Combining Graphics and Procedures in a VLSI Layout Tool: The Tpack System

Robert N. Mayo

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

ABSTRACT

Tpack is a system for VLSI module generation that uses both graphical and procedural information. A graphical editor is used to specify *tiles* of mask information, then procedures are written to arrange the tiles into modules. This technique combines the visual power of graphical systems with the programming power of procedural systems. Since all of the mask information is contained in the tiles, the same procedures may be used for different design rules or technologies, merely by supplying a different set of tiles. This paper describes the procedural and graphical interfaces, and discusses two module generators that have been built with them.

Introduction

There are two common methods of VLSI layout today: procedural and graphical. In the procedural approach the designer writes a program to generate the circuit layout. In the graphical approach, the designer interacts with a CAD system that displays the layout being designed. Each approach has its advantages and its shortcomings. The Tpack system described in this paper uses the best ideas from each approach by decomposing the layout process into a graphical portion and a procedural portion. This decomposition works especially well for building module generators (programs that produce standard structures from the designer's specification) because it allows the pieces to be changed and re-used for many different purposes.

Each of the procedural and graphical approaches has advantages and disadvantages [Row80, Tri81, Wil81]. Procedural layout allows the full power and generality of a programming language to be used in generating a circuit layout. Procedures are written to generate each structure of the circuit, and are combined hierarchically using the methods defined for the programming language [BMS81, Kar83]. Parameters may be passed to procedures in order to generate different versions of the same structure. In this way a property of one part of the layout (such as the size of an array of memory cells) can affect another part of the circuit (such as the location of power and ground busses). It is also possible to store non-mask information, such as power consumption, and use it to alter the topology of the layout.

Procedural layout offers great power and flexibility, but suffers from two drawbacks. First, it is hard to embed graphical information in a textual procedure: topological information is hard to visualize when specified textually. Second, the results of even a small change cannot be seen without "recompiling" the whole circuit, which can be a time-consuming process. Thus, procedural systems tend not to be very interactive.

The graphical approach, on the other hand, has the designer specify his layout by editing a two-dimensional picture of it on a color display [FaR78, IBM78, KeN82, Ous81]. The layout is modified by commands to add and delete mask geometries, commands to move sections of the layout, and commands to duplicate areas of the layout. Changes in the design are reflected immediately on the display. Because of the visual power and instantaneous feedback, graphical systems are especially well suited to editing small pieces of designs.

However, graphical tools provide only limited mechanisms for building up larger circuits. Cells may be composed hierarchically, with each cell containing some mask information and/or other cells. In addition, cells may be formed into arrays of identical elements. Unfortunately, current graphical systems do not generally allow cells to be parameterized, nor do they provide support for non-homogeneous arrays. Thus, graphical tools are often difficult to use for chip assembly and for layout of irregular modules.

This paper describes an intermediate mechanism between the above alternatives, called *tile packing*. Tile packing combines graphical information with simple procedures and is especially suitable for generating *semi-regular* modules. Semi-regular modules are those that consist of a few basic elements arranged in irregular patterns. PLAs and ROMs are obvious examples of semi-regular modules, but there are many other examples as well. For example, one kind of barrel shifter is composed of a uniform array of cells, except for the cells on the diagonal, which are different from the rest [MeC80,plate 13]. This kind of structure cannot be captured in the simple arrays provided by graphical editors such as Caesar [Ous81]. Other examples of these semi-regular structures include NOR decoders and Hamming encoders. Our experience is that virtually every chip design contains several of these structures.

Tpack is a set of procedures that may be used to write module generators based on tile packing. The following sections describe the concepts in Tpack and its interface as seen by the designer of a module generator. This paper then discusses two module generators that have been written and describe our experiences and conclusions.

The Tpack System

In the tile packing method, a module generator consists of two parts: a set of *tiles* that are designed graphically, and a set of procedures to arrange those tiles. Tiles are rectangular areas that contain layout information, and are designed using the Caesar graphical editor [Ous81]. A tile might contain a single pull-up, a CAM cell, or an ALU bit slice. In addition to the layout information in a tile, its size is also important and is used to control spacings. Some tiles are used only for spacing; any mask information they contain is ignored. Procedures, on the other hand, specify how tiles should be arranged to generate modules; they contain no knowledge of mask geometries or spacings. Since procedures only control the arrangement of tiles and not their contents, the tiles encapsulate all the information about design rules and technology. Because of this we can design different sets of tiles for different technologies or design rules, and the procedural information will not need to change.

In general, many tiles will be used when generating a layout, and it is important to see how they all fit together. To simplify this task, an example of the desired module is drawn (Fig. 1) and then rectangular labels are placed over the areas that are to be tiles. For example, a person designing a PLA generator will draw a small PLA that contains all of the features of a larger PLA. He will then

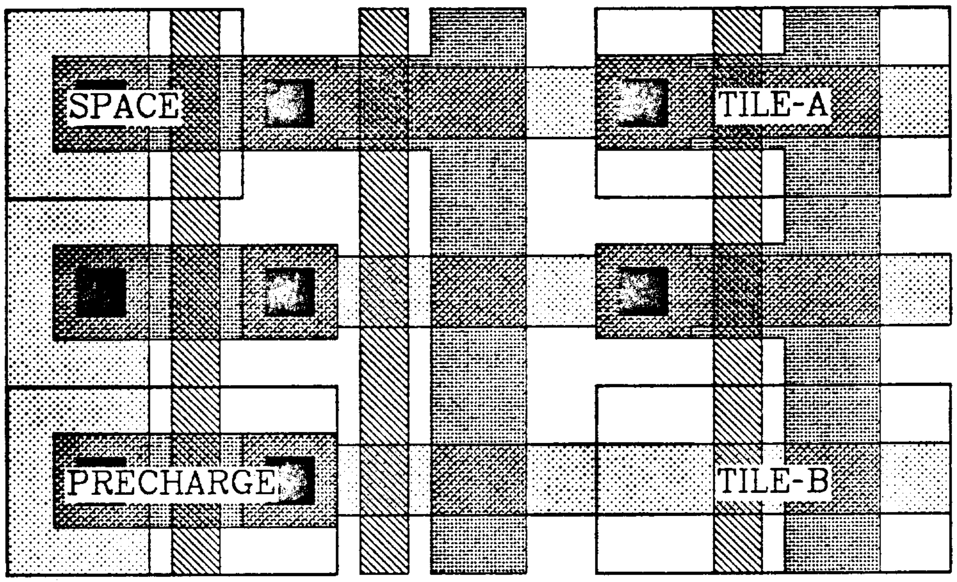

Figure 1: An example set of tiles. Tiles are areas of geometry defined by rectangular labels drawn on an example circuit. This allows tiles to be defined in their intended context, making it easier to see how a tile will fit together with other tiles. The information in some tiles, such as the SPACE tile, may be used by the program solely for spacing purposes.

