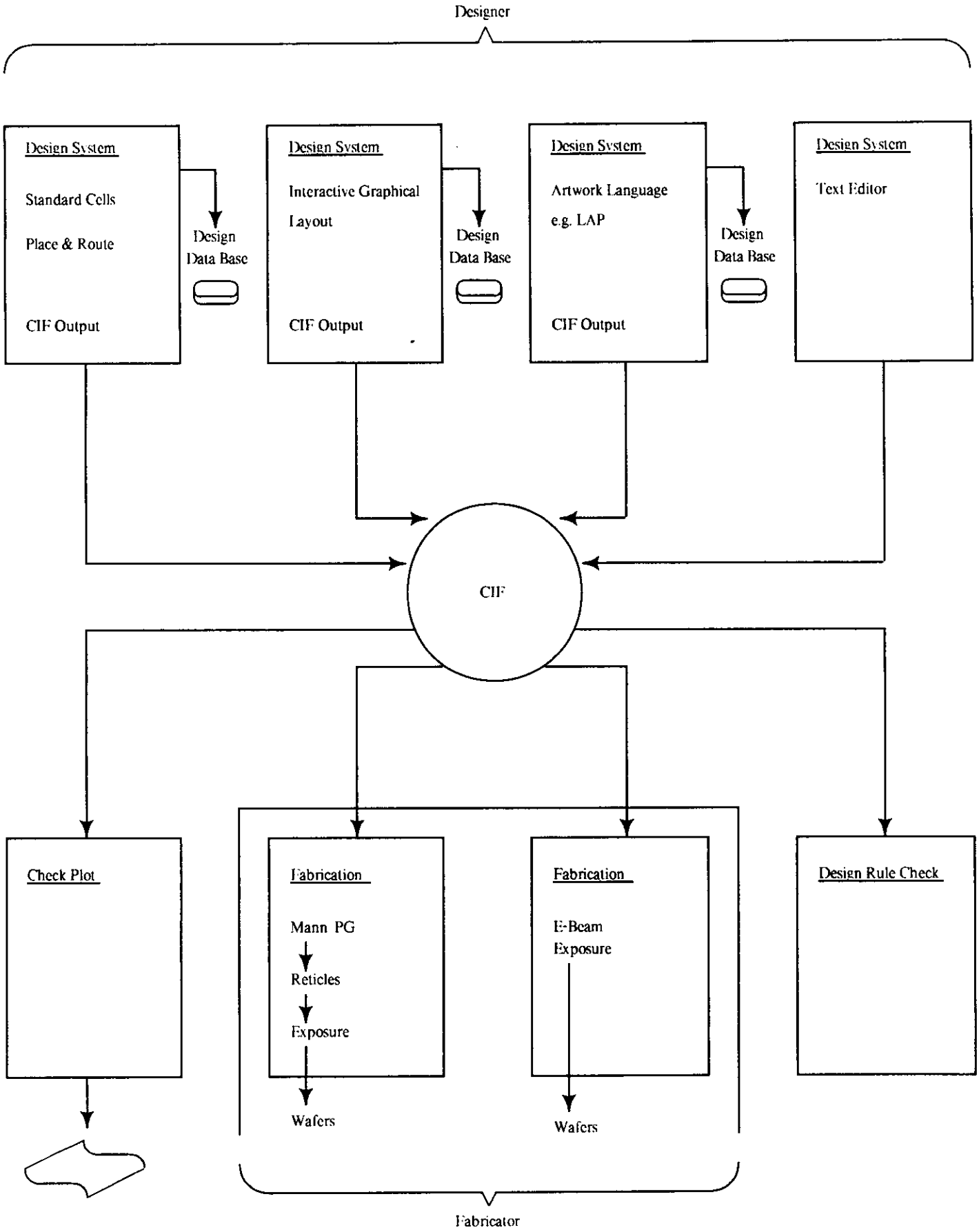

Designer
Design System
Standard Cells
Place & Route
CIF Output
Design Data Base
Design System
Interactive Graphical Layout
CIF Output
Design Data Base
Design System
Artwork Language e.g. LAP
CIF Output
Design Data Base
Design System
Text Editor
CIF
Check Plot
Fabrication
Mann PG
Reticles
Exposure
Wafers
Fabrication
E-Beam Exposure
Wafers
Design Rule Check
Fabricator

Figure 7.1

7.1 Definition of CIF 2.0

This section presents a concise definition of the CIF file format. Generally, the file may specify arbitrary geometries in an absolute coordinate system with a precision of 1/100th micron. These geometric patterns are expressed as a collection of *primitives*, such as boxes, wires, round flashes, and polygons of arbitrary shape. In addition, groups of these primitive items may be combined to form a *symbol*, which may be called to place *instances* of the symbol anywhere on the chip. The symbol may be mirrored, rotated, and translated at the time of the call. Although the symbol facilities add some complexity to CIF, they allow the geometry specification of a complex but regular design to be extremely compact.

The definition presented here is almost identical to that given in [Mead & Conway 1980]. Sections that are different or new are marked with an asterisk. It is intended that this chapter be the standard reference definition of CIF. As changes or clarifications become necessary, this section will be modified.

Portions of this section appear in [Mead & Conway 1980], and are reprinted by permission.

7.1.1 Syntax

A CIF file is composed of a sequence of characters in a limited character set. The file contains a list of commands, followed by an end marker; the commands are separated with semicolons. Commands are:

Command	Form
Polygon with a path	P path
Box with length, width, center, and direction (direction defaults to (1,0) if omitted)	B integer integer point point
Round flash with diameter and center	R integer point
Wire with width and path	W integer path
Layer specification	L shortname
Start symbol definition with index, a, b (a and b both default to 1 if omitted)	DS integer integer integer
Finish symbol definition	DF
Delete symbol definitions	DD integer
Call symbol	C integer transformation
User extension	digit userText
Comments with arbitrary text	(commentText)
End marker	E

A more formal definition of the syntax is given below. The standard notation proposed by Niklaus Wirth [Wirth 1977] is used: production rules use equals = to relate identifiers to expressions, vertical bar | for or, and double quotes " " around terminal characters; curly brackets { } indicate repetition any number of times including zero; square brackets [ ] indicate optional

factors (i. e., zero or one repetition); parentheses ( ) are used for grouping; rules are terminated by period. Note that the syntax allows blanks before and after commands, and blanks or other kinds of separators (almost any character) before integers, etc. The syntax reflects the fact that symbol definitions may not nest.

```
cifFile                = { { blank } [ command ] semi } endCommand { blank }.
command                = primCommand | defDeleteCommand |
                         defStartCommand semi { { blank } [ primCommand ] semi } defFinishCommand.
primCommand            = polygonCommand | boxCommand | roundFlashCommand | wireCommand |
                         layerCommand | callCommand | userExtensionCommand | commentCommand.

polygonCommand         = "P" path.
boxCommand             = "B" integer sep integer sep point [ sep point ].
roundFlashCommand      = "R" integer sep point.
wireCommand            = "W" integer sep path.
layerCommand           = "L" { blank } shortname.
defStartCommand        = "D" { blank } "S" integer [ sep integer sep integer ].
defFinishCommand       = "D" { blank } "F".
defDeleteCommand       = "D" { blank } "D" integer.
callCommand            = "C" integer transformation.
userExtensionCommand   = digit userText.
commentCommand         = "(" commentText ")".
endCommand             = "E".

transformation         = { { blank } ( "T" point | "M" { blank } "X" | "M" { blank } "Y" | "R" point ) }.

path                   = point { sep point }.
point                  = sInteger sep sInteger.

sInteger               = { sep } [ "-" ] integerD.
integer                = { sep } integerD.
integerD               = digit { digit }.

shortname              = c [ c ] [ c ] [ c ].
c                      = digit | upperChar.
userText               = { userChar }.
commentText            = { commentChar } | commentText "(" commentText ")" commentText.

semi                   = { blank } ";" { blank }.
sep                    = upperChar | blank.
digit                  = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
upperChar              = "A" | "B" | "C" | ... | "Z".
blank                  = any ASCII character except digit, upperChar, "-", "(", ")", or ";".
userChar               = any ASCII character except ";".
commentChar            = any ASCII character except "(" or ")".
```

7.1.2 Semantics

The fundamental idea of the intermediate form is to describe unambiguously the geometry of patterns for LSI circuits and systems. Consequently, it is important that all readers and writers of files in this form have exactly the same understanding of how the file is to be interpreted. This section is intended to contain sufficient information to allow unambiguous interpretation.

Measurements. The intermediate form uses a right-handed coordinate system shown in Figure 7.1.1, with x increasing to the right and y increasing upward. Directions and distances are always interpreted in terms of the front surface of the finished chip. The units of distance measurement are hundredths of a micron (μm).

Numbers. CIF uses numbers for several purposes: signed integers are used in coordinate measurements; unsigned numbers are used to specify widths, lengths, and symbol numbers. Signed numbers are restricted to lie in the range $-2^{24}+1 \leq x \leq 2^{24}-1$. Positive integers lie in the range $0 \leq x \leq 2^{24}-1$. These restrictions guarantee that a program reading CIF can represent any number in the file using a 25-bit one's or two's complement integer. *[Note: The restriction on the size of a number is not described in the CIF 2.0 description of [Mead & Conway 1980]. Some might argue that making such a restriction should require a change in CIF version number. However, since no programs yet written to generate or parse CIF will handle numbers larger than the size given here, adding the restriction induces no practical problems.]*

Directions. Rather than measure rotation by angles, CIF uses a pair of integers to specify a "direction vector." This technique eliminates the need for trigonometric functions in many applications, and avoids the problem of choosing units of angular measure. The first integer of a direction vector is the component of the direction vector along the x axis; the second integer is the component along the y axis. Thus a direction vector pointing to the right (the $+x$ axis) could be represented as direction (1 0), or equivalently as direction (17 0); in fact, the first number can be any positive integer as long as the second is zero. A direction vector pointing NorthEast (i.e., rotated 45 degrees counterclockwise from the x axis) would have direction (1 1), or equivalently (3 3), and so on.

7.1.2.1 Non-geometric Commands

User expansion: `5:NONSTANDARD DESIGN RULES: LAMBDA = 4.0;`

Several command formats (any command starting with a digit) are reserved for expansion by individual users; the authors of the intermediate form agree never to use these formats in future expansions of the standard format. For example, private expansions might provide for inserting instructions to a preprocessor that will be ignored by any program reading only standard intermediate form constructs; or recording ancillary information or data structures (e.g., circuit diagrams, design-rule check results) that are to be maintained in parallel with the geometry specified in the style of the intermediate form.

