CALIFORNIA INSTITUTE OF TECHNOLOGY

Computer Science Department

Silicon Structures Project

Technical Report 2686

The Caltech Intermediate Form for LSI Layout Description

by

Robert Sproull and Richard Lyon

revised by

Stephen Trimberger

February 11, 1980

Silicon Structures Project

sponsored by

Burroughs Corporation, Digital Equipment Corporation, Hewlett-Packard Company, Honeywell Incorporated, International Business Machines Corporation, Intel Corporation, Xerox Corporation, and the Natioal Science Foundation.

The Caltech Intermediate Form for LSI Layout Description

The Caltech Intermediate Form (CIF Version 2.0) is a means of describing graphic items (mask features) of interest to LSI circuit and system designers. Its purpose is to serve as a standard machine readable representation from which other forms can be constructed for specific output devices such as plotters, video displays, and pattern-generation machines. The intermediate form is not intended as a symbolic layout language: CIF files will usually be created by computer programs from other representations, such as a symbolic layout language or an interactive design program. Nevertheless, the form is a fairly readable text file, in order to simplify combining files and tracing difficulties.

The basic idea of the form is to specify literally every geometric object in the design using ample precision. Use of this form provides participating design groups easy access to output devices other than their own, enables sharing designs with others, allows combining several designs to form a larger chip, and the like. It is not necessary for all participating groups to implement the entire set of features of CIF, as long as their programs and documents contain warnings about unimplemented functions; nevertheless, the syntax must be correctly interpreted by all programs that read CIF, to assure a reasonable result.

CIF thus serves as the common denominator in the descriptions of various integrated system projects. No matter what the original input methods are (hand layout and coding, or a design system), the designs will be translated to CIF as an intermediate, before being translated again to a variety of formats for output devices or other design aids.

The original CIF was conceived by Ivan Sutherland and Ron Ayres in 1976. Subsequent improvements were contributed by Carlo Sequin, Douglas Fairbairn, and Stephen Trimberger.

This specification is divided into four parts: a description of the syntax of the form, a description of the semantics, an explanation of the transformations used, and a discussion of the conversion of wires to boxes.

