ANALYSIS OF ALTERNATIVES FOR A SINGULAR VALUE DECOMPOSITION PROCESSOR

Jaime Moreno

October 1985
CSD-850035

UNIVERSITY OF CALIFORNIA

Los Angeles

Analysis of Alternatives for a Singular Value Decomposition Processor

A thesis submitted in partial satisfaction of the
requirements for the degree of Master of Science
in Computer Science

by

Jaime H. Moreno

Report No. CSD-850035[†]

1985

[†]The publication of this report has been supported in part by the Office of Naval Research, Contract N00014-83-K-0493 "Specifications and Design Methodologies for High Speed Fault-Tolerant Algorithms and Structures for VLSI"

A mi esposa Marisa,
por acompañarme en este camino

A mis padres,
ellos me mostraron que existían
caminos

To my wife Marisa,
for coming with me in this journey

To my parents,
they showed me that journeys exist

ACKNOWLEDGEMENTS

I wish to express my gratitude to all persons who made it possible for me to accomplish this thesis.

My special thankfulness to my advisor, Prof. Tomás Lang, for his constant dedication to this research, his encouragement when things did not seem fine, his constructive ideas and suggestions. Without his help this thesis never would have been what it is.

My appreciation also to Prof. Vance Tyree, for his permanent interest in the topic and his suggestions to improve it.

Finally, I want to thank the United Nations Development Program who sponsored me during these two years. Their financial support has been essential for my studies.

ABSTRACT OF THE THESIS

Analysis of Alternatives for a

Singular Value Decomposition Processor

by

Jaime H. Moreno
Master of Science in Computer Science
University of California, Los Angeles, 1985
Professor Tomás Lang, Chair

In this thesis we present an evaluation of different alternatives for the implementation of a digital system to compute the Singular Value Decomposition (SVD), in terms of the throughput achievable with a given amount of hardware. An algorithmic model which captures the properties of the SVD computation and a methodology for the design of systems exploiting concurrency are presented and applied to the SVD.

The model uses a directed graph as a description of the algorithm, where nodes correspond to subcomputations and arcs to precedences among the subfunctions. Each node is described by its execution time as a function of the number of operation units used. This model is utilized to evaluate the cost and performance of implementations with replication, pipelining and parallelism.

The methodology for the design is essentially an iterative procedure consisting of top-down decomposition and bottom-up refinement of the nodes in the graph of the algorithm.

It is shown here that, for throughput higher than a certain value which depends on the computation, a pipelined approach for the implementation of the SVD algorithm is more attractive than other alternatives currently proposed for such computation. The architecture devised is a multilevel pipelined processor, which exploits concurrency at several levels through internal pipelines and the use of the parallelism in the subcomputations. This scheme offers better performance characteristics for a given amount of hardware than the other alternatives proposed, keeps the realization

at a level of complexity similar to those other alternatives, and is able to compute the decomposition of matrices of any size without hardware modifications. However, implementations with the scheme presented exhibit expansibility characteristics such that they can be upgraded if higher throughput for larger matrices is desired.

The algorithm used is a parallel version of Hestenes' one sided orthogonalization method proposed by Brent et al, which is adapted for implementation with few processors. The resulting scheme has the same characteristics of the original one regarding data transfers between units, and it is realizable with any number of parallel processors.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

CHAPTER 1
INTRODUCTION AND MOTIVATION.

1.1 The Singular Value Decomposition and its Applications.

In modern science and engineering a frequently used mathematical tool is matrix algebra, because data can often be represented by matrices and vectors and processing this data as required by the applications corresponds to operations between matrices and vectors. Examples exist in image processing, system theory, signal processing, pattern recognition and other fields [Klema80, Andre76a, Golub77]. For some of these fields (i.e. signal processing), it has been shown that the major computational requirements when processing the related data can be reduced to a common set of basic matrix operations [Speis80].

Several of the more elaborate processing functions required are actually matrix transformations and decompositions, where a given matrix is transformed into one or more matrices with specific properties. Those functions are useful in varied applications [Golub77, Klema80, Andre76a]. Examples are the Householder transformation to a bidiagonal form, the QR decomposition of a square matrix, the reduction of a square matrix to its Jordan canonical form, the LU decomposition, the LL^t (Choleski) decomposition, and the Singular Value Decomposition (SVD), among others.