place rectangular labels over the parts of the structure that are to be used as tiles by the PLA generator procedure. This method eliminates most problems associated with tiles not abutting or overlapping properly, since the designer can see the entire structure when its components are defined. In fact, if the procedure-writer ensures that tiles are only used in the same context as in the example then the design rule correctness of the example implies the that all generated modules will also be design rule correct.

Tiles are arranged by procedures that use the Tpack library of routines. These routines are written in the C programming language, and are designed to support tile-based module generation. Included are routines for initializing Tpack, reading tiles from files, creating new blank tiles, stretching tiles, and writing tiles into files. Also included are placement routines that allow information from a tile to be copied into another tile at a specific point. This allows us to build new tiles out of smaller tiles, and then use these new tiles, in turn, to produce even larger tiles. The final module is just a new tile generated in this fashion.

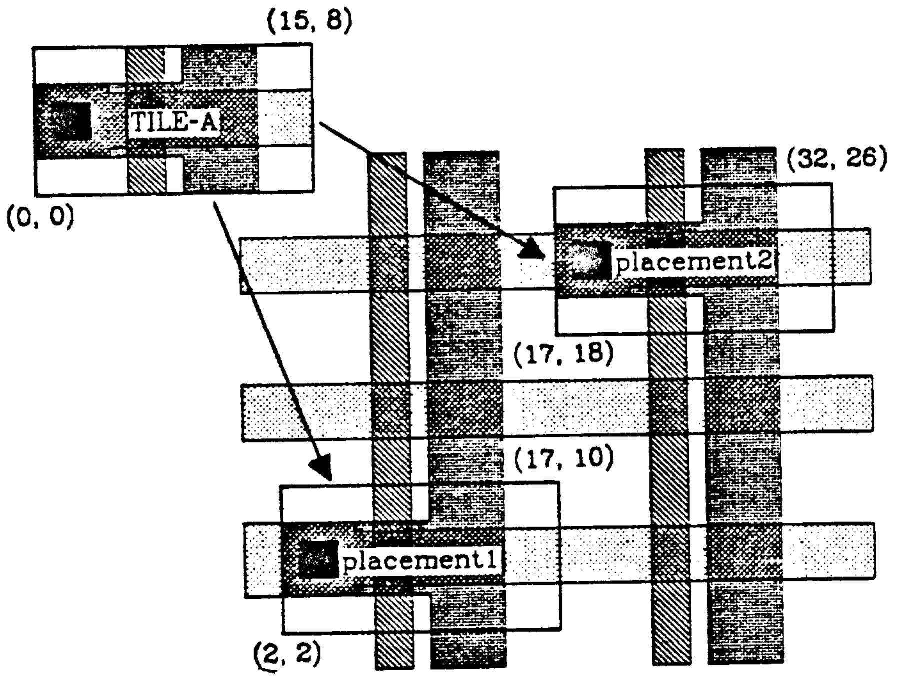

Figure 2: Tiles and their placements. Tiles contain mask information, and may be placed many times in a circuit. Each corner of each such *placement* has a fixed position in the layout, while the corner of a *tile* is considered to be relative to its lower left corner. For example, the upper right corner of TILE_A is (15, 8), while the same corner of its first use, *placement1*, is at (17, 10). The lower left corner of every *tile* is (0, 0).

The tile placement routine TPpaint_tile takes as parameters a ***source tile***, a ***destination tile***, and a ***point***. As a result of the call, the contents of the source tile are copied into the destination tile such that the lower left corner of the source tile appears at the specified point. A *placement* rectangle is returned that indicates the area in the destination tile that is now covered by the source tile (Fig. 2). Later on the corners of this placement may be used in positioning other tiles. This allows tiles to be placed by alignment with previously placed tiles; only the first tile in a module is placed at an absolute location. The alignment function TPalign takes two points as arguments: one corner of a placement and one corner of a tile. The result is a point that describes where the new tile should be placed so that the two points coincide. Note that the new tile is aligned with the placement of a previous tile, not with an absolute coordinate.

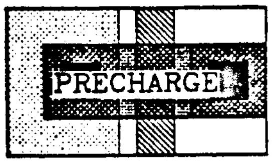

Figure 3a

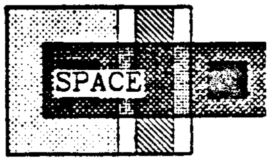

Figure 3b

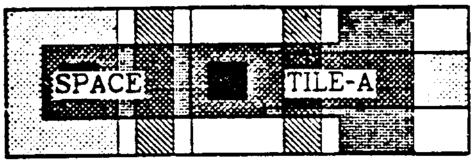

Figure 3c

Figure 3: Using tiles to control spacing. The PRECHARGE tile and TILE-A (from Figure 1) are to be placed so that their left edges are separated by a distance equal to the width of the SPACE tile. This is done by first placing down the PRECHARGE tile (Fig. 3a). The SPACE tile is then aligned with the placement of the PRECHARGE tile, but the information contained in the SPACE tile is not actually copied into the circuit (Fig. 3b). TILE-A is then placed down so that it aligns with the right edge of the SPACE tile placement (Fig. 3c). The result is that the contacts in the PRECHARGE tile and TILE-A overlap.

