6 The WOLCOMPII composition system

WOL provides a textual composition system, embedded in the Pascal language. The user writes a program that loads the component geometry cells, then composes them together. The WOLCOMPII composition system is intended to be a more user-friendly interface to an earlier composition system called WOLCOMP (written by Carver Mead; see the next section). In order to use these routines, a PASCAL program must search and link the following modules:

```
import
   $search '/LIB/WOLLIB/WOLCOMP/WOLCOMP'$  wolcomplib,
   $search '/LIB/WOLLIB/WOLCOMP/WOLCOMPII'$  wolcompii ;
```

A brief description of these routines follows:

PROCEDURE wolcomp_init(lamb : integer);

Initialize the WOLCOMP and WOLCOMPII libraries to generate CIF with λ = lamb / 100 microns. This must be the first call to the system, and is generally followed by commands to load the various geometry cells:

PROCEDURE load (filename : strg);

Reads a file containing a composition cell, and inserts all contained geometry cells into an internal dictionary. The global string variable `prefix` is prepended to `filename`; this is convenient while searching some library directory. The `load` command is identical to the WOLCOMP `load_library` procedure. Once all the cells are loaded, they can be composed into new cells:

PROCEDURE define_cell (cell_name : strg);

Begin the definition of the cell "cell_name". The current cell placement origin (x0,y0) is set to the lower left corner (0,0), and transformation, composition, and overlap modes are reset. The definition is terminated with the **endef;** statement.

PROCEDURE place (cell_name : strg);

Within a cell definition, position an instance of cell "cell_name" at position (x0,y0). The user can manually update (x0,y0) by hand as in WOLCOMP (perhaps making use of the "`cell_width`" and "`cell_height`" procedures below), *or* they can be updated automatically according to the current *composition mode*, and "overlap" distance (see below). For example, if we are currently composing horizontally, `x0` will be updated to `x0+cell_width-overlap`, and `y0` will be left undisturbed. If the global variable `include_geometry` is set to `FALSE` before a call to `place`, the coordinate (x0,y0) is updated, but *no geometry is actually drawn*; this is often useful in debugging to reduce the amount of output CIF generated.

PROCEDURE hcompose;

Set the current composition mode to horizontal. Successive calls to `place` will create a horizontal row of cells.

PROCEDURE vcompose;

Set the current composition mode to vertical.

PROCEDURE nocompose;

Set the current composition mode to "do nothing". This means that x0 and y0 are unaffected by a call to `place`. This is the default initial composition mode of WOLCOMPII.

PROCEDURE overlap (lambda : integer);

Sets the overlap between adjacent cells. For example, if `vcompose` is set, and `overlap(10)` is in effect, then every call to `place` will result in `x0` (the position of the *next* cell to be placed) to be 10 lambda to the left of the new cell's right side.

PROCEDURE make_array (cell_name : strg; count : integer);

Equivalent to the loop: `for i := 1 to count do place(cell_name);`

PROCEDURE transform (xfrm : integer);

Sets the current transform to `xfrm`, which can take the value of the predeclared constants:

```
R0 = 0;
R90 = 1;
R18 = 2;  R180 = 2;
R27 = 3;  R270 = 3;
M0  = 4;             MX  = 4;
M90 = 5;
M18 = 6;  M180 = 6;    MY  = 6;
M27 = 7;  M270 = 7;    MXY = 7;
```

This transform *remains in effect* until another transform is specified, or another cell is defined; all calls to `place` will employ the current transform. The initial transform in WOLCOMPII is R0

PROCEDURE vpad (cell1, cell2, cell3 : strg; to_width : integer);

Takes one instance of `cell1` and vertically adds as many instances of `cell2` as are required to give a cell of height `to_width` in lambda. It is legitimate to partially overlap instances of `cell2`. The resulting cell is defined as cell3. This function is useful in generating parameterized cells that must "stretch" to abut against some constraining cell. This function must not be called within a `define_cell` command, as cell definitions cannot be nested.

PROCEDURE hpad (cell1, cell2, cell3 : strg; to_width : integer);

Same as `vpad`, but the composition is horizontal. (identity transform).

