# A Data-Driven VLSI Array for Arbitrary Algorithms

**Israel Koren and Bilha Mendelson, University of Massachusetts at Amherst**

**Irit Peled and Gabriel M. Silberman,* Technion, Israel**

**Control-driven arrays like systolic arrays provide high levels of parallelism and pipelining for inherently regular computations. Data-driven arrays can provide the same for algorithms with no internal regularity.**

A variety of topologies and architectural designs of processor arrays have recently been proposed, and many computational algorithms for these arrays have been devised.[1] These include globally synchronous systolic arrays and globally asynchronous wavefront arrays.[2,3] Most existing algorithms for these arrays were developed for problems with inherent regularity, like vector and matrix operations, signal and image processing, pattern recognition, and some nonnumeric (such as sorting and language recognition) applications. These computations proceed in the processor array in a predetermined manner and achieve high performance through parallelism and pipelining.

In systolic arrays, all operations are synchronized by a global clock. This global synchronization ensures that all operands that have to be processed by every processing element in each computational step arrive at every PE simultaneously. Global synchronization greatly simplifies the internal control of the PE's operations. However, this simplification is achieved at the cost of a possible slowdown in the throughput of the entire array.[3] The period of the global clock has to be large enough to accommodate the slowest local operation (especially when the execution time is data dependent) and any clock skews. The latter may result in a slowdown if large arrays of PEs have to be synchronized or if fault-tolerance through reconfiguration is introduced into the array, since varied path length and, consequently, varied propagation delays have to be taken into account.[3]

Wavefront arrays are asynchronous arrays that substitute the requirement of correct timing with that of correct sequencing. Logically, the computation front in these arrays advances exactly as it does in systolic arrays, the difference being that the speed of propagation is not necessarily uniform across the front. This allows for differences in the timing of the operations being executed, either because some of them are inherently faster (for example, addition versus multiplication) or due to dependencies on the data (for example, multiplication by zero is faster).

Both systolic arrays and wavefront arrays have a computation front that propagates according to a predetermined *fixed* (that is, *data-independent*) sequence. Consequently, these arrays prove to be very effective for executing highly regular algorithms. Methods for mapping algorithms with inherent regularity onto these arrays have appeared in recent publications (see, for example, *Computer*, Vol. 20, No. 7[1]).

Many computationally demanding problems do not exhibit high regularity and, therefore, are unsuitable for these arrays. Still, many of these problems have an inherent parallelism, and it should be possible to exploit this parallelism through processor arrays that can also provide a high degree of pipelining.

To this end we suggest the design of specialized processing array architectures, capable of executing any given arbitrary algorithm. We adopt the approach proposed by Koren and Silberman,[4] where the algorithm is first represented in the form of a *dataflow graph* and then mapped onto the specialized processor array. The processors in this array will exe-

*Silberman is currently at Carnegie Mellon University, on leave from the Technion.

0018-9162/88/1000-0030$01.00 © 1988 IEEE

cute the operations included in the corresponding nodes (or subsets of nodes) of the dataflow graph, while regular interconnections of these elements will serve as edges of the graph.

Figure 1 shows a dataflow graph representing the computation of the two coefficients $A$ and $B$ in the solution $Y(t) = A \cos wt + B \sin wt$ of a spring-mass system with an external force $F(t) = F_0 \cos wt$. These coefficients are computed from

$$A = F_0 \frac{k - Mw^2}{(k - Mw^2)^2 + w^2c^2}$$

and

$$B = F_0 \frac{wc}{(k - Mw^2)^2 + w^2c^2}$$

where $k$, $M$, and $c$ are given parameters.

Figure 2 depicts a possible mapping of the graph in Figure 1 onto a regular processor array. The hexagonal topology of the array serves only as an example of a regular structure. The dataflow graph representation enables the exploitation of concurrency at the lowest possible level by treating each instruction as an independent activity. In this way, "fine-grain parallelism"[5] can be achieved. For example, the products $wc$ and $w^2$ in Figure 2 can be calculated simultaneously.

When an arbitrary algorithm executes on an array, in general no regular propagation of computational fronts takes place. Hence, to speed up the execution of arbitrary algorithms, we need a more flexible array. Such an array will allow the generation of new computation fronts and their cancellation at a later time depending on the arriving data operands. We call these arrays, therefore, *data-driven* arrays.

In these processor arrays, not all cells (processing elements) will perform exactly the same set of operations. As a result, the operation time of different cells in the array may differ substantially, and subsequent cells may receive their operands in an arbitrary order. Therefore, the cell should be capable of testing for the presence of its operands and execute only the instructions for which all the necessary operands have arrived. Thus, the order in which instructions are executed in a cell is data-dependent. We call such a cell a *data-driven* processing element. (Notice that wavefront arrays are also data-driven, in the sense that operations proceed as soon as operands become available. Nevertheless, these arrays are limited to map acyclic dataflow graphs, and their computational fronts are fixed and data-independent.)

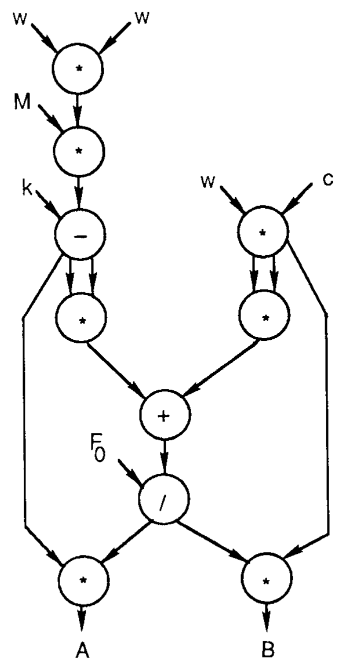


**Figure 1. A dataflow graph.**

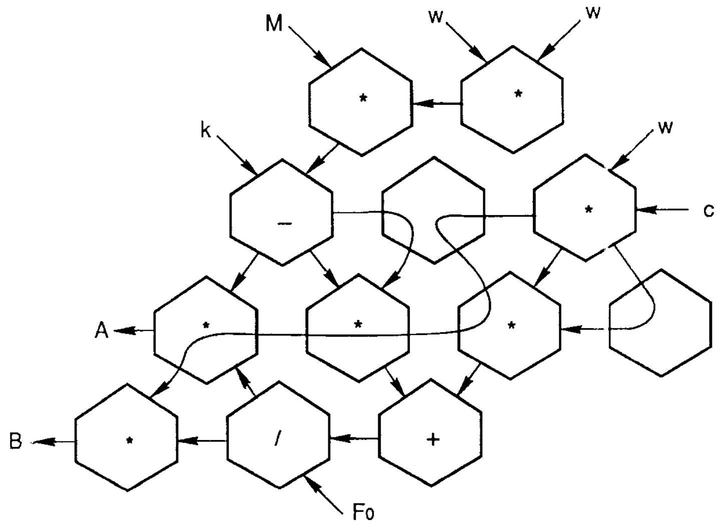


**Figure 2. The mapping of the graph in Figure 1 onto a hexagonally connected array.**

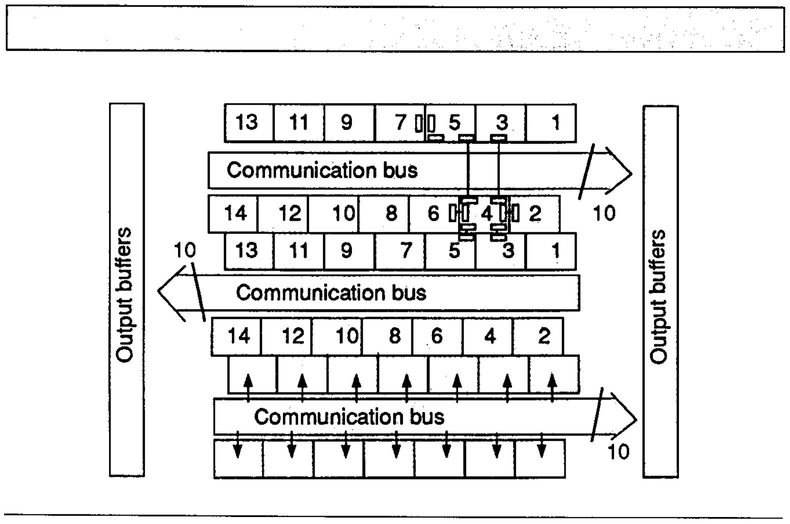


**Figure 3. The floor plan of the processor array chip.**

A key feature of our approach, then, is that it allows any arbitrary algorithm, expressed in a dataflow programming language (such as VAL[6], or Value-oriented Algorithmic Language), to be executed in a data-driven processor array that can provide high levels of concurrency and pipelining. The mapping of the algorithm is changeable, enabling the user to map various dataflow graphs (algorithms) onto the same array. Such a property increases the number of applications of the array, making it more appealing to the semiconductor industry and to users, alike. Also, programmable arrays admit simple fault-tolerance, since the dynamic and restructurable mapping of dataflow graphs will permit avoiding faulty processors and communication links, using any of the possible reconfiguration techniques.[7]

The programmability and reconfigurability features of the data-driven processor array somewhat resemble those of the Configurable, Highly Parallel Computer, or CHiP, array,[8] where a programmable interconnection structure is integrated with an array of processing elements. The user can program this interconnection structure to best suit a specific application.