Basically, the Singular Value Decomposition [Nash79] of an $m \times n$ matrix A consists of finding U, Σ and V such that

$$A = U \Sigma V^t$$

where U is an $m \times m$ orthogonal matrix, V is an $n \times n$ orthogonal matrix, and Σ is an $m \times n$ matrix with non-negative elements on the main diagonal and zeros everywhere else. It is assumed that $m \geq n$, that is A has at least as many rows as columns.

In matrix algebra as well as in many applications, the evaluation of singular values and singular vectors is of theoretical and practical interest. At the theoretical level, the singular value decomposition (SVD) yields a fundamental representation theorem that describes the basic properties of A . At the practical level, SVD is useful for the solution of overdetermined linear systems of equations, for the solution of least-squares problems, for the computation of eigenvalues and eigenvectors, among many other applications. It is generally acknowledged [Klema80] that the SVD is the

only generally reliable method for determining rank numerically; this is a crucial requirement in many situations.

Many modern signal processing tasks in communications, radar, sonar, speech, image processing, linear systems theory and others use the SVD [Golub77, Klema80, Andre76a]. These applications have been devised around the SVD, as a result of its particular characteristics. Speiser and Whitehouse [Speis80] have shown that the SVD could be used in adaptive filtering and data compression. Andrews and Patterson [Andre76a] demonstrated the use of the SVD in image enhancement and restoration. Huang and Narendra [Huang75] and Shim and Cho [Shim81] did the same, although their examples had less resolution (implying smaller data sets and therefore smaller matrices). Andrews and Patterson also applied the SVD to image coding [Andre76b]. Other applications available for this mathematical tool are rank estimation and pseudo-inverse computation [Golub77].

Recently a flood of new applications have been suggested and/or proved. Miao and Chen [Miao84] used the SVD in 2-D spectral estimation to extend some 1-D techniques. Konstantinides and Yao [Konst84] considered the evaluation of the order of a linear system transfer function; they used the SVD for efficient determination of the rank of a matrix, applying the SVD to system modeling in signal processing. Sullivan and Liu [Sulli84] extrapolated a band-limited signal in discrete-time by solving an underdetermined system of linear equations with the SVD. Sibul [Sibul84a] required this tool to deal with the ill-conditioned data arising in adaptive beamforming. Zhou et al [Zhou84] suggested the use of SVD and singular vectors for a discrete-time, discrete-frequency model for image restoration.

This increasing use of the singular value decomposition is not surprising. Klema et al [Klema80] predicted such a situation when they stated that, within five to ten years, SVD would be one of the most important and fundamental tools in many fields. The list of applications and areas is even longer than what has been shown here and is growing, as the recent developments indicated suggest.

All the examples and applications mentioned above have been computed on mainframe computers and, in some particular cases, minicomputers [Luk80, Nash75, Nash76, Busin69, Forsy77]. Some of these have only been simulations of the corresponding problems, with reduced matrix size, (the SVD is a compute-bound algorithm); therefore, they have been restricted just to show the feasibility of the corresponding method or algorithm. Andrews [Andre76a] described the situation by stating "the use of SVD techniques in digital image processing is of considerable interest for those facilities with large computing power and stringent imaging requirements"; this assertion is equally valid for the other fields interested in the SVD. However, some of the described applications [Sulli84, Speis83] have shown that, thanks to

the SVD, the problem complexity may be reduced considerably. Perhaps Speiser and Whitehouse assertion is one adequate example: "the computational wordlength requirement for such problems can be reduced by about a factor of two by solving the eigensystems problem indirectly via computing the singular value decomposition of the data matrix".

Speiser et al [Speis80], when discussing signal processing applications in some particular computing architectures, stated:

- matrix operations provide a large portion of the computational burden for real-time signal processing; this burden has limited adoption or even the comprehensive evaluation of new signal-processing algorithms, permitting them to be applied only to small problems in off-line computation, or to limited data sets

- different techniques, including the SVD, have been thoroughly studied and heavily used for non-real-time computation using conventional computers; many applications have been extensively studied via simulation but have not been implemented in real-time when the data block has more than a few degrees of freedom, because of the computational burden

- because of additional computational burden, certain methods have been applied so far only to low bandwidth signals or non-real-time analysis.

In the same work, they identified a set of basic matrix algorithms whose hardware implementation could allow certain designated tasks to be performed in real-time. Matrix algorithms which are of interest to compute in real-time but are not available yet include the SVD.