Once the project composition is complete, WOLCOMP is terminated with a call to:

PROCEDURE shut_down;

Other functions that are often useful include:

FUNCTION cell_width (cell_name : strg) : integer;

Returns the width of the cell in internal WOLCOMP units, which makes it suitable for addition/subtraction from variables such as (x0,y0).

FUNCTION cell_height (cell_name : strg) : integer;

Similar to "cell_width", but returns the height of the cell.

FUNCTION make_cell_name (root : strg; i : integer) : strg;

Returns a string which is the concatenation of the string `root`, and a string containing the integer `i`.

PROCEDURE push (xi, yi : integer);

Push a pair of integers onto a stack. This is very useful for recording some (x0,y0) position that you want to return to after going off and doing something else.

PROCEDURE pop (var xi, yi : integer);

Pop a pair of integers off the stack, into the variables xi and yi.

PROCEDURE include_file (fname : strg);

Include the contents of the file `fname` directly into the output CIF file. This operation can be safely performed only while no cell definition is in progress.

6.1 Sample program

See the PADGEN portion of this manual.

7 The WOL pad library and pad generator

A set of library pads exist for NEWOL. A complete pad family, containing both analog and digital pads is located in /LIB/WOLLIB/PADLIB. These pads are implemented in double-metal 3u feature-size CMOS, and provide a 150u square bonding area. The pads, and their applications, are:

INPAD	Input pad, 200Ω resistance, and static protection.
INORPAD	Input pad, static protection, no resistor.
METALPAD	Bare metal pad, no static protection.
OUTPAD	Non-inverting digital output pad, 50Ω drive.
FOLLOWPAD	Wide range follower to drive analog signals
VDDPAD	Provide Vdd to chip core and pad ring
GNDPAD	Provide Gnd to chip core and pad ring
CVDDPAD	Provide Vdd to chip core only
PVDDPAD	Provide Vdd to pad ring only
CGNDPAD	Provide Gnd to chip core only
PGNDPAD	Provide Gnd to pad ring only
CORNPAD	Corner of the pad ring
SPCPAD	No pad, just run pad ring and protection

A WOLCOMP module that implements a pad generator is also available: link your program with /LIB/WOLLIB/PADLIB/PADGEN. A sample program is shown below (it is taken from /LIB/WOLLIB/PADLIB/PADSAMPLE.TEXT):

```
program sample (input, output);
import general_2, graphics, misc, hpib_2, newci,

   { N O T E:   you will need to 'perm /lib/wollib/wolcomp'
                          and 'perm /lib/wollib/padlib/padgen'

             BEFORE trying to run this program!!!
    }
   $search '/LIB/WOLLIB/WOLCOMP'$  wolcomplib,
   $search '/LIB/WOLLIB/PADLIB/PADGEN'$  pad_gen ;

var my_pads  :  pad_spec;    { this is the structure used to specify to
                               the pad generator what pads you want, and
                               where they should go}

begin
   set_up;  lambda:=150;

   prefix:= '/lib/wollib/padlib/';
   getfile('cornpad');
   getfile('padwire');
```

```
  getfile('outpad');            { this is a digital output pad }
  getfile('inpad');             { input pad with static resitor --
                                  use this as an analog output pad }
  getfile('inorpad');           { input pad with no resistor }
  getfile('cvddpad');           { VDD pad for chip core - your circuit }
  getfile('pvddpad');           { VDD pad for PAD frame }
  getfile('vddpad');            { VDD pad for core and PAD frame }
  getfile('cgndpad');           { GND pad for chip core - your circuit }
  getfile('pgndpad');           { GND pad for PAD frame }
  getfile('gndpad');            { GND pad for core and PAD frame }
  getfile('followpad');         { Analog follower pad }
  getfile('plnpad');
  getfile('spcpad');            { "space pad" = no pad at all }

  with my_pads do begin
     xnum:= 2;                   {  NUMBER OF PADS ON TOP/BOTTOM }
     ynum:= 6;                   {  NUMBER OF PADS ON LEFT/RIGHT }
     olap:= 10;                  {  magic number -- do not change}
     corner:= cell('cornpad');   {  also magic }

{ define pads along each side:  north and south lists are left to right }
     north[1]:= c('pvddpad');
     north[2]:= c('cgndpad');

     south[1]:= c('cvddpad');
     south[2]:= c('pgndpad');

{ east and west lists are bottom to top }
     east[1]:=  c('plnpad');
     east[2]:=  c('inorpad');
     east[3]:=  c('plnpad');
     east[4]:=  c('outpad');
     east[5]:=  c('plnpad');
     east[6]:=  c('inpad');

     west[1]:=  c('inpad');
     west[2]:=  c('plnpad');
     west[3]:=  c('outpad');
     west[4]:=  c('plnpad');
     west[5]:=  c('inorpad');
     west[6]:=  c('plnpad');
  end;

  gen_pads(my_pads, 'samplepads');   { this actually does the work,
                                       and returns a cell called 'samplepads'}

  {alternate procedure for WOLCOMP users:

  surround_with_pads('mycore', my_pads, 'newcell');

          -- creates cell 'newcell' by wrapping pads around 'mycore'

  OR

  pad_wrap ('mycore', my_pads, 'newcell', 'routing');
```

```
          -- creates cell 'newcell' by wrapping pads around 'mycore',
          PLUS reads the routing wires in file 'routing' and incorporates
          them into cell 'newcell'.  This is most useful when repeatedly
          compiling some system in which the routing remains fixed.
          After creating the routing by hand the first time (in NEWOL),
          edit by hand the CIF file, putting the routing wires into
          file 'routing'.  You can now re-compile the entire project
          without ever having to re-route those wires.

   }


shut_down;
end.
```