The mapping of the application onto the CHiP array is performed differently, however. The mapping in our case is an automatic process, while the user must hand-tailor the CHiP array to each specific algorithm. (Support for this task is provided in the form of the Poker Programming Environment.)

Before an algorithm is mapped onto the data-driven array, it is translated into a dataflow graph. See the next section for a brief outline of the concept of dataflow graphs. We then describe in detail the principles of the design of a data-driven array. Finally, we present an algorithm for mapping a given dataflow graph onto the array. This mapping is independent of the capabilities of each data-driven cell in the processor array. In other words, each node in the dataflow graph is mapped onto a processor in the array, regardless of the node's function. The translation process, from a program in VAL to a dataflow graph, makes each node in the graph match the processing capabilities of an element in the array.[9]

## The dataflow concept

In most computation systems, execution of a program is performed according to its *control flow*. The program executes sequentially according to the order specified in the program. Even if the program is written in a concurrent programming language (such as CSP or Ada), the programmer has to identify the potential for parallelism in the algorithm. Furthermore, those segments explicitly identified as parallel are each executed in a control flow fashion, usually during execution of the concurrent program.

The dataflow approach to programming attempts to exploit the parallelism inherent in each algorithm, without requiring the programmer to indicate it explicitly in the program. In this approach, a program is written in a side-effect-free language (such as VAL[6]) and translated into a *dataflow graph*, or DFG. Dataflow program graphs explicitly represent the data dependencies within a program and, in so doing, identify program operations that may be executed independently (that is, implicit parallelism).

We can view a dataflow program graph formally as a directed graph whose nodes are operators and whose arcs represent data paths. "Execution" of a DFG occurs as tokens "flow" along the arcs, into and out of nodes, according to a set of *firing rules*. These firing rules specify that a node may fire whenever tokens are present on each of its input arcs and no token is present on any of its output arcs. Such nodes may fire in any order, and two or more nodes can fire concurrently. When a node fires, tokens are removed from each of its input arcs, the function represented by the node is computed using the absorbed data values, and the result is output as a token on the node's output arc(s).

Several authors have noted the advantages of using the dataflow approach to programming. (For a comprehensive review on the subject, refer to Dennis[5] and Khavi, Buckles, and Bhat.[10]) The main advantage is the "natural" parallelism. That is, we do not need special analysis of the algorithm, looking for portions of it that can be executed in parallel, prior to writing the program.

Clearly, the parallelism in a dataflow program can be affected by the way in which the program is written. The sequence of node firing depends only on data dependencies, thus the level of concurrency can be automatically detected by a compiler and expressed as a DFG.

In addition to exploiting parallelism, dataflow provides the potential for further algorithm acceleration by pipelining. When a node has fired and the result(s) from its operation are consumed, the node is ready to receive new data and carry out its function, even though overall processing of the previous cycle's data may not have finished.

## Processor array architecture and principles of operation

The feasibility of designing control-driven processor arrays was never in question; several processing elements for these arrays have already been designed (for example, see Fisher et al.[2]). However, the degree of hardware complexity required to add the data-driven property was not clear to us. Therefore, we made a preliminary design of a low-hardware-complexity data-driven element. The result of this design is very encouraging. The total hardware complexity of the cell (presented next) is less than 9,000 transistors in n-type metal-oxide-semiconductor technology. This low complexity should make possible the fabrication of a very-large-scale-integration chip containing about 50 to 100 cells. Several such chips can then be put together to form a larger processor array.

The first phase of the design has already been completed and is reported by Peled.[11] The final steps of the detailed design and layout of the VLSI chip are now being carried out.

The proposed floor plan of the chip is shown in Figure 3. It contains data-driven cells (all identical, including the boundary ones) arranged in rows such that the typical cell has six immediate neighbors in a hexagonally connected processor array. The high degree of connectivity of the hexagonal topology simplifies the task of embedding various graph structures. However, the hexagonal topology serves here only as an example; it has little impact on the principles of the design.

The cells communicate with an external host computer through communication buses that run between an odd-numbered row and the subsequent even-numbered one. The individual cells' programs are prepared in this host computer and then distributed to the cells. The host also supplies the necessary input operands and accumulates the final results.

In control-driven processor arrays, the host-to-array communication can in most cases be conveniently restricted to pass through the boundary cells. In data-driven arrays, the elements that are expected to either receive input operands or transmit final results may be distributed throughout the array.

Here, the host to array communication, if limited to the boundary cells, can substantially slow down the array operation. This is unacceptable in a data-driven cell, so provision is made for each cell to communicate directly with the host through the communication buses shown in Figure 3. The internal communication among cells does not use these buses. Instead, each cell communicates *directly* and *in parallel* with its six neighbors through dedicated registers.

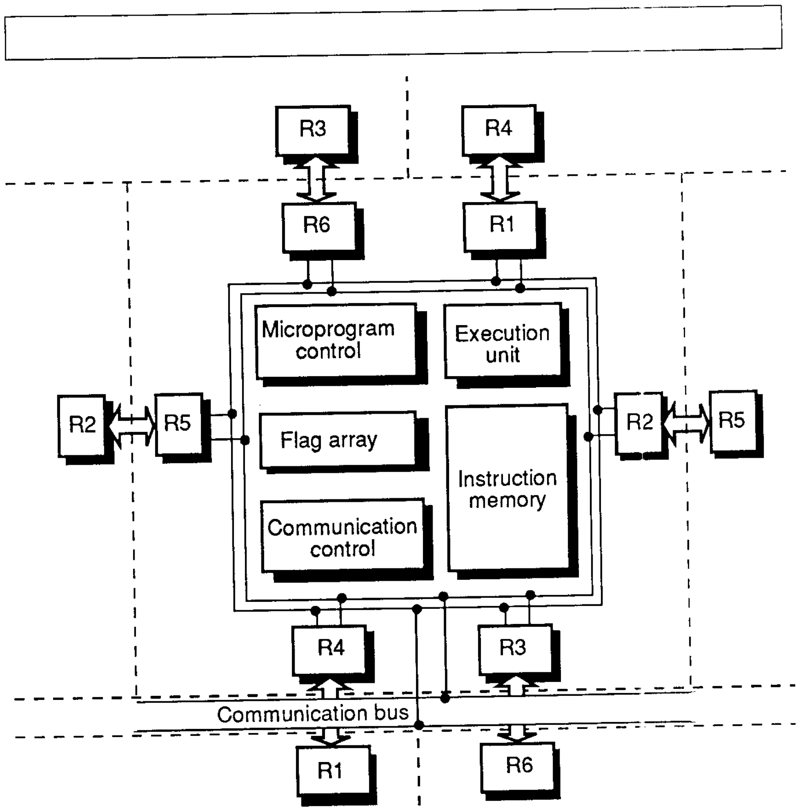


**Figure 4. Functional blocks in the data-driven cell.**

The width of the communication bus determines the maximum number of cells per row. In the current design the width of the bus is 10 bits, out of which 7 are used for addressing purposes yielding 128 different addresses (that is, at most 64 cells per row). The maximum number of rows is determined by $N$—the number of package pins. After subtracting from $N$ at least 10 (for clock signals, ground, and $V_{dd}$), we divide the result by 10 (the width of the bus) and then multiply by two. Currently, packages with $N = 244$ pins are available, yielding a maximum of 45 rows. Theoretically, the design presented here can yield a total of $64 \times 45 = 2{,}880$ cells.

