MADMACS: An environment for the layout of regular arrays

Eric Gautrin, Laurent Perraudeau

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

MADMACS : un environnement pour le dessin de masques de circuits réguliers

Eric Gautrin and Laurent Perraudeau

N° 1979

Avril 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

INRIA
RENNES

MADMACS : un environnement pour le dessin de masques de circuits réguliers

Eric Gautrin* and Laurent Perraudeau*

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet Api

Rapport de recherche n ° 1979 — Avril 1993 — 15 pages

Résumé : Ce papier présente un outil pour la réalisation automatique de dessins de masques pour circuits réguliers qui consistent en des cellules interconnectées avec leurs voisines. Puisqu'il existe une large gamme de circuits réguliers, un générateur unique n'est pas réalisable. Le système MADMACS est un environnement de développement pour des générateurs spécifiques dessinant de telles structures. Ces générateurs sont développés en utilisant le langage LISP. L'utilisation d'un langage de programmation permet une grande flexibilité dans la description d'un générateur pour un type particulier de structure. Un éditeur graphique est fortement couplé avec l'interpréteur LISP offrant un haut degré d'interactivité. Grâce à son mécanisme de macro commandes, on peut développer interactivement de nouvelles fonctions pour les tâches répétitives telle que le routage, et les utiliser dans les générateurs. De plus, des générateurs indépendants de la technologie peuvent être développés par l'utilisation de fonctions de MADMACS indépendantes de la taille des objets.
Mots-clé : CAO,VLSI

(Abstract: pto)

To be published in book on "Synthesis for control dominated circuits"

*{Eric.Gautrin}{Laurent.Perraudeau}@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

MADMACS : an environment for the layout of regular arrays

Abstract: This paper presents a tool for the automatic layout assembly of regular arrays which consist of cells interconnected with nearest neighbors. As there is a wide range of regular arrays, a single generator is not feasible. The MADMACS system is a development environment for designer specific generators of such structures. These generators are developed using the LISP language. The use of a programming language allows great flexibility in the description of a generator for a specific kind of structure. A graphic editor is tightly coupled with the LISP interpreter giving a high degree of interactivity. With its macro command mechanism, one can interactively develop new functions for repetitive tasks such as routing and use them in generators. Moreover, technology independent generators can be developed by using MADMACS object size independent functions.
Key-words: CAD, VLSI

1 Introduction

Many signal or image processing algorithms demand regular computations which can be implemented on massively parallel architectures such as systolic arrays. With the increasing density of integrated circuits, ASIC implementation of these algorithms becomes an evermore attractive solution. Kung [9] shows that a regular processor array simplifies VLSI integration: processors are identical, and interconnections are restricted to nearest neighbors. In recent years, much work has been devoted to automatic techniques for designing regular algorithms. For example, using the ALPHA DU CENTAUR[5], which is based on the recurrence equation formalism, a gate level description of a regular architecture can be derived from a system of recurrence equations through formal transformations. In this paper, we consider the problem of generating layouts from such descriptions.

Many circuit compilers have been presented in the recent literature, some of which are available as commercial products.

- Standard cell compilers are the most common systems available today and can produce compact layouts for random logic. In [4], an experience with the standard cell generation of a systolic circuit is described. The layouts produced by these compilers have been shown to be very much larger compared to designs made by hand. These tools break the natural topology to fit the floorplan of cell rows, and in so doing loose the regularity in placement and routing.

- Datapath compilers produce very dense layouts because they make use of the regularity inherent in datapath circuits [6, 12, 15, 18]. Generally speaking, such compilers lay out the different elements of the datapath in one dimension. Thus, these tools can efficiently produce linear arrays. As regular arrays can be composed of many processors, datapath compilation results in large aspect ratios. We do not know of a compiler which is able to improve the aspect ratio; this ratio improvement remains a manual task. Moreover, the linear floorplan used by the datapath compilers is not well-suited for bidimensional or triangular topologies.

- Array makers generate customized regular structures such as RAM, PLA[1], etc. These tools are based on cell tiling according to a user-defined template. This approach was used to generate a systolic spelling checker chip[7]. Many parts of this layout can be generated by a simple tiling but it includes some routing as well. To

be compatible with a tiling strategy, this routing must be embedded in the cells themselves or in dedicated routing cells. As simple routing must be generally broken into several cells to meet parameterization purpose, this results in a complicated template involving a large number of cells.

