Proc. of the 1983 Conference on Parallel Processing.

A DIRECT MAPPING OF ALGORITHMS ONTO VLSI PROCESSING ARRAYS BASED ON THE DATA FLOW APPROACH

Israel Koren

Computer Science Division
University of California
Berkeley, CA 94720
on leave from the
Dept. of Electrical Engineering
Technion - Haifa 32000, Israel

Gabriel M. Silberman

Dept. of Computer Science
Technion
Israel Institute of Technology
Haifa 32000, Israel

ABSTRACT

A new approach to the utilization of VLSI processing arrays by means of the algorithms running on them is presented. The idea is to represent algorithms as *data flow graphs*, and then map these graphs onto the array. This approach obviates the need to develop new concurrent algorithms to utilize the parallelism inherent in the array, while offering a general environment for the realization of algorithms on semi-custom VLSI.

1. INTRODUCTION

The approach taken in this research to achieve parallelism within a special purpose VLSI chip, without developing new concurrent algorithms, is the *data flow* approach [3-5]. In it, concurrency of activities is achieved at the lowest possible level by treating each machine instruction as an independent activity. This enables "fine grain parallelism" [3], not achievable when scheduling and synchronization of concurrent activities are controlled by software.

However, we do not propose to use one of the known general-purpose architectures of data flow machines [3-5]. Instead, we suggest to map the data flow graph which describes the problem in hand, on a regular array implemented in VLSI. These regular arrays of identical cells take considerably less time to design and manufacture [1,2]. Also, the mapping should not be fixed but changeable, enabling the user to map various data flow graphs (algorithms) on the same chip. Regularity and flexibility are thus obtained, increasing the number of potential applications for the chip and thereby making it more appealing to the semiconductor industry.

In the following we consider the hexagonal array as a basis for illustrating our approach. This array has a flexible structure [1,6], simplifying the task of mapping. In addition, fault-tolerance may be introduced into it [6,7] allowing it to recover from errors by reconfiguration. We then propose an architecture for the processing element (PE) which constitutes the basic cell in the array. Also presented is an outline of the general graph-to-array mapping process.

2. PRELIMINARIES

In contrast to control flow computers, data flow computers have no program counter. In the latter, an instruction is ready for execution when all its operands have arrived. Consequently, all such instructions may be executed in parallel. If the processing capabilities of the data flow computer are sufficient, the highest degree of parallelism may be achieved.

The program is represented by a data flow graph. The vertices correspond to operators, and data tokens move along the arcs. Parts of the graph may have to be executed iteratively. This might cause tokens to accumulate on certain arcs and result in the presence of tokens belonging to different iteration steps at the input arcs of an operator. This problem may be solved by either labeling (coloring) the tokens [4] or by preventing the accumulation altogether [3]. The latter is achieved by preventing an operator from producing a new output token until the previous one has been consumed [3,8]. This approach still enables pipelining through the data flow graph.

Maximum pipelining is not however, always possible. Bottlenecks may appear in parallel segments of the graph [8] (e.g., paths of a conditional expression), thus severly limiting the amount of concurrency. To eliminate these bottlenecks an optimization technique has been suggested in [8], inserting buffers (delay operators) in some of the parallel paths. However, these buffers may result in either an increase in the overall delay through the pipeline or a reduction in the throughput [8].

In the architecture suggested here dynamic length FIFO queues are employed. In this way, the level of concurrency is increased without the penalty of an increase in the overall delay. The labeling scheme as presented in [4] might be inappropriate for our purposes due to the additional hardware complexity.

3. PE ARCHITECTURE AND PRINCIPLES OF OPERATION

The basic PE, shown in Figure 1, is connected to its six immediate neighbors by dedicated busses, in an hexagonal processor array. The PE contains six registers $x_0, x_1, \cdots, x_5$ which are connected to the six communication busses. Each of these registers can either receive or transmit tokens and will accordingly be defined either as a primary input register or a primary output register.

Each basic PE contains in addition, an arithmetic and logic unit (ALU) and a *Pseudo Associative Memory unit* (PAM). Each location within the PAM contains a key and a data element. The PAM has therefore, two input registers k_i (key-in) and d_i (data-in), and two output registers k_o and d_o (Figure 1).

The ALU is capable of performing the basic arithmetic and logic operations. Its inputs may be connected to any primary input register, or to the PAM data-out register d_o. The ALU result is routed to either a primary output register or the PAM data-in register d_i.