Therefore, it is of interest to be able to implement the SVD in a real-time setting, using dedicated hardware. Given the high requirements from the applications, a useful SVD processor should be able to compute the decomposition of rather large matrices. In image processing, for instance, matrix dimensions in the order of 100 by 100 or even larger are needed [Andre76a, Huang75, Shim81]. However, some simplifications may be achieved in certain problems, as mentioned before. It is considered that for an SVD processor to be useful, it should compute at least the decomposition of a 20 by 20 matrix [Tyree85].

Computation time characteristics for such processor are hard to state, due to the lack of relevant related information. Most of the data available in the literature deals with the computation time of the algorithm in implementations in mainframes or minicomputers. For instance, an EISPACK SVD routine in an IBM 370/168 took

from 0.1 [sec] to compute the decomposition of a 16 by 16 matrix, to 30 [sec] for a matrix with 128 columns [Luk80]. The same problems were solved in the ILLIAC IV computer in 0.21 [sec] and 22 [sec] respectively, using Hestenes' method [Luk80]. These experiments were used to report the suitability of Hestenes' approach for parallel implementations, especially for large matrices (larger than the experiments performed).

In a different work, the decomposition of a 128 by 128 matrix took 3 [sec] in a CDC7600 computer [Andre76a]. Note that this implementation is different than the ones above, with different data and routines, so that the computation times can not be compared. Such data is given here only for illustrative purposes.

Most of the reported research about the SVD in the different fields presents the suitability of the algorithm for those applications, but they fail to specify actual requirements for them, particularly with respect to computation time. Perhaps the most quantitative and representative assertion is Speiser statement [Speis83] that real-time signal processing will increasingly require a hardware equivalent of the software running in the mainframes. As such hardware would not have the overhead involved in the execution of the software, it should be possible to obtain an implementation several times faster than those routines. Therefore, one approach is to target the design for a speed at least two orders of magnitude better than what could be obtained for a mainframe. That is, decomposition times of around 1 [msec] for a 20 by 20 matrix and around 30 [msec] for a 128 by 128 matrix. Or, as Tyree states [Tyree85], the speed requirement is simply "faster".

1.2 SVD Algorithms and Their Implementations

There are several algorithms for the computation of the SVD [Finn82a]. In 1958, Hestenes [Heste58] suggested a one-sided orthogonalization method. This algorithm was replaced by what became the standard method, one introduced by Golub and Kahan [Golub65] in 1965; it is currently known as the Golub and Reinsch EISPACK algorithm [Golub70]. Other less successful approaches include various Lanczos algorithms [Golub81].

The EISPACK algorithm has two basic steps: a Householder transformation to bidiagonalize the given matrix and then the QR method to compute the singular values of the resultant bidiagonal form [Luk80]. This method has been used the most, essentially because there are libraries of high quality (robust, numerically stable) software for it running on mainframe computers [Speis83].

Most applications described before would benefit substantially if fast dedicated hardware was available for the SVD, but this is a compute bound algorithm. Consequently, the quest for real-time processing has resulted in a push for the development of faster computing structures as well as algorithms of lower complexity, not only for the SVD but for all matrix arithmetic and signal processing [Ahmed82].

Ahmed et al reviewed computing structures for matrix arithmetic and signal processing applications [Ahmed82]. They showed why general purpose uniprocessor computers (specially microcomputers) have had little success in this area; they include, among others reasons, the inability to exploit the inherent parallelism in the algorithms. Therefore, a hardware implementation for the SVD will have to resort to concurrent computation capabilities if it is to be successful. Such approach is difficult, since its computation is necessarily iterative [Speis83].

The Golub and Reinsch EISPACK algorithm is undesirable for concurrent computation. Luk [Luk80] indicates three reasons why this method is not adequate: "first, although the Householder transformation is inherently parallel, the effective vector length decreases at each step causing inefficiencies; second, the parallel QR method may be numerically unstable; and third, data movement across the parallel processors memories can be very expensive".

These problems have led to the search of new algorithms which could take advantage of concurrent processing capabilities. It turned out that the previously superseded Hestenes' method was easily adapted to special purpose computations; it was first suggested by Chartres [Chart62] and implemented in a minicomputer by Nash [Nash75] because of its compactness. However, this application was still sequential due to the underlying hardware characteristics. Luk [Luk80] finally showed that Hestenes' method was suitable for parallel-processing machines by implementing it in the ILLIAC IV computer. He used the parallel capabilities of this machine to perform the different elementary operations required by the algorithm.in the 64 available processing elements; the time for those operations was reduced by an asymptotic factor of 64 [Luk80]. But this is a large machine, whose cost and turnaround properties restrict it from being widely used. Alternative schemes based on cheaper and dedicated hardware to compute the decomposition have been sought extensively; actually, Nash attempt in the minicomputer was already based on such premises [Nash75].

