REAL TIME GRAPHICAL SIMULATION OF SYSTOLIC ARRAYS

Hai Van Dinh Le
Marek A. Perkowski
Department of Electrical Engineering
Portland State University
P.O. Box 751
Portland, Oregon 97207
tel. (503) 464-3806 x 23.

ABSTRACT

This paper presents a new concept in visual simulation - Systolic Arrays Graphical Simulator, SAGS - that allows for the visualization of the data and control flow in various systolic arrays on the monitor screen. The systolic architecture is constructed interactively from various basic cells and arrays. The simulator has successfully been used to simulate several well-known and new architectures and has even proven itself to be useful in finding and correcting an error in a well publishized algorithm for general purpose matrix computations that is currently being realized in hardware and can have potential applications that require extreme reliability.

1. INTRODUCTION

Simulation techniques play an important role in design analysis, correctness verification, performance evaluation, and test generation for digital systems. Many simulators are widely available for the university and industrial user communities. Recently a new generation of simulators is becoming popular that permit the user to visualize the data flow, logic levels, or register contents in the form of animated schematics or timing diagrams [12], [15], [16], [17], [18], [19], [20], [21].

The arrival and wide acceptance of systolic processors [1], [2], [4], [5], [6], together with the difficulty involved in designing as well as the problems concerning the analysis of systolic algorithms, call for new kinds of CAD tools for their synthesis at many levels, optimization, verification, and simulation. It would be very advantageous if these tools were visual and combined the properties of the tools from [12], [15] - [21] with interactive editing ("schematic capture") and the modification of systolic architectures of various sizes, types of arrays, and cells linked in many configurations. Perhaps visual simulators will be the first tools of this type. Such a tool - the visual simulator of systolic arrays - is proposed below. The idea behind such a tool seems very natural and perhaps comes from the originator of systolic arrays, H. T Kung, who would overlap and shift several overhead transparencies to illustrate data flows in systolic arrays. With the exception of [11], however, we have not been able to identify such tools and we are also not aware of anyone using them in practical applications.

The classical method of displaying the simulation data and the results is totally useless for systolic arrays. A system designer must be able to observe the movement of every piece of data as it traverses through the array at all times to verify whether a given algorithm is correctly mapped into the corresponding array architecture. He must also be able to see the results from the operations performed on each piece of data by any of the cells. Furthermore, for debugging purposes, he must be able to look into the registers of every cell at any one time and see the values of all the control signals present in that cell. In short, he must have the most detailed view of the entire system that may consist of many arrays composed of many types of cells. This perspective must be available to him with the accuracy of single pulses during the entire simulation process.

To meet the above requirements, a new breed of simulator - a systolic array simulator - has been developed and built by us to aid a hardware or software designer in the task of designing, evaluating, verifying, debugging, and teaching systolic systems. It was deemed essential that this simulator should be graphics based, hence its name Systolic Arrays Graphical Simulator, or SAGS for short.

2. SYSTOLIC ARRAYS GRAPHICAL SIMULATOR

From the very beginning, SAGS was designed to simulate the systolic systems of any configuration. These configuration are specified to SAGS by means of *script files*. A script file contains all of the vital information about the system: its number of arrays, their types and sizes, the way they are linked together, and the microprograms used in each cell.

A script file also specifies when and where the input data and the control signals should be fed into, and the output data taken from, the system.

SAGS allows for systems with multiple input, control, and output data streams. Each input or control stream is stored into ASCII files prior to being accessed by SAGS. Similarly, SAGS outputs are written into ASCII files.

During the simulation run time, SAGS consistently and successively executes all of the steps involved in a problem, showing the results from each step on the screen. This is referred to as a *multistep mode of execution*. It can be stopped and restarted at any time. Alternatively, SAGS can single-step through the problem, allowing a more detailed inspection of the temporary results, which is referred to as a *single-stepped mode of execution*. Switching between these two modes can be easily accomplished at any time by typing the appropriate command in at the keyboard.