However, we cannot yet achieve the upper limit of 2,880 cells per chip because of the limited number of transistors possible on a single chip. For a maximum of 500,000 transistors per chip, the chip can include only 50 cells (based on the detailed design of the cell, which requires less than 9,000 transistors). A density of 1 million transistors per chip is expected in the near future, yielding almost 100 cells on a chip. These cells can be arranged in various ways, like $10 \times 10$, $11 \times 9$, and so on.

## Structure of a basic cell

The cell as designed operates on 8-bit operands. All internal registers and buses are 8 bits wide. Still, two or more cells can participate in a single (arithmetic) operation enabling the execution of 16- or 32-bit operations. All cells—identically designed—differ only in their ID number. Each cell has an identifying address that is unique within the pair of rows connected to the same bus (see Figure 3).

The functional blocks composing the basic cell are depicted in Figure 4. The communication control block contains the control logic and bus interface for communicating with the host computer, for such functions as loading the instruction memory and receiving and transmitting data operands. This block has a total complexity of about 150 transistors.

Six 8-bit registers, $R1$ through $R6$ (with a total complexity of 400 transistors), are positioned in the periphery of the cell. These registers connect to a common internal bus through which data can be transferred from one register to the other and from a register to the execution unit and vice versa. A register can also be loaded (or unloaded) through the internal bus with an operand provided by the host. In addition, each of the six registers connects directly to the corresponding register in one of the six neighboring cells (refer to Figure 4).

The carry-in and carry-out bits are also transferred through these registers, to allow operations on multibyte (that is, 16-bit or more) data. This implies that no other data can be transferred in parallel between cells with a carry connection. We could not justify the extra control logic required to support separate carry lines between any two adjacent cells, especially since no data (other than the carry bit) has to be transferred between cells participating in a multibyte operation. There is, however, a side benefit to this: since the carry-in from another cell is just another datum, it can be treated as a Boolean operand (as for testing some condition) instead of an arithmetic carry bit.

The transfer of data between the host and any of the six registers is under program control, while the transfer of data between a register and the corresponding register in the adjacent cell is under hardware control. The latter is a frequent operation, and its execution time is crucial. In our design, this transfer takes a single clock cycle and is done in parallel to all other operations in the cell.

The *instruction memory* consumes about 1,150 transistors and contains instructions that specify the operations the cell has to perform on operands it receives. These instructions are loaded from the host computer before the array's operation is started.

The *flag array* is a uniquely designed block that controls the data-driven operation of the cell. The instructions in the cell are not executed in any predetermined order. Instead, the arrival of all operands for a certain instruction enables the execution of that instruction. The flags monitor the movement of data operands internally, as well as in and out of the cell. For each register, a flag indicates whether the register has an operand or is empty and can receive a new operand. The flag array block (whose design is detailed in a later section) requires about 350 transistors.

The *microprogram control* unit is the largest block in the cell. It requires about 4,500 transistors. This unit translates the cell instructions stored in the instruction memory into sequences of control signals that execute these instructions. In the current design, most instructions take between two to eight clock cycles to execute. The execution of instructions can speed up if more parallelism is introduced into the microprogram, although this will increase the hardware complexity of the unit.

Finally, the *execution unit*, with a total complexity of 1,000 transistors, contains a 16-bit shift register and an arithmetic unit capable of executing all common Boolean and arithmetic operations, including multiplication and division (for which sequential algorithms are employed). The operands for the arithmetic operations can be either positive operands or negative ones represented in 2's complement. Carry bits are generated and used whenever applicable to allow multibyte operations.

## Programming the basic cell

One of the most important design decisions is the determination of the maximum complexity of the program each cell can execute. We can identify two alternatives:

(1) One or two operations in every cell.
(2) A separate instruction for each register (out of the six data registers in the cell).

In the first alternative, the design of the cell is simpler and its operation is faster. However, the overall utilization of the cell is very low. This results in the need for large arrays for even the simplest computations. The larger array required may decrease the throughput even though each individual cell operates at a higher speed.

The second alternative achieves the highest level of cell utilization, since each register has a separate instruction. Theoretically, we can have six different cell operations. In practice, however, we will have only two to four operations in a cell, since many operations involve more than a single register.

This alternative may also lead to a lower overall execution time. For example, the fact that a single variable can be an operand for several operations within the same cell eliminates the need for generating several copies of that variable and distributing them to other cells.

It may appear that the amount of hardware needed in this case is substantially higher than that needed for the first alternative, yet after completing detailed designs of both alternatives we have concluded that the second alternative requires less hardware. Also, as shown below, a data-driven element capable of performing up to six operations yields a more effective mapping process.

In view of the above, we selected the second alternative for implementation.

**The instruction format.** We defined about 108 instructions, most of them similar to those found in any ordinary 8-bit microprocessor. These instructions include

(1) Boolean instructions like AND, OR, and XOR.
(2) Arithmetic instructions like addition and subtraction in 2's complement, and 8-bit multiplication and division. An arithmetic operation may have an input carry and/or generate an output carry.
(3) Shift instructions for 8- and 16-bit shift operations with or without carry-in and -out.
(4) I/O instructions to control communication with the host.
(5) Flow control instructions, like Route an operand into one of two output paths according to the value of a Boolean control operand.[4]
(6) Cell initialization instructions indicating which registers will hold the carry-in, the carry-out, and the eight most significant bits in operations with 16-bit results.

A few instructions have up to six operands. For example, a shift operation (with carry-in and carry-out) that operates on a double word (16 bits) has three input operands and three output operands. Most instructions, however, have a substantially smaller number of operands. The commonly used solution of variable-length instructions must unfortunately be rejected here. In contrast to a control-driven PE, the data-driven cell does not execute its instructions sequentially. Whenever a new operand arrives, the cell has to look for instructions that need this operand and test whether they already have all their operands. If the instructions do not start in known positions (in the memory), the search operation becomes considerably more complex and time consuming. To prevent a lengthy search and simplify the test for ready-to-execute instructions, the instructions should be of fixed length and the operand fields within the instruction should be in fixed positions.

We therefore propose the instruction format shown in Figure 5. The leftmost nine bits constitute the operation-code field and the rightmost six bits, the operand field. The latter indicate only which registers hold the input operands. $LRi = 1$ ($i = 1, 2, \ldots, 6$) if the register $Ri$ is an input operand. Up to three bits out of the six may equal 1 in operations with carry-in.

We included no output operands in the instruction format. As mentioned above, the six instructions correspond to the six data registers in the cell such that $Ri$ is the destination of the final result for the $i$th instruction ($i = 1, 2, \ldots, 6$). Also, we assigned no instruction bits to specify which registers hold (whenever applicable) the carry-out and the eight additional bits in operations with a 16-bit result. Instead, we added special instructions to the instruction set invoked during cell initialization to indicate which registers are used for these purposes.

The implication of this design decision is that a cell can have at most one instruction generating a carry-out and at most one instruction with a 16-bit result. This limitation is reasonable in view of the reduced instruction size and simplified control logic.

A similar cell initialization instruction indicates which register is used for the carry-in, allowing the cell to distinguish between the register holding the carry-in and the two registers holding the input operands, in case three $LRi$ bits are on.

The above scheme is appropriate for commutative operations. However, for noncommutative operations like subtraction and division, we have to specify which of the two operands (not including the carry-in) is first. For this purpose we use the *Frst* bit; if this bit is 0, the first (rightmost) register in the instruction holds the first operand; otherwise, the second register holds the first operand.

The most significant bit in the op-code field is the *Act*, or active, bit. This bit always equals 1 except in the cell initialization instructions, which are executed only once when first loaded and are ignored from that point on. The *Cin* and *Cout* bits indicate whether the instruction needs a carry-in and/or generates a carry-out, respectively.

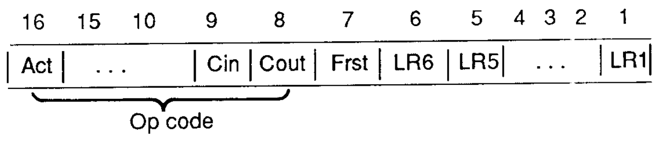


Figure 5. Proposed instruction format.