The alignment procedure described so far is fine for non-overlapping tiles, but there are cases where the designer wishes to overlap tiles. This cannot be done with the current scheme, because tiles may only be aligned with corners of previously placed tiles. The solution that we have adopted is to use some tiles solely for spacing purposes. All mask geometry contained in these tiles is ignored, but the placement of the tile provides additional corners with which to align. The TPspace routine is used for this purpose. This routine behaves in the same manner as the TPpaint_tile routine, except that no new mask information is placed; its only function is to return a placement indicating where the tile would have been placed by TPpaint_tile. Other tiles may then be aligned with this placement (see Figure 3). This allows tiles to be overlapped by an arbitrary amount, but note that this overlap distance is not contained in the the program: it is specified graphically in the collection of tiles.

Source Code Statistics		
component	lines code	total lines
tpack (caesar database part)	8316	18619
tpack (interface)	2421	4293

Table 1: Code size for the two Tpack components.

The tpack system is implemented as two pieces. All of the storage of mask geometries, as well as file input and output, is handled by a modified version of the Caesar database. On top of this database are routines for creating the abstraction of a tile, as well as routines for manipulating these tiles. Table 1 gives the code sizes for these components of Tpack.

Quilt — An Example

We have built two module generators in order to evaluate the effectiveness of tile packing. The first is Quilt, a program that assembles rectangular arrays of non-overlapping tiles. The output is a personalized array. Array elements may be different sizes, as long as each element abuts with its neighbors. Many VLSI structures can be viewed as personalized arrays, and Quilt is a convenient way of generating them.

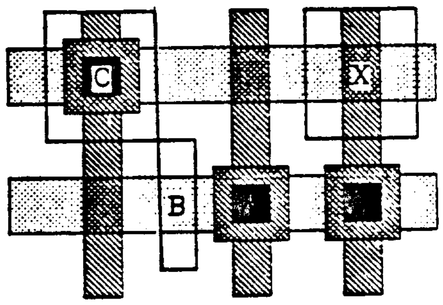

Figure 4a. a set of tiles for Quilt

```
CCBXXBX
XCBCCBC
CXBXCBC
```

Figure 4b: input text for Quilt

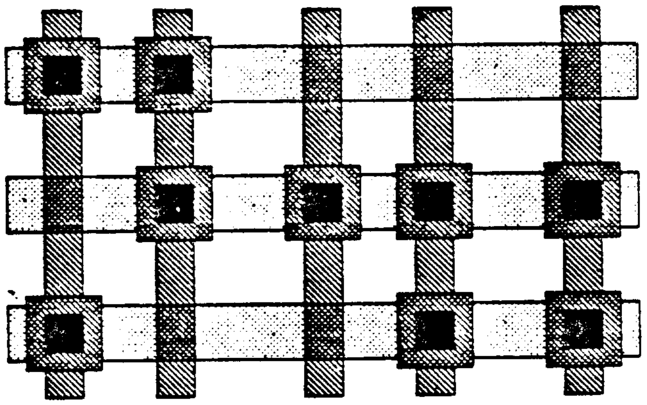

Figure 4c: the output of Quilt

Figure 4: Using the Quilt module generator. Quilt creates a personalized matrix of tiles from a set of tiles (Fig. 4a) and a matrix of characters (Fig. 4b). Each character in the matrix names a tile to be placed in the corresponding position in the output.

The user defines a set of tiles for Quilt to process, such as those shown in Figure 4a. Each tile is given a one character name. Quilt reads an array of characters from a text file, such as that shown in Figure 4b, and for each character in the file the tile of the same name is placed in the output. Figure 4c shows the result of running Quilt with the example data.

```
1  TPinitialize(argc, argv);
2  out_tile = TPcreate_tile();
3
4  strcpy(tilename,"?");
5  start_prev_row = origin_placement;
6
7  newrow = TRUE;
8  while ( (inchar = getc(infile)) != EOF) {
9     if (inchar == '0)
10        newrow = TRUE;
11    else if (inchar != ' ') {
12        tilename[0] = inchar;
13        this_tile = TPname_to_tile(tilename);
14        if (newrow) {
15           prev_placement = TPpaint_tile(this_tile, out_tile,
16              TPalign(pLL(start_prev_row), tUL(this_tile)) );
17           start_prev_row = prev_placement;
18           newrow = FALSE;
19        } else {
20           prev_placement = TPpaint_tile(this_tile, out_tile,
21              TPalign(pLR(prev_placement), tLL(this_tile)) );
22        }
23    }
24 }
25
26 TPwrite_tile(out_tile);
```

Figure 5: C code for the Quilt program. **The Tpack library provides functions to read in tiles, create new tiles, place and align tiles, and write out tiles. As a result, module generators like Quilt are easy to write and are very flexible.**

The code for Quilt is shown in Figure 5. The first line passes command line arguments to an initialization routine that loads in a set of tiles. In line 2 a new tile is created in which to form the result. After initialization Quilt enters the main loop, with each iteration processing a character from the input file. In line 10 Quilt sets a flag just before the beginning of each line. Lines 12 and 13 get the tile that has the same name as the input character, which is the tile that we want to place. If this is the beginning of a line, we want to place the new tile just below the left-most tile in the previous row. This is accomplished by the TPpaint_tile call on lines

15 and 16. The TPalign routine embedded in this call has 2 points as arguments: the lower left corner of a placement (generated by pLL) and the upper left corner of a tile (generated by tUL). The tile is placed such that these two corners align. The location of this new tile (a *placement*) is stored in *start_prev_row* for use at the start of the next row. The TPpaint_tile call on lines 20 and 21 is executed for the rest of the tiles in the row. In this call, each tile has its lower left corner aligned with the lower right corner of the placement of the previous tile.

Quilt is a good example of how flexibility can be increased by decomposing the layout process into stages. The designer supplies a set of tiles and a matrix of characters, and Quilt arranges the tiles accordingly. New modules can be generated by changing either the tiles or the matrix of characters. Quilt fits especially well in the Unix world where processes communicate over pipes: it reads the character matrix from its standard input, and thus can be used as the last filter in a pipeline. Many simple module generators can be built as programs that generate character arrays for Quilt. An example of this is the Vlsifont program for generating text logos. Appendix C describes Vlsifont, and appendix B gives more details on the Quilt program.