Users should be cautious about using local extensions that will render the CIF file meaningless to a program that cannot process the extension properly. For example, a common extension is to request that another file be "inserted" at this point in the processing, thus simplifying the use of symbol libraries. A standard CIF-reader will not insert the symbol definitions, causing the

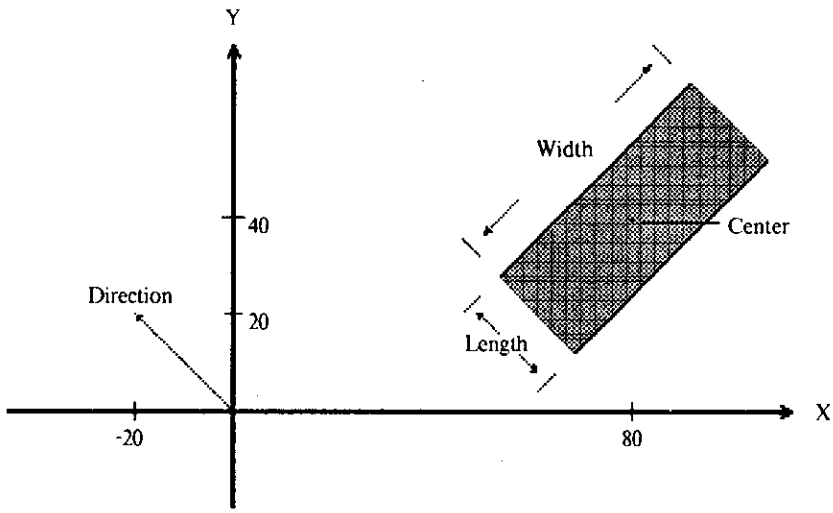


Figure 7.1.1 Box Representation in Intermediate Form

interpretation of the CIF file to be altered.

Comments: (HISTORY OF THIS DESIGN:);

The comment facility is provided simply to make the file easier to read. Note that a comment is a command: it must be terminated with a semi-colon, and can begin only in those places where commands can begin. The comment command can be used to deactivate any number of commands by simply enclosing them within a pair of parentheses, even if they already include balanced parentheses.

End Command: End of file.

The final E signals the end of the CIF file.

7.1.2.2 Geometric Primitives

The various primitives that specify geometric objects are not intended to be mutually exclusive or exhaustive. CIF may be extended occasionally to accommodate more exotic geometries. At the same time, it is not necessary to use a primitive just because it is provided. Notice in the examples below that lower case comments and other characters within a command are treated as blanks, and that blanks and upper case characters are acceptable separators.

Boxes: Box Length 25 Width 60 Center 80,40 Direction -20,20; (or B25 60 80 40 -20 20;);

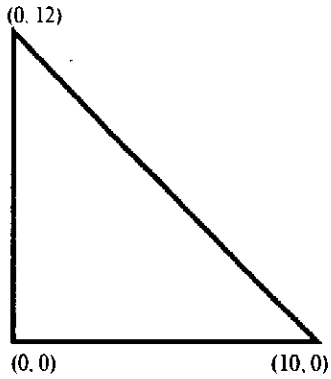
The fields that define a box are shown graphically in Figure 7.1.1. *Center* and *direction* specify the position and orientation of the box, respectively. *Length* is the dimension of the box parallel to the *direction*, and *width* is the dimension perpendicular to the *direction*. The *direction* may be omitted, in which case it will be defaulted to (1 0).

Polygons: Polygon A 0,0 B 10,20 C -30,40; (or P0 0 10 20 -30 40;);

A polygon is an enclosed region determined by the vertices given in the path, in order. For a polygon with n sides, n vertices are specified in the path -- the edge connecting the last vertex with the first is implied; see Figure 7.1.2a. Polygons with "holes" in them may be specified by the artifice illustrated in Figure 7.1.2b: the hole is part of the polygon boundary, and is joined to the outside boundary with a "channel" of zero width.

The polygon boundary may be self-intersecting, i.e., an edge may cross another edge. In this case, a point is on the interior of the polygon if the "winding number" (sometimes called "wrap number") of its boundary is non-zero. The wrap number can be visualized as follows: for each edge in the polygon, calculate the angle in radians subtended by the edge as viewed from the point; this angle will be positive if the edge is directed counter-clockwise with respect to the point, and negative if directed clockwise. Sum the angles for all edges; the wrap number is $sum/(2\pi)$. The wrap-number calculation is illustrated in Figure 7.1.3a. Figure 7.1.3b shows a polygon and the wrap

(a)



Polygon: P 0.0 10.0 0.12

(b)

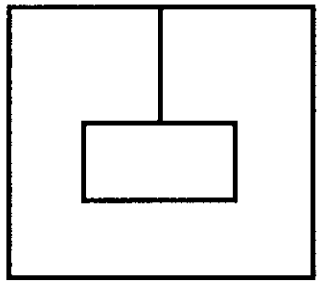


Figure 7.1.2

numbers of corresponding areas. Interior points (boundary wrap number non-zero) are shaded. *[This paragraph clarifies the definition of polygons in [Mead & Conway 1980].]*

Flashes: RoundFlash Diam 200 Center -500,800; (or R200 -500 800;);

This primitive calls for a circular shape, specified by its diameter and the location of its center (Figure 7.1.4a).

Wires: Wire Width 50 A 0,0 B 10,20 C -30,40; (or W50 0 0 10 20 -30 40;);

It is sometimes convenient to describe a long, uniform width run by the path along its centerline. We call this construct a wire (see Figure 7.1.4b). An ideal wire is the locus of points within one half-width of the given path. Each segment of the ideal wire therefore includes semicircular caps on both ends. Connecting segments of the wire is a transparent operation, as is connecting new wires to an existing one: the semicircular overlap ensures a smooth connection between segments in a wire and between touching wires.

Layer specification: Layer ND nmos diffusion; (or LND;);

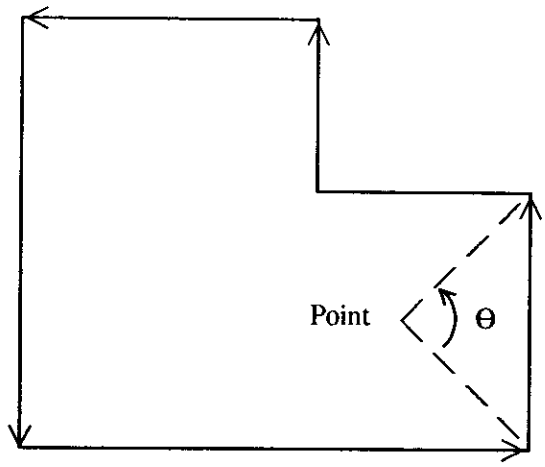
Each primitive geometry element (polygon, box, flash, or wire) must be labeled with the exact name of a fabrication mask on which it belongs. Rather than cite the name of the layer for each primitive separately, the layer is specified as a "mode" that applies to all subsequent primitives, until the layer is set again. It is illegal to specify a geometric primitive without having previously set the layer mode. Layer mode is preserved across symbol definitions and calls, which are discussed later.

The argument to the layer specification is a short name of the layer. Names are used to improve the legibility of the file and to avoid interfering with the various biases of designers and fabricators about numbers (one person's "first layer" is another's "last"). It is important that layer names be unique, so that combining several files in intermediate form will not generate conflicts. The general idea is that the first one or two characters of the name denote the technology, and the remainder is mnemonic for the layer. At present, the following layers are defined:

ND	NMOS Diffusion
NP	NMOS Polysilicon
NC	NMOS Contact cut
NM	NMOS Metal
NI	NMOS depletion mode Implant
NB	NMOS Buried contact
NG	NMOS overGlass openings

New layer names will be defined as needed in order to accommodate CMOS-bulk, CMOS-SOS and other technologies. Layer names will be added to the CIF definition without introducing incompatibilities in the format itself.

(a)



(b)

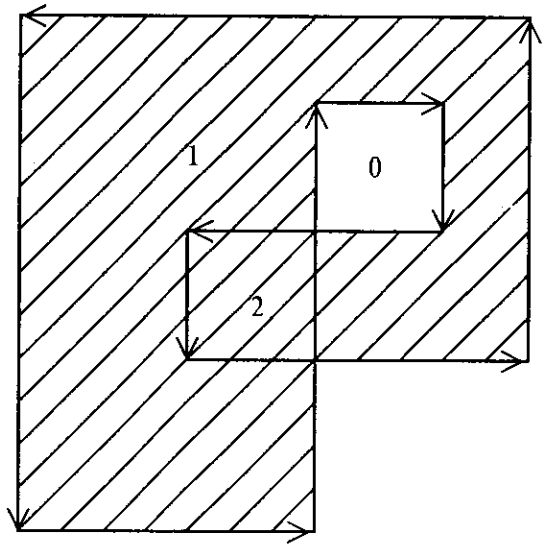
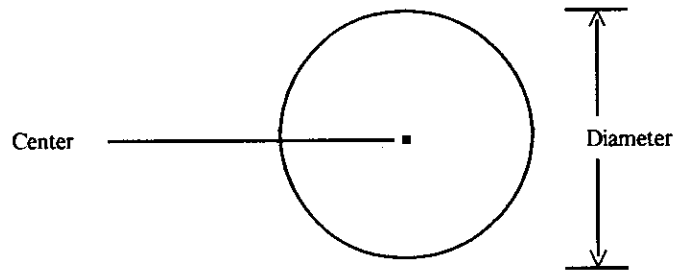


Figure 7.1.3

(a)



(b)

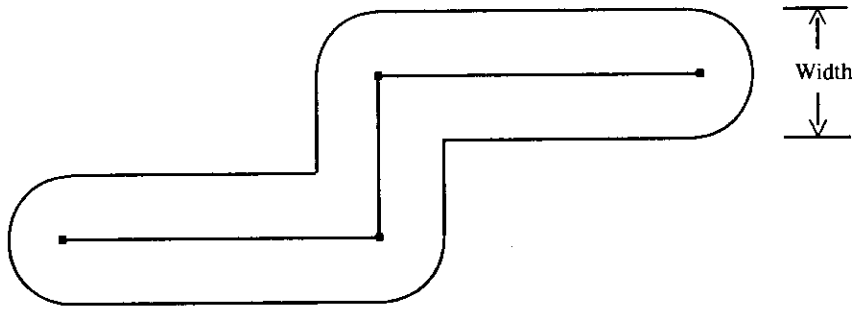


Figure 7.1.4

7.1.2.3 Symbols

Because many LSI layouts include items that are often repeated, it is helpful to define often-used items as "symbols." This facility, together with the ability to "call" for an instance of the symbol to be generated at a specific position, greatly reduces the bulk of the intermediate form.

The symbol facilities are deliberately limited, in order to avoid mushrooming difficulties of implementing programs that process CIF files. For example, symbols have no parameters; calling a symbol does not allow the symbol geometry to be scaled up or down; there are no direct facilities for iteration. If the symbol mechanism is not adequate for some application, the desired geometry can still be achieved with less use of symbols, and more use of explicit geometrical primitives. CIF files need not use symbols at all, although a rather large file will probably result.

Defining symbols: Definition Start #57 A/B = 100/1; ... ; Definition Finish; (or DS57 100 1; ... ;DF;);

A symbol is defined by preceding the symbol geometry with the DS command, and following it with the DF command. The first argument of the DS command is an identifying *symbol number*, unrelated to the order of listing of symbol definitions in the file.