8 The WOL PLA tools

A simple PLA generator has been built using WOLCOMP. A textual PLA personality file is written, and a PLA generated in a user-specified technology. Two interfaces are possible: a simple precompiled program that reads the `.pla` file and generates a `.CIF` file containing the PLA, and a Pascal procedure interface for use in the user's own WOLCOMP programs.

The PLAs generated have a simple form. The AND-plane is on the left, with inputs along the bottom. Each literal is presented in uncomplemented form (and is buffered and complemented in the PLA). The OR-plane lies on the right, with its uncomplemented outputs along the bottom. The reader is referred to documentation within the directory `/LIB/WOLLIB/PLAGEN` for specific details regarding the different PLA technologies (i.e. location of power rails, clock requirements, etc.).

8.1 The WOLCOMP interface to PLAGEN

To access this module, your program must contain the following import text:

```
import
   $search '/LIB/WOLLIB/WOLCOMP/WOLCOMP'$  wolcomplib,
   $search '/LIB/WOLLIB/WOLCOMP/WOLCOMPII'$  wolcompii,
   $search '/LIB/WOLLIB/PLAGEN/PLAGEN'$  plagen;
```

The interface to the module is the following (for an example of the use of these procedures, see the WOLPLA program below):

```
EXPORT

const
      MAXPLAROWS = 50;   { maximum number of rows in pla }
      MAXPLACOLS = 50;   { max. number of cols. in and/or planes }

type
      pla_plane =  array [1..MaxPlaRows,1..MaxPlaCols] of char;

var
   num_pla_rows, or_width, and_width  :  integer;
   or_plane, and_plane :  pla_plane;
```

```
PROCEDURE load_placells (tech : strg);
   { load cells from directory /lib/wollib/plagen/<tech>  }

PROCEDURE make_pla (cellname, filename : strg);
   { read <filename> for PLA personality, generate cell <cellname> }

PROCEDURE read_pla_file (filename : strg; var and_plane, or_plane : pla_plane);
   { read PLA personality into 'and_plane' and 'or_plane', set
     global variables 'num_pla_rows', 'or_width', and 'and_width' }

PROCEDURE build_pla (cellname : strg; and_plane, or_plane : pla_plane);
   { use 'and_plane', 'or_plane', and global variables 'num_pla_rows',
     'or_width', and 'and_width' to generate PLA in cell <cellname> }
```

8.2 PLA personality file format

A very simple file format is used to program the PLAs. The first line contains an integer representing the number of rows (implicants) in the PLA. A textual block then follows, one row per implicant, one column per literal, to define the AND plane. A "1" represents an uncomplemented literal, a "0" a complemented literal, and an "X" a dont-care (that literal does not appear in a particular implicant). A blank line separates the OR-plane specification, which is similar, but only "1" and "X" are allowed. Obviously, all rows in the AND-plane specification must contain the same number of characters, as must all rows in the OR-plane specification.