## Detailed design of selected blocks

In this section, we present some design details of two selected blocks. We designed the other blocks in the cell in a straightforward manner and do not detail their design principles here.

**Instruction memory.** The instruction memory consists of a 13-byte memory, an address counter and three registers that hold the numbers of the data registers containing the input carry, the output carry, and the eight additional bits generated in operations with 16-bit results.

The 13 bytes in the instruction memory contain six instructions, each two bytes in length. The 13th byte has only six meaningful bits (corresponding to the six data registers). Bit $j$ in this byte is 1 if the contents of $Rj$ have to be transferred to the corresponding register (whose index is $1 + (j + 2) \bmod 6$) in the neighboring cell (see Figure 4). Each bit in this byte has to be accessed separately by the hardware.

The address counter of the instruction memory has two modes of operation, instruction loading mode and execution mode. In the first mode, all 13 addresses have to be generated; in the second mode, only six addresses have to be accessed. Instead of implementing the address counter as an ordinary binary counter, we used a shift register. This implementation requires 24 additional transistors, but it allows the counter to skip addresses that do not contain executable instructions, considerably improving the performance of the cell.

Consider, for example, a cell having a single executable instruction. The use of an ordinary counter will result in a five-cycle delay before reexecution (of the same instruction) on a new set of operands can start. The designed counter (based on a shift register) enables immediate reexecution of the same instruction if new operands are available.

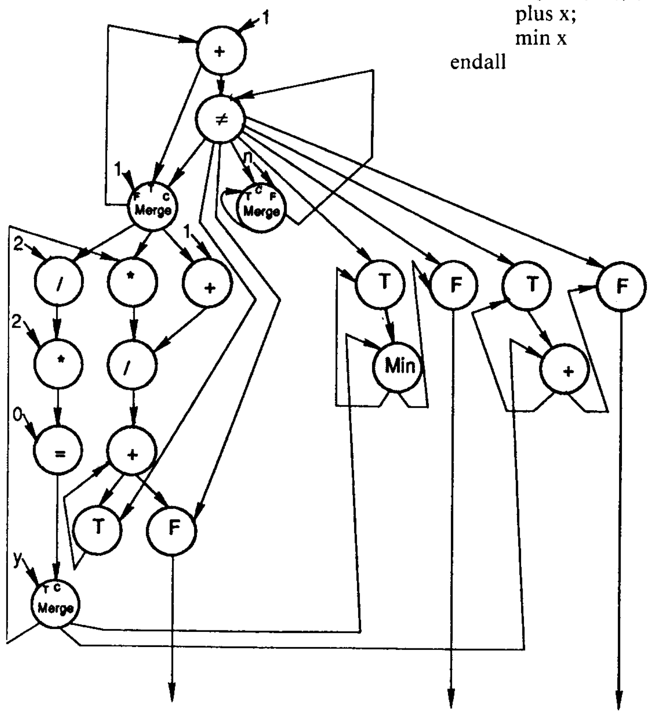
**Flag array.** The flag array controls the data-driven operation of the cell. The exact status of the six data registers is kept in this array. Through it, the cell can check whether a data register is empty and can be reloaded, or whether all the operands for a certain operation have arrived. The flag array contains only the dynamic flags that change during the execution of the instructions. The instruction memory contains the static flags (the $LR1$ through $LR6$ bits) that indicate which data registers are the operands for the given operations.

The dynamic flags are arranged in seven rows and six columns. The first six rows are identically designed and correspond to the six instructions that the cell executes. The six columns correspond to the six data registers. The arrival of a new operand in $Rj$ enables the setting of all the flags in the $j$th column. The set signal is ANDed with the static flags for $Rj$ in all six instructions. Thus, the number of 1's in column $j$ equals the number of operations for which $Rj$ is an operand.

The dynamic flags are reset by rows. Whenever the execution of instruction $i$ completes, the dynamic flags in row $i$ are reset. When all flags in column $j$ are reset, register $Rj$ is empty (its current value is not needed any more) and is ready to receive new data.

The seventh row in the flag array differs from the first six. The dynamic flags in row 7 correspond to the static flags in byte 13 of the instruction memory. If dynamic flag $j$ in the seventh row is set, it indicates that the data in $Rj$ is ready for transfer to the proper register in the neighboring cell. Since the transfer of data is controlled by hardware (and not the microprogram),

(a)
```
forall i in [1,n]
x: = if (i/2)*2 = i then x
     eval plus (x*i)/(i + 1);
       plus x;
       min x
endall
```



(b)

Figure 6. A program in VAL (a); its translation to a dataflow graph (b).

each flag in row 7 is tested and reset separately by hardware.

Before executing an instruction, the cell verifies that the dynamic flags (of the input registers) in the row of the current instruction equal the corresponding static flags, and that the flag of the output register and the carry-out register (if applicable) are reset (indicating that the registers are empty). If all these conditions are satisfied, the execution starts.

A single cycle suffices to test whether an instruction is ready for execution. The flag of the output register is also tested to prevent a possible deadlock. Suppose that, after the execution starts, it is found that the output register is not empty because another instruction still needs its value as an input operand. The current instruction cannot then proceed, and the other instruction (which would empty the output register) cannot start.

## Large arrays

The chosen design allows the implementation of more than 50 basic cells on a single chip. If we require a larger number of cells, we can connect several of these chips to the same host computer. Alternatively, we can make this connection through a control unit, which can reduce the overhead of the host and speed up the operation of the array. This control unit can be either a commercially available microcontroller like Intel's 8051 or a special-purpose design.

We decided on the latter, mainly because it can speed up interchip communication. Internally, the cells communicate directly with each other. However, because of the shortage of I/O pins, such direct communication is not possible between cells in different chips. Therefore, this communication has to take place through either the external host or, preferably, through a specially designed control unit. We plan to design such a control unit in the near future.

The interchip communication may still overload the external controller. Consequently, the algorithm to be mapped onto the array has to be partitioned so as to minimize interchip communication.

## The mapping process

In this section, we present and analyze a scheme for mapping a dataflow graph (program) onto a hexagonally connected PE array (first proposed by Mendelson and Silberman[9]).

The process of mapping a dataflow program onto a hexagonal array of processors consists of four phases. For the sake of modularity, we keep these phases as separate stages in the mapping process. This allows experimentation with various algorithms for each phase, as well as easy integration of changes driven by developments in technology. The four stages in the mapping process are

Stage 1: DF-program translation
Stage 2: DFG mapping onto array
Stage 3: Array partitioning
Stage 4: Task-to-PE assignment

The first stage in the mapping process is the translation of a dataflow program, written in a subset of VAL, to a dataflow graph. This subset excludes all dynamic data structures (such as heaps, stacks, multiple-field records) that require a central memory facility and their related language constructs (such as bound checking). This stage is general and does not depend on the specific architecture of the processor array.

The language VAL was developed to express algorithms for execution on computers with a high level of parallelism, more specifically, data-driven machines. As such, the language is free from side effects.

The translator that implements the first stage in the mapping process was built in a conventional fashion, by utilizing the Unix tools Lex (for Lexical Analyzer Generator) and YACC (for Yet Another Compiler Compiler). Translation proceeds by analyzing each construct in the program and translating it to a subgraph. These subgraphs, when combined, form the program DFG. This approach to translation creates a DFG with high locality; that is, each control structure in the program will be a connected subgraph in the DFG, a fact that simplifies later stages in the mapping process.

Figure 6 illustrates the translation process by showing a program in VAL and its translation to a DFG. The program takes a stream of $n$ data values, selects those values in the even-numbered positions of the stream, and calculates their average, their sum, and the minimum value among them.

The second stage in the process deals with the mapping of the DFG onto a (theoretically) unbounded, finite array of PEs. Here, the nonplanar graph is mapped onto a hexagonal array. Clearly, this stage depends on the specific topology of the grid, but it *does not* depend on technology limitations. By keeping this stage independent of the parameters influenced by these limitations (such as the chip size and pinout), we do not have to change the mapping process as a consequence of rapid technology developments in the fields of circuit density and chip size. These technology-dependent parameters will come into consideration in the later stages of the mapping process. We describe this stage and its related algorithms below.