The mechanism for symbol definition includes a convenient way to scale distance measurements. The second and third arguments to the DS command are called a and b respectively. As the intermediate form is read, each distance (position or size) measurement cited in the various commands (polygons, boxes, flashes, wires and calls) in the symbol definition is scaled to (a*distance)/b. For example, if the designer uses a grid of 1 micron, the symbol definition might cite all distances in microns, and specify a = 100, b = 1. Or the designer might choose lambda (characteristic fabrication dimension) as a convenient unit. This mechanism reduces the number of characters in the file by shrinking the integers that specify dimensions and may improve the legibility of the file. It provides neither scaling nor the ability to change the size of a symbol called within the definition.

Definitions may not nest. That is, after a DS command is specified, the terminating DF must come before the next DS. The definition may, however, contain calls to other symbols, which may in turn call other symbols. Additional information about symbol definitions is provided in *Symbol Interpretation Rules*, below.

Calling symbols: Call Symbol #57 Mirrored in X Rotated to -1,1 then Translated to 10,20;

The C command is used to call a specified symbol and to specify a transformation that should be applied to all the geometry contained in the symbol definition. The call command identifies the symbol to be called by its symbol number, established when the symbol was defined.

The transformation to be applied to the symbol is specified by a list of primitive transformations given in the call command. The primitive transformations are:

T point	Translate the current symbol origin to this point.
M X	Mirror in X, i.e., multiply X coordinate by -1.
M Y	Mirror in Y, i.e., multiply Y coordinate by -1.
R point	Rotate symbol's x axis to this direction.

Intuitively, each coordinate given in the symbol is transformed according to the first primitive tranformation in the call command, then according to the second, etc. Thus "C1 T500 0 MX" will first add 500 to each *x* coordinate from symbol 1, then multiply the *x* coordinate by -1. However, "C1 MX T500 0" will first multiply the *x* coordinate by -1, and then add 500 to it: the order of application of the transformations is therefore important. In order to implement the transformations, it is not necessary to perform each primitive operation separately; the several operations can be combined into one matrix multiplication (see [Newman & Sproull 1979]).

Symbol calls may nest; that is, a symbol definition may contain a call to another symbol. When calls nest, it is necessary to "concatenate" the effects of the transformations specified in the various calls (see [Newman & Sproull 1979]).

The layer mode is preserved across symbol calls and definitions. Thus, in the sequence:

```
LNM;
R6 20 0;

C 57 T45 13;

DS 114;
( ... definition of symbol 114);
DF;

LNM;
R3 0 0;
```

the second LNM is not necessary, regardless of the specification of symbols 57 and 114.

Deleting symbol definitions: Delete Definitions greater than or equal to 100; (or DD100;);

The DD command signals the program reading the file that all symbols with indices greater than or equal to the argument to DD can be "forgotten" – they will not be instantiated again. This feature is included so that several CIF files can be appended and processed as one. In such a case, it is essential to delete symbol definitions used in the first part of the file both because the definitions may conflict with definitions made later and because a great deal of storage can usually be saved by discarding the old definitions.

The argument to DD that allows some definitions to be kept and some deleted is intended to be used in conjunction with a standard "library" of definitions that a group may develop. For example, suppose we use symbol indices in the range 0 to 99 for standard symbols (pullup transistors, contacts, etc.) and want to design a chip that has 2 student projects on it. Each project

defines symbols with indices 100 or greater. The CIF file will look like:

```
(Definitions of library symbols);
DS 0 100 1;
( ...definition of symbol 0 in library);
DF;
DS 1 100 1;
( ...definition of symbol 1);
DF;
( ...remainder of library);

(Begin project 1);
DS100 100 1;
( ...first student's first symbol definition);
DF;
...
DS109 100 1;
( ...first student's main symbol definition);
DF;
C109 T403 -110; (call on first student's main symbol);

DD100; (Preserve only symbols 1 to 99);

(Begin project 2);
DS100 100 1;
( ...second student's first symbol definition);
DF;
...
DS113 100 1;
( ...second student's main symbol definition);
C1 T-3 45; (Call on library symbol, still available);
DF;
C113 T401 0; (call on second student's main symbol);

E
```

*7.1.2.4 Symbol Interpretation Rules**

The use of symbols in CIF requires a careful definition of the rules for interpreting a CIF file. We shall try to give a precise description without resorting to formalisms. The design of CIF and of the symbol facilities permits a straightforward interpretation scheme that processes commands in the file in order, in one pass over the file.

If no symbol facilities are used, the interpretation rule is very simple. Whenever a Layer command is parsed, the layer name it specifies is recorded as the *current layer.* Whenever a geometric primitive command is found, it can be "output" on the current layer. The kind of output generated will vary with the application of CIF: if we are making a check-plot, the primitive will be traced on a plotter, perhaps using a colored pen selected by the current layer.

Symbol facilities require that this simple interpretation rule be modified somewhat. The CIF file is still scanned once, from beginning to end. When the beginning of a symbol definition is encountered (DS command), subsequent commands are not "output" directly, but are instead retained, in the same order they appear in the file, as a *symbol definition.* When the end of the

symbol definition is encountered (DF command), the retained definition is entered in a *symbol table.* The symbol table maps a *symbol number* (the first argument following a DS command) into the corresponding *symbol definition*, a list of commands.

As the CIF file is scanned, we may also encounter commands that are not embedded in symbol definitions; these are *executable commands.* They are "output" directly, just as in the case mentioned at the outset where no symbol facilities are used. If the executable command is a Call on a symbol, the symbol definition is found by looking in the symbol table for the defintion with the corresponding *symbol number.* The interpreter now begins processing the commands saved as part of the symbol definition, with the additional need to apply to each geometrical primitive the transformation specified in the call. If, as the symbol is being interpreted, another Call command is encountered, its symbol definition is looked up in the symbol table, and the interpreter starts processing its commands, applying a transformation that is the concatenation of the transformations specified in the two calls. When all commands in the symbol definition have been processed, we resume interpretation of the first symbol where we left off, and so on. Eventually, the interpretation of the executable Call command is complete, and we continue scanning the file.

This model of the interpretation procedure provides intuitive answers to a number of common questions about the use of symbols:

1. Symbol definitions may occur in any order in the CIF file. The only restriction is that a symbol must be defined (i.e., entered into the symbol table) before its interpretation becomes necessary. Thus, the definitions for symbols 0 and 1 in the following file could be in either order:

```
DS 0 100 1;
( ...definition of symbol 0 );
C 1 T 14,32; ( call on symbol 1 );
DF;

DS 1 100 1;
( ...definition of symbol 1 );
DF;

C 0 ; ( Executable call; both symbols defined );
```

The call on symbol 1 inside symbol 0 is legal, because symbol 1 will be defined before symbol 0 is interpreted.

2. Deleting symbol definitions. The effect of the DD command is to scan the symbol table and remove any symbol definition with an identifying symbol number greater than or equal to the argument to DD. This may have the puzzling effect of retaining symbol definitions that *call* symbol numbers that, after the deletion is performed, do not exist. These are called "dangling references." It is helpful to issue a warning message if any dangling references remain after deleting symbols. The message should read: "Warning: dangling references after DD." *It is not considered normal practice to leave references dangling intentionally.*

3. Multiple symbol definitions. If a symbol definition is encountered that defines a symbol number already recorded in the symbol table, the old definition is replaced by the new one, and a warning message is issued. The message should read: "Warning: symbol *n* redefined." After the new symbol is entered, dangling references may remain, as described above; a warning message may be helpful. *Again, it is not considered normal practice to redefine symbols in CIF.*

7.1.3 The Relationship Between CIF and Fabricated Chips*

The relationship between geometric patterns specified in a CIF file and the patterns on a chip fabricated from the file cannot be controlled precisely. To attempt to do so would make the transformation from a CIF file to mask patterns difficult or even impossible.

CIF specifies the "nominal fabricated geometry" on the chip. By speaking of "fabricated geometry," we mean that the file specifies what the patterns should look like after fabrication. By "nominal" we mean that any given chip will depart in various ways from the exact CIF specification, due to variations occurring throughout the manufacturing process: alignment variations, slight variations in widths of objects, etc.

In most cases, extremely precise correspondence between the CIF geometry and the fabricated geometry may not be necessary. CIF will usually be used in conjunction with some *geometric design rules* associated with a particular IC technology. Adhering to these design rules precludes geometries that cannot tolerate small distortions during fabrication. A designer expects that a layout conforming to the rules will be fabricated sufficiently precisely to operate properly. Thus the design rules and the choice of a characteristic dimension are a measure of the fidelity with which the lithography and fabrication processes can manufacture the geometry specified in the CIF file. CIF that does not meet design rules cannot be guaranteed to be manufactured properly.

The following paragraphs expand on the notion of "nominal fabricated geometry," and try to answer the most common questions that arise about the interpretation of CIF geometry:

1. Features below the resolution of the manufacturing process. Although a CIF file may contain geometric shapes of very small size (.01 μm), the manufacturing process will not reproduce features smaller than a certain size.

When describing the relationship between CIF geometry and what appears on the chip, we often need to discuss the characteristic dimension of the lithographic and fabrication processes. In the design rules for NMOS given in [Mead & Conway 1980], this dimension is denoted by λ. In those rules, the single parameter is quite closely related to the geometric errors that are introduced in the manufacturing process. Design rules for other technologies may employ other parameters, which may not reflect only geometric fabrication limitations. In our discussion, we shall use λ to express the geometric fabrication tolerances, with the understanding that a similar parameter could be devised for technologies other than NMOS.

By following the geometric design rules for a particular process, the designer will be prevented from specifying patterns too small to fabricate. It should be pointed out that common right-angle corners technically constitute design-rule violations, because they require fabrication of an object with a width dimension less than λ (Figure 7.1.5a). Even if a design-rule checker is instructed to overlook such errors, the designer cannot expect perfect construction of the corner; a hypothetical result of manufacturing the shape with a process of minimum feature size 2λ is shown in Figure 7.1.5b.

2. Compensation for line-width distortion. The CIF file specifies the sizes that features should have once fabricated, which are not necessarily equivalent to the sizes of these features on a mask or to the sizes specified to pattern-generation equipment. Depending on the amount of etching done in certain process steps, on the differences between positive and negative resist steps and other effects, line widths extracted from the CIF file may need to be broadened or narrowed slightly before being passed to pattern-generation equipment. This compensation usually applies equally to all shapes, and thus requires a geometric "shrink" or "expand" to be applied to all geometric figures in the CIF file before generating a mask.

3. Connections. Two shapes on the same layer are assumed to be connected electrically if they abut one another and meet the geometric design rules. The shapes need not overlap in order to be connected. Figure 7.1.6a shows a legal connection made by abutting two shapes; the design in Figure 7.1.6b is illegal because a 2λ minimum-width design rule is violated at the joint.

4. Overlap. CIF allows objects on the same layer to overlap arbitrarily. Unfortunately, not all pattern-generation processes accommodate overlap. Repeated overlapping flashes on most present-day optical mask-generation equipment will overexpose the reticle, with a consequent "blooming" of the desired shape due to scattering and flare. A CIF file should be processed to remove overlaps before generating instructions for a pattern generator with this problem. Because this process depends on details of the pattern-generation process used, it should be the responsibility of the fabricator, not of the designer.