Visually, SAGS allows for all of the arrays of the system to be seen on the monitor screen as long as each array has a reasonable number of cells. Because the real estate of the monitor screen is limited, arrays can be *overlapped*. This means that one from the background can be brought into the foreground to be scrutinized at any time. In addition, individual arrays can be interactively positioned anywhere on the screen to match the system-schematic closely.

SAGS allows for an array to be viewed in two different ways: *a real perspective*, with the array and its cells appearing smaller and therefore containing less information, and *a full perspective* in which the cells show all their register contents. The perspective of an array can be specified in the script file, or changed at the run time. All visual changes made to a system configuration at run time can be recorded back to the script file for reuse. A *status bar* on the top of the screen displays additional information such as the current step number, the total execution time, and the array actually being selected. The user can measure an execution time for a cell from their microcodes. This time unit is used as a step which allows for the evaluation and the comparison of the speed of various algorithm variants. Other properties of secondary importance like, for example, the shape of the cells or the array color, are also implemented.

SAGS has been used for debugging, verification and performance evaluation of various algorithms. It aids in experimenting with new designs, for example, in comparing various systolic architectures for a systolic algorithm or in playing with different cell algorithms for the same architecture such as Gausian elimination and Givens rotations.

SAGS was developed on an IBM Personal Computer, running the MS DOS operating system. It was written in Turbo Pascal [13], a dialect of the Pascal programming language as described by Wirth and Jensen in [14]. Its current version limits the architecture to 15 arrays using the *Turbo Graphix Toolbox™*, but SAGS itself can handle an unlimited number of arrays depending on the memory available. Each cell can have two inputs and two outputs on each of its four sides. There is practically no restriction concerning the number of inputs and outputs from the architecture or with respect to the number of different types of cells and arrays. The SAGS source code and many sample script files are available in Appendix C to [9].

ISCAS '89

They are also in ASCII format on floppy disks and are available on request to interested parties. To produce an executable SAGS copy, two software packages are needed: a copy of the DOS-based *Turbo Pascal™* compiler (version 3.0) and a copy of the *Turbo Graphix Toolbox™* (version 1.07), both available from Borland International, Inc. Since SAGS is graphics-based, a video card with bit-mapped graphics capabilities is needed to run the program. Because computer graphics and simulations are floating-point intensive applications, the use of a numeric co-processor is highly recommended. For SAGS to take advantage of the numeric co-processor, it must be compiled using a compiler version that supports 8087 floating point mathematical routines.

3. ONE EXAMPLE OF SAGS APPLICATION

SAGS was created to help in verifying and debugging systolic processors for linear algebra applications, particularly for various implementations of Faddeev algorithm [3]. Many architectures, including those from [2], [4], [5], [6], [7] and [8] have been simulated in [9]. Indeed, it was, during the use of SAGS, while simulating the Nash implementation of Faddeev's algorithm [6], [7], that the mentioned bug in its boundary cell microprogram was discovered and identified.

In the Appendix A, six snapshots from a series of 13 will illustrate the simulation of Nash's systolic design. Each snapshot is a SAGS screen output for one execution step. To appreciate the usefulness of SAGS in finding and correcting the error, let us first explain the Nash's architecture which is shown in Fig. 1. It is based on the Faddeev algorithm. This is a general purpose algorithm useful for a wide class of matrix operations and especially suited for systolic implementation. It calculates the value of CX + D, given AX = B, where A, B, C, and D are known matrices of order n, and X is an unknown matrix. The problem is formulated as

$$\begin{array}{c|c} A & B \\ \hline -C & D \end{array} \qquad (1)$$

If by some means, a suitable linear combination of the rows of A and B is found and added to the rows of -C and D as follow