In fact, all of these compilers usually employ a unique placement and wiring strategy. They produce dense layouts only if this strategy fits the circuit topology[8]. We are interested in a wide class of regular array topologies which consist of active elements (processors, memories, latches, etc.) interconnected with neighbors. Past experiences in the design of regular arrays with available compilers have shown poor results. To produce compact layouts of regular arrays, the generation must be done in two steps:

Active element generation : the layout of active elements is generated by classical compilers. However, this generation must be constrained in order to retain the routing regularity at the array level.

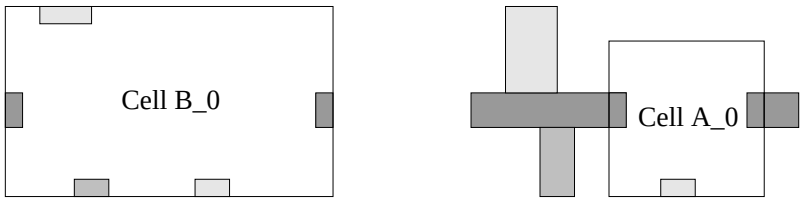
Array assembly : the array is then produced by active element tiling and regular routing to interconnect them.

There is a need to develop a circuit generator for the assembly of these structures. However, since there is a wide range of possible array structures (linear, bidimensional, triangular, etc.), it is not practical to define a single compiler for all conceivable array structures. The alternative we have chosen is to develop array specific generators. As a result, procedural layout systems [3, 10, 11, 14] are the best approach. A generator is then a program in which a designer can apply any place and route strategy. However, these systems have a poor human interface. It can be very difficult to give a textual representation of some simple graphical structures and there is poor interaction between a text editing session and the visualization of the resulting design. Some investigations [2, 13, 17] have been made regarding the use of a programming language tightly coupled with an interactive graphics editor. MADMACS is such a tool.

This paper gives an overview of the MADMACS system. The basic concepts are presented in the next section. A linear systolic array generator is developed in section 3.

2 MADMACS

The MADMACS system is a development environment to create designer specific generators for the assembly of regular arrays. MADMACS has four major components:



a) Bounding box of Cell B b) Contents of Cell B

Figure 1: Different types of objects

- LISP *interpreter*: A programming language is indispensable as a base upon which to develop specific generators. This approach gives much flexibility. The LISP language allows the use of parameters and conditional operations so that a generator can produce an entire class of regular arrays.

- *Friendly graphical front end*: Programming languages are known for their lack of interactivity. A graphics editor is tightly coupled with the LISP interpreter. Graphics system commands are transcribed into LISP functions. In our approach, a designer uses the editor to update the design data base or to interactively develop new functions which can be used in a generator.

- *Technology independence*: To be efficient, a generator must be technology independent. MADMACS features coordinate free commands or design functions. Using these object size independent operations rather than absolute coordinate operations, a designer can produce symbolic layouts and define technology independent generators.

- *Data base manager*: Three types of objects are used in the MADMACS system.

  A *figure* is the bounding box of a named cell. It contains a collection of rectangles and/or instances of figures. These objects are used to manage the hierarchy.

  A *connector* is associated with a figure and represents one input/output of the figure. A connector is always on the figure edges and has a color, a name and a size.

  A *rectangle* is a piece of material on a layer. A rectangle has a color, a size and can be named.

  Figure 1 illustrates the different object types. Creation, deletion and modification operations on these objects are processed by the database manager written in C++. Object oriented languages[19] are well suited for object manipulation such as MADMACS operations, and for developing new commands and reusable code

In the next subsections, the three first points are developed.

2.1 LISP interpreter

MADMACS is a design tool based on LISP language extended with some specific functions for layout design, such as *(create-rect)* or *(make-instance)*. As the interpreter evaluates a design function, it calls the MADMACS database manager which executes the associated operation and returns an execution status. For instance, *(make-instance F)* returns *t* if the figure *F* exists, *nil* otherwise.

