(12)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: 86-07-01
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): Coordinate Free LAP
5. TYPE OF REPORT & PERIOD COVERED: Technical
6. PERFORMING ORG. REPORT NUMBER:

7. AUTHOR(s): William Beckett
8. CONTRACT OR GRANT NUMBER(s): MDA903-85-K-0072
ARPA-4563, #2
Code 5D30

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
UW/NW VLSI Consortium, Dept. of Computer Science
University of Washington, FR-35
Seattle, WA 98195

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:

11. CONTROLLING OFFICE NAME AND ADDRESS:
DARPA - IPTO
1400 Wilson Boulevard
Arlington, Virginia 22209

12. REPORT DATE: July 1986
13. NUMBER OF PAGES: 14

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):
ONR
University of Washington
315 University District Building
1107 NE 45th St., JD-16, Seattle, WA 98195

15. SECURITY CLASS. (of this report): unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Coordinate Free LAP, CFL , Design Generators, VLSI, Caesar, Magic, Symbol, C, PLAP

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Coordinate Free LAP (CFL) is a library of subroutines written in C intended to facilitate the construction of VLSI circuit layouts. The Operators of CFL generate new cells by forming combinations of existing cells using only relative positioning, that is without reference to a system of coordinates. CFL is able to assemble sets of large cells very quickly because its positioning and routing operations work from descriptions of the boundaries of cells and, therefore, avoid direct references to the geometry within cells.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

Coordinate Free LAP

William Beckett

University of Washington
Seattle, WA 98195

Technical Report 86-07-01
July 1986

Coordinate Free LAP

Abstract

Coordinate Free LAP (CFL) is a library of subroutines written in C intended to facilitate the construction of VLSI circuit layouts. The operators of CFL generate new cells by forming combinations of existing cells using only relative positioning, that is without reference to a system of coordinates. The external data representation used by CFL may be made compatible with either of the UCB graphics editors, Caesar and Magic, so these editors may be used in conjunction with CFL. CFL is able to assemble sets of large cells very quickly because its positioning and routing operations work from descriptions of the boundaries of cells and, therefore, avoid direct references to the geometry within cells.

This paper bears on topics 3 (IC Layout) and 4 (Silicon Compilation).

Coordinate Free LAP

Introduction

Coordinate Free LAP (CFL) is a library of subroutines written in C intended to facilitate the construction of VLSI circuit layouts. The system is organized algebraically in that there is a data type called SYMBOL, a set of operands of this type, and a set of operators which generate new SYMBOLs by forming combinations of existing SYMBOLs.

The system has only two geometric primitives, **box** and **label**, which may combined to make objects. There is a larger set of non-primitive objects called macros, which may be used to generate frequently used structures such as contacts. Routing facilities are provided which generate a variety of planar and non-planar wiring patterns used to connect functional blocks. Additionally, there is a coordinate dependent facility called **wire** for generating arbitrary configurations of material.

Although CFL has sufficient functionality to allow definitions to be developed for all artwork including the lower level cells in a design, it is intended to be used more in the mode of chip assembly. Hence the typical application involves using a graphics editor to generate lower level cells or tiles and then using CFL facilities to assemble these leaf cells into higher level modules. Currently, the system may be used in conjunction with either of the UCB graphics editors, Caesar[3] and Magic[4].

To insure that a wide variety of assembly situations can be accommodated, CFL includes approximately 70 variants of operators for juxtaposing, transforming, and replicating hierarchies of symbols. The positioning performed by most of these operators is with respect to several abstract locations associated with objects, for example, 'the top', rather than to a set of coordinates.

The syntax of these operators is quite compact since generated symbols are simply stored in program variables of type SYMBOL *. The embedding of the language in C is such that sequences of CFL operators admit to both procedural and declarative interpretations. The resulting coordinate free form for defining the structure of complex objects is grammatical in character and fairly easy to manipulate.