$$\begin{array}{c|c} A & B \\ \hline -C+WA & D+WB \end{array}$$

where W specifies the appropriate linear combination such that only zeros appear in the lower left hand quadrant then the lower right hand quadrant will become matrix E = CX + D. This is because annihilating -C requires W = CA^{-1} so that D + WB = $CA^{-1}B$, and since AX = B, D + WB = D + CX. The elegance and simplicity of the algorithm is apparent when one notes that to carry it out it is only necessary to annul the lower left hand quadrant by applying a suitable matrix triangularization procedure to the left side of (1) while extending the operation to its right side. This can be done using Gaussian elimination. To improve the numerical stability of the Faddeev algorithm, Nash suggested a modification to it by replacing the Gaussian elimination procedure used to triangularize matrix A of (1) with *orthogonal triangularization*. The array consists of two types of cells: *the boundary cells* (represented by circles) and the *internal cells* (represented by squares). Each cell stores a microprogram. Changing the microprograms of cells will allow the array to execute different procedures.

For clarity, it is useful to divide Nash's algorithm into a two-phase procedure. In the first phase, A is triangularized by a series of *Givens rotations* (simultaneously applied to B); in the second phase, the diagonal elements of the resulting triangular matrix are used as *pivoting elements* in the *Gaussian elimination* procedure on C and D, where columns of C will be zeroed out and D will become the result. Note that for the Gaussian procedure to work properly, it is necessary that these pivoting elements be non-zero. Hence, the requirement that A be a full rank, i.e. at least one of its square submatrices of order n has a non-zero determinant.

Nash's systolic implementation, shown in Fig. 1, consists of a triangular array and its right extension: a square array. The triangular array, based on Kung's design in [4] for *orthogonal triangularization*, performs Givens rotations on A (first phase) and ordinary Gaussian elimination on C (second

The cell's execution codes are shown in Figure 2.

Furthermore, the added processing of ordinary Gaussian elimination requires the extra codes shown in Figure 3.

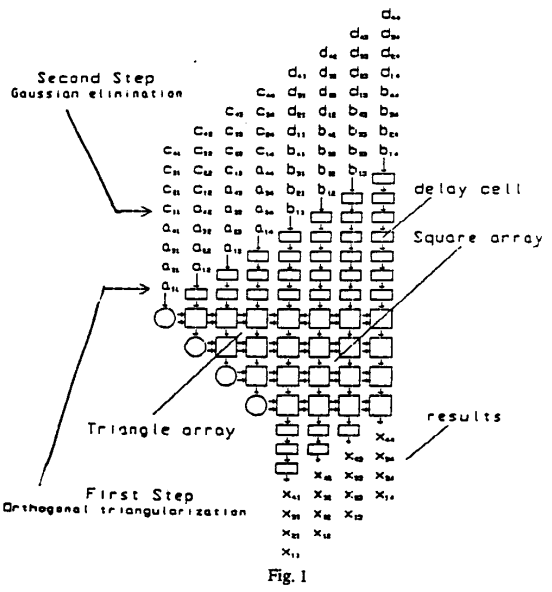


Fig. 1

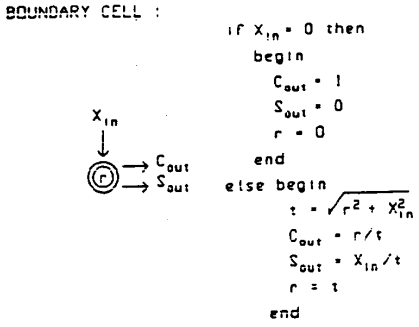


Fig. 2

Department of Electrical and Computer Engineering and
The Coordinated Science Laboratory

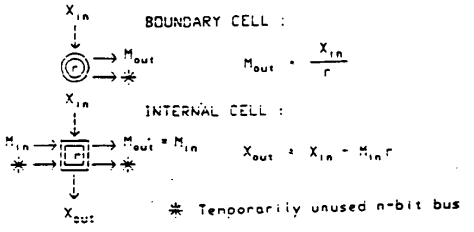


Fig. 3

The square array simply extends the corresponding processing to **B** and **D** and thus only consists of square cells. The input data flow involves feeding **A** and **B** through the system from the top with cells executing the microprograms codes shown in Figure 2 on each incoming row. This corresponds to the first phase of the modified algorithm. Notice that the required skewing of the data flow is performed by a triangular array of delay cells (represented by rectangles) above the system. The second phase is accomplished by a similar flow of **C** and **D**. Only this time the cells execute the microprograms shown in Figure 3 on the data elements and the resulting matrix will appear row by row coming from the bottom of the square array. These output rows are straightened back to normal by another triangular array of delay cells below the square array. The input data flow can be contiguous, i.e. matrices **A** and **B** and then **C** and **D** can enter the array without any interruption in between.