Tpla — A PLA Generator

Tpla is a more complex example of tile packing. It is a PLA generator written using the Tpack library. Several different styles of PLAs may be produced: nMOS with buried contacts, nMOS with butting contacts, precharged CMOS, and CMOS with ratioed pull-ups. While there is a different set of tiles for each style, the procedural part of the PLA generator remains the same.

A large number of tiles are needed to specify a PLA structure. Each location in the core of the AND and OR planes may contain one of three tiles: a transistor, no transistor but a continuation of the input line, or no transistor and no input line.* The AND plane uses these tiles in two orientations, and the OR plane uses these in two more orientations. We therefore have 6 tiles for the core of the AND plane and 6 for the OR plane. The vertical and horizontal pitch of the AND plane is specified by a tile, and the horizontal pitch of the OR plane is set by another tile. The vertical pitch of the OR plane is the same as that of the AND plane.

Fourteen tiles surround the periphery of the planes. These tiles include tiles for power and ground routing, input and output drivers, pull-ups or precharge tiles, and tiles that connect the two planes. Eight tiles are used to control the overlapping of the peripheral tiles with the core region (1 tile for each side of the AND and OR planes). Some tiles have more than one form: there are both clocked and unclocked input drivers, for example. This gives a total of 36 tiles for the PLA.

At first we felt that 36 tiles was a lot for a PLA, especially since all 36 tiles must be redesigned when converting to a new technology. However, it is much easier to generate new tiles than to write new procedural code. Thus, for example, separate tiles are defined for two symmetries in each of the AND and OR planes, even though the tiles are just flipped and rotated versions of each other. Even with this many tiles it has been easy to retarget the PLA generator. Defining tiles is a

* The last item is included so that we can extend the input lines only as far as is needed. This helps reduce the capacitance of the circuit and makes for faster PLAs.

quick process, since it only involves drawing a picture of a PLA and labeling the tiles. Our experience has been that designing a new style of PLA is more time consuming than retargeting our PLA generator for the new style. A person familiar with PLAs in some new technology should be able to retarget our PLA generator for that technology in an afternoon, assuming that the PLA fits our model of an *and-plane* and an *or-plane* surrounded by peripheral circuitry.

Debugging a retargeted set of tiles in Tpla is easy in comparison to debugging a retargeted purely procedural PLA generator. Few bugs appear in the first place, due to the fact that the tiles are specified by outlining areas on an example PLA. Those bugs that do exist show up in the graphical output and are easy to relate back to the graphical collection of tiles. A purely procedural approach to generating PLAs is likely to result in more bugs due to the lack of immediate visual feedback. The correction of these bugs involves modifying the procedural description and recompiling the PLA generator. Correction of bugs in a retargeted Tpla only requires modifying the tiles and regenerating the PLA, therefore removing the compile phase from the edit-compile-generate cycle of the purely procedural approach.

The code for Tpla is about 720 'C' statements. A similar PLA generator, Mkpla, does not use the tile packing scheme and consists of about 900 lines of Pascal code [Lan80]. While we have not saved much code by using tile packing, we have gained technology independence. Retargeting a program like Mkpla for a new technology requires major modifications since every rectangle that appears in the PLA is specified in the code. It is interesting to note that the flexibility of procedural information is increased by separating out all the of the graphical information.

Performance Statistics						
test case				time in seconds		
truth table	# of inputs	# of outputs	# of product terms	mkpla	tpla (no merging)	tpla (merging)
A	3	5	8	1.3	10.6	14.0
B	10	23	34	3.7	43.2	170.2

Table 2: Performance of Tpla versus Mkpla.

Table 2 shows the performance of Tpla versus Mkpla. As can be seen, the hard-coded Mkpla program runs much faster than the more flexible tile-based module generator. Part of this is due to the fact that the Tpack system is built upon the Caesar database which is tailored to interactive applications. The Tpack system has an option that causes it to merge adjacent rectangles into one in order to decrease the size of the database. Table 2 shows performance data for PLA generation both with and without merging.

Appendix D describes Tpla in more detail. Tpla has many options, some of which choose tiles for the input and output drivers. It would be useful to extend this mechanism in a general way so that other tiles could have alternative definitions, such as alternative pullups for lower power consumption, or special input drivers for high speed PLAs.

Conclusions & Ideas for Improvement

The major contribution of the tile packing method is the decomposition of module generation into a graphical portion and a procedural portion. This decomposition provides great flexibility. A different set of tiles may be designed for a new technology, for new design rules, or in order to generate new styles of modules. In all cases the procedural information stays the same. This means that standard programs, like Tpla, may be used to generate customized modules — all the designer has to do is modify the set of tiles that the program assembles.

Tpack is also useful when a program can be written that generates many different structures. Quilt is a example of such a program — we expect that many other simple module generators can be built as front-ends that pipe data into Quilt. Tpla is a much longer program that generates a large class of PLA structures in different technologies. Both Tpla and Quilt may be used with different sets of tiles, spreading the cost of writing the code over many designs.

Designing the tiles for a VLSI structure is a quick process due to the use of an example layout and the immediate feedback of a graphical editor. The design of tiles in the context of an example helps ensure that the tiles will fit together properly. Any tile design errors that do exist will appear in the final structure graphically and are therefore easy to relate back to the graphical tile description.

Writing the procedures that assemble the tiles is more time consuming. For simple structures, such as personalized arrays of cells, the code is fairly short. When we write procedures for assembling more complicated structures, such as PLAs, the code gets much longer. Tpack is most useful when the overall structure of a layout is simple and the low level details can be encapsulated inside of a small set of tiles. As a layout becomes more and more complex, Tpack's advantages over purely procedural generators are reduced. In complex modules, most of the irregularity can not be forced down into the tiles, and writing the Tpack code is not much simpler than writing a completely procedural module generator. Of course, the tile based technique still provides technology independence for the module generator.

Although our decomposition of module generation into a procedural portion and a graphical portion seems to work well, there are some areas where it is awkward. The information about which tile aligns with which other tile is contained in the procedural code and thus is not graphically available to the tile designer. This can be confusing — the tile designer must guess and then run the program to try out his guess. It would be a good idea for a future system to provide the alignment information in a graphical form for the tile designer, or else allow the tile designer to specify this graphically with the tiles.

