Layout generation of bidimentional regular arrays in the MADMACS environment

Eric Gautrin, Laurent Perraudeau, Oumarou Sié

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Layout generation of bidimensional regular arrays in the MADMACS environment

Eric Gautrin , Laurent Perraudeau and Oumarou Sié

N° 1980

Avril 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

Rapport de recherche

1993

ISSN 0249-6399

Layout generation of bidimensional regular arrays in the MADMACS environment

Eric Gautrin* , Laurent Perraudeau* and Oumarou Sié*

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

Rapport de recherche n ° 1980 — Avril 1993 — 11 pages

Abstract: This paper presents an approach for the automatic layout generation of regular arrays. These arrays consist of processor 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 customizing the 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 object size independent functions in MADMACS. The MADMACS environment is illustrated with a generator for bidimensional regular array layout.

Key-words: CAD, VLSI

(Résumé : tsvp)

Submitted to VLSI 93

*{Eric.Gautrin}{Laurent.Perraudeau}{Oumarou.Sie}@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

Génération de dessins de masques pour circuits réguliers bidimensionnels dans l'environnement MADMACS

Résumé : Ce papier présente une approche pour la génération automatique des masques de circuits réguliers. Ces circuits sont constitués d'un tableau de processeurs et d'interconnexions entre processeurs voisins. Puisqu'il existe de nombreux types de circuits réguliers, un générateur global n'est pas réalisable. Le système MADMACS est un environnement de développement pour la réalisation de générateurs 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 de structure donné. Un éditeur graphique est fortement couplé avec l'interpréteur LISP pour permettre une importante interactivité. Avec son mécanisme de macro commandes, on peut développer interactivement de nouvelles fonctions pour les tâches répétitives telles que le routage et les utiliser dans les générateurs. De plus, des générateurs indépendants de la technologie peuvent être produits en utilisant des fonctions indépendantes de la taille des objets. L'environnement MADMACS est présenté avec un générateur pour le dessin de circuits réguliers bidimensionnels.

Mots-clé : CAO, 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 [7] 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, the ALPHA DU CENTAUR[3], is based on the recurrence equation formalism. Through formal transformations such as time and space allocation or signal pipelining, a system of recurrence equations is derived into a realistic architecture. This description contains all information for layout generation:

- the size of the array,
- a gate level description of the processors,
- a description of processor interconnections.

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. The layouts produced by these compilers have been shown to be very much larger compared to designs made by hand[3]. These tools break the natural topology to fit the floorplan of cell rows, and in doing so, lose the regularity in placement and routing.
- Datapath compilers produce very dense layouts because they make use of the regularity inherent in datapath circuits [9]. Generally speaking, such compilers lay out the different elements of the datapath in one dimension. This linear floorplan is not well-suited for bidimensional topologies.
- Array makers generate customized regular structures such as RAM, PLA, etc. These tools are based on cell tiling according to a user-defined

template. To be compatible with a tiling strategy, processor interconnections 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[6]. To produce compact layouts of regular arrays, the generation must be done in two steps:

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

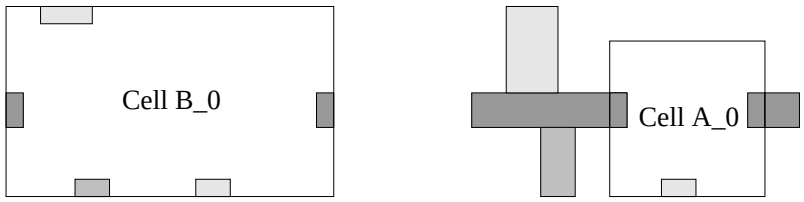
Array assembly : the array is then produced by processor tiling and regular routing to interconnect them.

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 [2, 8] 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 [1] have been made regarding the use of a programming language tightly coupled with an interactive graphics editor. MADMACS is such a tool.

The next section gives a short presentation of the MADMACS system[4]. The following sections focus on layout generation of bidimensional regular arrays.

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

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 or modification operations on these objects and update of the current drawing are processed by the database manager written in C++. Object oriented languages are well suited for object manipulation such as MADMACS operations, and for developing new commands and reusable code[10].