Although Nash's modification of Faddeev's algorithm is mathematically sound, its systolic implementation, unfortunately, contains some serious deficiencies. For instance, it is possible for the array to produce erroneous results, as illustrated by the following example. Suppose we have a linear system $\mathbf{AX} = \mathbf{B}$ of order $n = 3$ where **X** is an unknown matrix, and one or more entries in column 1 of matrix **A** are zeroes, in this case, a_{21}:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 7 \\ 2 & 1 & 3 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 5 \\ 9 \\ 7 \end{bmatrix} \qquad (2)$$

Since the determinant of **A**, $\Delta(\mathbf{A}) = 9$, is a non-zero, **A** is full rank. Thus, guaranteeing that a solution to the system exists and that it is unique with $x_1 = 1.33$, $x_2 = -0.67$ and $x_3 = 1.67$. When **A** is fed into the array shown in Figure 1 during the second step, however, the boundary cell in row 1, column 1 will clear its r register (it had $a_{11} = 1$ stored prior to this) because $a_{21} = 0$. This effectively transforms **A** into another matrix, say **E**, whose entries are identical to **A**'s except for e_{11}, which is zero, and all further processing will be on the resulting linear system.

$$\mathbf{E} = \begin{bmatrix} 0 & 2 & 3 \\ 0 & 4 & 7 \\ 2 & 1 & 3 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 5 \\ 9 \\ 7 \end{bmatrix} \qquad (3)$$

In this case, since $\Delta(\mathbf{E}) = 4$ is non-zero, **E** is also full rank and therefore the procedure is completed successfully, but with $x_1 = 3$, $x_2 = 4$ and $x_3 = -1$, which is the solution to (3) instead of (2).

The cause of the above error can be traced to a bug in the microprogram in the boundary cell. As Figure 2 reveals, this microprogram has a line of code

$r = 0$

which always clears the content of register r whenever $x_{in} = 0$. In fact, if at any time during processing the boundary cell of a row i receives a zero-valued x_{in} from an internal cell of row i - 1, an erroneous result will appear when the processing ends. Thus, to correct the problem, this line should be removed. The corrected Nash algorithm has not been published to our knowledge and we do not think we would have been able to find and correct this bug without SAGS.

For the purpose of verification, we first found the correct solutions to examples (2) and (3) manually and then by using other algorithms applying the Faddeev algorithm with Givens rotations: $x_1 = 1.33$, $x_2 = -0.67$, $x_3 = 1.67$. Now that the correct result is known, the reader is encouraged to examine the series of snapshots in Appendix A which shows the graphics simulation of Nash's array computing (2). Only steps 2, 3, 4, 5, 12, and 13 are shown from the series of 13, but these pictures clearly illustrate the sequence of events leading up to the erroneous results. (The screen views have been overlapped slightly so that they will fit on to a single page).

4. CONCLUSION

We have presented a new approach to simulate systolic processors. A visual simulator allows for the visual analysis of data flows in systolic arrays of various configurations. This approach allows for the visualization of the behavior of such systems and is, therefore, also **an excellent system design educational tool.** It proved to be very useful in describing and analyzing several well-known and new architectures. A microcode error has been identified and corrected in a well-known algorithm thanks to this simulator.

5. LITERATURE

[1] H. T. Kung, "Why Systolic Architectures?", *IEEE Computer Magazine*, Vol. 15, No. 1, January 1982, pp. 37 - 46.