For consistency with the graphical front-end, many design functions use some data base manager variables such as *cursor*: the current position in the design, *current-mark*: a stored position, or *current-object*: the object pointed to by *cursor*. For example, the *(make-instance)* design function uses the *cursor* variable to set the position of the top left corner of the figure.

Like other programming languages, a designer can define powerful design functions using parameters and conditional generation instructions. For example, the following function generates a column of cells by abutment. In this *(abut)* function, the column is generated recursively. The *(cursor)* function modifies the current position in the design.

```
(defun abut (name nb xpos ysize)
  (COND ( (≤ nb 0) nil)                          ; test if finish
        ( t (cursor xpos (* ysize nb))           ; update the current position
            (make-instance name)                 ; insert a figure
            (abut name (- nb 1) xpos ysize))))   ; recursive call
```

In the above function, the execution status of the *(make-instance)* design function is not used. This function returns *t* only if the figure exists. In the following version of *(abut)*, the column generation is stopped if the figure does not exist:

```
(defun abut2 (name nb xpos ysize)
  (COND ( (≤ nb 0) nil)
        ( t  (cursor xpos (* ysize nb))
             ( COND ((make-instance name) (abut2 name (- nb 1) xpos ysize))
                    (t nil)))))
```

The above functions can be reused in different array assembly generators. The MADMACS system comes with a library of such design functions.

2.2 Graphical front-end

The MADMACS system provides the designer with a graphic editor tightly coupled with the LISP interpreter. Every editor command has its functional form in LISP: cursor movements, object manipulations, zoom facilities, etc. When an editor command is issued, the associated design function is evaluated by the LISP interpreter which activates the database manager. The database manager is also responsible for updating the current drawing after processing commands issued to the LISP interpreter or to the editor. Mixing editor commands and design functions, a designer can easily lay out structures difficult to simply describe by a textual representation alone.

MADMACS features a graphics variable (or mark) mechanism. A designer will frequently return the cursor to a previous location. The coordinates of this position are identified by a named graphic variable. MADMACS provides commands for direct cursor movement such as *(goto-mark "A")*, or cursor alignment commands to a graphic variable such as *(mark-horizontal-alignment "A")*. These commands are particularly useful for laying out a wire with manhattan angles.

Like many systems, MADMACS can memorize a sequence of commands called a macro-command (macro for short). To build a macro, the user enters the *Macro Learning* mode. The system executes and stores each editor command or LISP function issued. The macro construction is finished when the user leaves the *Macro Learning* mode. A macro is saved as a LISP function with a user specified name and is callable by that name. With the addition of parameters and conditional instructions, this macro/function can be generalized, and used as part of a generator.

Macros are mainly used for repetitive tasks where the same sequence has to be executed several times. For instance, two sets of connectors must be connected by a bus as shown in figure 2.1. In this example, the connectors have the same size and are evenly spaced. The first connectors are pointed to by the two graphic variables A and B. First, a rectangle is created between the connector pointed to by A and the horizontal alignment with B (see figure 2.2):

```
(goto-mark "A")                 ; snap current position to "A"
(current-mark)                  ; store the current position
(cursor-down 2)                 ; update current position to take connector size
(mark-horizontal-alignment "B") ; alignment with "B"
(create-rect)                   ; create the horizontal rectangle
```

Secondly, a rectangle is created between the new current position and the connector pointed to by B (see figure 2.3):