7.1.4 Common Conventions for Using CIF*

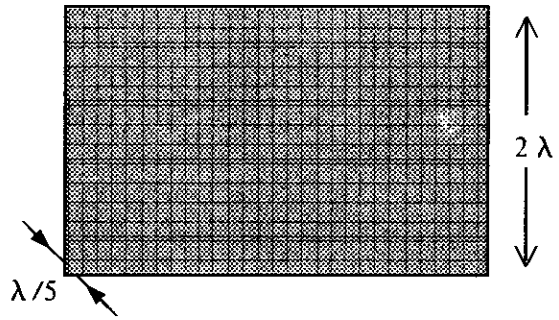
CIF was designed to be used in a variety of ways in the preparation of LSI artwork. The definition of legal CIF does not reflect all of the conventions that make CIF easy to use. This section outlines some conventions, and gives names to them. By giving names to the common ways to use CIF, we anticipate individual designers or design systems may simply adopt one or more of the conventions.

1. *Version comment convention.* This convention requires a CIF file to contain a comment near the beginning that describes the version number of the CIF format used in the file. For example:

```
(CIF 2.0);
```

This convention clearly labels the file with the assumptions used to generate it. It is also wise to

(a)



(b)

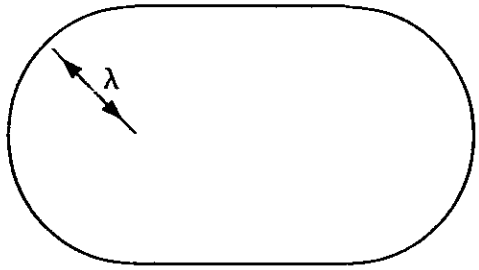
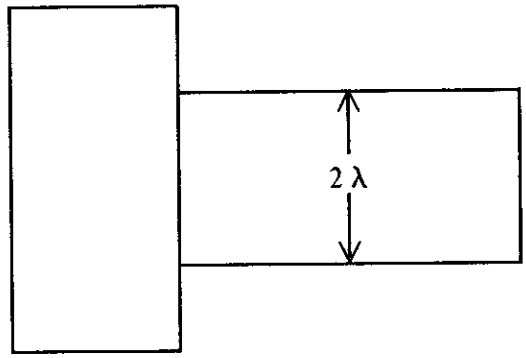


Figure 7.1.5

(a)



(b)

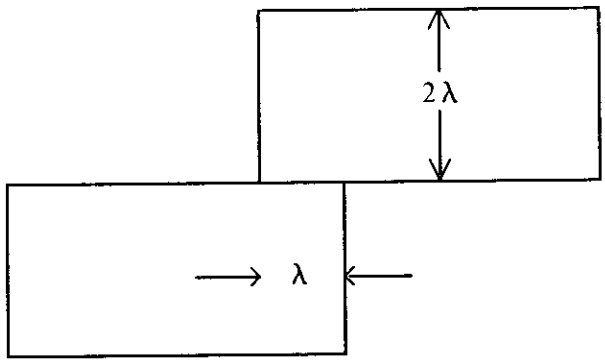


Figure 7.1.6

identify the designer and his or her organization at the beginning of the CIF file, perhaps including a date and design revision code.

2. *Fabrication comment convention.* A CIF file specifies only a portion of the information required to fabricate a chip. In addition to the geometry in the file, we need to specify some parameters of the fabrication process. This convention places such parameters in a comment near the beginning of a file; eventually a fully digital interface to fabrication may define these parameters precisely in machine-readable form. Until it becomes clear exactly what parameters must be supplied, the comment can contain arbitrary text:

```
(FAB nMOS silicon gate, Mead&Conway design rules
    lambda=3 microns; design can be scaled if necessary
    from lambda=2 min to lambda=5 max );
```

The comment explains that the geometry is for nMOS and has been defined using design rules in [Mead & Conway 1980] and that the parameter λ in those rules is 3 microns. It also says that the entire design could be scaled over a range from $\lambda=2$ to $\lambda=5$ microns in order to match the properties of a particular fabrication line. Indicating some flexibility in actual fabrication parameters reduces the constraints on combining separate experiments on a multi-project chip or on identifying a fabrication line that can process the design.

3. *No call convention.* Some CIF files will not use the symbol facilities of CIF at all; these are said to be "no call files." They contain no instances of the DS, DF, DD, or C commands.

4. *One call convention.* A common way to use CIF is to define a layout by an arbitrary number of symbol definitions, following by a *single* Call command. A CIF file conforming to this "one call convention" might look like:

a. Symbol definitions for "primitive" components, such as transistors, contacts, pads, etc.

b. Symbol definitions for building-blocks such as PLA cells, adder cells, register cells, pad drivers, super-buffers, etc.

c. Symbol definitions for functional parts of the design: a particular adder, PLA or shift register.

d. A single symbol definition that specifies the entire design by calling other symbols (presumably mostly those in category c), often using additional wiring to connect functional parts.

e. A single call on the symbol defined in (d). The transformation in the call determines where on the chip the entire design will lie.

The ordering of symbol definitions (a,b,c,d) is not critical; what is essential for this convention is that the only executable CIF command is the single call at the end (e).

This convention greatly simplifies the assembly of several projects on a single chip. By altering the transformation in the single call, the project can be placed in the correct position and orientation on the actual chip. Moreover, several projects can be combined into a single CIF file as follows:

1.1 Symbol definitions for project 1 (a,b,c,d).

1.2 The call for project 1, modified suitable to place the project properly.

1.3 A DD 0; command to cancel all symbol definitions associated with project 1.

2.1 Symbol definitions for project 2, similar to 1.1 above.

2.2 Call for project 2, similar to 1.2 above.

2.3 DD 0; to cancel project 2 symbols.

.

.

n.1, n.2, n.3 As many projects as you like.

s.1, s.2, s.3 A "project" to define scribe lines, test patterns, alignment marks, etc.

E The final "end" command.

It is possible to convert any CIF file into a "one call" file. For example, a no call file can be made into a single symbol by placing at the beginning a DS 0; command and DF; and C 0; commands before the End command. Files that already contain symbol definitions and executable commands interspersed require more complex modifications to make a "one call" file (e.g., renumbering symbols to create unique numbers, removing DD commands, etc.).

5. *x-Skeletal connectivity convention.* CIF files generated using this convention require that two objects on the same layer that are intended to make electrical contact must overlap by at least $2x$. Thus we speak of λ-skeletal connectivity convention or a 1-micron-skeletal connectivity convention. This requirement for connection is stronger than the abutting requirement of arbitrary CIF files.

The convention is called "x-skeletal connectivity" because the *skeletons* of each object are connected. The x-skeleton of an object is the object formed by shrinking it uniformly by x. Figure 7.1.7 shows an example of a wire connecting to a box; λ-skeletons are shown in solid lines and outlines are dashed. The λ-skeleton of a wire 2λ wide is simply a line (or a connected set of line segments). The λ-skeleton of a box is another box whose center is identical with the first, but whose width and length are diminished by 2λ.

The advantages of skeletal connectivity arise when geometric processing of the CIF file must be done. For example, if a pattern generator requires that all objects be shrunk slightly before exposure, shrinking a CIF file that uses the λ-skeletal connectivity convention can be done on an object-by-object basis: shrinking by any amount less than λ will preserve all electrical connections. If a file does not use the skeletal connectivity convention, shrinking two abutting objects independently will introduce a gap between them.

Design-rule checkers may also find processing of λ-skeletally-connected files easier. All minimum-width checks can be made simply by checking skeletons of objects. Clearance checks may also be conveniently expressed in terms of skeletons. Since many designs use lots of 2λ wires that have simple (line) skeletons, the checker may be able to substitute simple checks on lines for complex checks on solid objects, and consequently run faster.

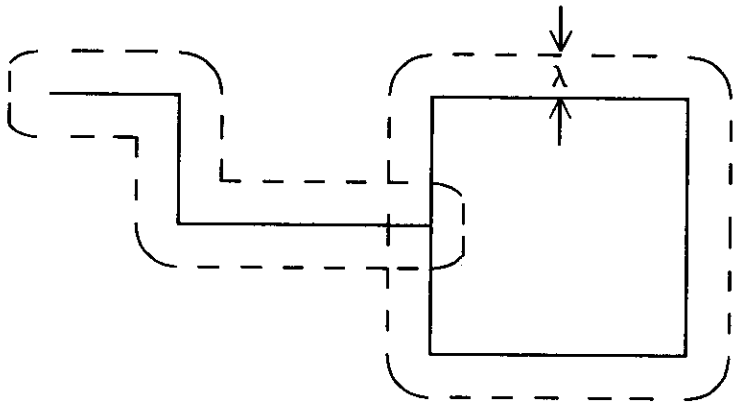


Figure 7.1.7

6. *No-polygon convention.* Files that observe the no-polygon convention do not use the Polygon primitive.

7. *Modest line-length convention.* CIF places no limits on the number of characters in a text line, although many computer systems find processing extremely large lines (or records) impractical. A CIF file that adheres to the modest line-length convention has no lines with more than 132 characters. On record-oriented systems, these files can be accommodated in 132-character fixed-length records. On character-oriented systems, these files have line breaks (carriage-return in the ASCII standard) so that no line is longer than 132 characters. Note that carriage return is a "blank" in CIF syntax, and may not appear at arbitrary points (e.g., in the middle of an integer).

7.1.5 Future Plans for CIF

CIF 2.0, as defined in this chapter, is being used heavily by university communities to transmit experimental designs and student projects to fabrication facilities. Several dozen serious projects have been successfully described in CIF and fabricated. The use of CIF is growing.

Our emphasis at the moment is on gaining experience with CIF 2.0 rather than on extending it. How well does CIF work in practice? What are the problems integrating it into a commercial fabrication environment? Is the definition given above clear and unambiguous? What are the problems with implementing computer programs to generate or parse CIF? Is it feasible for commercially-available design systems to generate CIF output? Does CIF apply equally well to all integrated-circuit technologies? These are the questions we are currently exploring.

When sufficient experience has been accumulated, it is likely that a new version of CIF will be specified. Currently, the suggestions for improvements are:

1. A mandatory "version" command (see *Version comment convention*, section 7.1.4).

2. A sensible convention for defining polygons with "holes." One proposal that has been advanced is to define a polygon with a series of boundary contours, together with a convention about the direction of tracing a boundary (e.g., the interior of the shape is always on the left as the boundary is traversed in the order specified).

3. Better scope rules for symbol definitions. The present symbol machinery has difficulties: DD leads to dangling references; it is difficult, in the general case, to make a "one call" file from an arbitrary CIF file – this makes the assembly of multi-project chips rather difficult. Using nested symbol definitions and some corresponding scope rules will solve these problems.

4. Some mechanism to iterate symbol calls so that regular arrays can be generated easily. Iteration was deliberately omitted from CIF 2.0 because we could not devise a method that avoided machine-dependent computational problems. It is still possible in CIF 2.0 to achieve substantial file compaction when defining arrays by using several layers of symbol (e.g., cell, row, double-row, etc.). In a future extension, we may choose to make the "simple" iterations a part of CIF (e.g., two-

dimensional orthogonal arrays), and omit more complex iterations such as hexagonal arrays. The more complex designs will simply have to call symbols explicitly, rather than using an iterative construct.