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 [1] 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 ")".
```

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. Many of the decisions in designing the file format were made to avoid ambiguity or small but troublesome errors: floating point numbers are avoided; there are no iterative constructs, though there may be in future additions to CIF.

A simple file format might include only primitive geometric constructs, such as polygons, boxes, flashes and wires. Unfortunately, the geometric description of a chip with hundreds of thousands of rectangles on it would require an immense file of this sort. Consequently, we have made provision for defining and calling symbols; this should reduce the size of the file substantially.

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 description below mentions implementation suggestions or cause for caution inside brackets [ ].

Measurements. The intermediate form uses a right-handed coordinate system shown in Figure 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, not in terms of the various sizes and mirrorings of the intermediate artifacts.) The units of distance measurement are hundredths of a micron (um); there is no limit on the size of a number. [Programs reading numbers from CIF files should check carefully to be sure that the number does not overflow the number of bits in the internal representation used, and should specify their own limits, if any.]

Directions. Rather than measure rotation by angles, CIF uses a pair of integers to specify a "direction vector." (This eliminates the need for trigonometric functions in many applications, and avoids the problem of choosing units of angular measure.) The first integer is the component of the direction vector along the x axis; the second integer 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. [A (0 0) direction vector may be defaulted to mean the +x axis; a warning should be generated].

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 which define a box are shown graphically in Figure 1. Center and direction (optional, defaults to +x axis) 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.

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 2). [Programs that try to interpret

polygons may place various restrictions on their paths; no set of constraints has been generally accepted, and no program currently exists for converting completely general polygons to pattern generator output.]

Flashes: RoundFlash Diam 200 Center -500,800; (or R200 -500 800;); The diameter of a flash is sufficient to specify its shape, and the center specifies its position. (see Figure 2). [Some programs may substitute octagons, or other approximations, for round flashes.]

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 2). 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. [For output devices that have a hard time constructing circles, we approximate the ideal wire with squared-off ends. Notice that squared-off ends work nicely for segments meeting at right angles, but cause problems if wires or wire segments are connected at arbitrary angles. A way to circumvent this problem is to convert, prior to output, any wires in a file into connected sets of boxes of appropriate length, width, angle and center position (Figure 3). The width of each box is the same as the width of the wire. The length of the boxes must be adjusted to minimize unfilled wedges and overlapping "ears". An algorithm for constructing boxes from a wire description is given in a later subsection. If the wire is specified within a symbol definition, the approximation need be computed only once, and can then be used each time the symbol is instantiated.]

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 (layer mode is preserved across symbol 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"). [The intention of the layer specification command is to label locally the layer for a particular geometry. It is therefore senseless to specify a box, wire, polygon or flash if no layer has been specified. In order to detect this error, the command LZZZZ is

implicitly inserted at the beginning of the file, and as the first command of a symbol definition (DS: see below). Any attempt to generate geometric output on layer ZZZZ will result in an error.]

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 character of the name denotes 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 name layer names will be defined as needed.

[Programs that read CIF should check to be sure that layer names used do in fact correspond to fabrication masks 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.]

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 the 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. The main reason for symbol facilities is to limit the file size: if the symbol mechanism is not adequate for some application, the desired geometry can still be achieved with the use of symbols, and more use of explicit geometrical primitives. [Symbols need not be used at all; this eliminates the need for intermediate storage for symbol definitions, but results in larger design files. Machines which must process a fully-instantiated representation of a layer (such as pattern generators) might only accept CIF files without symbol definitions, to reduce the cost of implementation. Therefore, it would be useful to have a program that would convert general CIF files to fully instantiated CIF files, and maybe to sort by layer, location, or whatever.]

The ability to call for iterations (arrays) of symbols is not provided in CIF Version 2.0. This is primarily due to the difficulty of defining a standard method of specifying iterations, without introducing machine-dependent computation problems. It is possible to achieve a great deal of file compaction by defining several layers of symbols (e.g. cell, row, double-row, array, etc). However, the ability to iterate symbol calls is a likely prospect for a future addition to CIF.

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 the symbol 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 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 does not provide scaling or 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. [If a definition redefines a symbol that already exists, the previous definition is discarded: a warning message should be generated. When several people contribute to a design, some symbol management is therefore necessary: see Deleting symbol definitions below.]

There is only one restriction on the placement of symbol definitions in the file: a symbol must be defined before its instantiation becomes necessary. This constraint can be satisfied by placing all symbol definitions first in the file, and then calls on the symbols. In fact, it is often convenient to have the file consist exclusively of symbol definitions and ONE call on a symbol. This call will be the last command in

the file before the end command. [A CIF file is meant to be interpreted in one pass. Symbol calls not embedded within definitions are meant to be instantiated before any more of the file is read. This is significant in some cases. See also the note following the first paragraph of Deleting symbol definitions, 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 with its "symbol index", established when the symbol was defined. [The symbol index refers to whichever cell happens to be defined with that index when the symbol is called, not the symbol that might have been defined with that index when the symbol in question was defined. Symbol calls are resolved in the context in which they are called, not the context in which they are written. See also the note following the first paragraph of Deleting symbol definitions, below.]

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 the 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 transformation 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 the subsection on transformations).

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 the subsection transformations). [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 should 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, 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.]

Layer settings are preserved across symbol calls and definitions. Thus, in the sequence:

```
LNM;
R6 20 0;
C 57 T45 13;
DS 114...;
DF;
LNM;
R3 0 0;
```

the second LNM is not necessary, regardless of 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 intermediate form files can be appended and processed as one. In such 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 proper interpretation of the DD is as text removal. All definitions of symbols greater than or equal to the argument of the DD are removed. Further processing of the file continues as if the text of those symbols had not been included in the file. Thus, a symbol call always refers to the most recent definition the symbol with the given symbol index. This can be seen in the following example in which the call on symbol 1 produces a call on the second definition of symbol 2:

```
DS 2: ... DF:
DS 1: ... C 2: ... DF:
DD 2:     (Def 2 is deleted and removed):
DS 2: ... ... DF:   (This is now the only version of symbol 2):
C 1:  (This calls symbol 1 which calls the new symbol 2):]
```

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
```

User expansion: `3'SYMBOL.LIBRARY'; 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 (1) requesting that another file be "inserted" at this point in the processing, thus simplifying the use of symbol libraries; (2) inserting instructions to a preprocessor that will be ignored by any program reading only standard intermediate form constructs; ore (3) 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.

Comments: `(HISTORY OF THIS DESIGN:);` The comment facility is provided simply to make the file easier to read. [It is possible 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. [Programs that read CIF should gave an error message if the file ends without an End Command, or a warning if more text other than blanks follows the E.]

Data Conventions for Transporting CIF Code

The description of CIF syntax makes no mention of record lengths or line lengths or end-of-line information. However, in order to transport CIF files among installations, it is necessary to standardize the data representation.

CIF files are composed of ASCII characters. CIF should be implemented without a line length or record length restriction. However, if this is impossible, CIF processing programs should accept lines at least 132 characters long. Every installation should be able to produce CIF output with lines less than 132 characters. Since ends of lines are treated as blanks in the CIF syntax, a program to break CIF lines is easy to write.