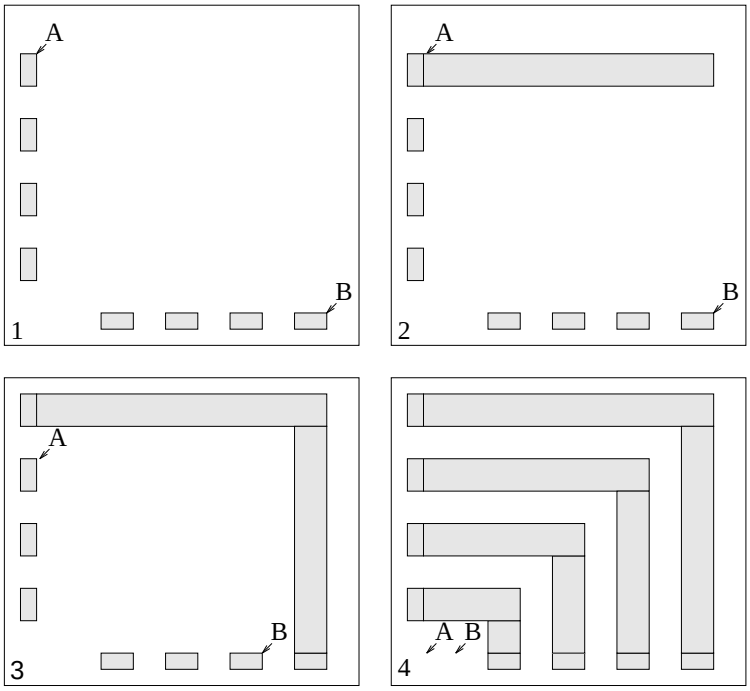


Figure 2: Macro execution

```
(current-mark)          ; store the current position
(goto-mark "B")         ; snap current position to "B"
(cursor-left 2)         ; update current position to take connector size
(create-rect)           ; create the vertical rectangle
```

Finally, the graphic variables are moved to the next connectors to be wired (see figure 2.3):

```
(goto-mark "A")         ; snap current position to "A"
(cursor-down 4)         ; update current position to below connector
(put-mark "A")          ; store current position in "A"
(goto-mark "B")         ; snap current position to "B"
(cursor-left 4)         ; update current position to left connector
(put-mark "B")          ; store current position in "B"
```

The new context is similar to the original one, so the macro can be executed again. Figure 2.4 illustrates a multiple execution. This macro can be applied in other parts of the design

as long as the context remains similar to the context of the definition. However this macro depends strongly on the exact sizes and spacings of the connectors, by using coordinate free functions, the macro could be even more general.

2.3 Coordinate free functions

Coordinate free functions allow procedures to adapt to context and to define technology independent generators. MADMACS offers facilities for graphic cursor movement and *cursor* variable updating: displacement of n microns in any direction; snap to absolute coordinates. In addition, the system provides logical and coordinate free cursor movements functions such as:

- move to a edge of the current object, such as *(cursor-left-in-cell)*,
- move to a neighboring cell, such as *(cursor-to-upper-cell)*,
- move to a connector of a figure, such as *(connector-up)*.

These functions are size-independent and rely on the local context of the layout around the cursor location. Therefore, they allow the designer to ignore the absolute coordinates of layout geometries and to define size-free command sequences. For instance, by using coordinate free functions rather than absolute cursor movements, the macro *(busconnect)* defined previously is rewritten:

```
(defun busconnect ( )                    (defun busconnect2 ( )
  (goto-mark "A")                          (goto-mark "A")
  (current-mark)                           (current-mark)
  (cursor-down 2)                  ⇒       (connector-nextside)
  (mark-horizontal-alignment "B")          (mark-horizontal-alignment "B")
  (create-rect)                            (create-rect)
  (current-mark)                           (current-mark)
  (goto-mark "B")                          (goto-mark "B")
  (cursor-left 2)                  ⇒       (connector-nextside)
  (create-rect)                            (create-rect)
  (goto-mark "A")                          (goto-mark "A")
  (cursor-down 4)                  ⇒       (connector-down)
  (put-mark "A")                           (put-mark "A")
  (goto-mark "B")                          (goto-mark "B")
  (cursor-left 4)                  ⇒       (connector-left)
  (put-mark "B"))                          (put-mark "B"))
```

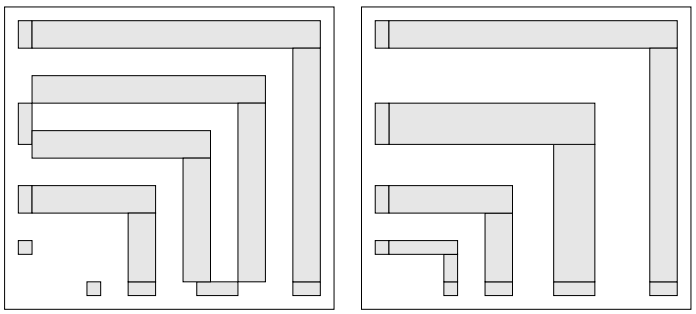


Figure 3: Bus between connectors of different sizes and spacings.
Left: execution of (busconnect). Right: execution of (busconnect2)