This list will doubtless grow as experience with CIF grows. Please help improve CIF by sending us comments and suggestions.

7.2 Ways to Generate CIF

CIF is an interchange format, usually generated by a computer-aided design system (Figure 7.1). There are thus as many ways to generate CIF as there are design systems. This section simply surveys some of the techniques.

7.2.1 Keyboard Interface

For small-scale projects, it is possible to generate a CIF file directly with an interactive text editor. The designer will generally make a detailed drawing of the layout, and read off coordinates from his drawing and enter the appropriate CIF commands one at a time. After the design is entered, it must be check-plotted in some way to verify that the file is in proper CIF format and correctly encodes the layout. In this way, a person can design a small experiment with minimal facilities.

Creating CIF files directly is much easier if the designer has a library of CIF text that defines useful symbols: pads, pad drivers, PLA cells, shift register cells, and so forth. The symbols must be accompanied by documentation that illustrates how to make connections to them, etc. Such a library is a tremendous aid to a beginning designer because it allows him to concentrate on his design ideas rather than on details of pad drivers and the like. The library is, in effect, a primitive design data base. Although CIF is not intended to be used for complex data bases, it can be used very effectively in this crude form.

Direct creation of CIF files can also be made more convenient and less error prone with a suitable preprocessor program. For example, an interactive input program might allow the designer to use relative coordinates, provide suitable default values, permit convenient iteration of cells, and check for obvious errors. It might prompt the user for the next entry and give feedback as to the current symbol definition or current layer. Ideally it would interact with the user in terms of symbolic symbol names and handle the generation of proper symbol numbers internally. (See section 2.2 for a further discussion of such a program.)

7.2.2 Programming Languages

If the designer is also a computer programmer, he may prefer to write a program that, when executed, writes a CIF file. This allows all the facilities offered in programming languages to be used in determining the geometry: arithmetic expressions, conditionals, procedures, etc. Often a cell is used with some minor modifications in many different places of the overall design. This cell could be defined by a procedure, with parameters that govern the modifications. The procedure in turn calls on "primitive" procedures that generate CIF geometric primitives directly. These primitive procedures might also generate a display or check-plot at the same time the CIF file is being constructed. (See [Newman & Sproull 1979] for a discussion of "display procedures." See [Locanthi 1978] for an example of a simple set of procedures for IC design.)

If a fully interactive programming environment is available, the user can change the program and re-execute it very quickly. Most BASIC, APL and LISP systems are interactive in this way, and do not incur the delay of compiling the program before it can be executed. If the execution of the program generates a display, the designer can very quickly modify the program to achieve the design he desires.

An important advantage to this approach is that the programs and procedures created by the designer can be saved for use in subsequent designs. Procedures corresponding to cell designs can be generalized and parameterized to apply in many situations. In effect, the collection of procedures becomes a very powerful design data base.

7.2.3 Interactive Graphical Layout Systems

A common design tool in use in the IC industry today is the interactive graphical layout system, for example the commercially available Calma or Applicon systems. A user views a display of all or part of his design, and requests changes by entering commands with a keyboard and graphical input device such as a tablet. The graphical editing commands are designed to make simple and complex changes naturally. The system should allow the user to insert and move geometrical shapes as easily as moving cardboard pieces on a floor plan. With the same ease the user should be able to change the shape of geometrical features. If the system automatically snaps geometrical shapes to a selectable grid and aligns all shapes parallel to the coordinate axes (or possibly lines at 45 degrees), it frees the designer from the tiring task of paying attention to exact coordinate values.

Present-day graphics systems are not without their shortcomings. Often it is cumbersome to define a collection of items as a symbol and to generate a two-dimensional array of such symbols. A further drawback results from the the limited resolution and size of the screen. It can be difficult to keep track of the overall context in which particular items are being manipulated. For example, it may be hard to select the proper wire out of a large group of wires extending over a significant

distance on the chip.

It is usually a simple matter for these systems to generate CIF, as they retain a precise description of the layout geometry. It may be difficult, however, for them to exploit the CIF symbol mechanism if the layout system has no concept of symbols or if its symbol semantics do not correspond closely with those of CIF.

7.2.4 Standard-cell and Gate-array Systems

Some IC design systems in use today free the designer from most of the details of geometric layout. The design is described in terms of gates or functional blocks and the interconnections among them. The description may be entered with an interactive graphics system, but it contains no geometric information; it is a "logic diagram." The design system processes the logic diagram, drawing on a library of gate or standard-cell geometries, and devises a layout that implements the desired logic diagram.

Ultimately, these systems generate geometric mask information, which can easily be expressed in CIF form. These systems can probably take advantage of CIF's symbol machinery to define standard cells, standard gates, etc.

7.2.5 Silicon Compilers

The ultimate goal is to interact with the computer on a much higher level, where the system designer enters a detailed functional description of his system. From there a sophisticated program, drawing upon the resources of a large collection of properly parameterized subsystems or building blocks in a library, would automatically assemble a possible layout and the corresponding CIF file.

Steps toward this goal are already being taken, see for example [Johannsen 1979] or [Ayres 1979].

7.3 Processing CIF files

The processing of CIF files differs somewhat from that for conventional pattern-generation formats. This section attempts to provide some suggestions and guidelines for implementing the programs that read CIF files.

Any program to read a CIF file will probably have the general structure illustrated in Figure 7.3.1. The *parser* is responsible for reading the CIF text file, parsing commands, assembling arguments, and issuing warning or error messages if the syntax of the CIF file is found to be invalid in some way. The *interpreter* is responsible for retaining symbol definitions, expanding symbol calls, performing transformations, and issuing warnings or errors if difficulties are encountered. Finally, the *output* module receives geometric information from the interpreter for each geometric object in the design, and sends it to the appropriate output device. If they are properly designed, the same parser and interpreter can be used for all CIF processing. Different output modules will be needed to drive different output devices: plotters, pattern-generation equipment, displays, and so forth.

7.3.1 CIF Implementation Guidelines

It is important that programs processing CIF files operate cautiously, maintaining a constant vigilance for mistakes or entries that will not be processed properly. The next three sections provide suggestions corresponding to the three modules: parser, interpreter, and output. It must be remembered that these are suggestions; they are not part of the CIF standard.

These sections describe the construction of program that implement *all* of the CIF standard. Implementations of subsets are often a good idea for getting started (e.g., by restricting CIF coding to use only wires and boxes a somewhat simpler check-plotting program can be built). Any serious use of CIF, however, that accepts CIF files from various sources, must implement the full language.

7.3.1.1 Parser

The parser is responsible for reading the CIF text file, checking syntax, and the like. It should carefully implement the syntax given in section 7.1.1. In addition, several semantic checks should be made:

Numbers. The parser should check to be sure that no number exceeds the representation ability of the computer.

Nonsense arguments. Various arguments to CIF primitives do not make sense, and should generate warning messages. This includes boxes with length or width of zero, round flashes with zero diameter, and direction vectors (either in the box command or in transformations) in which

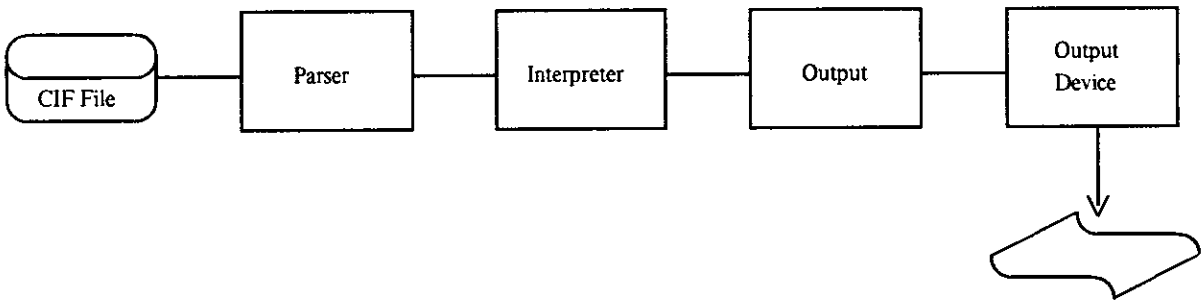


Figure 7.3.1 The Structure of a CIF Processing Program

both elements are zero. A polygon with one or two vertices should generate a warning message; this is almost certainly a mistake. A wire with one point in its path is perfectly legal (it is the same as a round flash at that point, with diameter equal to the wire width), but might deserve a warning message because it is an unusual way to achieve that effect.

Layers. Programs that read CIF will want to check to be sure that layer names used do in fact correspond to fabrication masks being constructed or to plotting conventions if a check-plot is being constructed. However, the file may cite layer names not used in a particular pass over the CIF file. It would be helpful for the program to provide a list of the layer names that it ignored. In this way, a user could spot typographical errors in layer names.

The intention of the layer specification command is to label locally the layer for a particular geometric primitive. It is therefore senseless to specify a box, wire, polygon or flash if no layer has been specified. There are a variety of ways to detect this error. One possibility is to insert the command LZZZZ implicitly at the beginning of the file and as the first command of each symbol definition. Any attempt to generate geometric output on layer ZZZZ will result in an error message.

User extensions. User extensions in CIF provide a means of including non-standard features, which may be installation-dependent. Programs that process CIF should flag and ignore user extension commands that are not implemented. Unfortunately, it is possible to devise user extensions that prevent the CIF file from being interpreted "correctly" by a program that does not implement the same user extensions (see the first example, below). Consequently, there is an informal desire to have similar user extensions. The remainder of this section lists a few machine-independent user extensions that have been found useful. They are described to give readers some feel for the types of features that may be incorporated into CIF. Note that far more than 10 extensions are possible because extensions may have more than one digit (e.g., "889 my own extension;").

Include Files. Format: 0 FileName;

This command has the effect of substituting the contents of FileName for the command. This allows easy merging of a number of separate files to make a project or a multi-project chip. Include File commands may be nested to any level. Programs implementing this command should be careful to flag possible errors, for example if symbol definitions are open across files, or if an End command is encountered within one of the included files. Warning: using this extension makes direct transmission of the CIF file to another site impossible, because it may not provide the same extension. Before transmission, the included file or files must be copied into the file being transmitted.

Comments from CIF Files. Format: 1 "Comment to be typed";

This command provides a mechanism for displaying progress notes to the standard output device (probably the user's terminal) as they are encountered in the CIF file. This is useful

to monitor the progress of the execution of an unobservable output device, e.g. E-beam mask maker.

Text Output on Plots. Format: 2 "TextOnPlot" *transformation*;

This extension cites text to be included on a check-plot to ease identification of pads, parts of the design, or wires. The text is treated as a symbol called with the stated transformations. The first character is located at (0,0) in its coordinate system; the text string extends in the $+x$ direction.

Symbolic Names. Format: 9 SymbolName;

This extension is used to associate a symbolic name with a symbol definition. Example:

```
DS 1 100 1; 9 InputPad;
LNM; (...rest of definition);
DF;
```

End command. The parser might issue a warning message if it finds any non-blank characters following an End command.