All of the calculations which support the operators of CFL are performed from descriptions of the borders of the symbols. The information in the border descriptions includes the bounding box and lists of rectangles representing the intersection that each kind of material in the symbol makes with the bounding box. If there is a label near this intersection, the border description will also contain the label. If a border description is available for a particular symbol, CFL will not require access to any of the rest of the geometry of symbol.

The system will automatically generate border descriptions from the geometry whenever the need arises but it will also automatically save them on disk when library symbols are written out. In this way, modules which have a large number of rectangles may be

accessed from the library without the need of reading all of the geometry files associated with their sub-modules. This capability allows CFL to assemble large blocks of circuitry extremely quickly.

CFL provides automatic hierarchy compression when symbols are written to disk so that only those symbols which represent meaningful functional groups need be saved.

Entities

The operations provided by CFL are defined with respect to a number of basic entities. These entities include primitive geometric objects and compound objects, called SYMBOLS; the boundaries of these symbols, called BORDERS; and individual symbolic points along these boundaries.

CFL has the following two primitive objects -

box(*layer,dx,dy*)	- box
label(*name,dx,dy,pos*)	- rectangular label

These are the same two primitives used by Caesar and Magic. (In the case of Magic, the label primitive also specifies a layer.) All coordinates are dimensionless. **box** creates a box on the specified layer with dimensions *dx* and *dy*. **label** creates a label. Labels consist of a rectangle with dimensions *dx* and *dy* and a *name*. *pos* is used to specify the position of the name of the label relative to its center when it is displayed by the graphics editor.

A CFL symbol is either a primitive object or an object formed by combining primitive objects and other symbols using the CFL operators. Each symbol is a collection of geometry (boxes), calls to other symbols (calls) and labels. CFL represents symbols internally as data structures having lists of boxes, calls, and labels and all references to symbols within a CFL application program are made through pointers to these structures. The pointers are declared with the declarator SYMBOL *.

For example,

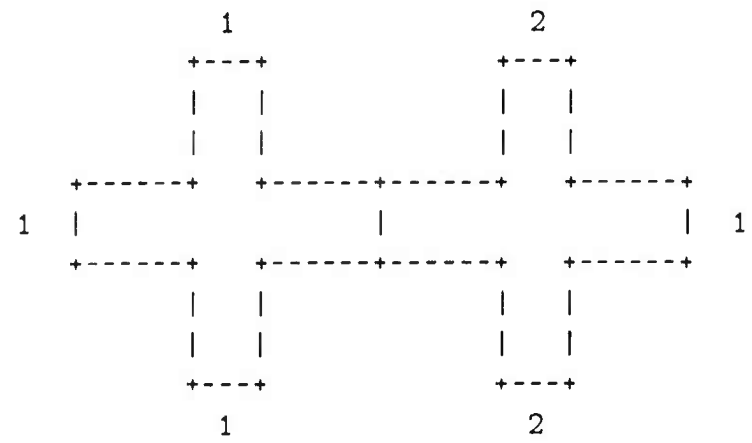
```
SYMBOL *box1,*box2,*cross1,*pair1;
box1 = box("metal", 3,10);      /* vertical bar   */
box2 = box("metal",10, 3);      /* horizontal bar */
cross1 = cc(box1,box2);         /* metal cross    */
pair1  = cx(cross1,cross1);     /* two adjacent crosses */
```

In this example, **cc** is the center to center alignment operator of CFL. It creates a new object by juxtaposing the center of the vertical bar and the center of the horizontal bar. The operator **cx** constructs a horizontal pair of crosses, aligned by their horizontal center lines, with the right edge of the first cross abutting the left edge of the second cross. (All

CFL operators are declared SYMBOL * by the include file *cfl.h* which must be included in CFL application programs.)

For each symbol, CFL maintains a list of coordinates which mark the centers of all intersections of mask layers and the bounding box. These sets of coordinates, called crossings, are maintained separately for each mask layer and for each of the four sides of the bounding box. Each crossing may be referred to by specifying its symbol, side of the bounding box, layer and ordinal along the side.