As an example:

```
4
.001
0.11
10..
1.00

1..
.1.
..1
11.
```

8.3 The WOLPLA program

This program is executed by typing `/LIB/WOLLIB/PLA/WOLPLA` (you may need to type `USEWOLCOMP` first). The user is prompted for (i) the technology for the PLA, (ii) the programming file for the PLA, and (iii) the cell name and CIF file name for the generated PLA.

The listing of the program follows:

```
program wolpla(input,output);

import
   $search '/LIB/WOLLIB/WOLCOMP/WOLCOMP'$  wolcomplib,
   $search '/LIB/WOLLIB/WOLCOMP/WOLCOMPII'$  wolcompii,
   $search '/LIB/WOLLIB/PLAGEN/PLAGEN'$  plagen;

var
   tech_name,      { technology name for PLA }
   file_name,      { PLA personality file }
   cell_name       { cell name of generated PLA }
         : string[255];
   pla_lambda : integer;  { lambda in 1/100 micron }

begin
   writeln ('WOLPLA Version 1.00');  writeln;
   write ('Enter technology name:  '); readln (tech_name);
   write ('Enter PLA personality file name:  '); readln (file_name);
   write ('Enter cell name of generated PLA:  '); readln (cell_name);
   write ('Enter lambda in 1/100 of microns:  '); readln (pla_lambda);

   wolcomp_init (pla_lambda);
   load_placells (tech_name);
   make_pla (cell_name, file_name);

end.
```

8.4 Building a PLA cell library

The user can supply their own cell library to PLAGEN for composition. The `load_placells(tech);` procedure searches in the directory `/LIB/WOLLIB/PLAGEN/TECH` for the following files (each file contains a single cell of the same name as the file):

p_and	if a literal appears uncomplemented in an implicant
p_nand	if a literal appears complemented in an implicant
p_null	if a literal does not appear in an implicant
p_andend	left-hand side of AND-plane
p_buffer	input buffers on bottom of AND-plane
p_buss	power jumper between bottom of AND- and OR-planes
p_andcorner	lower-left hand corner cell
p_or	an implicant does appear in an output
p_ornull	an implicant does not appear in an output
p_orcorner	lower-right hand corner cell
p_orend	right-hand side of OR-plane
p_pcinv	cell between AND- and OR-planes
p_outinv	output buffer along bottom of OR-plane

8.5 The WOL Finite State Machine Compiler

A powerful tool for compiling finite state machines into PLAs has been developed, and is available. Talk to Mass Sivilotti for more info.

9 The WOL Decoder Generator

A WOLCOMP module exists for generating decoders. The user must specify the number of decoded outputs (which must be even), the CIF layer they should be on, and the pitch between them (minimum 36λ). The decoder generator is accessed by "perm"ing /LIB/WOLLIB/DECODER/DECODER, and including the following import text in the WOLCOMP program composing the user's chip:

```
import
$search '/LIB/WOLLIB/WOLCOMP/WOLCOMP'$     wolcomplib,
$search '/LIB/WOLLIB/WOLCOMP/WOLCOMPII'$ wolcompII,
$search '/LIB/WOLLIB/DECODER/DECODER'$ decoder;
```

The cell library of decoder cells is loaded by a call to

PROCEDURE load_decodercells;

The actual decoder is generated by

PROCEDURE make_decoder (name : strg; rows, a, b : integer; ciflayer : strg);

where `name` is the name of the cell that is to be the decoder, `a` and `b` encode the pitch of the decoder outputs (i.e. outputs alternate between being `a` apart and `b` apart), and `ciflayer` is the layer of the outputs (valid choices are PG, MF, and MS, for poly, metal1, and metal2).

A multiplexer generator is also available, with similar syntax:

PROCEDURE make_mux (name : strg; rows, a, b : integer; ciflayer : strg);

10 The WOLCOMP composition system

This section is taken almost verbatim from a document prepared by Carver Mead. For more information, please refer to the **WoLery**. Changes and additions are indicated by horizontal bars.