7.3.1.2 Interpreter

The job of the interpreter is to retain symbol definitions and to instantiate and transform symbols when they are called.

Transformations. (See also [Newman & Sproull 1979].) When we are expanding a symbol, we need to apply a transformation to the specification of an item in the symbol definition to get the specification into the coordinate system of the chip. There are three sorts of measurements that must be transformed: distances (for widths, lengths), absolute coordinates (for "points" in all primitives) and directions (for boxes). Distances are never changed by a symbol call, because we allow no scaling in the call. Thus a distance requires no transformation.

A point (x,y) given in a symbol definition is transformed to a point (x',y') in the chip coordinate system by a 3x3 transformation matrix T: [x' y' 1] = [x y 1] T
T is itself the product of primitive transformations specified in the call: T = T1 T2 T3, where T1 is a primitive transformation matrix obtained from the first transformation primitive given in the call, T2 from the second, and T3 from the third (of course, there may be fewer or more than 3 primitive transformations specified in the call). These matrices are obtained using the following templates for each kind of primitive transformation:

Tab.	Tn =	1	0	0
		0	1	0
		a	b	1

$$\text{MX.} \quad Tn = \begin{matrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

$$\text{MY.} \quad Tn = \begin{matrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

$$\text{Rab.} \quad Tn = \begin{matrix} a/c & b/c & 0 \\ -b/c & a/c & 0 \\ 0 & 0 & 1 \end{matrix} \quad \text{where } c = \text{Sqrt}(a^2 + b^2)$$

Transformation of direction vectors (x y) is slightly different than the transformation of coordinates. We form the vector [x y 0], and transform it by T into the new vector [x' y' 0]. The transformed direction vector is simply (x' y'). Some output devices may require rotations to be specified by angles, rather than direction vectors. Conversion into this form may be delayed until necessary to generate the output format. Then we calculate the angle as ArcTan(y/x), applying care when $x = 0$.

Nested calls require that we combine the transformations already in effect with those specified in the new call. Suppose we are expanding a symbol a, as described above, transforming each coordinate in the symbol to a coordinate on the chip by applying matrix Tac. Now we encounter, in a's definition, a call to b. What is to happen to coordinates specified in b? Clearly, the transformations specified in the call will yield a matrix Tba that will transform coordinates specified in symbol b to the coordinate system used in symbol a. Now these must be transformed by Tac to convert from the system of symbol a to that of the chip. Thus, the full transformation becomes

$$[x' \; y' \; 1] = [x \; y \; 1] \; Tba \; Tac$$

The two matrices may be multiplied together to form one transformation Tbc = (Tba Tac) that can be applied to convert directly from the coordinates in symbol b to the chip. This procedure can be carried to an arbitrary depth of nesting.

To implement transformations, we proceed as follows: we maintain a "current transformation matrix" T, which is initialized to the identity matrix. We use this matrix to transform all coordinates. When we encounter a symbol call, we:

1. "Push" the current transformation and layer name on a stack.
2. Collect the individual primitive transformations specified in the call into the matrices T1, T2, T3 etc.
3. Replace the current transformation T with T1 T2 T3 ... T; i.e., premultiply the existing transformation by the new primitive transformations, in order).
4. Now process the symbol, using the new T matrix.
5. When we have completed the symbol expansion, "pop" the saved matrix and layer name from the stack. This restores the transformation to its state immediately before the call.

Precision in numeric calculations. The numbers given in a CIF file generally undergo substantial processing before being delivered to a pattern generator or a plotter. We must take care that the processing not introduce numerical errors due to insufficiently precise arithmetic in the computer. An argument given below suggests that floating-point arithmetic with 24 or more bits of mantissa will suffice.

First, we note that it is not necessary for CIF to perform calculations with a precision greater than, say, $\lambda/5$. Here we are using λ to denote the characteristic resolution of the lithographic process. Variations in position on the order of $\lambda/5$ will not reproduce.

CIF describes geometry in units of 1/100 micron. This precision should suffice to describe integrated-circuit devices to a precision of $\lambda/5$ at the limit of device operation, roughly .2 μm, $\lambda = .1$ μm. If we imagine the largest design being about 10 centimeters on a side, the largest CIF coordinate would need to be 10^7. To store coordinates precise to 1 part in 10^7 we require 24 bits. An additional sign bit is required because transformations can mirror or translate negative coordinates to positive ones. Happily, these arguments mean that 32-bit floating point numbers, such as are used in the IBM/370 and many other computers, easily suffice for holding CIF coordinates.

These arguments also mean that coordinates may be saved as integers, in units of 1/100 micron. A representation using 25 or more bits (24 plus sign) is required. Note that the scaling by a/b of numbers in a symbol definition may give rise to coordinate numbers with a grain finer than 1/100 micron; these can safely be rounded to units of 1/100 micron.

This discussion ignores any problems with precision in the coordinate systems used to drive plotters, pattern generators, etc. If integer representations are used, conversion from the 1/100th micron system into the system of the output device must be delayed as long as possible to avoid losing precision.

Precision in transformations. Were it not for rotations, the transformation functions would introduce no numerical difficulties. There are two problems introduced by rotations: imprecision in the matrix values (a/c, b/c, etc. in the notation above), and the accumulation of error due to nested symbol calls. A single rotation may introduce some error due to the matrix multiplication (e.g., x' = x*a/c-y*b/c). This calculation introduces approximately 4 round-off errors (the error will either be 1/2 least significant bit if "rounded" arithmetic is used or 1 least significant bit if "chopped" arithmetic is used).

The nesting problem is more dangerous, because the computation of the transformation matrix is *incremental.* If a symbol calls another with a 1-degree rotation, which in turn calls a third with a 1-degree rotation, and so on to a depth of 90 calls, the calculation of the current transformation matrix may have accumulated considerable error. Very roughly, each level of nesting may introduce 4 round-off errors in a coefficient. So to allow nesting to a depth of n we should use arithmetic with a precision of 1 part in $10^7(4n+4)$. Equivalently, we should use floating-point arithmetic with

$26+\log_2 n$ bits in the mantissa. Calls that do not use rotation or that involve only rotations by multiples of 90 degrees (which use only perfectly accurate coefficients 0 and 1, and therefore give rise to perfectly accurate matrix multiplication) do not need to be counted in n. All of this suggests that the extremely cautious interpreter will keep track of symbol nesting depth, and issue a warning if it suspects arithmetic precision is insufficient.

Symbols. There is no sensible way in which a symbol may be invoked recursively (i.e., call itself, either directly or indirectly). Programs that read the intermediate form might check that no recursion occurs. This can be achieved by retaining a single flag with each symbol to indicate whether the symbol is currently being instantiated; the flags are initialized to "false." When a symbol is about to be instantiated, we check the flag; if it is "true," we have detected recursion, and so print an error message and do not perform the call. Otherwise, we mark the flag "true," instantiate the symbol as specified, and mark the flag "false" when the instantiation is complete.

7.3.1.3 Output

The most difficult problem faced in the output module is that of *device independence*: how is the geometry specified in the CIF file to be generated on a particular output device with sufficient precision? A complete discussion of methods used to drive different output devices is beyond the scope of this document. This section attempts instead to discuss the device-independence problem in general terms.

Driving any device will require choosing techniques for generating the geometry CIF specifies on that device. These choices are simple for devices such as pen plotters, which can trace arbitrary shapes with modest precision. By contrast, a pattern generator may allow only rectangular "flash" exposures of limited size. Moreover, the precision with which the CIF geometry is reproduced on the reticle is extremely important.

It is often impossible to guarantee that the geometry will be reproduced *exactly*; instead we require that the software and pattern generator exert their "best efforts" to conform to the CIF. Once again, the parameter λ enters: the design rules allow us some leeway in making mask patterns. The exact details of how much error can be tolerated will depend on the entire fabrication process, not just the pattern-generation step.

The fact that pattern generators cannot reproduce CIF exactly leads to a requirement for *approximations*. Examples of the kinds of approximations that may be necessary are:

1. Round flashes might be approximated with octagons, or with two overlapping square flashes rotated 45 degrees (Figure 7.3.2). Another approximation might use more flashes at smaller rotational increments.

2. Wires cannot be constructed exactly on output devices that cannot construct circles. One possibility is to use the round flash approximations described above. Another is to "square off" the ends of wires by extending them by the half-width of the wire, and thus generate only box shapes

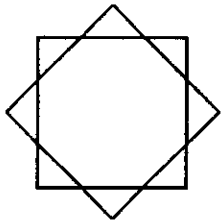


Figure 7.3.2 Approximation of a Round Flash

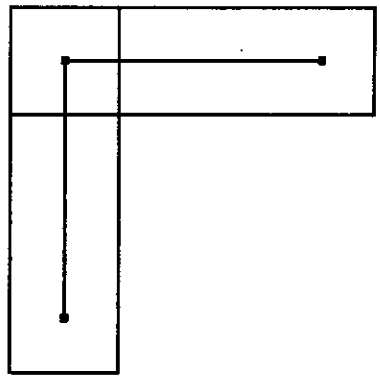
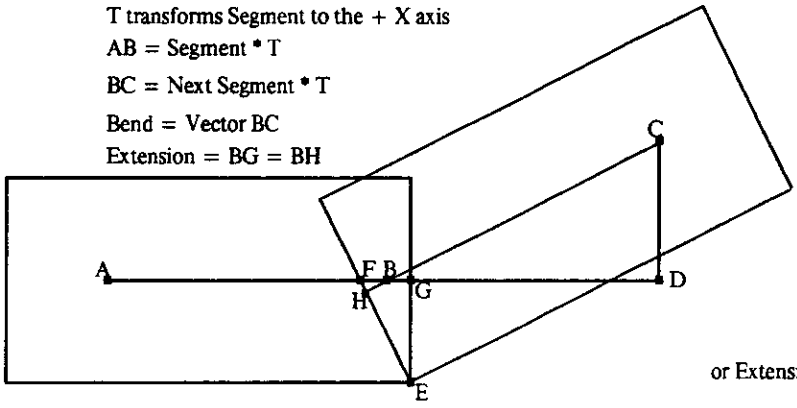


Figure 7.3.3a Approximation of a Wire



Similar triangles BCD, EFG, BFH
BC:CD:DB::EF:FG:GE::BF:FH:HB

FG = FB + BG
= BH * (BC/DB) + BG
= (1 + BC/DB) * BG

BG = FG/(1 + BC/DB)
= GE * (CD/DB) / (1 + BC/DB)
= GE * (CD/(DB + BC))

or Extension = w/2 * Y [Bend] / (X[Bend]] + LENGTH[Bend])

Figure 7.3.3b Converting Wires to Boxes

(Figure 7.3.3a). This approximation works nicely for segments that meet at right angles, but may cause problems if wires or wire segments are connected at arbitrary angles. The technique can be improved by adjusting wire lengths and positions (Figure 7.3.3b). The width of the boxes is the same as the width of the wire, but the length of the box is adjusted to reduce unfilled wedges are overlapping "ears." An algorithm for constructing boxes in this way from a wire is given below. If the wire is specified in a symbol definition, the approximation need be computed only once, and can then be used each time the symbol is instantiated.