[2] H. T. Kung, and C. E. Leiserson, "Systolic Arrays (for VLSI)", *Sparse Matrix Proc. 1978*, SIAM, 1979, pp. 256 - 282.

[3] D. K. Faddeev, and V. N. Faddeeva, *Computational Methods of Linear Algebra*, W. H. Freeman and Company, 1963, pp. 150 - 158.

[4] W. W. Gentleman, and H. T. Kung, "Matrix Triangularization by Systolic Arrays", *Proc. SPIE - The International Society of Optical Engineering*, Vol. 298, 1981, pp. 19 - 26.

[5] H. T. Kung, "Systolic Array for Orthogonal Triangularization", *Proc. SPIE*, San Diego, CA, 1981, pp. 19 - 26.

[6] J. G. Nash, "A Systolic/Cellular Computer Architecture for Linear Algebraic Operations", *Proc. 1985 IEEE International Conference on Robotics and Automation*, March 1985, pp. 779 - 784.

[7] J. G. Nash, and S. Hansen, "Modified Faddeev Algorithm for Matrix Manipulation", *Proc. SPIE*, Vol. 495, August 1984, pp. 39 - 46.

[8] H. Y. H. Chuang, and G. He, "A Versatile Systolic Array For Matrix Computations", *The International Symposium on Computer Architecture*, 1985, pp. 315 - 322.

[9] V. H. D. Le, "A New General Purpose Systolic Array for Matrix Computation", *M. S. Thesis*, Department of Electrical Engineering, Portland State University, P.O. Box 751, Portland, Oregon 97207, April 1988.

[10] V. H. D. Le, and M. A. Perkowski, "A New General Purpose Systolic Array for Matrix Computation", submitted to IEEE TC, 1988

[11] H. Hellwagner, and T. Muller, "LISAS - Loops-implemented systolic array simulator", *Proc. of VLSI & Computers Conference*, COMPEURO '87, IEEE, 1987.

[12] Caominh Tuan, and M. A. Perkowski, "Logic/Register-Transfer Simulator with Visualization", *Record of Northcon/86*, paper 11.3, September 30 - October 2, 1986, Seattle, Washington.

[13] "Turbo Pascal Manual".

[14] Wirth, and Jensen, "Pascal User Manual and Report".

[15] G.P. Brown, R.T. Carling, C.F. Herot, D.A. Kramlich, and P. Souza, "Program Visualization: Graphical Support for Software Development", *Computer*, pp. 27-35, August 1985.

[16] D.D. Hill, "Edisim: A Graphical Simulator Interface for LSI Design", *IEEE Trans. CAD ICS*, Vol. CAD-2, No. 2, pp. 57 - 61, April 1983.

[17] R.J.K. Jacob, "A State Transition Diagram Language for Visual Programming", *Computer*, pp. 51 - 59, August 1985.