WoLcomp is a very simple composition system embedded in Pascal. It takes cells from only those CIF files prepared on **WoL**, and composes them into larger cells which can be further composed. Being an embedded language, the user builds a chip by writing a program. There are a number of built-in functions and procedures to make all that a lot easier. To compose a design, one must first get cells—each of which resides as an individual file on your file system—into **WoLcomp**. The procedure that brings in a cell is

procedure getfile(name:strng10);

To get a cell from file `XOR.CIF` the call is **getfile**('`XOR`'); Note that the file must have a `.CIF` extension and you must be prefixed to the directory where the file resides. If you need to access a cell in a different directory, set the global string variable **prefix** to the appropriate directory (don't forget to terminate it with a "/").

EDITOR'S NOTE: It is no longer required that all geometry cells be in separate files. The `load_library` *command described below can accomodate multiple cells per file (as created by NEWOL, particularly by the DUMP command). Since it can also accomodate one cell per file,* `load_library` *is a superset of* `getfile`*, and can be used in its place everywhere!*

WoLcomp builds a symbol table and places the cells in it so they can be accessed by name. A pointer to an entry in the symbol table is of type 'cptr'. To get a pointer to a cell use

function cell(name:strng10):cptr;

To instantiate a copy of the cell, there are two commands:

procedure draw(c:cptr);

places an untransformed instance of the cell pointed to by the argument with its lower left corner at the position given by the global coordinates x_0, y_0.

For situations where cells need to be mirrored or rotated

procedure drawx(c:cptr,tform:integer);

places an instance with its lower left corner at x_0, y_0 and transformed by one of the 8 transforms in the dihedral group defined by mirrors in x and y and rotations by integral multiples of 90°.

The transforms are defined as follows:
0–3 unmirrored cell rotated by the number times 90°.
4–7 cell first mirrored in x, then rotated as above.

When a cell is originally drawn, all of the shapes within it are defined with respect to some *origin* local to the cell. Many embedded systems use the CIF convention that positions an instance of the cell by placing the origin at some x, y location. Since the instance may be transformed by rotation or mirror operators, the placement of the origin depends upon the transformation applied. This approach places the burden of keeping track of these coordinate offsets on the user. **Wol_comp** automatically applies the proper offsets such that all cells are placed with their lower left corner at x_0, y_0 rather than with their origin at that point. In this way, the user need never be aware of the cell origin. Cell instances are treated as tiles, and composition is the proper tiling of the plane. Both procedures **draw** and **drawx** leave coordinates of the the upper right corner of the instance in global variables x_1, y_1.

The main program must start with a call to **start_up** and end with a call to **shut_down**. You may of course, add procedures, variables, etc., above the main program to make your life easier. Since you are thus mixing your own stuff with the **Wol_comp** primitives, your file will be unique to a chip design. I suggest that you name it after the chip.

A large design is always built up by defining a cell as a composition of already defined cells. To define a new cell use

procedure define(name:strng10);

Then put the required **draw** or **drawx** commands for the subcells and end with

procedure endef;

which buttons the whole thing up and adds it to the symbol table.
Note—definitions cannot be nested !

The methodology I have used to build complex chips size is to define all the "real" cells (those with transistors in them) in the context of the others to which they must abut. The odd corners are then filled up with *wiring cells*, to connect the whole thing together. I keep a floorplan with cell names as I go along. The cell names are also the file names. To date, **WoL** is not smart enough to do this floorplanning job for us, but it does give a nicely annotated floorplan plot as it is run. At the end, I write a little program in **WoLcomp** to put the whole thing together. A small example follows to illustrate the technique. Note that there is not a single absolute coordinate in the entire text. All of the placement coordinates are done by the draw commands interacting with global variables. In a pinch, there are times when you will really want to stick a long wire down, and don't want to make an enormous wiring cell in **WoL** just for that wire. For those occasions there is

procedure box(c:strg);

The string argument is the CIF layer name of the box. This procedure creates a box of the appropriate layer with one corner at x_0, y_0 and the other at x_1, y_1. You can make a whole pile of boxes (even a whole design) by using this primitive and the composition functions described above. **WoLcomp** is thus a complete embedded layout language. *However, it was never intended to be used that way*—**WoL** is really a much easier and more idiot-proof way to design cells.