The following algorithm for decomposing wires into boxes was developed by Carver Mead, and first implemented at Caltech by Ron Ayres; it was was further modified to be consistent with the use of direction vectors, to allow more general path lengths, and to avoid use of trigonometric functions. Note that this decomposition covers more area than the locus of points within w/2 of the path for small angles of bend, but less area for sufficiently sharp bends; in particular, if a path bends by 180 degrees (reverses) it will have no extension past the point of reversal (it is missing a full semicircle). Other decompositions are possible, and may better approximate the correct shape.

```
Let the wire consist of a path of n points p1,...,pn.
Let w represent the width of the wire.
    IF n = 1 THEN
        {MAKEFLASH[Diameter ← w; Center ← p1]: "single-point gets a flash";
        DONE;};
    i ← 1;
    OldExtension ← w/2; "initial end of wire"
    Segment ← p2 - p1; "p1 and p2 are points in path, Segment is a vector (a point)"
    "LoopConditions:"
    FOR pi, pi+1 in path UNTIL pi+1 is last DO
        "calculate the box for the segment from pi to pi+1:"
        IF pi+1 is last THEN
            { Extension ← w/2; "final end of wire" }
         ELSE
            { "compute Extension for intermediate point:"
            NextSegment ← pi+2 - pi+1; "next vector in path"
            T ← MATRIX[ X[Segment],  -Y[Segment],
                        Y[Segment],   X[Segment] ];
            "T transforms Segment to +x axis."
            Bend ← MULTIPLY[ NextSegment, T ]; "relative direction vector"
            "if Bend is (0 0), delete pi+1, reduce n, and start over"
            Extension ← w/2 * ( ABS[Y[Bend]] /( LENGTH[Bend] + ABS[X[Bend]] ) ); };
        MAKEBOX[
            Length    ← LENGTH[Segment] + Extension + OldExtension;
            Width     ← w;
            Center    ← ( pi + pi+1 )/2 +( Segment / LENGTH[Segment] ) *(Extension - OldExtension)/2;
            Direction ← Segment; "careful, may be zero vector" ];
        i ← i + 1;
        OldExtension ← Extension;
        Segment ← NextSegment; "next vector in path"
        ENDLOOP;
    DONE;
```

This is only one of many possible algorithms. When a particular approximation for wires is chosen, the ideal concept of a wire with semi-circular ends should always be kept in mind.

Polygons. Polygons may need to be approximated by boxes. In some cases, the approximation can be exact or nearly so (Figure 7.3.4a). In others, it will be impossible to avoid introducing "ears" or unfilled wedges (Figure 7.3.4b). Alas, we have no polygon-approximation algorithm to recommend.

The designer of the output module must be careful to handle self-intersecting polygons properly. If the implementation cannot deal with these polygons, the program should at least detect self-intersecting polygons and issue appropriate error messages.

The correct processing of self-intersecting polygons may depend on the type of output device. A geometric transformation can be applied to generate a new set of polygons that are not self-intersecting [Sutherland 1978]; these can then be approximated with the same technique as normal polygons. For line-by-line raster-scanned devices the following approach will give the proper result. With each polygon edge, an orientation is maintained (upward-moving or downward-moving), given by the original ordering of the points in the descriptive path. Moving across a scan line, a counter keeps track of how many intersections with the two types of edges have been encountered: the counter is incremented when an upward-moving edge is crossed, and decremented when a downward-moving edge is crossed. Whenever the the counter is non-zero, the scan line is inside the polygon. In these regions of the scan-line, the image of the polygon is generated.

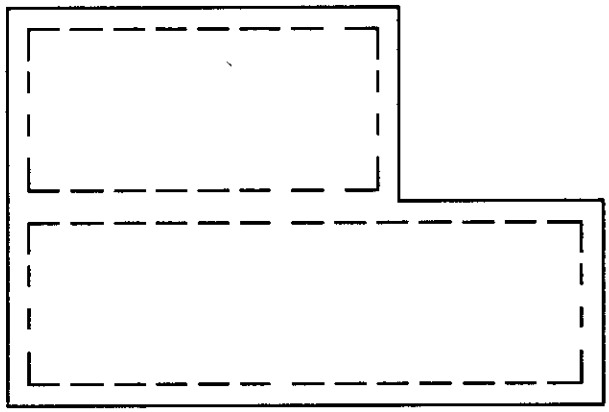
7.3.2 A Program for Processing CIF

Once an IC design has been expressed in CIF, a number of different programs are needed to create different representations. An obvious need is for a program to convert CIF into a form suitable for generating masks. Other programs may create checkplots from CIF, a file listing design rule violations, or an input file for a simulator. All of these programs require a CIF file as input and produce output in some other form. Accordingly, the "front ends" of these programs are likely to be very similar and could therefore be shared.

The remainder of this section details a modularization of one program that inputs CIF and plots it. The document is intended to provide guidelines for one reasonable structure for such a program, which is not optimized for any particular output device. The reader should be familiar with the syntax and semantics of CIF, and have given the problem of processing CIF some thought before trying to fully understand the program structure. The procedure declarations are written in a Pascal-like language, the main difference being that routines may return arbitrary structures. *All parameters are passed by value.*

The system is composed of a Parser, an Interpreter, and an Output stage. The Parser reads characters from an input file, checks syntax and generates well-defined, parameterized procedure calls on routines in the Interpreter. These calls closely reflect the structure of CIF; for each input statement, one command to the Interpreter is generated. The Interpreter takes the command and either generates a call, such as *Output a wire...*, to the Output module or saves it (in some internal

(a)



(b)

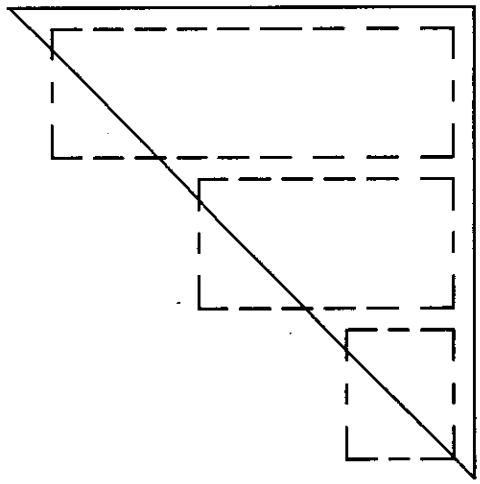


Figure 7.3.4 Approximations of Polygons

form) for later recall. The Output module produces an output file, plot, or display containing the result of output commands. Together the Parser and the Interpreter comprise a front end that may be combined with individual Output stages for any of a number of devices. The Parser and the Interpreter can be written with about one man-month of effort. Depending on the complexity of the output device, new Output stages may require from one man-week to one man-month of effort to implement.

The section on each major component contains suggestions on how it might be implemented. It bears repeating that the implementation suggested is not intended to be particularly efficient for all output devices. Raster scan devices, for example, may require output commands to be sorted in some order. This affects the internal structure of the Interpreter, and may dictate major changes in the way the Interpreter is implemented. Nevertheless, the routines that the Interpreter provides to the Parser should have the same input/output behavior as before.

Several modules are needed by all three phases of the CIF processing program; these modules provide services for reporting errors, typing messages on the user's terminal, and other utility functions. Each module contains a routine to initialize the module that must be called before any of the others, and a routine called to terminate the interaction with the module. There must also be a main program that initializes each module, interacts with the user to obtain input parameters (e.g. file names), contains a loop to scan the next CIF statement, and cleans up.

Data Type Definitions. A number of data type definitions are required by various modules and are included here. It is suggested that procedures that support these data types (such as routines to add a Point to a Path) be packaged together in one module with the TYPE declarations to form suitable data abstractions. The types used are only guidelines, as each implementor has his/her own preferences for data structures.

```
Boolean InitTypes()

Boolean FinishTypes()

Type Point =
        Record
        Integer X, Y;
        End;

Type LinkedPoint =
        Record
        Point Value;
        ↑LinkedPoint Next;
        End;

Type PathRecord =
        Record
        ↑LinkedPoint First, Last;
        Integer Length;
        End;

Type Path = ↑PathRecord;

Path AllocatePath()
        Paths are queues of Points; used to specify the perimeter of polygons and the trail that wires follow.

FreePath(Path Name)
        Release all of the storage held by Name.
```

```
AppendPoint(Path Name, Point Value)
        Adds a point onto the end of Name.

Boolean, Point RemovePoint(Path Name)
        Returns True and the next point on Name or False and garbage if Name is an empty path.

Integer PathLength(Path Name)
        Returns the number of points remaining on Name.
```

TLists are queues of transformation commands (the commands may be represented in any convenient manner, say as strings -- "R 1 1" -- or perhaps as triples of integers -- "0 1 1" -- where the "0" is interpreted to mean "Rotate"). Used to pass a list of CIF transformations to the Interpreter, which ultimately converts the list into a transformation matrix.

```
Type TType = {Mirror, Translate, Rotate};

Type TEntry =
        Record
        Select TType: ThisOne FROM
        Mirror => [Coords: {X,Y}],
        Translate => [Integer X, Y],
        Rotate => [Integer XRot, YRot],
        End;

Type LinkedTEntry =
        Record
        TEntry Value;
        ↑LinkedTEntry Next;
        End;

Type TRecord =
        Record
        ↑LinkedTEntry First, Last;
        Integer Length;
        End;

Type TList = ↑TRecord;

TList AllocateTList()

FreeTList(TList Name)

AppendTList(TList Name, TEntry Value)

Boolean, TEntry RemoveTList(TList Name)

Integer TListLength(TList Name)
```

Error Reporter. Each module reports errors that it encounters through a call to "Report" in the Error Reporter. This module takes care of identifying the input file context of the error and reporting and logging errors in a consistent manner.

```
Type Error = {FatalSyntax, FatalSemantic, FatalOutput, FatalInternal, Advisory, Fatal, Other};

Boolean InitError()
        Initializes the Error module, returns False on error.

Boolean FinishError()

Report(String Message; Error Kind)
        Indicates the current context (by a call to Identify() in the Input Module) and prints Message (a call
        to SendMessage() in the Output Module), keeps a count of each type of error. Advisory errors are
        preceded by a "Warning" tag.

Array of Integer ErrorSummary()
        Returns the number of each type of error encountered so far (the array is as long as there are
        different kinds of errors).
```

Input. All Input functions are carried out by this module, in addition, it keeps track of the input context and provides a means of displaying it (for use by the Error Reporter).

Boolean InitInput()
Initializes the Input module, returns False on error.

Boolean FinishInput()
Cleans up the Input operations (e.g. closes open files), returns False on error.

Boolean InFromFile(String Filename)
Fixes things so that input characters come from FileName (i.e. the CIF file).

Character GetChar()
Removes and returns the next character from the currently open file. Succesive calls to Getchar return successive characters from the file. Returns EOF when the end of file is reached.

Character Peek()
Returns the next character from the currently open file without removing it from the input stream (providing one character look ahead).

Boolean EndOfFile()
Returns True if the end of the input file has been reached.