As an extension to this idea, it would be convenient to allow alignment of arbitrary points within tiles — not just their corners. This would eliminate the need for special spacing tiles in many cases, and would be more intuitive to the tile designer. It would also help move more of the alignment information out of the code and into the graphical world.

For complicated modules, it would be convenient to rotate and mirror tiles before placing them down. The Tpack system currently does not provide rotation or mirroring.

As a practical consideration, is not a good idea to require that the tiles be prepared with the Caesar editor. Editors come and go, and we would like our module generators to be independent of this evolution. It would be easy to define a protocol which allowed the Tpack system to get tile information interactively from a editor, and then ask that editor to place down geometry in certain places. This would allow designers to interactively run module generators and would allow tiles to be defined using whatever editor the designer was used to. This would even work on editors that provide different views of mask geometries. For example, the tiles could contain sticks drawings and the module generator would not be concerned with that fact, just with the fact that a certain tile needed to be placed at a certain location.

Acknowledgements

I would like to thank John Ousterhout for his continual help and guidance in all areas of my research at Berkeley, and especially for suggesting the basic ideas that lead to the Tpack system. It is often said that graduate students develop ideas and their advisors publish them — in this case I feel that the reverse is partially true.

The students and faculty at Berkeley have created a research environment that is top-notch. I am glad to be a part of it, and I thank them all. Many thanks to Dave Patterson for providing VLSI design projects that motivate and push CAD research, and John Ousterhout for leading the CAD research in the CS division. Grace Mah has used Tpack extensively, and I thank her for her comments and patience in waiting for bug fixes. Dave Ungar deserves thanks using the Quilt program with enthusiasm. Howard Landman designed mkpla, and the PLAs produced by that program provided the tiles for the first PLAs from Tpla. Howard's program also provided the equations that are used in Tpla to place additional ground and power lines. Randy Katz taught the VLSI class in the fall of 1983, and provided useful comments on this report.

Special thanks go to several other students at Berkeley for their enthusiasm and general good nature: Michael Arnold, Artie Chang, Tom Conroy, Susan Eggers, Gordon Hamachi, Mark "the hop" Hofmann, Ken Keller, Herb Ko, Grace Mah, Joan Pendleton, Dave Petersen, Harry Rubin, Walter Scott, and George Taylor.

References

[BMS81] J. Batali, N. Mayle, H. Shrobe, G. Sussman and D. Weise, The DPL/Daedalus Design Environment, in *VLSI '81*, Academic Press, 1981, pp. 183-192.

[FaR78] D. G. Fairbairn and J. H. Rowson, ICARUS: An Interactive Integrated Circuit Layout Program, *Proc. 15th Design Automation Conference*, 1978, pp. 188-192.

[IBM78] B. Infante, D. Bracken, B. McCalls, S. Yamakoshi and E. Cohen, An Interactive Graphics System for the Design of Integrated Circuits, *Proc. 15th Design Automation Conference*, 1978, pp. 182-187.

[Kar83] K. Karplus, CHISEL: An Extension to the Programming Language C for VLSI Layout, Report No. STAN-CS-82-959, PhD Thesis, Stanford University, 1983.

[KeN82] K. H. Keller and A. R. Newton, KIC2: A Low-Cost, Interactive Editor for Integrated Circuit Design, *Digest of Papers, Compcon 82*, 1982, pp. 305-306.

[Lan80] H. A. Landman, Automatic Layout of Optimized PLA Structures, *Masters Report -- EECS Department*, 1980.

[MeC80] C. A. Mead and L. A. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[Ous81] J. K. Ousterhout, Caesar: An Interactive Editor for VLSI Layouts, *VLSI Design*, Fourth Quarter, 1981, pp. 34-38.

[Row80] J. Rowson, Procedural vs. Graphical Design of Integrated Circuits, *Lambda (now VLSI Design)*, Fourth Quarter, 1980, pp. 6-7.

[Tri81] S. Trimberger, Combining Graphics and a Layout Language in a Single Interative System, *Proc. 18th Design Automation Conference*, 1981, pp. 234-239.

[Wil81] J. Williams, Symbolic Artwork Systems, *Lambda (now VLSI Design) 2*,2 (Second Quarter, 1981), pp. 64-67.

NAME

tpack – routines for generating semi-regular modules

DESCRIPTION

Tpack (tile packer) is a library of 'C' routines that aid the process of generating semi-regular modules. Decoder planes, barrel shifters, and PLAs are common examples of semi-regular modules.

Using Caesar, a tpack user will draw an example of a finished module and then break it into tiles. These tiles represent the building blocks for more complicated instances of the module. The tpack library provides routines to aid in assembling tiles into a finished module.

MAKING AN EXAMPLE MODULE

The first step in using tpack is to create an example instance of the module, called a *template*. The basic building blocks of the structure, or *tiles*, are then chosen. Each tile should be given a name by means of a rectangular label which defines its contents. If the tiles in the module do not abut (e.g. they overlap) it is useful to define another tile whose size indicates how far apart the tiles should be placed.

Templates should be in Caesar format and, by convention, end with a **.tp** suffix. With some programs, it is possible to generate the same structure in a different technology or style by changing just the template. If this is the case, each template should have a filename of the form basename-*style*.**tp**. The *style* part of the filename interacts with the **-s** option (see later part of this manual).

WRITING A TPACK PROGRAM

A tpack program is the 'C' code which assembles tiles into the desired module. Typically this program reads a file (such as a truth table) and then calls the tile placement routines in the tpack library.

The tpack program must first include the file **~cad/lib/tpack.h** which defines the interface to the tpack system. Next the **TPinitialize** procedure is called. This procedure processes command line arguments, opens an input file as the standard input (**stdin**), and loads in a template.

The program should now read from the standard input and compute where to place the next tile. Tiles may be aligned with previously placed tiles or placed at absolute coordinates. If a tile is to overlap an existing tile the program must space over the distance of the overlap before placing the tile.

When all tiles are placed the program should call the routine **TPwrite_tile** to create the output file that was specified on the command line.

To use the tpack library be sure to include it with your compile or load command (e.g. **cc** *your_file* **~cad/lib/tpack.lib**).