As shown in figure 3, the *(busconnect2)* macro can be applied in more contexts than *(busconnect)* itself. By using coordinate free functions, the function *(abut2)* can be simplified. Now, the exact figure size is not a parameter and the *cursor* variable is used implicitly.

```
(defun abut3 (name nb)
  (COND ( (≤ nb 0) nil)
        ( t  (cursor-down-in-cell)
             ( COND ((make-instance name) (abut3 name (- nb 1)))
                    (t nil)))))
```

These two examples show the importance of the coordinate free functions. They allow the user to design a fully technology independent generator. Generally, a technology file is read by the generator. This file contains the definition of a few variables describing the minimal width and spacing of routing layers.

3 A linear array generator

In this section, a linear array generator is described. In order to achieve a better aspect ratio for the layout, the array is broken into several columns of processors.

3.1 Linear Array Topology

Before designing the generator itself, the first task is to define the topology of the structure to be laid out. Every placement and routing problem must be planned in order to determine

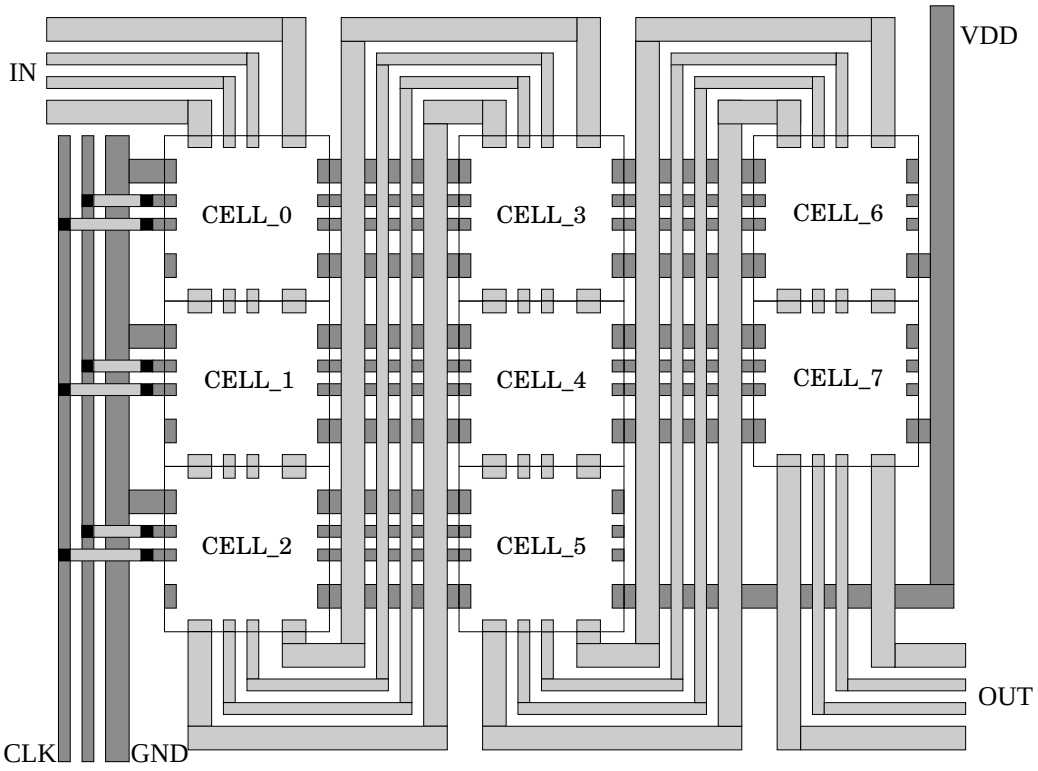


Figure 4: Linear array topology

a place and route strategy for the generator (layer selection, control versus data, power routing, etc.).

In our example, the processor cell is generated automatically by a classical circuit compiler under the following constraints: control signals run horizontally through cells in metal 1 layer, such that a direct interconenction to the next column would be possible; data signals run vertically across cells in metal 2 layer for vertical abutment. The generator then considers a processor cell as a figure with connectors on each side: control connectors on the sides and data connectors on the top and bottom.