Character Flush(Character BreakChar)
Tosses out characters up to and including the first BreakChar, returns the character it stopped on (either BreakChar or EOF). Useful in error recovery.

Identify()
Identifies the context within the input file by line number, contents, or whatever. Calls SendMessage in the Output module to have the line printed on the terminal.

7.3.2.1 Parser

The Parser is responsible for reading the input file and generating calls to the Interpreter. Its primary responsibility is syntax checking (and perhaps some limited semantic checking).

Type CommandType = {Wire, DefineStart, DefineEnd, DeleteDef, CallSymbol, Layer, Flash, Polygon, Box, End, Comment, NullCommand, UserCommand, SyntaxError, SemanticError};

These are the codes returned by ScanStatement; there is one for each type of command possible (cf. CIF syntax).

Boolean InitParser()

Boolean FinishParser()

CommandType ScanStatement()
Scans the statement (= {*blank*} [*command*] *semi*) beginning at the current character, munching chars up to and including the closing *semi*. Once a syntactically valid statement has been scanned a procedure in the Interpreter is called. Returns a code for the type of statement scanned.

Implementation. One simple way to do syntax analysis is to write a procedure to parse each of the nonterminals (e.g. *semi*, *sep*, *blank*) in the CIF syntax. Characters from the input file are obtained using the procedures in the Input module, and the user is notified of errors through the Error Reporter. A CASE statement on the next character in the input file suffices to decide which type of statement is expected; each arm of the CASE statement is a sequence of calls on the routines that parse nonterminals. Error recovery can be relatively crude inititally, for example an error might cause the Parser to ignore characters through the next *semi*. Front ends that are intended to process CIF that was generated by hand should be especially careful to catch machine dependent errors (e.g. word length limitations). The function of the Parser is essentially limited to

syntax analysis; it should do as little interpretation as possible.

7.3.2.2 Interpreter

The Interpreter receives commands from the Parser as each CIF command is parsed. It keeps track of the current state (whether or not a definition is in progress, the current layer, etc.) and based on that information makes the appropriate calls on the Output module. Each call on ICallSymbol, IWire, IFlash, IPolygon, or IBox in the Interpreter will either cause an output command to be generated, or will be stored (because it is part of a symbol definition). Those items that are stored will be output each time the symbol that they are part of is called outside of a symbol definition.

Boolean InitInterpreter()
Initializes the Interpreter, returns False on error.

Boolean FinishInterpreter()
Cleans up after the Interpreter, returns False on error.

IDefineStart(Integer SymbolNumber, Multiplier, Divisor)
Get things set up so that items are stored (probably in some compiled form) as part of this symbol definition. IDefineStart will have to create a symbol table entry for SymbolNumber after making sure that no other symbols are currently being defined. If SymbolNumber is already defined, a warning message should be given. Multiplier and Divisor are used to scale incoming points and dimensions.

IDefineEnd()
Finish up the current symbol.

IDeleteDef(Integer NSym)
Delete all symbols numbered NSym and above.

ICallSymbol(Integer SymbolNumber; TList A)
Make a call to SymbolNumber with the transformations given by list A. If a symbol definition is in progress the call will be stored for later recall. Otherwise the Interpreter must retrieve the primitive objects from SymbolNumber and generate a call on the Output module for each.

ILayer(String LayerName)
Switch to layer LayerName.

IWire(Integer Width; Path A)

IFlash(Integer Diameter; Point Center)

IPolygon(Path A)

IBox(Integer Length, Width; Point Center; Integer XRotation, YRotation)
A call on any of these routines will cause one of two things to happen: if a symbol definition is in progress, some compiled form of the command will be saved in the internal data structures within the Interpreter. If there is no symbol definition in progress, a call will be made on Output routines.

IComment(String Contents)
The interpretation of comments will vary with the application of the CIF processing program. Where some form of checkplotting is being done, comments will likely be ignored.

IUserCommand(Integer Command; String UserText)
The meaning of user commands will also vary, those user commands not recognized by the interpreter should be passed on to the Output module.

IEnd()
An End command has been scanned.

Implementation. The Interpreter contains all of the symbol table routines necessary to maintain the symbol number / symbol definition correspondence, a storage manager that allows

arbitrarily large symbol definitions to be stored, and also does semantic checking for nonsense arguments or other problems (e.g. output to layer ZZZZ). For every command parsed the Interpreter takes one of four actions:

1. *It passes the command along to the Output module.* This happens when a primitive (Wire, Box, Polygon, Flash) occurs outside of symbol definition.

2. *It stores the command in the current symbol definition.* This happens when a Call or a primitive is encountered within a symbol definition. Symbol numbers may be hashed into a table that holds a pointer to the first and last primitives in the definition. The primitives can be stored as a linked list; primitives are added to a symbol by tacking them on after the last entry. Each type of primitive requires that different information be stored with it. Calls need an incremental transformation matrix (see GetLocal below) and the symbol number called. Boxes require a length, width, center and layer.

3. *It changes some internal state information (e.g. DF causes the current symbol to be closed) and no output action is taken.* This case includes layer changes, DS, DF, DD.

4. *It recalls a symbol definition, making a number of calls on the Output module.* In calling a symbol, the Interpreter must set up the transformation stack correctly. For example, the command "C 114 T 10000,0 M X" is encountered. The Interpreter does a SaveTransformation() to begin a new frame of reference. It then calls Translate(10000,0) and Mirror(True) to set up the incremental transformations. It looks up symbol 114, obtaining a pointer to the first primitive in the definition; at the same time, symbol 114 is marked as "being expanded." It then makes a call on the Output module for each piece of primitive geometry. The Output module will make calls on TransformPoint in order to interpret coordinates in the current context. If a symbol call is encountered in the symbol being expanded, the entire process is nested one level deeper. When the end of the symbol definition is reached, the Interpreter resets the expansion flag, and calls RestoreTransformation() to reset the frame of reference.

Transformations. The Transformation module contains a stack of transformation matrices; the top matrix is the "current" matrix. A new transformation system is created by a call to SaveTransformation, which pushes a new matrix (initialized to an identity matrix) onto the stack. Subsequent calls to Mirror, Translate, and Rotate modify the top matrix. On the first call to TransformPoint, the top matrix is postmultiplied by the matrix that is underneath it on the stack and replaced by the result. This represents the new frame of reference, which is the previous frame with the new transformations applied; the point passed to TransformPoint is postmultiplied by this new matrix to give the final point. Further calls on TransformPoint use this matrix, until the context is changed; it is illegal to issue more transformation commands without changing the context (by a call on SaveTranformation or RestoreTransformation). The previous context is

restored by a call to RestoreTransformation, which pops the stack.

```
Type TransformationMatrix =
        Record
        Real a11,a12,a21,a22,a31,a32,a33;
        Boolean IdentityMatrix;
        End;

Boolean InitTransformation()

Boolean FinishTransformation()

Rotate(Integer XRotate, YRotate)
        Builds the rotation into the current transformation matrix.

Translate(Integer XTrans, YTrans)
        Builds the translation into the current transformation matrix.

Mirror(Boolean XCoord)
        Builds the mirroring (either the X coordinates or the Y coordinates) into the current transformation
        matrix.

Point TransformPoint(Point A)
        Replaces the top matrix with the product of the top matrix and the previous matrix (only if the two
        have not been multiplied yet, i.e. on the first call to TransformPoint) and returns a point as
        transformed by the current (top) transformation matrix.

SaveTransformation()
        Pushes a new identity transformation matrix onto the stack, thereby starting a new relative coordinate
        system.

RestoreTransformation()
        Pop the transformation stack, return to previous coordinate system.

TransformationMatrix GetLocal()
        Returns the top matrix on the stack, which represents the incremental transformations applied since
        the last SaveTransformation. Useful for saving transformations for a symbol call.

ApplyLocal(TransformationMatrix T)
        Apply T to the top of stack, i.e. top ← top x T. Used to set up the context for a symbol call.
```

7.3.2.3 Output

The Output module produces the output (e.g. raster scan bit maps, CalComp plot commands, MEBES trapezoids and rectangles, or some intermediate form that requires further processing) that is the end result. It contains routines for each of the primitive geometric constructs defined in CIF. The CIF geometric primitives have not been subdivided (to the level of rectangles, for instance) so that redundant work is eliminated and so that the advantages of each output device may be fully utilized. For example, while a vector plotter would like to have rectangles expressed as four corners, a PG machine would require that a four-corner representation be converted to a length, width, and angle. By forcing the Output stage to do the conversion from CIF primitive to whatever form is best for the output device, such convert/unconvert problems are avoided.

Coordinates passed to the Output stage are always interpreted within the current transformation context, that is, the final coordinates are obtained by a calls to "TransformPoint" in the Transformation Module (which is part of the Interpreter, see above). The Output module may modify the transformation context (through other calls to the Transformation Module) in order to implement transformations that it may require, but should reset the context to the state it was in prior to the output call before returning.

Boolean InitOutput()
Initializes the Output module, returns False on error.

Boolean FinishOutput()
Cleans up, returns False on error.

Boolean OutputToFile(String FileName)
Fixes things so that the output goes to FileName (may not be needed in some cases).

Boolean MessagesToFile(String FileName)
Routes messages sent by SendMessage to FileName instead of the terminal, useful for keeping a log of messages printed to the user.

SendMessage(String Text)
Prints Text wherever messages get printed, probably the terminal, intended for reporting error messages somewhere.

OutputWire(String Layer; Integer Width; Path A)

OutputPolygon(String Layer; Path A)

OutputBox(String Layer; Integer Length, Width; Point Center; Integer XRotation, YRotation)

OutputFlash(String Layer; Integer Diameter; Point Center)
These commands cause each of the primitive geometric items to be output.

OutputUserCommand(Integer Command; String UserText)
The meaning of user commands will vary, those user commands not recognized by an installation should be flagged with warning messages.

7.4 A Final Note

It is our ambition to refine and improve CIF and its description in this chapter. Comments and questions about CIF or about its presentation here are most welcome. Please send them to:

Robert F. Sproull
Computer Science Dept.
Carnegie-Mellon University
Pittsburgh, Pa. 15213

or to ARPANET address:

Sproull@CMUA

References

[Ayres 1979]
R. Ayres, "Silicon Compilation — A Hierarchical Use of PLAs", *Proc. Caltech Conf. on Very Large Scale Integration*, January 1979.

[Johannsen 1979]
D. Johannsen, "Bristle Blocks: A Silicon Compiler", *Proc. Caltech Conf. on Very Large Scale Integration*, January 1979.

[Locanthi 1978]
B. Locanthi, "LAP: A Simula Package for IC Layout", Caltech Computer Science Display File, July 1978.

[Mead & Conway 1980]
C. A. Mead and L. A. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA., 1980.

[Newman & Sproull 1979]
W. M. Newman and R. F. Sproull, *Principles of Interactive Computer Graphics*, 2nd Ed., McGraw-Hill, 1979.

[Sutherland 1978]
I. E. Sutherland, "The Polygon Package," Caltech Computer Science Display File, March 1978.

[Wirth 1977]
N. Wirth, "What Can We Do About the Unnecessary Diversity of Notations for Syntactic Definitions?", *Communications of the ACM*, November 1977.