LISP interpreter: A programming language is indispensable as a base upon which to develop specific generators. This approach gives much flexibility. We choose the LISP language extended with some specific functions for layout design, such as *(create-rect)* or *(make-instance "cell")*. 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 "cell")* returns *t* if *cell* exists, *nil* otherwise.

In its simplest form, a circuit generator is a command script. The LISP language allows the use of parameters, conditional operations and supplies functions to test the execution context, for instance *(cell-on-the-right?)*, so that a generator can produce an entire class of regular arrays, or a function

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

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: cursor movements, object manipulations, etc. When an editor command is issued, the associated design function is evaluated by LISP. Mixing editor commands and design functions, a designer can easily lay out structures difficult to simply describe by a textual representation alone.

Like many systems, MADMACS can memorize interactively 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. Macros are mainly used for repetitive tasks where the same sequence has to be executed several times. With the addition of parameters and conditional instructions, this macro/function can be generalized, and used as part of a generator.

Technology independence: To be efficient, a generator must be technology independent. MADMACS offers facilities for graphic cursor movement: displacement of n microns in any direction; snapping to absolute coordinates, etc. In addition, the system provides logical and coordinate free cursor movements functions such as:

- move to an 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-independent macros or functions. For instance, two sets of connectors must be connected by a bus as shown in figure 2. In this example, the connectors have different size and are unevenly spaced. The idea is to develop a macro to perform this repetitive task. Using only absolute cursor movements, it is difficult to

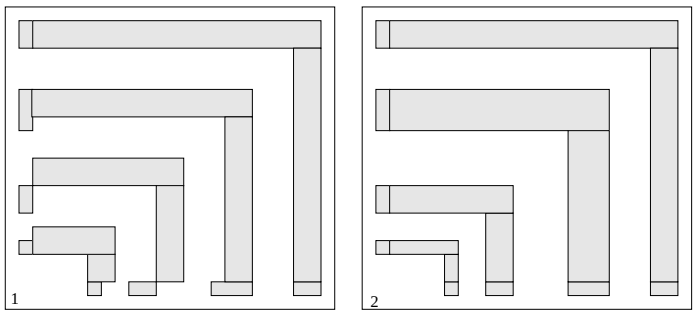


Figure 2: Bus between connectors of different sizes and spacings.

adapt the macro to the different connector sizes or spacings. Probably, the bus connection will be erroneous as shown in figure 2.1. With coordinate free cursor movements, such as *(cursor-to-left-cell)* to move the cursor from an horizontal connector to the next one, the macro is adapted to different size and spacing as shown in figure 2.2.

This example shows the importance of the coordinate free functions. With the addition of a technology file containing the definition of a few variables describing the minimal width and spacing of routing layers, a designer can produce symbolic layouts and define fully technology independent generators. An example of such a generator is detailed in the next section.

3 A bidimensional array generator

In this section, a bidimensional processor cell array generator is described. As input, it takes three parameters:

- the netlist of a processor cell assuming that all processors in the array are identical. For the sake of simplicity, only bit-level processors are considered, so that a processor is laid out with standard cells. The use of a datapath generator is under investigation.
- a description of processor interconnections.
- the size of the array.

Array Topology
Before designing the generator itself, the first task is to define the topology