ROUTINES

Initialization and Output Routines

TPinitialize(*argc, argv, base_name*)

The tpack system is initialized, command line arguments are processed, and a template is loaded. The file descriptor **stdin** is attached to the input file specified on the command line. The template's filename is formed by taking the *base_name*, adding any extension indicated by the **-s** option, and then adding the **.lp** suffix if no suffix was provided. The **-t** option allows the user to override *base_name* from the command line.

Argc and *argv* should contain the command line arguments. *Argc* is a count of the number of arguments, while *argv* is an array of pointers to strings. Strings of length zero are ignored (as is the flag consisting of a single space), in order to make it easy for the calling program to intercept its own arguments. *Argc* and *argv* are of the same structure as the two parameters passed to the main program. A later section of this manual summarizes the command line options.

TPload_tiles(*file_name*)
The given *file_name* is read, and each rectangular label found in the file becomes a tile accessible via TPname_to_tile. No extensions are added to *file_name*.

TILE TPread_tile(*file_name*)
A tile is created and *file_name* is read into it. The tile is returned as the value of the function.

TPwrite_tile(*tile, filename*)
The tile *tile* is written to the file specified by *filename*, with **.ca** or **.cif** extensions added. See the description of the **-o** option for information on what file name is chosen if *filename* is the null string. The choice between Caesar or CIF format is chosen with the **-a** or **-c** command line options.

Tile creation, deletion, and access

TPdelete_tile(*tile*)
The tile *tile* is deleted from the database and the space occupied by it is reused.

TILE TPcreate_tile(*name*)
A new, empty tile is created and given the name *name*. This name is used by the routine **TPname_to_tile** and in error messages. The type **TILE** returned is a unique ID for the tile, not the tile itself. Currently this is implemented by defining the type TILE to be a pointer to the internal database representation of the tile.

int TPtile_exists(*name*)
TRUE (1) is returned if a tile with the given *name* exists (such as in the template or from a call to TPcreate_tile).

TILE TPname_to_tile(*name*)
A value of type **TILE** is returned. This value is a unique ID for the tile that has the name *name*. This name comes from a call to TPcreate_tile(), or

from the rectangular label that defined it in a template that was read in by TPread_tiles() or TPinitialize(). If the tile does not exist then a value of NULL is returned and an error message is printed.

RECTANGLE TPsize_of_tile(*tile*)
A rectangle is returned that is the same size as the tile *tile*. The rectangle's lower left corner is located at the coordinate (0, 0). All coordinates in tpack are specified in half-lambda.

Painting and Placement Routines

RECTANGLE TPpaint_tile(*from_tile, to_tile, ll_corner*)
The tile *from_tile* is painted into the tile *to_tile* such that its lower left corner is placed at the point *ll_corner* in the tile *to_tile* . The location of the newly painted area in the output tile is returned as a value of type RECTANGLE. The tile *to_tile* is often an empty tile made by **TPcreate_tile()**. The point *ll_corner* is almost never provided directly, it is usually generated by routines such as **align()**.

TPdisp_tile(*from_tile, ll_corner*)
A rectangle the size of *from_tile* with the lower left corner located at *ll_corner* is returned. Note that this routine behaves exactly like the routine TPpaint_tile except that no output tile is modified. This routine, in conjunction with the **align** routine, is useful for controlling the overlap of tiles.

RECTANGLE TPpaint_cell(*from_tile, to_tile, ll_corner*)
This routine behaves like **TPpaint_tile()** except that the *from_tile* is placed as a subcell rather than painted into place. The tile *from_tile* must exist in the file system (i.e. it must have been read in from disk or have been written out to disk).

Label Manipulation Routines

TPplace_label(*tile, rect, label_name*)
A label named *label_name* is place in the tile *tile*. The size and location of the label is the given by the RECTANGLE *rect*.

int TPfind_label(*tile, &rect1, str, &rect2*)
The tile *tile* is searched for a label of name *str*. The location of the first such label found is returned in the rectangle *rect2*. The function returns 1 if such a label was found, and 0 otherwise. The rectangle pointer *&rect1*, if non-NULL, restricts the search to an area of the tile.

TPstrip_labels(*tile, ch*)
All labels in the tile *tile* that begin with the character *ch* are deleted.

TPstretch_tile(*tile, str, num*)

The string *str* is the name of one or more labels within the tile *tile*. Each of these labels must be of zero width or zero height, i.e. they must be lines. Each of these lines define a line across which the tile will be stretched. The amount of the stretch is specified by *num* in units of half-lambda. Stretching such a line turns it into a rectangle. Note that if the tile contains 2 lines that are co-linear, the stretching of one of them will turn both into rectangles.

Point-Valued Routines

POINT tLL(*tile*)
POINT tLR(*tile*)
POINT tUL(*tile*)
POINT tUR(*tile*)

The location of the specified corner of tile *tile*, relative to the tile's lower left corner, is returned as a point. LL stands for lower-left, LR for lower-right, UL for upper-left, and UR for upper-right. Note that **tLL()** returns (0, 0).

POINT rLL(*rect*)
POINT rLR(*rect*)
POINT rUL(*rect*)
POINT rUR(*rect*)

The location of the specified corner of the rectangle *rect* is returned as a point. LL stands for lower-left, LR for lower-right, UL for upper-left, and UR for upper-right.

POINT align(*p1, p2*)

A point is computed such that when added to the point *p2* gives the point *p1*. *p1* is normally a corner of a rectangle within a tile and *p2* is normally a corner of a tile. In this case the point computed can be treated as the location for the placement of the tile.

For example, TPpaint_tile(outtile, fromtile, align(rUL(rect), tLL(fromtile))) will paint the tile *fromtile* into *outtile* such that the lower left corner of *fromtile* is aligned with the upper-left corner of *rect*. In this example *rect* would probably be something returned from a previous TPpaint_tile() call.

Point and Rectangle Addition Routines

POINT TPadd_pp(*p1, p2*)
POINT TPsub_pp(*p1, p2*)

The points *p1* and *p2* are added or subtracted, and the result is returned as a point. In the subtract case *p2* is subtracted from *p1*.

RECTANGLE TPadd_rp(*r1, p1*)
RECTANGLE TPsub_rp(*r1, p1*)

The rectangle *r1* has the point *p1* added or subtracted from it. This has the effect of displacing the rectangle in the X and/or Y dimensions.