After mapping of the DFG onto an unbounded PE array, the next stage partitions this array into several (subarray) chips according to the technology-imposed limitations on the number of PEs per chip. Performing this partition to achieve a minimal number of interchip connections has been shown to be an NP-complete problem. Therefore, we adopted the heuristic solution suggested by Krishnamurthy,[12] which starts from an arbitrary division of the array into two portions, then attempts to improve upon it by moving single elements from one side of the division to the other. When no further improvements are possible, the same process is recursively applied to each side of the final division. Obviously, we stop this process as soon as each group is fitted onto a single chip.

In the last stage of the mapping process, each PE in the array is physically assigned its task(s).

**Mapping the DFG onto an unbounded finite PE array.** First, we need to define certain terms. A *path* is a set of links in the array that form a continuous, directed chain between two PEs. The PE from which the path emanates is called the *source* of the path, and the PE where the path terminates is referred to as its *destination*. We define the *length* of a path as the number of links that make it up. We say that a path $p$ in the array *represents* an arc $a$ in the DFG if the node at the source (destination) of $a$ has been mapped onto the source (destination) of $p$.

**An optimal mapping would require only as many PEs as there are nodes in the DFG.**

We have implemented the DFG mapping process using the following four steps:

Step 1: Limiting the outdegree
Step 2: Dividing into layers
Step 3: Ordering in each row
Step 4: Routing paths

The mapping process is based on the routing ability of the PEs and attempts to minimize the number of additional PEs required for the mapping. An optimal mapping would require only as many PEs as there are nodes in the DFG. Additional PEs are required because we must find, for each arc in the DFG, a path to represent it in the array. This search is complicated by the nonplanarity of the DFG.

We next describe each of the steps in the DFG-map process. First, we illustrate these steps for the case where each PE performs a single operation (besides any routing duties assigned to it). Subsequently, we will take into account the ability to gather several operations into a single PE.

**Limiting the outdegree of DFG nodes.** The DFG generated during the translation stage of the mapping process had no limitations imposed by the array architecture. As a result, some of the nodes in the DFG have several output arcs (the number of such being the node's *outdegree*). In addition, DFG operators (nodes) take between one and three input tokens. However, the architecture we chose (a hexagonal array) limits the total number of links that can be used for input or output to six.

Furthermore, since most results flow downwards in the DFG, and since each PE has two neighbors in this direction, we limited the number of outputs from each node in the DFG to two. This limitation also facilitates use of a PE for both computational and routing tasks concurrently by employing the unused ports for routing.

The first step in the DFG-map process is thus the introduction of *Split* operators at the output(s) from DFG nodes with an outdegree larger than 2. For example, Figure 7 shows the DFG in Figure 6(b) after the addition of Split nodes (marked as SP).

**Dividing the DFG into layers.** The next step of the DFG-map process divides the nodes in the DFG into layers. Each of these layers will then be mapped onto a single row in the PE array.

Srini described a somewhat similar approach to layering,[13] using it for assigning nodes in a dataflow graph to processors in a task-level dataflow distributed computer system. The algorithm used in this step consists of a depth-first traversal of the DFG to mark backward-pointing arcs (that is, arcs that close loops). Experiments with a breadth-first approach led to consistently less satisfactory results.

This DFG traversal is then followed by a variant of topological sort on the DFG, which ignores these marked arcs. During the sorting process, which begins by labeling with level = 1 all these nodes in the DFG that receive their inputs from outside the array, each node $N$ is labeled with level = $i$, where $i$ is one greater than the highest-labeled node whose output(s) feed node $N$.

In Figure 7, we have marked the layers in the DFG by dashed lines.

**Ordering the nodes in a layer.** After dividing the nodes in the DFG into layers, we place each layer on a row of PEs in the array in an arbitrary order according to the initial placement (given by the compiler). As a preliminary step towards the definition of paths to represent all the arcs in the DFG, we now arrange the nodes in each row. Since we would like to keep paths as short as possible, an ideal order for the nodes in a row is one that minimizes the sum of these lengths. But, since at this point in the DFG-map process we have not yet determined the exact length of each path $p$, we use the *column distance*, denoted by $D_c(p)$, between the path's source and destination PEs. For example, if the source PE for a path is currently located in the third column of the array and its destination PE is in the tenth column (their row locations are not taken into account), then the column distance for the path is seven.

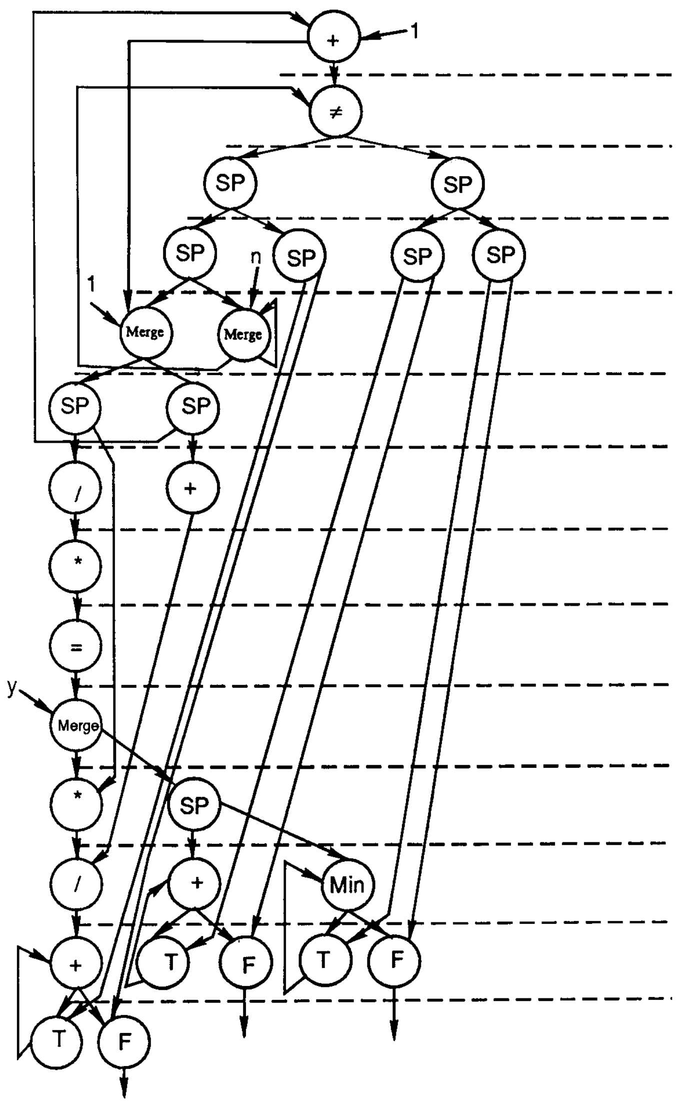


Figure 7. The DFG of Figure 6b with outdegree limited by 2.

Now, we can order the nodes in a single row $r$ so as to try to minimize the sum $\sigma_r = \Sigma_{p\epsilon P} D_c(p)$, where the set $P$ contains all those paths with either a source or destination PE in row $r$. We have found that a good heuristic for determining the place of a node $N$ within a layer is to put it in a column as close as possible to the center of mass for all the paths for which $N$ acts as either its source or destination. Each such path is assigned a weight equal to its column distance. Our criterion for evaluating a given ordering of nodes is the sum of the $\sigma_r$ for all the rows in the array.

The above makes it clear that the ordering of nodes within layers is an iterative process. We begin with the nodes arranged in the order in which they were found in the DFG. During each iteration we traverse the array, layer by layer, attempting to reduce the $\sigma_r$. In each layer, nodes are processed in the order in which they are currently found, and a new order is determined starting with an empty row. This processing involves, for each node in the layer, determination of the column closest to the node's center of mass. If another node already occupies this column, we look for the closest available column and place the node there.

Iterations are started alternately from the top and bottom rows, until either no improvement is possible or a maximum number of iterations is exceeded. From our experience, a good limit is 10, but usually less than five iterations suffice for convergence. We believe that this results from the high locality obtained by keeping language constructs as subgraphs in the DFG.

We have compared the length of paths resulting from ordering the array nodes using the above algorithm, with those achieved by using the order determined by the DFG (the starting point for the ordering algorithm). This comparison shows an average reduction (over several examples) of 46 percent in the average path length and also a drop of 52 percent in the length of the longest path.