Transformations (see also [2])

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 the symbol is transformed to a point (x',y') in the chip coordinate system by a 3x3 transformation matrix T:

$$[x'\ y'\ 1] = [x\ y\ 1]\ T$$

[It is a good idea to check either the last column of T, or the 1 at the end of the transformed vector, even though they never need to be computed.]

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:

T a b. $T_n = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{matrix}$

M X. $I_n = \begin{matrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$

M Y. $T_n = \begin{matrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{matrix}$

R a b. $T_n = \begin{matrix} a/c & b/c & 0 \\ -b/c & a/c & 0 \\ 0 & 0 & 1 \end{matrix}$ where $c = \mathrm{Sqrt}(a^2 + b^2)$

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'). [Note that 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 file. 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 be. 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. Set layer name to ZZZZ.
3. Collect the individual primitive transformations specified in the call into the matrices T1, T2,T3,etc.
4. Replace the current transformation T with T1 T2 T3 ... T: i.e., premultiply the existing transformation by the new primitive transformations, in order).
5. Now process the symbol, using the new T matrix.
6. 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.

Decomposing Wires Into Boxes

The following algorithm for decomposing wires into boxes was developed by Carver Mead, and first implemented at Caltech by Ron Ayres; it 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.

```
"Initialization:"
If n = 0 THEN DONE; "no path"
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; "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; "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;
```

References

[1] N. Wirth, "What Can We Do about the Unnecessary Diversity of Notations for Syntactic Definitions?", Communications of the ACM, Nov. 1977.

[2] W.M. Newman, R.F. Sproull, Principles of Interactive Computer Graphics, McGraw-Hill, 1973.

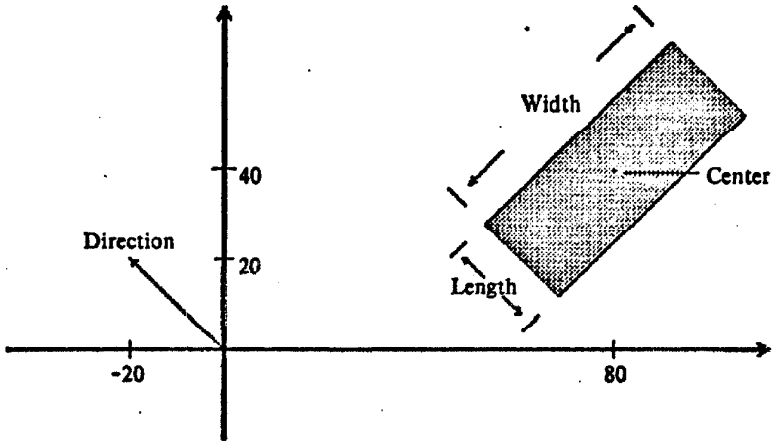

Fig. 1. Box Representation in CIF

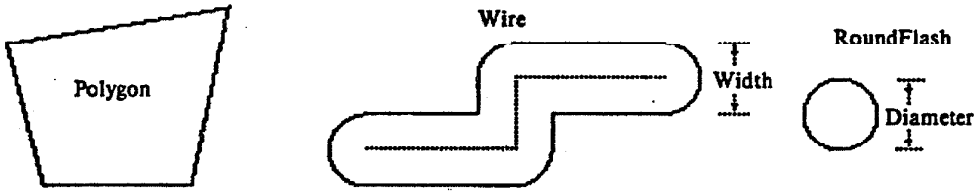

Fig. 2. Other Items in CIF

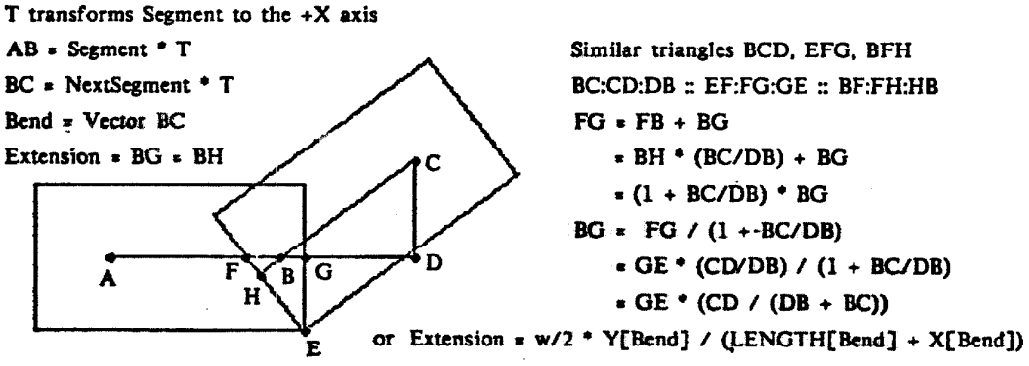

Fig. 3. Converting Wires to Boxes