Miscellaneous Functions

int TPget_lambda()
This function returns the current value of lambda in centi-microns.

INTERFACE DATA STRUCTURES

In those cases where tiles must be placed using absolute, (half-lambda) coordinates, it is useful to know that **RECTANGLE**s and **POINT**s are defined as:

```
typedef struct {
   int x_left, x_right, y_top, y_bot;
} RECTANGLE;

typedef struct {
   int x, y;
} POINT;
```

The variable **ORIGIN_POINTER** is predefined to be (0, 0). **ORIGIN_RECT** is defined to be a zero-sized rectangle located at the origin.

OPTIONS ACCEPTED BY TPinitialize()

Typical command line: *program_name* [-t *template*] [-s *style*] [-o *output_file*] *input_file*

-a produce Caesar format (this is the default)

-c produce CIF format

-v be verbose (sequentially label the tiles in the output for debugging purposes; also print out information about the number of rectangles processed by tpack)

-s *style* generate output using the template for this style (see TPinitialize for details)

-o The next argument is taken to be the base name of the output file. The default is the input file name with any extensions removed. If there is not input file specified and no -o option specified, the output will go to **stdout**.

-p (pipe mode) Send the output to **stdout**.

-t The next argument specifies the template base name to use. This overrides the default supplied by the program. (see TPinitialize)

-l *num* Set lambda to *num* centimicrons. (200 is the default)

input_file
The name of the file that the program should read from (such as a truth table file). If this filename is omitted then the input is taken from the standard input (such as a pipe).

-M *num*
Normally tpack merges rectangles to form maximal horizontal strips, just like

Caesar(CAD). If the **-M** option is present tpack will only look back through the last *num* rectangles on each layer when doing merges. A small value for *num* will make tpack run faster, but not all possible merges will be found. The **-v** option gives information about the number of merges done.

-D *num1 num2*

The *Demo* or *Debug* option. This option will cause **tpack** to place only the first *num1* tiles, and the last *num2* of those will be outlined with rectangular labels. In addition, if a tile called "blotch" is defined then a copy of it will be placed in the output tile upon each call to the *align* function during the placing of the last *num2* tiles. The blotch tile will be centered on the first point passed to *align*, and usually consists of a small blotch of brightly colored paint. This has the effect of marking the alignment points of tiles. The last tile painted into is assumed to be the output tile.

EXAMPLE

It is highly recommended that the example in **~cad/src/quilt** be examined. Look at both the template and the 'C' code. A more complex example is in **~cad/src/tpla**.

FILES

~cad/lib/tpack.h	(definition of the tpack interface)
~cad/lib/tpack.lib	(linkable tpack library)
~cad/src/quilt/*	(an example of a tpack program)
~cad/lib/caesar/*.tech	(technology description files)

ALSO SEE

Caesar(CAD)
'C' Manual
Quilt(CAD)
Tpla(CAD)
Robert N. Mayo and John K. Ousterhout, *Pictures with Parentheses: Combining Graphics and Procedures in a VLSI Layout Tool,* Proceedings of the 20th Design Automation Conference, June, 1983.

AUTHOR

Robert N. Mayo

BUGS

When a tile contains part of a subcell, or touches a subcell, then the whole subcell is considered to be part of the tile. The same goes for arrays of subcells.

NAME

quilt – assemble tiles into a rectangular array

SYNOPSIS

quilt [-acv] [-s standardTemplate] [-t *template*] [-o *output_file*] *text_file*

DESCRIPTION

The user of Quilt first creates a Caesar file, called the *template*, containing a circuit layout over which single-character rectangular labels have been placed. These labels define blocks of the circuit called *tiles*. Using a text editor, the user then creates an array of characters (each line defines one row in the array). Quilt reads in the array of characters and produces a layout where each character is replaced by the tile of the same name. Spaces and blank lines in the text file are ignored.

For example, we can produce a 3X3 checkerboard with this input file:

```
ABA
BAB
ABA
```

The template file would contain rectangular labels called A and B. The paint and subcells underneath these labels would be placed in the output file in a checkerboard fashion.

Tiles are normally placed so that they abut with each other in the following fashion: the lower edges of all tiles in a row are aligned, tiles are packed together horizontally as closely as possible within a row, and the first tile in a row touches the first tile in the row above it and the first tile in the row below it.

If we wish tiles to be spaced a certain distance apart, instead of what was described previously, we can use *spacing* tiles. Spacing tiles are tiles which indicate, by their size, how far apart two tiles should be spaced. For horizontal spacing, the single-character name of a spacing tile should be placed in parentheses between the names of the two tiles on either side of it. The left edges of the two tiles will be spaced apart by the width of the spacing tile. For example, the form "AB" places tiles A and B next to each other while "A(C)B" places them apart by a distance determined by C. If C is of zero width, A and B will be placed on top of each other. If C is the same width as A, A and B will abut (note that "A(A)B" is the same as "AB"). If the width of C is less than the width of A the tiles will overlap, and if C has a width greater than A they will be separated.

Spacing tiles may also be used to control the vertical spacing. A spacing tile at the beginning of a row (such as "(C)AB") will cause the bottom of the first tile in this row (in this case tile A) to be separated from from the bottom of the first tile in the row above by a distance equal to the height of the spacing tile.

Quilt is a small program written with the Tpack system.

OPTIONS

-a	produce Caesar format (this is the default)
-c	produce CIF format
-v	be verbose (sequentially label the tiles in the output, for debugging purposes)

-o The next argument is taken to be the base name of the output file. The default is the input file name with any extensions removed.

-t The next argument specifies the template to use. A **.tp** suffix is added if no suffix was specified.

-s *style* Use the template with the name **q-***style* located in ~cad/lib/quilt.

text_file
The name of quilt's text file. If this filename is omitted then the input is taken from the standard input (such as a pipe). If the input comes from the standard input and the **-o** option is not specified then the output will go to the standard output.

other options
Several other options are inherited from tpack(CAD).

FILES

~cad/bin/quilt	– executable
~cad/src/quilt/*	– source
~cad/lib/quilt/q-*	– location of standard templates
~cad/examples/quilt/*	– quilt example

SEE ALSO
tpack(CAD)

AUTHOR
Robert N. Mayo

BUGS
This program inherits any bugs that may exist in tpack(CAD).