The advent of massively parallel computer architectures has aroused much interest in parallel SVD procedures [Brent82a, Finn82a, Helle83, Kung82, Luk80, Schre83]. Such architectures may turn out to be indispensable in settings where real-time computation is desired [Brent83]. Speiser and Whitehouse [Speis80] surveyed parallel processing architectures; they concluded that systolic architectures offer the best combination of characteristics for utilizing VLSI/VHSIC technology to do real-

time signal processing.

As a result of those studies, systolic architectures have emerged as attractive alternatives for VLSI implementation of some matrix computations [Kung82, Ahmed82]. Their characteristics are: simple and regular data and control flows, simple and uniform cells, but above all, the ability to use each input data several times, achieving high computation throughput with rather low input/output bandwidth requirements.

Hestenes' method is not readily suitable for a systolic array or other concurrent architectures for the computation of the SVD, with concurrency at levels higher than the basic arithmetic operations between columns elements [Brent82a, Finn82a]. The method is an iterative algorithm that forces pairwise column orthogonality by orthogonal plane rotations; it essentially consists of a serial Jacobi procedure for finding an eigenvalue decomposition of the matrix A^tA , without actually forming it. At each step in the computation, two columns are made orthogonal; because of the iterative procedure, in the long run all columns will be orthogonal. This process involves a significant number of operations not only on the elements of a column pair but on different columns pairs of the matrix too, which could be performed in some concurrent fashion.

Parallelism on the columns elements was exploited by Luk [Luk80] in his implementation of the method in ILLIAC IV. However, some problems exist for concurrent computation of the orthogonalizations of different columns. When orthogonalizing columns i and j , the values of the inner products $<a_i,a_j>$, $<a_i,a_i>$, $<a_j,a_j>$ are required. But the rotation changes a_i and a_j; thus, any other inner product involving a_i or a_j must wait for this rotation to finish before it can start, introducing a serial dependency in the process. Additionally, Hestenes' method assumes that orthogonalizations are performed using the columns in the traditional Jacobi ordering, which is essentially a serial process but it guarantees convergence. Therefore, this method has to be adapted for concurrent computation of the SVD.

Several solutions have been suggested for these problems. Finn, Luk and Pottle [Finn82a] developed two algorithms which are variants of Hestenes' method. To avoid the implicit serialization described, these two new algorithms essentially compute approximations to the angle used in each rotation. Their first method is similar to Hestenes one, except that all inner products are computed first and then the rotations are performed; the second is a more complex approximation whose description is based on the wavefront concept and distance metric [Finn82a, Kung82]. Unfortunately, there is no formal proof of convergence for either of these algorithms; only empirical evidence tested with different kinds of matrices, including both rank deficient and full rank matrices [Finn82b].

The architecture proposed by Finn et al is based on a quadratic array of microprogrammed processing elements (PE); each of them is composed of an ALU, a microprogram memory, control logic, some registers and working store. They stated that an implementation with current technology would probably use byte, nibble or bit serial PE to maximize integration, as also time multiplexed communication paths. This architecture supposely should provide linear time computation for the SVD, although no actual time complexity is discussed. The scheme's attractiveness is reduced as a result of the lack of guaranteed convergence and also because of the number of processing elements required.

The work by Brent, Luk and Van Loan has been the most extensively reported so far and has received the most attention. It also includes two approaches to solve the problem: the first one [Brent82a] uses a linear array with $n/2$ processors that requires $O(m\ n\ logn)$ time to compute the SVD; the second uses a square array of processing elements, which takes $O(m + n\ logn)$ to converge [Brent83]. There is also some related work in [Brent82b] and [Schre82] but for the more restricted case of a symmetric $n\ x\ n$ matrix.

Hestenes' method is also used in [Brent82a] and [Brent83]. But, instead of computing an approximation of the rotation angle as Finn et al to solve the implicit serialization, an ordering for the column orthogonalizations different than the classic Jacobi one is presented. The scheme suggested decomposes the iterative process into steps; at each step all rotations can be performed in parallel, because there is no dependency among the different columns involved. Data exchanges occur from one step to the next and they are only with neighbor processors; this is very attractive because of its simplicity [Brent82a].