The general topology is presented in figure 4. Processor cells are arrayed in columns, but the last one can have fewer cells. The interconnections are laid out using two layers: metal 2 layer is preferred for data signals whereas metal 1 for control signals. The routing remains regular and does not require vias except for the global control distribution on the left of the array.

3.2 The generator

The generator skeleton consists of four main functions corresponding to the place and route strategy:

```
(defun genarray (nbcell nbrow)
  (celltiling nbcell nbrow)      ; 1- processor cell tiling
  (controldistribution)          ; 2- control distribution through columns
  (globalcontrol)                ; 3- global control distribution to first and last columns
  (datarouting))                 ; 4- data signal routing between columns
```

The linear array generator uses only two parameters defining the aspect ratio of the array: *nbcell* is the total number of processors, *nbrow* is the number of rows. In the next paragraphs, these functions are detailed.

Processor cell tiling The *(celltiling)* function uses the *(abut3)* function presented in section 2.3, to generate the different columns.

```
(defun celltiling (nbcell nbrow)
    (COND ( (≤ 0 (- nbcell nbrow))                ; test if last column
              (abut3 "processor" nbcell))         ; last column generation
          ( t (abut3 "processor" nbrow)           ; column generation
              (go-up-column)                      ; cursor snaps to the column top
              (spacing-column 0 0)                ; compute the column spacing
              (celltiling (- nbcell nbrow) nbrow))))  ; recursive call
```

The *(go-up-column)* function just snaps the current position to the top of the column. It uses some coordinate free functions to ignore the exact size of the processor cell. The *(spacing-column)* function computes recursively the number of data connectors and the sum of their sizes, then snaps the current position to the top of the next new column. The *(connector-right)* design function returns *t* if there is a connector on the right, *nil* otherwise.

```
; current position at the top left of a column
(defun spacing-column (nb size)
    (COND ( (connector-right)
              (spacing-column (+ nb 1) (+ size (connector-size))))
          ( t (cursor-right-in-cell)
              (cursor-right (+ size (* nb spacing-cme2))))))
```

These functions are independent of the processor cell size and connectors. They can be applied with any processor cell meeting the original specifications.

Control distribution Processor cell constraint and tiling guarantee that a control connector and its opposite on next column are on the same line and have the same size. Control distribution is achieved by two functions: one to successively snap the cursor on the each columns and a second to generate the horizontal wires.

```
; current position at the top of first column
(defun controldistribution ()
  (COND ((cursor-to-right-cell)       ; test if last column
         (horizontal-wire)            ; current column wiring
         (controldistribution))       ; recursive call
        (t nil)))

(defun horizontal-wire ()
  (COND ((connector-down)             ; test if last connector of the column
         (current-mark)               ; store the current position
         (connector-nextside)         ; take the connector size
         (cursor-to-left-cell)        ; current position to left column
         (create-rect)                ; create connection
         (cursor-to-right-cell)       ; current position to right column
         (horizontal-wire))           ; recursive call
        (t  (go-up-column))))         ; snap to the column top and return
```

Some control signals can be wider than others, for instance *GND, VDD*, etc. The *(connector-nextside)* function takes the connector size into consideration allowing the creation of a wire with same width.

Global control connection: First, the generator creates a vertical line for each control signal. *GND* and *VDD* lines are laid out by default. For the others, the connector names are used. For each different signal, a single vertical line is produced and a graphic variable is associated. This graphic variable has the same name as the connector. To avoid multiple creations for the same signal name, the *(vertical-wire)* function tests a graphic variable to see if a vertical line has already been created for that signal name.

```
; current position at the top of first column
(defun vertical-wire ()
  (COND ((connector-down)
         (COND ((= "VDD" (connector-name)) (vertical-wire))
               ((= "GND" (connector-name)) (vertical-wire))
               ((exist-mark (connector-name)) (vertical-wire))
               (t      (create-vertical-wire (connector-name))
                       (vertical-wire))))
         (t nil)))
```