NAME

vlsifont - create text logos for VLSI chips

SYNOPSIS

vlsifont [-k key] [-f font] word | quilt -s vlsifont

DESCRIPTION

The 'word' on the command line is rasterized into a matrix of characters suitable for input to quilt or viewing on a text terminal. 'word' may be surrounded by quotes to allow embedded spaces. The background characters in the rasterized image will be the same as the first character of *key*, while the foreground characters will be the same as the second character of *key*. *Key* defaults to "**em**".

If the output is piped to quilt, the user should use the standard template **~cad/lib/quilt/vlsifont.tp** by specifying the **-s vlsifont** switch, or else supply his own (see quilt(CAD) for how to do this using Caesar). The standard template recognizes these foreground and background characters:

e	– a small empty square
p	– a small poly square
d	– a small diffusion square
m	– a small metal square
E, P, D, or M	– larger versions of the above

FILES

~cad/lib/quilt/q-vlsifont.tp	– standard template for quilt
/usr/lib/vfont/*	– standard place for fonts
~cad/bin/vlsifont	– executable
~cad/src/vlsifont/*	– source

SEE ALSO

quilt(CAD), caesar(CAD), vfont(5), vfontinfo(1)
The Berkeley Font Catalogue

AUTHOR

Robert N. Mayo

BUGS

If the font does not specify the width of a space character then the width of the letter 'e' is used instead.

NOTES

MOSIS will not fabricate chips that contain logos or text over 50 microns high, unless permission is obtained first. (As of January 1983.)

HISTORY

This program is a modified version of the tool 'vfontinfo' from Berkeley.

NAME

tpla – technology independent PLA generator

SYNOPSIS

tpla [-acv] [-s *style*] [-o *output_file*] *input_file*

DESCRIPTION

tpla is a PLA generator that generates PLAs in several different styles and technologies. The input format is compatible with **eqntott**, see PLA(5) for details. **Tpla** does not handle split and folded PLAs.

Tpla is a program written with the Tpack system.

STYLES OF PLAs AVAILABLE

The following styles of PLAs are currently supported:

Bcis Buried contacts, nMOS, cis version (inputs and outputs on same side of the PLA). Clocked inputs and outputs are supported. Berkeley design rules.

Btrans
Buried contacts, nMOS, trans version (inputs and outputs on opposite sides of the PLA). Clocked inputs and outputs are supported. Berkeley design rules.

CD4cis
CMOS p-well process, 4 micron Berkeley design rules, dynamic PLAs with two separate precharge lines for the AND and OR planes, no inverting buffers between planes, cis version. Since the default for extra grounds lines is based on an nMOS PLA, use "-G 10" for this style. Clocked inputs and outputs are not supported.

CS3cis
CMOS p-well process, 3 micron MOSIS design rules (August, 1982), static PLA with p-channel pullups (pullup/pulldown = 1/2), cis version. Since the default for extra grounds lines is based on an nMOS PLA, use "-G 10" for this style. Clocked inputs and outputs are not supported.

CS3trans
CMOS p-well process, 3 micron MOSIS design rules (August, 1982), static PLA with p-channel pullups (pullup/pulldown = 1/2), trans version. Since the default for extra grounds lines is based on an nMOS PLA, use "-G 10" for this style. Clocked inputs and outputs are not supported.

Tcis Just like **Bcis** except that it has protection frames and terminals added (a special mod for EECS at Berkeley).

Ttrans
Just like **Btrans** except that it has protection frames and terminals added.

Mcis Mead & Conway design rules. Butting contacts, nMOS, cis version (inputs and outputs on same side of the PLA). Clocked inputs and outputs are supported.

Mtrans
Mead & Conway design rules. Butting contacts, nMOS, trans version

(inputs and outputs on opposite sides of the PLA). Clocked **inputs** and outputs are supported.

It is easy to create a template for a new style of PLA, and tpla(CAD5) has information on how to do it. If you develop a particularly nice template and would like to share it, send it to "mayo@Berkeley" or "ucbvax!mayo".

OPTIONS

-I Clock the inputs to the PLA, if this feature is supported for this style.

-O Clock the outputs to the PLA, if this feature is supported for this style.

-G Insert an extra ground line every *num* rows in the AND plane and every *num* columns in the OR plane. This defaults to whatever is appropriate for the corresponding **nMOS** PLA.

-S Stretch power and ground lines by *num* lambda. This defaults to whatever is appropriate for the corresponding **nMOS** PLA.

-v Be verbose, and show (in the Caesar output) how the PLA was constructed from its basic components.

-V Be verbose, and print out information about what tpla is doing. This option implies **-v**.

-a produce Caesar format (this is the default)

-c produce CIF format

-o The next argument is taken to be the base name of the output file. The default is the input file name with any extensions removed. If the input comes from the standard input and the **-o** option is not specified then the output will go to the standard output.

-s The next argument specifies the style of PLA to generate. (This causes tpla to use the file **~cad/lib/tpla/p-***style***.tp** as its template).

-l Set lambda to *num* centimicrons. (200 is the default)

-t The next argument specifies the template to use, this normally defaults to the standard library. A **.tp** suffix is added if no suffix was specified. This option is useful for generating styles of PLAs that are not included in the standard library.

input_file
The file containing the truth_table. If this filename is omitted then the input is taken from the standard input (such as a pipe).

FILES

~cad/bin/tpla	– executable
~cad/src/tpla/*	– source
~cad/lib/tpla/p*.tp	– standard templates for PLAs

SEE ALSO

eqntott(CAD1), presto(CAD1), plasort(CAD1), pla(CAD5), tpla(CAD5), tpack(CAD3), mkpla(CAD1)

AUTHOR

Robert N. Mayo, program and the Bcis, Btrans, Mcis, Mtrans, Tcis, and Ttrans templates.

CD4cis, CS3cis, and CS3trans templates by Grace H. Mah

BUGS

The -G and -S options have no way of knowing what the grounding requirements are for the style of PLA actually being generated.

This program inherits any bugs that may exist in tpack(CAD3).