The overall function of the PE is specified by the designation of each of the x_i registers as primary input or output register, by the internal connections of these registers to the ALU and the PAM unit registers, and by the operation performed by the ALU. Thus, the operation of a PE may be defined by a set of statements like

$$PE : \left\{ \begin{array}{ll} d_i := x_o + x_1 & x_4 := d_o \\ x_3 := x_1 & x_5 := x_2 \end{array} \right.$$

The PAM unit has two modes of operation, *First-in First-out* (FIFO) mode and *Associative* mode. In the FIFO mode the PAM unit serves as an input or output buffer for token accumulation. In the associative mode the PAM serves as a queue in which a key is attached to each data element. This mode of operation is useful when implementing recursion and in it data elements are accessed by using keys instead of addresses. However, a fully associative memory is not necessary. Instead, a sequential access memory unit with added logic can be employed, using shift registers, CCD's, or magnetic bubbles.

In the FIFO mode of operation the PAM unit may serve as a queue for accumulating either incoming or outgoing tokens. The purpose of this FIFO queue is to dynamically equalize the length of parallel paths in the graph in order to achieve maximum pipelining. The fixed capacity of the PAM might limit the maximum length of the FIFO queue. However, the PAM units in two or more neighboring PEs may be chained and used for accumulating tokens from a single source.

For the sake of brevity, the exact principles of operation of the proposed PAM unit are not detailed here.

4. IMPLEMENTING BASIC DATA-FLOW STRUCTURES

This section shows how to use an array of PEs in the implementation of data-flow structures. We begin by examining the basic data-flow elements, which can be directly mapped onto a single PE. These are the *Arithmetic and Logical Operators* (like addition, negation, And, complement, etc), and the *Conditional Operators* (like arithmetic comparisons, test for zero, etc). Also requiring only a single PE are the *Flow Control Operators*. They do not affect the contents of the token, but rather its progress and/or destination.

The simplest such operator is *Copy* (denoted by C). It makes two identical copies of its input token.
Merge (M) operator is used when a data token may come from two different sources (paths), and it is to be merged into a single path for further processing. This operator is also capable of producing a Boolean token whose value depends on the input path which supplied the token for the output.
The *Router* (R) operator receives two inputs, a data token and a logical value. The data token is copied into exactly one of two output registers, depending on the value of the logical input. This is analogous to "distribute" in [9].
The *Gate* (G) operator transfers the incoming token to an output register if a second token is present at another input register. The G and M operators may be used to implement the "Select" operation [9].
Self-Iterating Operator (L) is used in those cases where the result produced by the PE is immediately used as argument to the next operation. This saves the need to create "external" loop structures (e.g., [10]).

Figure 2 depicts a data flow graph which calculates the factorial function, using C, M, R and L operators. Notice the labeling of the outputs of the R operator by T and F. This is used to specify which output path corresponds to each logical value. The *L* operator receives two inputs, one being the initial value for the iteration (in this case n), and the other a Boolean value which determines, for each iteration, whether to load a new initial value, or use the one from the latest iteration.

Having defined the basic data-flow elements, we now show how these may be combined to yield the basic data-flow *structures*.

4.1. Conditional (if-then-else) -

This construct, has the general format:

```
    if <condition> then <expression1>
                   else <expression2>,
       endif
```

and is evaluated as <expression1> *or* <expression2>, depending on the logical value of <condition>. The above statement may be implemented in general as shown in Figure 3.

Notice that when a certain branch of the conditional is taken, the tokens corresponding to the other branch are not produced at all. This is achieved by using an R operator with only one output; this way tokens corresponding to different computations are not mixed.

We also deal here with the problem of keeping in correct sequence the results being produced by a conditional construct. The ordering is achieved by using an extra R and two Gs as shown in Figure 3. The initial token present at the R input is routed to the G in the appropriate branch of the conditional, thus allowing only its result to flow through. When a result arrives at the M, a token (its value is immaterial) is recirculated to the R to enable further output tokens.

Note that conditional constructs may result in token accumulation, because of different path lengths between the two branches. Here we can benefit from the PE's capability of dynamically adjusting to token traffic, by using the FIFO mode of the PAM.