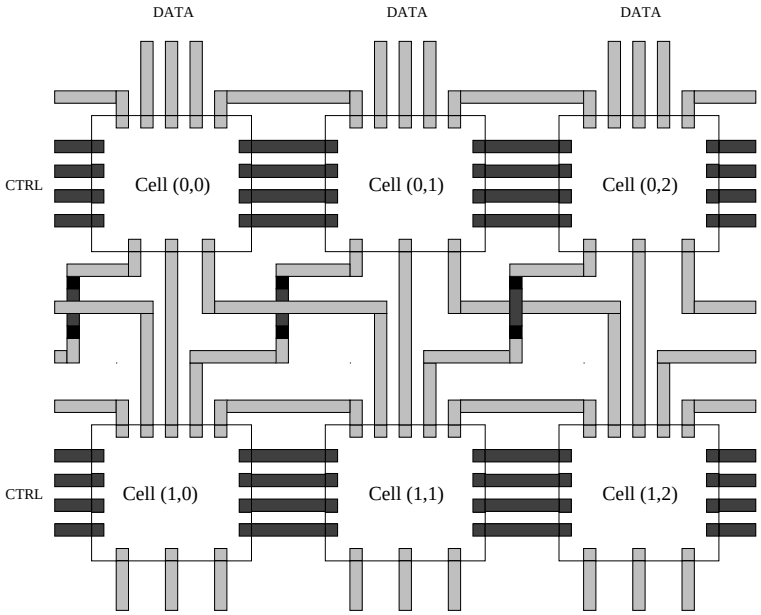


Figure 3: Array topology

of the structure to be laid out. Every placement and routing problem must be solved in order to determine a place and route strategy for the generator (layer selection, control versus data, power routing, etc.).

The general topology is presented in figure 3. The interconnections are laid out using two routing layers: metal 1 layer is preferred for data signals whereas metal 2 for control signals. This choice comes from the standard cell approach where a signal is connected to a standard cell input/output in metal 2 layer.

Control signals run horizontally through processors in metal 2 layer, such that a direct interconnection to the next column would be possible. Data signals run between the processor rows. Horizontal and vertical connections are laid out using only metal 1 layer. In the case of connections in the two diagonal directions, one diagonal must be partly routed in metal 2 for crossing. To minimize the number of vias, the generator tests the number of connections in each direction, chooses the direction with fewer connections for crossing in metal 2 layer, and doing so, minimizes the number of vias.

To retain the regularity of the routing, a strict data connector ordering must be respected during the processor generation.

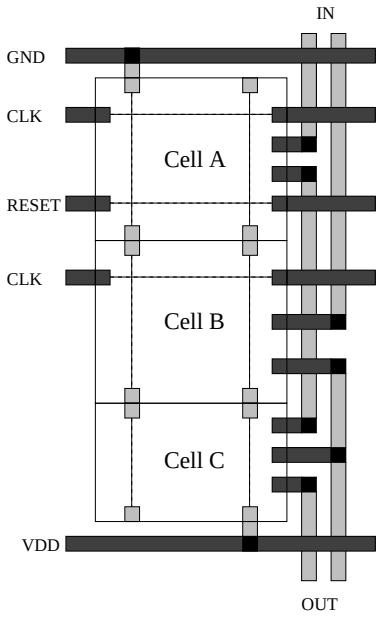


Figure 4: Processor floorplan

Processor Generation

Processor layout generation is based upon a standard cell library. This approach allows us to easily migrate from one technology to another. Standard cells are placed in a single column as shown in figure 4. Power and ground supplies are connected by cell abutment, and then routed horizontally such that a direct connection to the next column is possible. To the right of the cell column we use a channel for data signal routing. The control signals (clock, reset, etc.) run horizontally through standard cells and the data routing channel for direct connection to the next column.

The area of the data routing channel depends on the linear ordering of the processor cells. To minimize this area, a min-cut problem has to be solved, using the Gurari and Sudborough algorithm [5]. The processor netlist is transcribed in a graph which includes two special nodes. The *source* (respectively *sink*) node represents data connections to processor upper edge (respectively lower edge) and is placed at the top of the linear ordering (respectively at the bottom). Although this algorithm is time consuming, it is practical for our examples where the processors are simple and rarely exceed

a hundred cells. For a graph with 103 nodes and 113 nets, a linear ordering minimizing the min-cut is found in 57 seconds.

The next step consists of net assignment to the vertical routing channel tracks. In this part, the data connection ordering must be respected to preserve the regularity of the array routing. Then, the input parameters for the processor generator are produced.

Array Generator
The generator skeleton consists of 6 main LISP functions corresponding to the place and route strategy:

1. *(processor_generation)* takes as inputs the linear ordering and the net assignment.
2. *(processors_tiling)* takes as inputs the size of the array and a list of processor connections. The distances to allow routing space between columns and rows are evaluated according to the connections themselves and the technology variables.
3. *(ctrl_connection)* consists of a direct wiring between columns.
4. *(horizontal_connection)* is in fact a *bridge routing.*
5. *(diagonal_connection)* tests the number of connections in the two diagonal directions. First, it performs the diagonal connections without vias, then the next diagonal ones. Each case is in fact a *river routing.*
6. *(vertical_connection)* consists of a direct wiring between rows.