**Building paths in the array to represent arcs in the DFG.** Once each node in the DFG has been assigned to a PE in the array, we have to determine how the arcs in the DFG will be represented by paths in the array. Links in the array *cannot* be shared by paths; that is, they can belong to at most one path representing a DFG arc.

We approach the problem in a piecemeal fashion, moving from PE to PE. At each PE we add a single link to one or more (incomplete) paths that go through it. A different approach, which attempts to build whole paths one path at a time, suffers from the disadvantage that in some cases a short path (such as one that connects two neighboring PEs) is lengthened because other paths occupy the needed link(s).

Now, at a given PE we must assign links to certain paths—those originating at the PE (meaning the PE is the source of the path) or those that are segments of a longer path. In the first case, we say that the path is a *full path*, whereas in the latter case we call it a *partial path*.

We process both full and partial paths according to their priorities. A priority is assigned to a path $p$ according to the heuristic formula $Priority(p) = A \times s + B \times n - [D_c(p) + D_r(p)]$, where $s = 1$ (0) if $p$ is a full (partial) path and $n = 1$ (0) if the destination for $p$ is (is not) a neighboring PE. We select the coefficients $A$ and $B$ so that

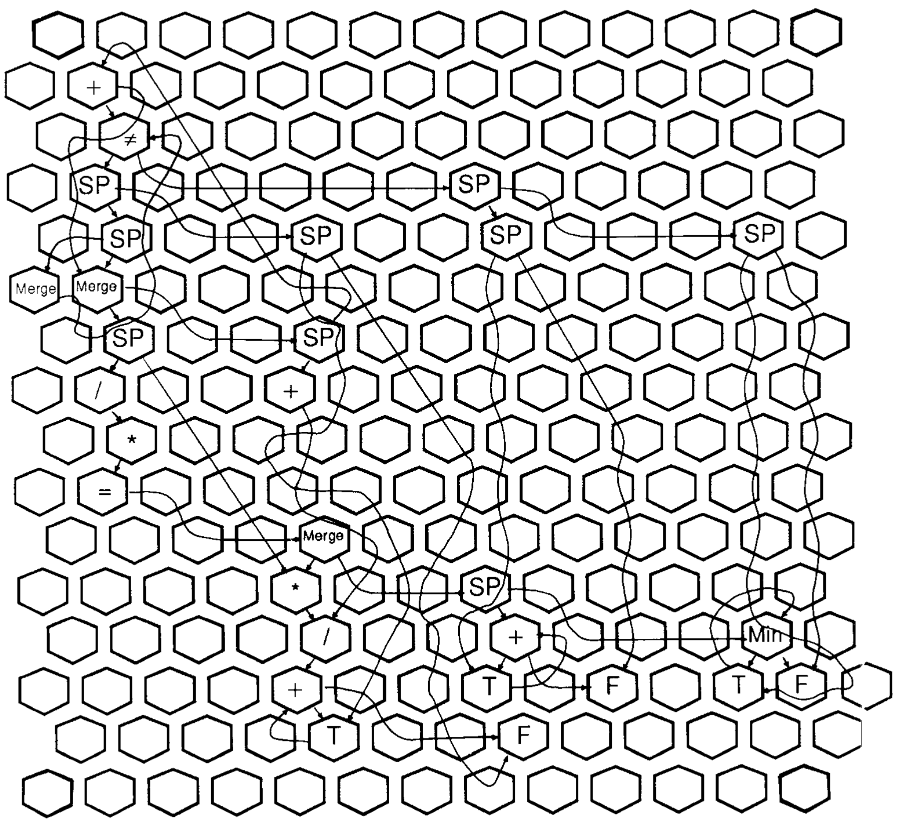


**Figure 8. Mapping of the DFG in Figure 7.**

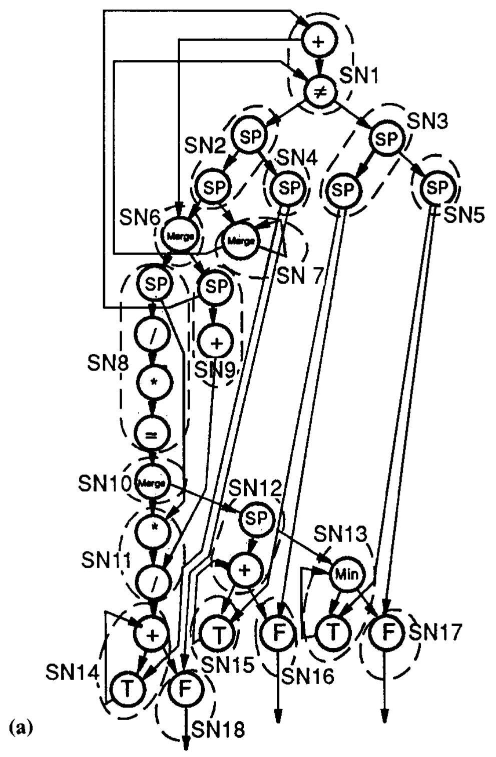
$A \ll B$ to give a higher priority to paths whose destinations are one of the neighboring PEs. $D_r(p)$ is the *row distance* for the path $p$, which is analogous to $D_c(p)$ when we consider rows rather than columns. The factor $[D_c(p) + D_r(p)]$ in the above formula results in a lower priority for longer paths.

Note that paths originating at a PE are never blocked, since we always check for link availability before attempting to route a path through a PE. Once we have selected a path for routing, we assign a link closer to the path's destination. If this link has already been assigned to another (higher priority) path, we select another available link.

In some cases, a situation develops where a large number of paths have to be routed through a small area in the array. This can cause a bottleneck at a given PE, because there are no available links to route one (or more) path(s). Rather than performing an extensive backtracking process to search for other possible links-to-paths assignments, we limit the amount of backtracking to change only the last decision made in the routing process. If this does not solve the problem, we add a row or column of PEs and use these free PEs to resolve the situation. From our experience, this case happens rarely and the waste of additional PEs for routing purposes remains small.

Figure 8 shows an example of applying the DFG-map process to the graph in Figure 7. This figure clearly shows that the PE utilization is very low—only 13.9 percent of the total number of PEs are utilized for computational tasks. One of the reasons for the low PE utilization is that we have not taken advantage of the capability of each PE to perform several operations, as supported by the chosen design. The easiest way to incorporate this capability into the DFG-map process is to redefine the DFG in terms of *supernodes*, each representing a number of (up to six) nodes in the original DFG, and to formally adopt the *generalized firing rules* presented by Khavi, Buckles, and Bhat.[10] These rules allow for the firing of the (super) node even though not all operands are present (as would be the case where some operands are input to one of the supernode's operations, while others are used by different operations).

Therefore, we add a *node compression* step that creates supernodes in the DFG. We refer to the DFG obtained by the intro-



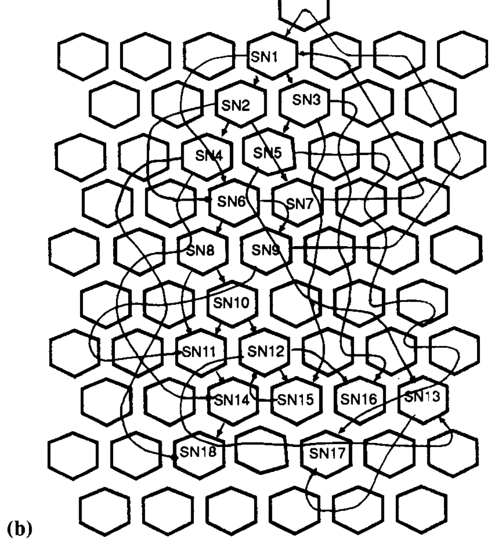


**Figure 9. A CDFG corresponding to the DFG in Figure 6b (a); its mapping onto a hexagonally connected array (b).**

**Table 1. Performance measurements for the DFG-map process.**

| | Conditional | | Random | | Spring-Mass | | Runge-Kutta | | Even Process | | Inner Product | | Average | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | DFG | CDFG | DFG | CDFG | DFG | CDFG | DFG | CDFG | DFG | CDFG | DFG | CDFG | DFG | CDFG |
| Number of rows | 13 | 11 | 7 | 4 | 8 | 3 | 22 | 19 | 16 | 10 | 6 | 4 | 12 | 8.5 |
| Number of columns | 8 | 7 | 4 | 3 | 3 | 3 | 13 | 10 | 13 | 7 | 4 | 4 | 7.5 | 5.7 |
| Number of utilized PEs | 25 | 14 | 9 | 5 | 12 | 5 | 50 | 28 | 31 | 17 | 12 | 6 | 23.2 | 11.7 |
| Percent of utilized PEs | 7.4 | 18.2 | 32.1 | 41.7 | 50.0 | 55.0 | 7.5 | 14.2 | 13.9 | 25.7 | 50.0 | 75.0 | 26.8 | 38.3 |
| Maximum P | 22 | 14 | 6 | 2 | 1 | 1 | 21 | 19 | 12 | 11 | 3 | 1 | 10.8 | 8 |
| Average P | 3.34 | 4.63 | 2 | 1.25 | 1.0 | 1.0 | 2.53 | 4.45 | 3.7 | 3.53 | 1.4 | 1.0 | 2.33 | 2.64 |