For example, the symbol, *pair1*, generated above looks some thing like this -



The crossings are given by the following four-tuples -

```
(pair1, "top",   "metal", 1)
(pair1, "top",   "metal", 2)
(pair1, "bot",   "metal", 1)
(pair1, "bot",   "metal", 2)
(pair1, "left",  "metal", 1)
(pair1, "right", "metal", 1)
```

The string literals "top", "bot", "left", and "right" are used by CFL to indicate the sides of bounding boxes. Layer names like "metal" are, of course, technology dependent. For each technology, CFL uses the long format Caesar or Magic layer names. All crossing ordinals start at 1 and increase along the coordinate corresponding to the bounding box edge in question.

Several of the routing operators in CFL have symbolic points as arguments. These arguments are declared to be of type PT * and are generated by the symbolic point descriptor constructor, **pt**. For example, to construct symbolic points which refer to the leftmost and rightmost metal crossings in *pair1* above, the following program statements are used -

```
PT *p1,*p2;
p1 = pt(pair1, "left",  "metal", 1);
p2 = pt(pair1, "right", "metal", 1);
```

Whereas symbolic points are used to refer to specific crossings, CFL borders are used to refer to sets of crossings. CFL borders are used as arguments to some of the routers. A border is similar to a symbolic point in that it is referenced through a descriptor, declared BORDER *, and constructed using a constructor, in this case, **bd**. In the simplest case, a border contains all the crossings associated with a given symbol, side and layer. Hence,

```
BORDER *b1,*b2;
b1 = bd(pair1, "top", "metal");
b2 = bd(pair1, "bot", "metal");
```

constructs two borders; *b1*, containing all the metal crossings on the top of *pair1*, and *b2*, containing all the metal crossings on the bottom of *pair1*.

In addition to the basic border constructor which, by default, includes all crossings in its resulting border description, CFL provides operators **bdin** and **bdex** for including and excluding specific crossing ordinals from border descriptions. In general then, the border description facilities are capable of directing the routers to consider any subset of crossings along the side of a particular symbol. For example, the following statements construct a description of the top of pair1 which includes only the second crossing:

```
b1 = bd(pair1, "top", "metal");
b1 = bdex(b1,1);
```

In the special instance that the ordinal argument is zero, **bdin** will include all crossings in its resulting border and **bdex** will exclude all crossings from its resulting border.

Operators

CFL has six classes of operators -

1. Alignment operators
2. Linear transformations
3. Array constructors
4. Tiling operators
5. Library access operators
6. Miscellaneous operators

The alignment operators combine a pair of symbols by placing them in one of several relationships with respect to each other. The coordinate free nature of CFL stems largely from the fact that the alignment operators typically specify the position of one symbol relative to another rather than the position of either of them relative to a more global set

of coordinates. CFL has six categories of alignment implemented as the following thirteen alignment operators -

1. Center to center	**cc**
2. Center line to center line	**cx**,**cy**
3. Edge to edge	**ll**,**rr**, **tt**,**bb**
4. Border to border	**bx**,**by**
5. Point to point or center	**pax**,**pay**, **cp**
6. Origin to origin	**oo**

Each of these operators has two arguments *s1* and *s2* which are symbol pointers, declared SYMBOL *. The operators form a new symbol containing *s1* and *s2* positioned according to the indicated alignment criterion. The position of *s2* relative to *s1* in this new symbol is called the (0.0) position. All of the alignment operators have three additional variations which allow the specification of offsets from this (0,0) position in the *x*, *y*, or both directions. The variations are formed by suffixing the operator name with **dx**, **dy**, or **dxy**. For example, the **cx** operator has the following four forms:

cx(*s1*,*s2*)	- pair in x, center lines aligned
cxdx(*s1*,*s2*,*dx*)	- pair in x, center lines aligned, x offset
cxdy(*s1*,*s2*,*dy*)	- pair in x, center lines aligned, y offset
cxdxy(*s1*,*s2*,*dx*,*dy*)	- pair in x, center lines aligned, xy offset

The center to center, center line to center line, and edge to edge alignment operators depend only on the bounding boxes of the symbols being aligned. The border and point alignment operators .however, depend on the border crossings. For example, the **bx** operator forms the horizontal pair of symbols (*s1*,*s2*) such that the right side of the bounding box of *s1* is adjacent to the left side of the bounding box of *s2* and the symbols are aligned so that corresponding patterns of material along the common edge match up.