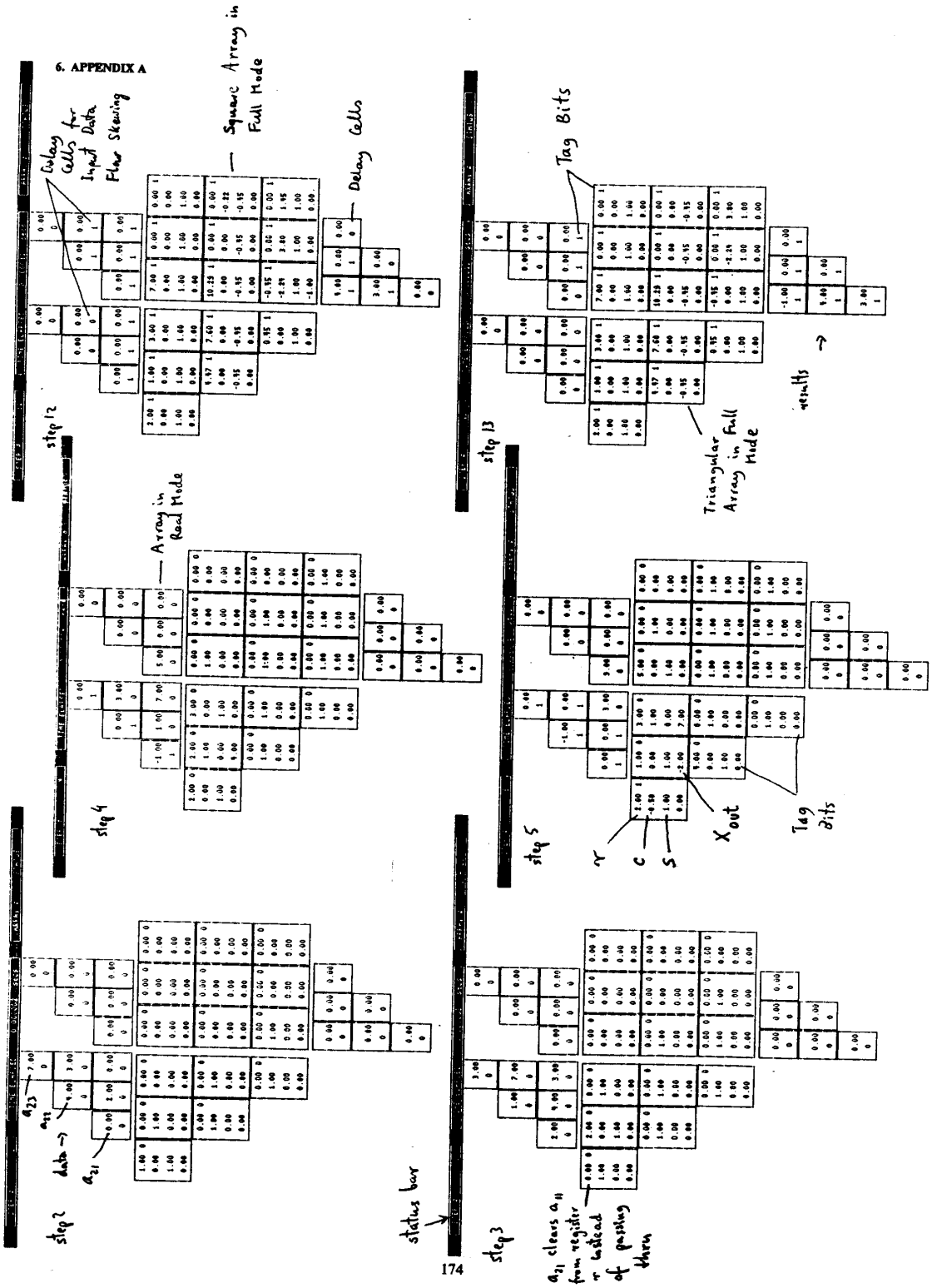
[18] B. Melamed, R.J.T. Morris, "Visual Simulation: The Performance Analysis Workstation", *Computer*, Vol. 18, No. 8, pp. 87 - 94, August 1985.

[19] A. Sugimoto et al, "VEGA" A Visual Modeling Language for Digital Circuit Design", *Proc. IEEE ICCD'84*, pp. 807 - 812, 1984.

[20] A. Sugimoto, " "VEGA" A Visual Modeling Language for Digital Systems", *IEEE Design and Test*, pp. 38 - 45, June 1984.

[21] A. Sugimoto, S. Abe, M. Kuroda, and Y. Kato, "An Object-Oriented Visual Simulator for Microprogram Development", *Proc. 23-rd DAC*, pp. 138 - 144, June 29 - July 2, Las Vegas, 1986.

6. APPENDIX A


Delay cells for Input Data Flow Skewing
Square Array in Full Mode
Delay Cells
step 2
step 3
step 4
step 5
Array in Real Mode
Triangular Array in Full Mode
Tag Bits
Tag Bits
Status bar
a_{21} clears a_{11} from register & instead of passing them
X_{out}