4.2. Iterative (Do-While, Repeat-Until) -

Iterative data-flow constructs make use of conditionals much in the same way traditional programming languages do. In general, we have the two iterative constructs, *do-while* and *repeat-until*, depending on whether the test for loop repetition is placed before or after the loop body, respectively. Figure 4 shows how a repeat-until loop is used to approximate the square root of a (positive) value c, using Newton's iterative method.

4.3. Recursion -

This construct is by far the most involved in the data-flow context. Actually, most current data-flow architectures do not handle recursion at all. However, recursion is generally recognized as a good programming technique. When used, it leads in many cases to simpler and shorter algorithms which are easier to understand and to prove correct.

For the sake of brevity we do not present here the way recursion is implemented, we would like however to indicate that any possible implementation of the recursion construct will be substantially more complex than all previous ones. The benefits of its use should therefore, be carefully examined before incorporating it.

5. DATA FLOW GRAPH MAPPING ALGORITHM

In the following, we show a simple (by no means optimal) scheme for mapping complete data flow graphs onto an hexagonally connected PE array. The mapping algorithm presented is executed externally by some host, and the results are then fed into the array for distribution. The graph mapping process is clearly dependent on the array topology. Therefore, different such topologies result in different mapping algorithms. Nevertheless, they all must tackle the same problem, namely, the non-planarity of the graph, arising from both ordering of operands and iterative constructs (loops).

We begin by assigning levels to the vertices (operators), where an operator (mapped on a PE) is at level i if all its operands come from operators at level $i-1$ or above. Clearly, our objective is to find minimal levels for all operators. In the case of loops, we do not consider the target of the loop to be a descendant of the source.

A second pass is now made to insure that no two operators which are either a loop source or target, are at the same level. If this is the case, the level is split until the condition no longer exists. The reason for this is to enable the use of the horizontal busses between PEs for connecting the source to the target.

In the next step of the mapping we connect the operators in the various levels. The outputs of level i have to be ordered so as to fit the inputs to level $i+1$.

After ordering the operands, (possibly by introducing extra levels which exchange operands), we connect the loop source with its target by using the horizontal connections between PEs. We first route the operand from the source to the boundary of the current graph. Then, we route it to the level of the loop target. Finally, we use again the horizontal connections to reenter the graph up to the target operator.

An example of the mapping of the factorial function from Figure 3 is shown in Figure 5(a). After the mapping process, has been completed, reduction techniques may be applied to reduce the size of the final mapping. For example, two levels may be collapsed into one, taking advantage of unused horizontal connections. Such a reduction procedure, when applied to the example in Figure 5(a), results in the mapping shown in Figure 5(b).

The mapping algorithm described above is by no means optimal and the only purpose it serves is to show the feasibility of mapping arbitrary data flow graphs onto hexagonal arrays. More efficient mapping algorithms for hexagonal arrays as well as for other array topologies are clearly needed.

Once the mapping algorithm is completed, we have to convey its results toward every relevant PE in the array. A simple way of doing this is to input a "configuration string", each component of which is addressed to a specific PE, and contains the setup information for it. Another possibility that is being investigated, is to execute the mapping algorithm within the array itself in a distributive fashion.. This may enable a dynamic mapping, allowing PEs which have completed their current processing task, go into a configuring phase, change their function and execute another part of the data flow graph. Such a dynamic mechanism may allow the mapping of larger data flow graphs on a given VLSI chip. It will also facilitate the handling of faulty PEs and/or connections.

6. CONCLUSIONS

The idea of directly mapping an arbitrary algorithm on a VLSI array has been shown to be feasible. However, further research has to be carried out before the effectiveness and practicality of this approach are established.

Clearly, not all algorithms that can be mapped on a array will use it effectively. Some algorithms may require a too large chip area, other may not execute fast enough. Consequently, methods have to be developed for estimating the chip area that will be used by a given algorithm, and its execution time.

REFERENCES

[1] C.Mead and L.Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA, 1980.

[2] M.J.Foster and H.T.Kung, "Design of Special-Purpose VLSI Chips: Examples and Opinions," *Proc. of the 7th Symp. on Comp. Arch.*, April 1980, pp.300-307.

[3] J.B.Dennis,"Data Flow Supercomputers," *Computer* Vol.13, Nov.1980, pp.48-56.

[4] I.Watson and J.R.Gurd,"A Practical Data Flow Computer," *Computer*, Vol.15, Feb.1982, pp.51-57.