The point alignment operators are similar to the border alignment operators except that the symbols are aligned so that specfic symbolic points along the respective borders are adjacent.

The origin to origin operator is used in conjunction with CFL's routers and will be discussed later.

There are three linear transformations -

mx(*s*)	- mirror in x
my(*s*)	- mirror in y
rot(*s*,*n*)	- rotate

The argument to **rot** is in degrees and must be an integer multiple of 90.

There are three array constructors, **nx**, **ny**, and **nxy**, which can be used to generate horizontal, vertical, or rectangular arrays of a given symbol. As in the case of the alignment operators, offset variants of these operators are also defined. The interpretation of the offsets is, however, slightly different. *dx* and *dy*, when supplied, are taken to be the spacings between the bounding boxes of successive array elements. The (0,0) position is when the bounding boxes are adjacent. The variants of the array operators which would produce a non-rectangular structure are not defined, for example, **nxdy**.

nx(*s*,*n*)	- repeat in x
nxy(*s*,*nx*,*ny*)	- repeat in x and y
ny(*s*,*n*)	- repeat in y

There are three additional array constructors **repx**, **repy** and **repxy** which construct arrays of particular spatial periods. The arguments to these routines are given as *dx* and *dy* but they specify the periods rather than offsets. These operators do not have variants for providing additional offsets.

repx(*s*,*n*,*dx*)	- repeat in x with period dx
repxy(*s*,*nx*,*ny*,*dx*,*dy*)	- repeat in x and y, with periods dx dy
repy(*s*,*n*,*dy*)	- repeat in y with period dy

Tiling is similar to an array operation except that each element of the generated array can be a different symbol. There are three tiling operators, **vx**, **vy**, and **vxy**, which can be used to generate horizontal, vertical, or rectangular tilings. These operators are similar to the array operators except that the first argument is an array of symbol pointers rather than a single symbol pointer. The tiling operators, then, operate on vectors of symbols so their mnemonic starts with **v**. There are no offset variants for the tiling operators since the offset for each tile could potentially be different.

vx(*s*,*n*)	- vector in x
vxy(*s*,*nx*,*ny*)	- vector in x and y
vy(*s*,*n*)	- vector in y

There are two operators for accessing library symbols -

gs(*cell*)	- get library symbol
ps(*y*,*s*)	- put symbol in the symbol table

gs will read a library symbol in either Caesar or Magic format, place the symbol in the data base and return a pointer to it. If the symbol is already in the data base, gs simply returns the pointer, that is, it will read the symbol only once.

ps compresses the hierarchy below its argument and marks that argument as a library symbol. The hierarchy compressor removes from the hierarchy all cells which are not

marked as library cells, that is, cells which were not read in with **gs** or cells which have not been marked as permanent by a call to **ps**. Therefore, **ps** can be used to not only to save symbols but also to control the actual structure of the hierarchy.

CFL is designed to be able to be used with any desired technology. It obtains its table of layer names from technology files in the CFL path. A call to the routine **cflstart** initializes the package and specifies the name of the technology file to be used. **cflstart** must be called before invoking any other CFL functions. **cflstop** causes all permanent symbols to be written to disk and should be called just prior to exiting a CFL application.

Routers

CFL does not currently provide high level routing facilities such as a general channel router or switchbox router. Rather, the CFL routers consist of a set of wiring pattern generators each of which is specialized to a particular kind of routing situation. These routers, which are designed to be used in conjunction with each other and the other CFL operators, support a set of elementary routing operations from which more sophisticated patterns may be constructed.