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 and any interconnections. It takes less than 10 seconds for the generator to lay out an array of 10 by 10 processors.

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

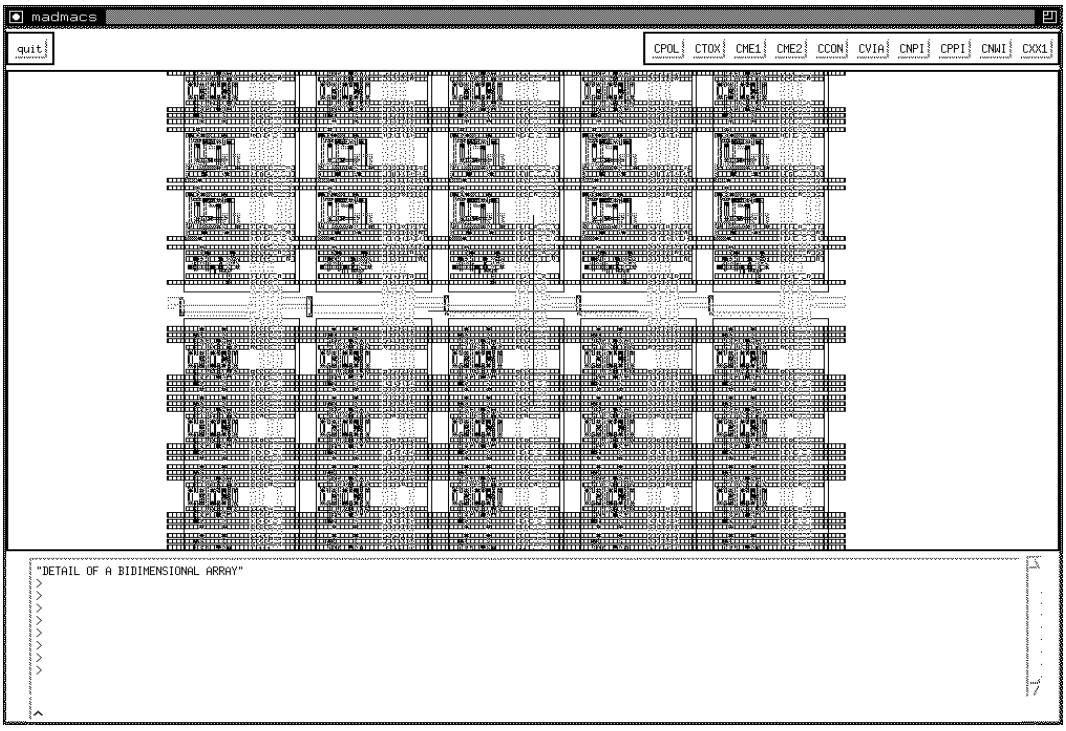


Figure 5: MADMACS interface

4 Conclusion

In the MADMACS environment, a designer creates his/her own 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 any generator with a particular place and route strategy. On the other hand, the graphical front-end gives a high degree of interactivity. In particular, the macro mechanism memorizes sequences of graphic editor commands and/or LISP functions. 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 paper presents a generator for bidimensional array layout. The generation is made in two steps. First, the processor is generated respecting the

array connection constraints to retain the regularity of the array. Then, the array is assembled where each connection follows a simple routing scheme: bridge, river or direct routing. Constraining the processor generation to retain the regularity through the hierarchy simplifies the array routing and also facilitates some analysis such as design rule checking or extraction.

References

[1] 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.

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

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

[4] E. Gautrin, L. Perraudeau. MADMACS: a tool for the layout of regular arrays. In *WG 10.5 IFIP Workshop on Synthesis, Generation and Portability of Library Blocks for ASIC Design*, Grenoble, France, March 1992.

[5] E. M. Gurari, I. H. Sudborough. Improved Dynamic Programming Algorithms for Bandwidth Minimization and the MinCut Linear Arrangement Problem. In *Journal of Algorithms*, Vol 5, pp 531-546, 1984.

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

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

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

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

[10] 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