Finally, the connections to vertical lines are created. Three different cases can happen: a connection to the *GND* line is direct, a connection to the *VDD* line is direct but on the right side, a connection to any other control signal needs vias to bypass. For sake of simplicity, the *(global-connection)* function is not fully detailed:

```
; current position at the top of first column
(defun global-connection ()
  (COND ((connector-down)
         (COND ((= "VDD" (connector-name))
                       (right-direct-connection)
                       (global-connection))
               ((= "GND" (connector-name))
                       (left-direct-connection)
                       (global-connection))
               (t      (left-connection-with-via)
                       (global-connection))))
         (t nil)))
```

Data signal routing There are three different cases for data signal routing (see figure 4): from the design top left to the first column, between the columns, and from the last column to the design bottom right. One specific function for each case is written. They are all extensions of the *(busconnect2)* macro presented in section 2.3. This routing is done in metal 2 layer, respecting minimal metal 2 spacing and the connector size.

Conclusion: By using coordinate free functions, a fully technology independent generator has been written. As long as the processor cell respects the connector placement constraints, this generator can be applied with any processor cell. The first version of this generator was written in a few hours. It takes less than 10 seconds for the generator to

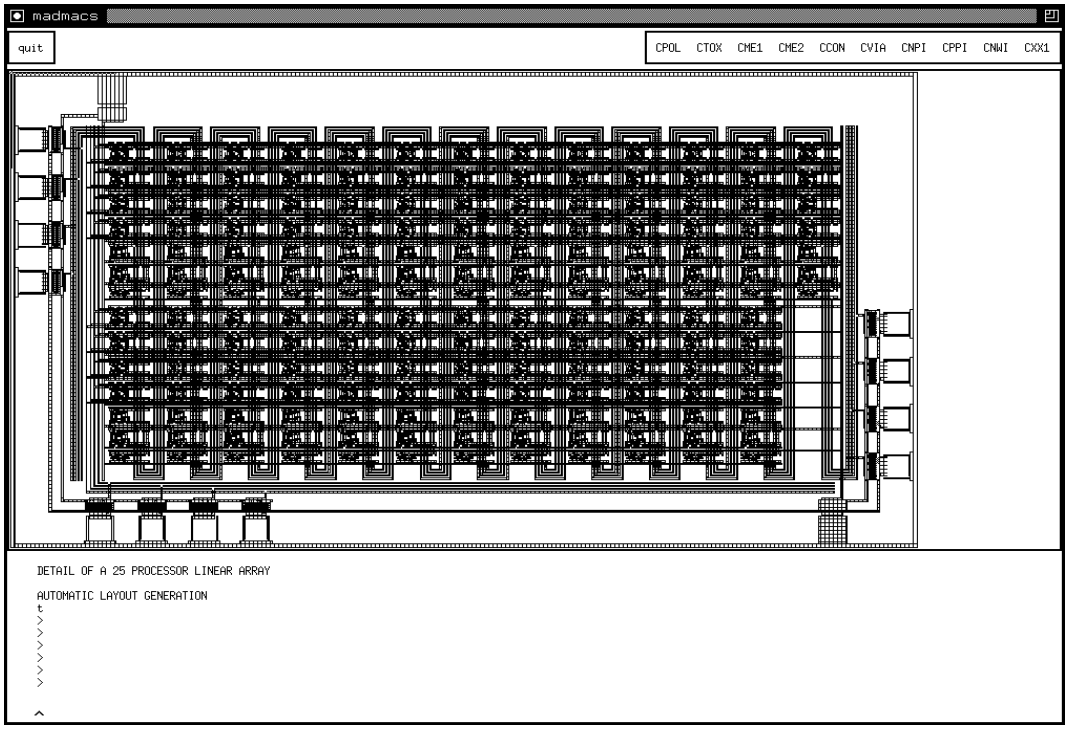


Figure 5: MADMACS interface

produce a design with 32 cells. So, the designer can try different *nbrow* values to find the best aspect ratio.

Figure 5 shows the MADMACS interface. The *graphics window* displays a part of a design produced by this linear array generator. Through the *textual window* the designer can interact with the LISP interpreter.