In [Brent83], a modification of a two-sided Jacobi SVD method for square matrices is presented and it is shown how it can be implemented in a systolic array. The array is very similar to the one proposed in [Brent82b], but a pre-processing step is required to be able to handle m by n SVD problems; in it, the QR-factorization $A = QR$ is obtained and then the array computes the SVD of R.

The pre-processing step can be done in $O(m)$ time, using a different structure for the systolic array; the proposed scheme computes the SVD of R in $O(n \log n)$ time using $O(n^2)$ processors [Brent83].

Of the two approaches, linear and quadratic arrays, the linear systolic array is perhaps a more feasible architecture than the quadratic array, because it only requires $n/2$ processing elements instead of $O(n^2)$. Ahmed et al arrived to this conclusion in their study about computing structures for matrix operations and signal processing [Ahmed82]. Furthermore, they showed that a linear array can be viewed as a cut

through a 2-D array along a computation wavefront; as a result, it enjoys higher average utilization than the 2-D array. However, the time complexity for a linear array is necessarily larger than for the quadratic one.

In [Schre83], Schreiber points out that Brent, Luk and Van Loan linear systolic architectures are the most promising idea for computing the SVD in real time, although these arrays are only able to solve problems of fixed size. He presents two modified algorithms and a modified array that do not have such disadvantage. Therefore, Schreiber's work deals with the issue of how to solve problems of arbitrary size with an array of fixed size. However, his alternatives have some drawbacks. The first method suggested implies that some columns pairs are orthogonalized several times, introducing a 20% to 60% overhead, depending on the size of the matrix and the number of processors available [Schre83]. The second method modifies the original Brent-Luk array by converting it into a ring and performing only the required orthogonalizations, but in a different ordering; there is no evidence of how effective it might be nor how the convergence characteristics of the method are affected.

Schreiber concerns should also be seen from another point of view: many problems require the decomposition of large matrices [Andre76a] for which an n/2 linear array might be impractical. Therefore, a scheme will be successful as long as it can deal with those matrices but in a smaller size array.

Symansky [Syman83] presented an experimental implementation of an SVD processor, using a two-dimensional systolic array testbed; each processing element (PE) consisted of an arithmetic processor, a general-purpose microprocessor, memory and some support devices. This testbed was used, among other applications, to compute the SVD; only the linear systolic array alternative was tested although the hardware was arranged as a quadratic array.

Symanski's scheme mainly allows to prove the feasibility of a linear systolic array but, because of the underlying hardware generality, no attempt was made to optimize it. Furthermore, the control function was implemented as a program stored in each PE, with frequent interaction with the host system (actually, after every orthogonalization).

One deviation from the schemes mentioned so far is the work by Sibul and Fogelsanger [Sibul84b]. They discussed how the coordinate rotation algorithm [Walth71] could be used to compute the SVD and described a basic cell for a processor. This scheme is attractive, as the hardware requirements are simple. However, CORDIC is a serial procedure, and questions regarding the speed of a SVD processor based on it have not been answered yet. Sibul and Fogelsanger did not include time/cost evaluations for their proposal.

1.3 Research Objectives and Contributions

Most of the reported research about the design of an SVD processor has been focused on the suitability of Hestenes' method for parallel computation in a systolic array, either linear or quadratic. Only some attention has been given to the actual hardware implementation characteristics and how they affect the architectures. No analysis has been reported regarding the actual complexity in computation time and hardware resources required for those schemes. Therefore, there is no data on time/cost characteristics for these approaches.

Systolic architectures are promising and the experimental results attractive. However, a deeper analysis of the algorithm characteristics raises questions about the optimality of systolic approaches for the SVD. Indeed, those schemes efficiently solve communications and control problems between processing elements (PE), and these PEs allow for simple and regular designs. Furthermore, they seem to satisfy Kung's [Kung82] criteria for systolic architectures: multiple use of each input data item, extensive concurrency, few types of simple cells and simple and regular data and control flow.

However, the concept of simplicity at each cell is questionable in the architectures proposed. Kung states that "a systolic system consists of a set of interconnected cells, each capable of performing some simple operation". The SVD implementations using the systolic arrays, particularly the linear one, assume that each PE performs a full orthogonalization between a pair of columns of the matrix. This task does not seem to correspond to "some simple operation", because it actually includes many arithmetic computations and several of them are complex, like division and square-root. Furthermore, the concurrency, data transfers, and functions at the different steps vary widely, making the orthogonalization not one but many operations, and not simple ones.