When your design is complete, compile the program and run it! You will be prompted for the name of an output file. Some generally helpful information is displayed on the screen as things progress. If you have forgotten to define a cell, or to get it off the disk, **WoLcomp** will tell you that the cell is not loaded. Once all is well, you will be asked if you want a floorplan plot. *Say yes!* You will find it much easier to debug your composition this way before you start staring at all that layout. **WoLcomp** can plot any cell you have defined, so be sure the entire project is defined as a cell. Floorplans are only plotted to one level—if you want to see the subcells of one of the subcells, you should plot it as a separate cell *right?—O.K.—I'm sorry I asked!* The name of the cell whose floorplan is plotted is shown in the upper right corner of the plot. The names of the subcells are centered in their bounding boxes. Of course, the output CIF generated by **WoLcomp** can be read into NEWOL, and viewed or modified there.

By now, programs in The **WoLery** have been used to do substantial designs in all three technologies. There are many more sophisticated members of the **PooH** family nearly ready to use. Meanwhile, the simple programs described in this document are a good example of which portions of a design are naturally done in an interactive way, and which are more appropriate to do as a language. It sounds silly, but I know of no other system that has got that issue right. I hope the listeners in our far-flung radio audience are enjoying the latest episode in the life of my favorite stuffed animals. *They are so much more interesting than stuffed people!*

A Small Example

The following example—a 6 bit pipeline multiplier—illustrates many of the composition principles used in this style of design. The variables `cell1`, `cell2`, etc., are pointers to cells; `x0`, `y0`, `x1`, `y1`, etc., are integers. All of these variables were declared as globals in the program.

WoLcomp programs all follow the pattern shown. First, all of the needed cells are brought in. Next, cells are defined in terms of smaller cells. Finally, the entire design is defined as a cell, and brought into being with the one naked *draw* command at the end. This arrangement produces CIF conforming to the "one call convention" described in certain obscure books on the subject.

The cells `rend`, `lend`, `lep1`, `lep2`, `rep1` and `rep2` are wiring cells, and contain no active elements. The `lco` and `rco` cells are the VDD and GND pads. The multiplier itself is made out of `plbit` cells, which in turn are made out of subcells. All of the `pd` and `pad` cells are pads with drivers or static protection. The **box** procedure is used to create two horizontal metal wires that tie the VDD and GND busses in the pad ring between the two center `inpad` cells. The number `3` is the wire coordinate relative to the cell bounding box. The `16` is the width of the wire. The `4` is the spacing between the two wires. The original design was 32 bits long, and was obtained by changing the 5 in the FOR loop to a 31. The floorplan generated for this example by **WoLcomp** is shown in the figure.

10.1 WOLCOMP Procedure Calls

In order to link in the **WoLcomp** module, the following file should be searched, linked, and perm'ed:

```
import
   $search '/LIB/WOLLIB/WOLCOMP/WOLCOMP'$  wolcomplib;
```

The following is a brief synopsis of the calling sequence for all the routines in the current **WoLcomp** (version #1.00).

procedure getfile(filename:strg);

Load a single cell out of a .CIF file created with either WOL or NEWOL. The file must contain one geometry cell. "filename" refers to the name of the file that contains the cell. If "prefix" has been specified, that string will be pre-appended to "filename", thus providing a facility to specify a search directory. The cell will be added to the internal symbol table. If the CIF file contains a cell name (i.e. user extension 9), this name will be associated with the cell; otherwise, a cell name will be generated from the supplied filename.

procedure getcompfile(filename:strg);

Load a file containing many cells. These cells will be renumbered to guarantee uniqueness and consistency within each top-level cell, and the top-level cell will be added to the internal symbol table. This is intended for use as a MPC composition facility.

procedure draw(c:cptr);

Draw the cell "c" at the position currently specified by x0, y0. The origin of the cell (usually the lower left corner) will be positioned at x0, y0. Upon return, the global variables x1, y1 will point to the upper right corner of the cell that was just placed.

procedure ndraw(c:cptr; drawit:boolean);

If "drawit" is TRUE, then this is the same as a call to "draw"; otherwise, the pointers x0,y0,x1,y1 will be updated, but no geometry will be drawn. This is very useful for testing compositions of large, repetitive structures.

procedure drawx(c:cptr; tform:integer);

Draw the cell "c" at (x0,y0), but apply a transformation "tform" to it. Possible transformations are:

0 - no transformation;
1 - rotate CCW by 90;
2 - rotate CCW by 180;
3 - rotate CCW by 270;
4 - mirror in the y-axis;
5 - mirror in the y-axis, then rotate CCW by 90;
6 - mirror in the y-axis, then rotate CCW by 180;
7 - mirror in the y-axis, then rotate CCW by 270;

procedure ndrawx(c:cptr; tform:integer; drawit:boolean);

Same as "ndraw", but draws a transformed cell.

procedure plot(cell:cptr);

Plot out a cell, either to screen or to pen plotter.

function cell(name:strg):cptr;

Lookup cell named "name" in symbol table, and return a pointer which can then be passed to "draw", "drawx", etc.

procedure define(name:strg);

Start the definition of a composition cell.

procedure endef;

Close the definition of a composition cell.

procedure box(c:strg);

Draw a box on layer "c" from (x0,y0) to (x1,y1).

procedure set_up;

Should be called first to set up all **Wol_comp** internals. This sets a default "prefix" (null), and "lambda" (1.5 microns). If a different lambda is desired, it should be changed (by assignment to the exported variable **lambda**) after **set_up** is called. Remember: **lambda** is in hundreths of microns. Similarly, **prefix** should be set after **set_up** is called.

procedure shut_down;

Closes the output file, and optionally plots a floorplan for a generated cell. The floorplan can be drawn on either the screen or a pen plotter. Also, the internal symbol table can be displayed (useful for seeing how large cells are).

procedure dumplist;

Prints out the internal symbol table. This is primarily a debugging aid for **Wol_comp** maintainers.

procedure set_mbb(llx,lly,urx,ury:integer);

Forces the MBB of a composition to be set to the specified values. This is primarily useful for handling non-WOL generated CIF, in conjunction with the **cif_type** attribute.

procedure load_library(filename:strg);

Reads a file containing a composition cell, and inserts all contained geometry cells into a "library table". If the global variable **search_library** is **true**, any subsequent calls to cells with names that match that of a cell in the library will cause a call to the library cell to be generated, instead of the supplied cell. This is most useful for commonly maintained cells such as pad libraries, etc.

function lam(x:integer):integer;

Returns the value of **x** lambda in **WoLcomp** internal units. This is the recommended way to incorporate fixed (i.e. non-relative) offsets. (e.g. x0 := x1 + lam(4); would cause the two cells to be separated by 4 lambda).

function fix_name(fname:strg):strg;

Fix up file names. You should never have to use this.

Routing wire primitives

procedure width (x : integer);
procedure wire (layer : strg; startx, starty : integer);
procedure x (delta : integer);
procedure y (delta : integer);
procedure dx (delta : integer);
procedure dy (delta : integer);
procedure xy (x, y : integer);
procedure dxdy (deltax, deltay : integer);
procedure endwire;

These routines allow routing wires to be included in WOLCOMP-generated cells. All dimensions are specified in WOLCOMP internal coordinates (basically, $1\lambda = 2$ units, and the cell origin is in the lower left corner). The width of the wire is set by **width**; the wire is started by a call to **wire** with the layer (pass the CIF layer name–for example, metal in the SCG technology is CMF–and the start coordinates of the wire). The procedures **x**, **y**, and **xy** set the next corner of the wire to have that x,y and (x,y) coordinate, respectively. The procedures **dx**, **dy**, and **dxdy** are analogous, but are passed the *relative* displacement of the next vertex from the current point. To terminate the wire, call **endwire**.

Exported Variables

```
EXPORT
TYPE
   strg = string[80];
   strng10 = string[10];
   cptr = ^cel;
   sptr = ^subcel;
```

```
   cel = record
          name : strg;
          lx, ly, cifnum  :integer;
          next : cptr;
          contents : sptr;
       end;

   subcel = record
              cell : cptr;
              x, y, xfrm : integer;
              next : sptr;
            end;

   cif_types = (LYON, TELLE, WOL);

VAR
        { cell position, last cell position   }
   x0, y0, x1, y1 : integer;

        { value of lambda in 1/100 of microns }
   lambda : integer;

        { type of CIF being parsed            }
   cif_type : cif_types;

        { should the library be searched for each subcell? }
   search_library : boolean;

        { if true, print cell list on line printer }
   print_cells : boolean;

        { filename prefix to specify directories }
   prefix : strg;
```