4 Conclusion

In this paper, we have described a development environment in which to create designer specific generators for the assembly of regular arrays. The system combines programming and graphics facilities. On the one hand, the LISP language gives the flexibility needed to develop a designer specific generator with a particular place and route strategy. On the other hand, the graphical front-end gives a high degree of interactivity. The macro command mechanism memorizes sequences of graphic editor commands and/or LISP functions.

Saved in their LISP form, macros are available as new design functions. Generalized by addition of parameters and conditional generation instructions, these macros can be made coordinate free and used in a generator. Indeed, it is a good method for capturing the designer's floor plan methodology. An important MADMACS feature is the size-independent design functions. The designer can work in a logical way, produce symbolic layouts and define fully technology independent generators.

This approach has several advantages. The ability to define size-independent functions is particularly important: the designer can define libraries of design functions, and use them to create new generators with less effort. A generator can be executed in a few seconds, so the designer can try different parameter values to produce the best design aspect ratio.

References

[1] R. Ayres. Silicon Compilation: A Hierarchical Use of PLAs. In *16th Design Automation Conference*, pp 314-326, June 1979.

[2] J. Batali, N. Mayle, H. Shrobe, G. Sussman, D. Weisse. The DPL/Daedalus Design Environment. In *VLSI'81*, John P. Gray ed., Academic Press, pp 182-193, 1981.

[3] JM. Berge, L. O. Donzelle, J. Rouillard, D. Rouquier. LOF. Technical note, CNET-FRANCE, 1985.

[4] C. Dezan, E. Gautrin, H. Le Verge, and P. Quinton. Synthesis of systolic arrays by equation transformations. In *ASAP 91*, Barcelone, Spain, September 1991.

[5] C. Dezan, H. Le Verge, P. Quinton and Y. Saouter. The ALPHA DU CENTAUR environment. In P. Quinton and Y. Robert, editors, *International Workshop Algorithms and Parallel VLSI Architectures II*, North-Holland, Bonas, France, June 1991.

[6] P. Drenth and C. Strolenberg. Datapath layout generation with in-the-cell routing and optimal column resequencing. In *Euro ASIC' 91*, pp 373-376, IEEE, May 1991.

[7] P. Frison and D. Lavenier. A fully integrated systolic spelling co-processor. In *VLSI91: International Conference on Very Large Scale Integration*, August 1991.

[8] D. Johansen. Silicon Compilation. *Advanced Research In VLSI,* Proceedings of the Decennial Caltech Conference on VLSI, March 1989.

[9] H.T. Kung. Why systolic architectures? *IEEE Computer*, Vol 15, pp 37, 1982.

[10] J. A. Lewis, A. A. Berlin, A. J. Kuchinsky, P. K. Yip. Integrated Circuit Procedural Language. Hewlett-Packard Journal, pp 4-10, June 1986.

[11] R.J. Lipton and al. ALI: a Procedural Language to Describe VLSI Layouts. In *19th Design Automation Conference*, pp 467-474, June 1982.

[12] T. Mashburn and al. Datapath: a CMOS data path silicon assembler. In *23rd Design Automation Conference*, pp 722-729, 1986.

[13] R.N. Mayo, J.K. Ousterhout. Pictures with Parentheses: Combining Graphics and Procedures in a VLSI Layout Tool. In *20th Design Automation Conference*, pp 270-276, June 1983.

[14] S. Sastry, S. Klein. Plates: a Metric-Free VLSI Layout Language. In *Proc Conf on Advanced Research in VLSI*, Cambridge, Massachusetts, pp 165-174, January 1982.

[15] H.E. Shrobe. The datapath generator. In *CompCon82 High Technology in the Information Industry*, pp 340-344, IEEE Computer Society, 1982.

[16] G. Steele. Common Lisp: The language. Digital Press, 1984.

[17] S. Trimberger. Combining Graphics and a Layout Language in a Simple Interactive System. In *18th Design Automation Conference*, Nashville, pp 234-239, June 1981.

[18] VLSI Technology Inc. *Datapath Compiler*. Technical Report, VLSI Technology Inc., San Jose, CA, USA, June 1989.

[19] W. Wolf. Object-Oriented Programming for CAD. In *IEEE Design and Test of Computers*, pp 35-42, March 1991.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399