duction of supernodes as the *compressed dataflow graph*, or CDFG. As mentioned above, the number of operations in a PE is limited to six; thus, a supernode can contain up to six regular nodes. In creating supernodes, we take into account the fact that operations within a single PE cannot proceed in parallel, contradicting the basic dataflow approach and negating one of its main advantages. Therefore, we create supernodes by gathering nodes directly connected by arcs in the DFG (so that results from one node are used as operands by the other). Thus, the natural place in the DFG-map process for introducing node compression is immediately after the DFG has been partitioned into layers (refer to Figure 7).

Notice that the node compression step also eliminates from the DFG some Split nodes whenever both nodes fed by a Split node become a single supernode in the resulting CDFG. Once the CDFG is obtained, the last two steps in the DFG-map process proceed as usual, using the CDFG as input instead of the original DFG. Figure 9 shows the DFG of Figure

6b after the introduction of supernodes (9a) and its corresponding mapping (9b).

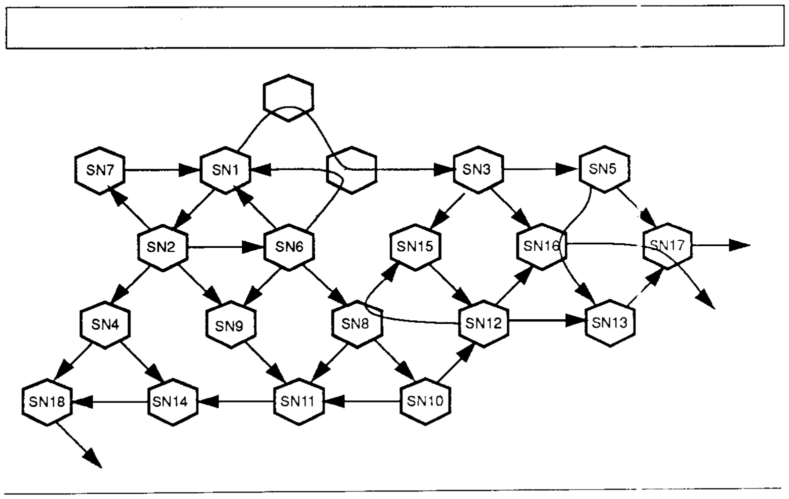


**Figure 10. An optimal mapping of the CDFG in Figure 9a.**

## Performance studies

We discuss two distinct performance measures in this section. One measure is the *efficiency* of the mapping process, meaning the size of the array required to map a given algorithm. In case the size of the array is given, the efficiency can be expressed as the array *utilization*, or the portion of the array actually used by the DFG. The second measure is the expected *computation time* of the data-driven array, which is the time required to execute the algorithm mapped onto the data-driven array.

We begin by examining the array utilization profile for several programs.

**Array utilization.** We chose a set of programs to demonstrate the mapping process. These programs appear in Table 1, where "Conditional" is a nested If expression from Brock and Montz,[14] "Random" is a simple (additive) random number generator, "Spring-Mass" is the Spring-Mass system example presented in Figure 1, "Runge-Kutta" is the Runge-Kutta program, "Even Process" is the example in Figure 9, and "Inner Product" is the sum of two inner products of four independent vectors.

Several parameters appear in Table 1: the number of rows and columns occupied by the mapping (their product is the array size in number of PEs), and the overall array utilization in number and percentage of PEs that perform computations. Also shown are the length of the longest path in each graph (Maximum P) and the average path length (Average P). These measures indicate the propagation delays incurred when transmitting operands among PEs.

Table 1 shows the impact of using either the DFG or the CDFG on the array utilization. Clearly, using the CDFG instead of the original DFG results in smaller array size and higher overall PE utilization. The large variation in PE utilization results from routing conflicts.

Also notice that the longest path becomes shorter, even though in some cases the mapping process increases several paths. This phenomenon simply reflects the increased difficulty in routing paths in the CDFG, since some of the links originally available for routing in the DFG have been used for intermediate results in the supernodes. Average path length tends to grow for the CDFGs in some cases. The reason for this apparent problem is that some of the short paths (which caused a lower average in the DFG case) are eliminated from the graph by the node compression step.

The utilization results indicate that automatic mapping is feasible and yields reasonable results, even though it is not optimal and we have more efficient ways to perform the mapping. For example, Figure 10 depicts an "optimal" mapping of the CDFG in Figure 9a. This mapping, done by hand, yields a higher array utilization.

**Table 2. PE operation timings (in clock cycles).**

| Operation | Clock Cycles |
|---|---|
| Route | 2 |
| Split | 2 |
| Merge | 3 |
| Magnitude (<, >,...) | 3 |
| Addition | 3 |
| Subtraction | 3 |
| Multiplication | 11 |
| Division | 25 |

**Array performance.** One can consider several measures for array performance. The data-driven array is mostly useful for repetitively executed algorithms. Therefore, the most important performance measure is the *throughput*, the rate at which results are produced. To compare the performance of the data-driven array to that of a sequential processor, we employ the *speedup* and *processor utilization* metrics. Speedup is the ratio between the execution time of an algorithm on a sequential processor and the execution time of the same algorithm on the array. Processor utilization indicates what portion of time each of the working PEs is actually used.

We examined the expected performance on the data-driven array of the algorithms in Table 1. The timing of PE operations (computation and routing) shown in Table 2 is based on the microcode developed by Peled.[11]

Table 3 compares the execution time of the six algorithms using either the original DFGs or the corresponding CDFGs. Table 4 then compares the execution time of those algorithms on the data-driven array and on a sequential processor.

From Table 3 we can conclude that neither the time for the first result to emerge from the array nor the interval between subsequent results are strongly influenced by the node compression. This agrees with

**Table 3. Array performance—DFG versus CDFG (in clock cycles).**

| | Conditional | | Random | | Spring-Mass | | Runge-Kutta | | Even Process | | Inner Product | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | DFG | CDFG | DFG | CDFG | DFG | CDFG | DFG | CDFG | DFG | CDFG | DFG | CDFG |
| First result | 50 | 45 | 69 | 69 | 88 | 84 | 414 | 404 | 132 | 128 | 60 | 58 |
| Next result | 14 | 12 | 26 | 26 | 26 | 26 | 372 | 370 | 80 | 76 | 48 | 48 |

**Table 4. Performance comparison (in clock cycles).**

| | Conditional | | Random | | Spring-Mass | | Runge-Kutta | | Even Process | | Inner Product | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 68000 | Array | 68000 | Array | 68000 | Array | 68000 | Array | 68000 | Array | 68000 | Array |
| $(\text{Throughput})^{-1}$ | 93 | 12 | 234 | 26 | 425 | 26 | 691 | 370 | 386 | 76 | 202 | 48 |
| Speedup | | 7.7 | | 9.0 | | 16.3 | | 1.9 | | 5.1 | | 4.2 |
| Number of PEs | | 9 | | 4 | | 5 | | 23 | | 18 | | 6 |
| PE utilization | | 0.86 | | 2.25 | | 3.26 | | 0.08 | | 0.28 | | 0.7 |

intuition, since supernodes were created by merging nodes forming a chain of operations dependent on one another. It is also encouraging to see that these supernodes do not have an adverse effect on performance. In fact, they cause a slight improvement by reducing the longest path length. The main advantage of using the supernodes is the smaller array size needed to contain the CDFG, as opposed to the original DFG.