There are two types of routing facilities available in CFL, planar routers and non-planar routers. The planar routers are -

pp(*s0,p1,p2,w*)	- point to point router
pr(*s0,b1,b2,w*)	- general planar router
ext(*b,d,w*)	- border extender
fill(*s,side,d*)	- Caesar fill operation

and the non-planar routers are -

plx(*s0,p1,p2,w,ct*)	- horizontal point to line router
ply(*s0,p1,p2,w,ct*)	- vertical point to line router
elb(*s0,b1,b2,w,ct,rev*)	- general elbow
tee(*s0,b1,b2,w,ct,rev*)	- tee

Since CFL is coordinate free, the routers operate from border descriptions and from symbolic point designations. The generation of symbolic point and border descriptors is described in the earlier section, Entities.

Most CFL operators produce a new symbol by combining existing symbols. The arguments to these operators have no particular spatial relationship to each other before the operation takes place. The routers, on the other hand, rather than combining symbols, must form connections between them. This process requires that the symbols to be connected have a previously established fixed spatial relationship.

Symbols acquire a fixed spatial relationship as soon as they become constituents of some higher level symbol. CFL refers to a higher level symbol, *s0*, which contains symbols *s1* and *s2* as a container of *s1* and *s2*. Within any container, the relative positions of *s1* and *s2* are fixed.

The routers, like all other CFL operators, are SYMBOL * valued functions. When a router is invoked it produces a pattern of wiring as its result. This pattern of wiring is not 'written' into place directly by the routing operation, rather it is a separate symbol in its own right. Therefore to connect two symbols using the routers, two steps are necessary:

1. Use one of the routers to generate the pattern of wiring necessary to form the required connections.
2. Use the origin to origin alignment operator to locate the generated wiring pattern in the container so that the intended connections are made.

In all cases, the wiring is generated in the coordinate system of the container and often the two steps above may be combined using a statement of the following form -

$$result = \mathbf{oo}(container, \mathbf{router}(container, ...));$$

The rationale for requiring that the generation and placement of wiring patterns be performed in steps rather than as an atomic operation is that in many cases routing problems require the generation of complex patterns in which wiring generated by one call to a router must itself be connected to the wiring generated by another call to a router. Separating the generation allows the generated wiring to become a separate symbol which may be then operated on using any CFL operator.

The point to point router generates a single wire for connecting two symbolic points (see Entities). The layers of the points should match and both points must be uniquely locatable within the containing symbol.

The planar router generates a planar wiring pattern for connecting the points in two borders (see Entities). The borders must contain the same number of points. Also they must be uniquely locatable within the containing symbol. All wires will have the same width.

To simplify the diagnostic process, **pr** will construct wiring patterns whether or not there is sufficient space for the number of wires requested. It will, however, issue a warning message if any of the generated wires are closer than a specified tolerance.

ext generates a pattern of wiring for extending all points in a given border perpendicularly for a specified distance. **fill** is similar to **ext** except that all layers crossing the indicated side are extended. The extensions have the same widths as the crossings. For example, suppose it is desired to generate a symbol *s2* which consists of five instances of a symbol *s1* placed a distance 10 apart and connected by extending the material of the right side of *s1*. The following CFL statement generates this configuration -

```
s2 = cx(nx(oo(s1,fill(s1,"right",10)),4),s1);
```

Generally speaking the planar routing facilities of CFL are technology independent whereas the non-planar routing facilities are technnology dependent since contacts must be specified.

plx generates a single wire for connecting a symbolic point the vertical line running through another symbolic point. The connection is made horizontally. The layer of the first point taken to be the layer of the wire. The points must be uniquely locatable in the container symbol. If requested, a contact is placed with its origin at the intersection of the vertical line and the generated wire.

ply is similar to **plx** but generates a single wire for connecting a symbolic point to the horizontal line running through another symbolic point. The connection is made vertically.

elb generates a wiring pattern for connecting the points in two borders, say, *b1* and *b2*. Wires from *b1* and *b2* may have different widths. The pattern generated must form an elbow but it is not necessary that *b1* and *b2* be on the same layer. If requested, a contact will be placed with its origin at the intersections of the wires from *b1* and the wires from *b2*.

elb may generate either forward or reversed elbows. For a forward elbow the low order points in *b1* will connect to low order points in *b2*. For a reversed elbow, low order points in *b1* will connect to high order points in *b2*.