[5] A.L.Davis,"The Architecture and System Methodology of DDM1," *Proc. 5th Symp. Comp. Arch.*, April 1978, pp.210-215.

[6] D.Gordon, I.Koren, and G.M.Silberman,"Embedding Tree Structures in Fault-Tolerant VLSI Hexagonal Arrays," submitted for publication.

[7] I.Koren,"A Reconfigurable and Fault-tolerant VLSI Multiprocessor Array," *Proc. of the 8th Symp. on Comp. Arch.*, May 1981, pp.300-307.

[8] J.D.Brock and L.B.Montz,"Translation and Optimization of Data Flow Programs," *Proc. of the 1979 Int'l Conf. on Parallel Processing*, Aug.1979, pp.46-54.

[9] A.L.Davis and R.M.Keller,"Data Flow Program Graphs" *Computer*, Vol.15, Feb.1982, pp.26-41.

[10] Arvind and K.P.Gostelow,"The U-Interpreter" *Computer*, Vol.15, Feb.1982, pp.42-49.

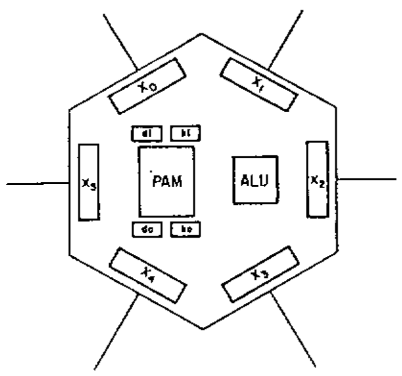


Fig. 1: The basic processing element.

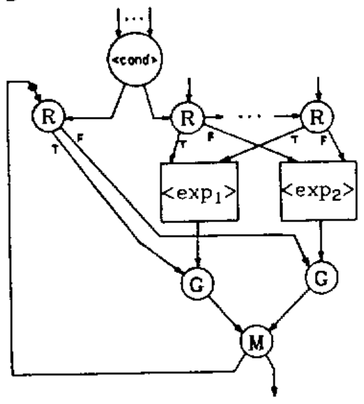
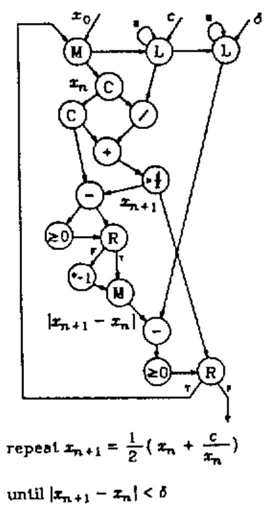


Fig. 3: Conditional construct.



x_0
c
δ
M
L
x_n
C
/
+
x_{n+1}
≥0
R
$|x_{n+1} - x_n|$
repeat $x_{n+1} = \frac{1}{2}(x_n + \frac{c}{x_n})$
until $|x_{n+1} - x_n| < \delta$

Fig. 4: The Newton method.

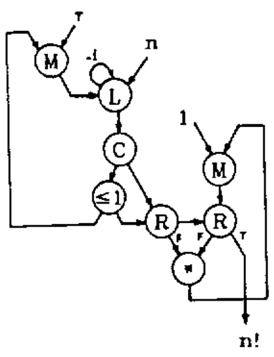


Fig. 2: The data flow graph for the factorial function.

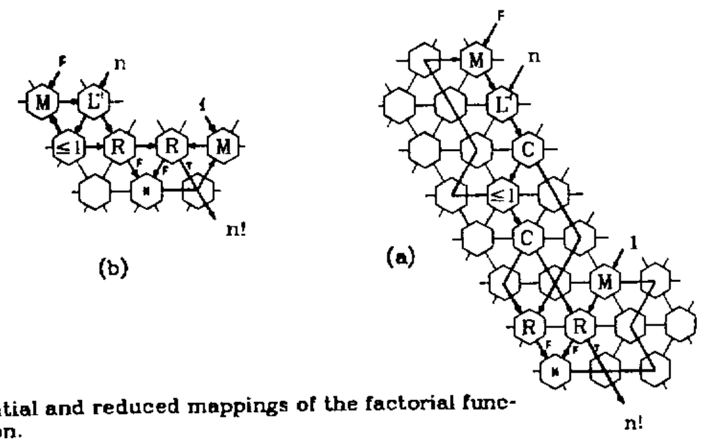


Fig. 5: Initial and reduced mappings of the factorial function.