Table 4 compares the potential performance of the data-driven array to that of a sequential processor to illustrate the effect of pipelining and parallelism in the data-driven array. For this comparison, we manually mapped the CDFG onto the array instead of applying the above mapping algorithm. The table includes the execution time (in clock cycles) of the same six algorithms on the array and on the 68000 microprocessor.

No commercially available processor is fully suitable for comparing the performance of the data-driven array to that of a sequential processor. However, to obtain some quantitative comparison, even if not a completely satisfactory one, we selected the commonly used 68000 microprocessor. This microprocessor also executes 8-bit operations, and the number of cycles per instruction in the 68000 and in our PE are about the same for most instructions. The only difference is in the multiply and divide operations. To obtain a reasonable comparison, we took multiply and divide execution time as half of the actual time (35 cycles instead of 70 for the multiply and 70 cycles instead of 140 for the divide).

Three performance measures appear in Table 4: $(\text{Throughput})^{-1}$, speedup (the number of cycles needed to produce a result on the array divided by the number of cycles needed to compute the same result on the 68000), and processor utilization (speedup divided by the number of PEs in the array).

The algorithms that we can map onto the data-driven array can be divided into two major classes: *noniterative* and *iterative*. Iterative algorithms have corresponding DFGs with feedback edges, while noniterative algorithms have no cycles. The first three examples in Table 4 are noniterative and the last three are iterative algorithms. The highest speedup and PE utilization are achieved by the iterative algorithms. The introduction of cycles into the DFG, as in the last three examples of Table 4, results in a significant reduction in speedup. The reduction is due to the fact that in iterative algorithms maximum pipelining cannot be achieved.

There are two types of iterative structures: *dependent* and *independent*. In a dependent structure, the computation within the iteration body depends on the previous iteration result. The dependency point can be at the beginning of the iteration body (such as Runge-Kutta) or at some internal point (such as Even Process). In the first case only parallelism in the body can be exploited, while in the second case some pipelining between successive iterations is feasible. In an independent structure (such as Inner Product), both parallelism and pipelining can be exploited, resulting in higher throughput and, consequently, higher speedup. In these algorithms, as the body part increases in complexity, a higher speedup can be achieved.

From the above discussion we can conclude that high speedup and PE utilization can be achieved when using the data-driven array. A sequential processor has a large overhead due to operand handling (memory references and temporary storage) and branching. We avoid this overhead in the data-driven array and consequently can obtain a PE utilization greater than one.

The main characteristics of two well-known types of processor array architectures (namely, systolic arrays and wavefront arrays) have been discussed. One can expect high throughput when executing highly regular algorithms on these arrays. This leads to the conclusion that a different type of architecture can better suit algorithms that lack inherent regularity. We have presented one such architecture, VLSI data-driven arrays, in this article. The data-driven array is substantially more flexible than previously suggested architectures and is, therefore, more suitable for high-speed execution of arbitrary algorithms.

From the examples in the previous section, we can conclude that the algorithms that achieve a high speedup when executed on the data-driven array are those with intensive arithmetic computations and with either no cycles or independent cycles.

We can expect lower speedup and/or processor utilization when mapping dependent structures onto the data-driven array. In such cases, we need to carefully examine the cost-effectiveness of using this array.

Further work is underway to finalize the design of the array and, more importantly, to develop more efficient mapping algorithms that will achieve higher speedup ratios and better utilization of the processor array.□

## Acknowledgments

This work was supported in part by SRC under contract 88-DJ-125.

## References

1. J.A.B. Fortes and B.W. Wah, eds., "Special Issue on Systolic Arrays—From Concept to Implementation," *Computer,* Vol. 20, No. 7, July 1987.
2. A.L. Fisher et al., "Design of the PSC: A Programmable Systolic Chip," *Proc. Third Caltech Conf. VLSI,* March 1983, pp. 287-302.
3. S.Y. Kung et al., "Wavefront Array Processors—Concept to Implementation," *Computer,* Vol. 20, No. 7, July 1987, pp. 18-33.
4. I. Koren and G.M. Silberman, "A Direct Mapping of Algorithms onto VLSI Processor Arrays Based on the Data Flow Approach," *Proc. 1983 Int'l Conf. Parallel Processing,* Aug. 1983, pp. 335-337.
5. J.B. Dennis, "Data Flow Supercomputers," *Computer,* Vol.13, Nov. 1980, pp. 48-56.
6. W.B. Ackerman and J.B. Dennis, *VAL—A Value-Oriented Algorithmic Language: Preliminary Reference Manual,* MIT Laboratory for Computer Science Technical Report, MIT/LCS/TR-218, Cambridge, Mass., June 1979.
7. J.A. Abraham et al., "Fault Tolerance Techniques for Systolic Arrays," *Computer,* Vol. 20, No. 7, July 1987, pp. 65-75.
8. L. Snyder, "Introduction to the Configurable, Highly Parallel Computer," *Computer,* Vol. 15, No. 1, Jan. 1982, pp. 47-56.
9. B. Mendelson and G.M. Silberman, "Mapping Data Flow Programs on a VLSI Array of Processors," *Proc. 1987 Int'l Conf. Computer Architecture,* June 1987.
10. K.M. Kavi, B.P. Buckles, and U.N. Bhat, "A Formal Definition of Data Flow Graph Models," *IEEE Trans. Computers,* Vol. C-35, No. 11, Nov. 1986, pp. 940-948.
11. I. Peled, "A Data-Driven Processing Element," MSc thesis (in Hebrew), Dept. of Electrical Engineering, Technion, 1986.
12. B. Krishnamurthy, "An Improved Min-Cut Algorithm For Partitioning VLSI Networks," *IEEE Trans. Computers,* Vol. C-33, No. 5, May 1984, pp. 438-446.
13. V.P. Srini, "A Fault-Tolerant Dataflow System," *Computer,* Vol. 18, No. 3, March 1985, pp. 54-68.
14. J.D. Brock and L.B. Montz, "Translation and Optimization of Data Flow Programs," *Proc. 1979 Int'l Conf. Parallel Processing,* Aug. 1979, pp. 46-54.

**Israel Koren** is currently a professor in the Department of Electrical and Computer Engineering at the University of Massachusetts at Amherst. His current research interests are fault-tolerant VLSI and WSI architectures, models for yield and performance, floor planning of VLSI chips, and computer arithmetic.

Koren is chair of the 1988 IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems and a guest editor for a special issue of *IEEE Transactions on Computers* on high-yield VLSI systems. He is also a senior member of the IEEE.

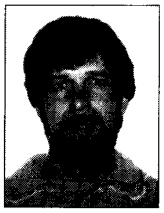
Koren received the BS, MS, and DSc degrees from the Technion, Israel Institute of Technology, Haifa, in 1967, 1970, and 1976, respectively.

**Bilha Mendelson** is a PhD student in the Department of Electrical and Computer Engineering at the University of Massachusetts at Amherst. Her current research interests include dataflow and parallel and distributed systems.

Mendelson received the BS and MS degrees in computer science from the Technion, Israel Institute of Technology, Haifa.

**Irit Peled** is currently a visiting scholar at Stanford University. Her current research interests include VLSI architecture, dataflow modeling, circuit design, and software design.

Peled received a BS in electrical engineering from Tel Aviv University in 1976. She spent three years in VLSI design at Intel Israel and received an MS in electrical engineering in 1986 from the Technion, Israel Institute of Technology, Haifa.



**Gabriel M. Silberman** serves on the faculty of both the computer science and electrical engineering departments at the Technion, Haifa, Israel. He is currently a visiting associate professor at the Department of Electrical and Computer Engineering at Carnegie Mellon University. His current research interests include computer architecture, operating systems, VLSI design verification, test generation, and fault simulation.

Silberman is a member of the Association for Computing Machinery, the IEEE, and the IEEE Computer Society. He received BS and MS degrees in computer science from the Technion, Israel Institute of Technology, Haifa, and a PhD in computer science from the State University of New York, Buffalo.

Readers may write to Koren at the Dept. of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003.