Through combinations of selecting subsets of the borders with **bdin** and **bdex** and utilizing the normal and reverse options, a succession of **elb** invocations may be used to form a set of elbows between *b1* and *b2* which implement any desired ordering of the connections.

tee generates a wiring pattern for connecting the border of a tee connected symbol to the wiring of a transverse routing symbol. The wiring in the routing symbol is assumed to run perpendicular to the wiring generated for connecting the tee connected symbol. The routing symbol, presumably generated by a prior call to a router, is also assumed to consist strictly of parallel lines, no elbows. All generated wires will have the same width. If requested, a contact will be placed with its origin at the intersection of the generated wires and the wires existing in the routing symbol. The connection order for tee may be either forward or reversed.

All of the non-planar routers have a contact argument *ct*. The provision for positioning this contact in generated routing is coordinate dependent in that the contacts are always positioned so that their origins, coordinate (0,0), coincides with the intersections of wires on different layers. If the contacts are symmetric and generated with the CFL box primitive, as is the case with the NMOS macros **gb** and **rb**, the origins will be in the geometric centers because the box primitive is designed to make boxes which are symmetric about the origin whenever possible. If other, asymmetric, forms of contacts are needed they may be generated according to the above criterion using the CFL wire facility described later.

Use of the routers generally requires that three pointers into a symbol hierarchy be supplied - the container and the two symbols to be connected. When symbols are retrieved

from the library using **gs** only one pointer is provided. A typical problem of this form is to retrieve from the library both a complete circuit and a pad frame and then to connect the circuit to the pads. The CFL procedure **locate** may be used to obtain a pointer to any named sub-symbol within a symbol hierarchy. All symbols saved with **ps** are named symbols.

For example, suppose a circuit called *memory* is to be placed in a pad frame and connected. Suppose further that the section of the pad frame containing the output pads is named *outputs* and that the memory outputs are available on the boundary of a sub-symbol called *planes*. The following CFL code accomplishes the task:

```
SYMBOL *memory,*pads,
       *outputs,*planes,*chip;

/* get the memory and the pad frame from the library          */

     memory = gs("memory");
     pads   = gs("pads");

/* establish pointers to the planes sub-symbol of the memory */
/* and the outputs sub-symbol of the pad frame                */

     outputs = locate(pads,"outputs");
     planes  = locate(memory,"planes");

/* position the memory within the padframe                    */

     chip = ccdx(memory,pads,120);

/* connect the outputs from the memory planes to the          */
/* corresponding output pads                                  */

     chip = oo(chip,pr(chip,bd(planes, "top","metal"),
                            bd(outputs,"bot","metal"),3));
```

Macros

CFL has two groups of macros - technology independent macros and technology dependent macros. The technology independent macros are -

alpha(*s,layer,w*)	- character string, width w
cross(*layer1,dx1,dy1,layer2,dx2,dy2*)	- two boxes, centers aligned
letter(*c,layer,w*)	- alphanumeric letter, width w
lne(*layer,w,dx,dy*)	- el, north east
lnw(*layer,w,dx,dy*)	- el, north west
lse(*layer,w,dx,dy*)	- el, south east
lsw(*layer,w,dx,dy*)	- el, south west

alpha generates a string of characters which are 5w wide, 8w high with 2w spacing in between. The same rules apply to **letter**. The character set that is available is

A - Z
0 - 9
- . , : ; ? ! / [ ] + =

Currently, space (or blank) is not available.

The technology dependent macros available generate commonly used structures like contacts, pullups and and components of standard pad frames.

Wire Facility

In order to provide for the parametric generation of particularly complex leaf cells, or cells with specific coordinate requirements like router contacts, CFL includes the wire facility which allows the use of symbol relative coordinates. Note that the use of this facility can introduce significant coordinate dependency into a design so it should not in general be used in cases where the coordinate independent operators are able to serve. The procedures associated with the wire facility are the following -

wire(*layer,width*)	- Initialize a wire
at(*x0,y0*)	- Move to the point (x0,y0)
dx(*dx0*)	- Draw to the point (x+dx0,y)
dy(*dy0*)	- Draw to the point (x,y+dy0)
iso(*s*)	- Include symbol origin
wl(*layer*)	- Reset the wire layer
ww(*width*)	- Reset the wire width
x(*x0*)	- Draw to the point (x0,y)
y(*y0*)	- Draw to the point (x,y0)

wire is of type SYMBOL *. All of the procedures apply to the wire generated by the last call to **wire**. Note that the symbol generated by **wire** may contain an arbitrary number of physical 'wires' which need not be connected. The only thing they have in common is their coordinate system.

The procedure iso has a symbol as its argument. iso includes that symbol positioned so that its origin coincides with the current wire position. Note that the current wire position, or more precisely, the position within the coordinate system of the current wire, is initialized with the at procedure and maintained by all move and draw procedures.

Experience Using CFL

The UW/NW VLSI Consortium has been using CFL for the last year to make a set of module generators. These generators are designed to produce instances of general structures which meet various specifications. For example, a CMOS multiplier generator has been developed which produces two's complement multipliers for either signed or unsigned operands of varying sizes. Flexible generators have also been developed for a PLA, a CAM, several kinds of ROM's and a multiplexer. In general, the generators include features like automatic adjustment of driver and buss sizes as a function of the modules' speed and power requirements.

In addition to the module generators, CFL has been used for assembling and routing the components of the Quarter Horse microprocessor that the Consortium has developed.

So far indications are that CFL is easily learned by those familiar with C. The number and sohpistication of the projects that have been completed using CFL indicate that the system is substantially more convenient than coordinate based systems while still retaining a sufficient degree of flexibility.

Due to the border abstraction, the system has an excellent speed advantage over many other procedural and graphical approaches for assembly of larger modules. For example, the sample program shown earlier, which places a ROM in a pad frame, executes in less than two seconds on the VAX 11/780. The ROM has approximately 5000 transistors. The ROM generator produces an eight by eight instance in about 16 seconds. The main limitations are that the SYMBOL data structure consumes about 2KB of memory per symbol and that the routers are not always straightforward to apply due to their somewhat specialized formulation.

Acknowledgements

CFL has taken about two years to develop. I would like to thank the management of the Consortium, in particular, Larry McMurchie, for his encouragement and patience during the critical initial phase of the development. Also, the system would not have achieved its current level of capability without the efforts of several staff members and students who have spent considerable time exploring the package, developing new techniques, new features, and discovering and helping to correct a number of bugs. Most particularly I would like to thank Dave Morgan and Wayne Winder of the Consortium staff, Barry Jinks, Consortium liaison from Microtel Pacific Research, and Jim Schaad, Amilesh Tyagi, and Chyan Yang of the Department of Computer Science here at the University of Washington.

References

1. W. Beckett *Coordinate Free LAP Reference Manual*, UW/NW VLSI Consortium *Design Tools Release 3.0*, University of Washington (June 1985)

2. V. Corbin and B. Yanagida *PLAP Reference Manual*, UW/NW VLSI Consortium *Design Tools Release 2.1*, University of Washington (October 1984)

3. J. Ousterhout, *Editing VLSI Circuits with Ceasar*, 1983 VLSI Tools, Report No. UCB/CSD 83/115 (March 1983)

4. J. Ousterhout, R. N. Mayo, and W. S. Scott, *Magic Tutorials*, Berkeley VLSI Tools, Report No. UCB/CSD 85/225 (March 1985)