None of the studies has actually considered the peculiar characteristics of the algorithm, particularly the orthogonalization process; this has been seen as a sequence of arithmetic operations to be performed in a general purpose arithmetic unit. No attempt has been reported to try to exploit additional concurrency within PEs or in different architectures, although there are many issues in there which require consideration. These issues are also important, and could produce a different architecture as a better design alternative for a SVD processor.

Furthermore, there are some inherent inefficiences in systolic array techniques for this type of applications, as pointed out by Speiser and Whitehouse [Speis83]. For some architectures, at any time only a fraction of the cells is performing useful computations. Although this fraction is independent of the size of the array, so constant

efficiency may be maintained as additional cells are added to increase throughput, the schemes are inherently inefficient [Speis83].

Given the conditions described, in this work we contribute to the field by reviewing the SVD algorithm, the different subfunctions in the orthogonalization process and their hardware related issues (i.e. resources required, computation times, data communications, and control). From that analysis, relevant information regarding the architectural features for the design of a digital system to compute the SVD is obtained.

The goal has been to search for an efficient design alternative for a SVD processor, in terms of the throughput achievable with a given amount of hardware, while keeping the implementation at moderate complexity. At the same time, information regarding the characteristics and/or requirements for the devices needed in the particular design has also been sought.

Pipelined, parallel, and systolic array architectures are examined and compared in terms of throughput and hardware required. The emphasis is in the arithmetic hardware, since it is the predominant factor. This study considers matrices ranging from 20 by 20 to 40 by 40, because these sizes represent a minimum for practical applications [Tyree85]. However, the methodology and analysis performed are extensible to higher dimensions, as also the architectures devised.

To formalize the analysis, an algorithmic model which captures the characteristics of the SVD process is presented. This model gives insight into the properties of this type of computations and how to exploit them in different architectures. The model uses a directed graph as a description of the algorithm, where nodes correspond to subcomputations and arcs to precedences among the subfunctions. Each node is described by its execution time as a function of the number of operation units used. This model is utilized to evaluate implementations with replication, pipelining and parallelism.

Cost and performance measures are defined for the evaluation of the different schemes; these measures are computation time, hardware requirements, speedup with respect to a reference system, efficiency and hardware utilization. The model assumes that only one type of operation units are used for the implementations.

A methodology for the design is also suggested, in terms of the model; this methodology is essentially an iterative procedure consisting of several steps of top-down decomposition and bottom-up refinement of the implementation for the nodes in the graph of the algorithm.

The analysis and evaluation developed here show that, in most cases, a multilevel pipelined approach to compute the singular value decomposition is more convenient, because for a given amount of hardware it provides higher throughput with better efficiency than other alternatives, including the linear systolic array. This pipelined approach exploits concurrency at several levels in the implementation, through pipelines and the use of the parallelism in the subcomputations of the algorithm.

It is also shown that only the lowest cost linear systolic array (i.e. with one arithmetic unit per processing element) has better characteristics than a pipelined system with a similar amount of hardware. However, if larger throughput than what is achieved with that linear array is desired, any increase in hardware is better utilized in the pipelined approach.

The data dependencies in the computation are solved with the ordering of columns proposed in [Brent82a], but adapted for implementation with fewer processors. The resulting scheme presented here has the same characteristics of the original one regarding data transfers between neighbor units, and it is useful for a system composed of any number of parallel processors and stages per processor, including a single pipelined processor.

It turns out that the linear systolic array approach to compute the SVD, which has been regarded as the most promising architecture so far, is less effective than a multilevel pipelined system. This last one can, for instance, achieve the decomposition of a 40 by 40 matrix in the equivalent to 19600 multiplication times (i.e. 1.96 [msec] at 100 [nsec] per floating point multiplication) with less than 80 pipelined arithmetic units; in contrast, a linear systolic array with similar hardware would take 29400 multiplication times. Furthermore, systems implemented using the proposed scheme are not restricted to fixed size problems, but can be used to compute the SVD of matrices of any size, without modifications (assuming the memory is large enough to hold the data).

It is concluded that a multilevel pipelined approach for the SVD processor is more convenient than other currently proposed alternatives, in terms of throughput achievable and hardware required. The realization of such scheme has a level of complexity similar to those other alternatives. For the matrix dimensions discussed here, the speed-up factor obtained is 1.2 to 1.5 times larger than what is achieved in a linear systolic array with similar hardware, depending on the matrix size and the amount of hardware.

To reach the results stated above, in Chapter 2 we present the SVD algorithm according to Hestenes' method properly adapted for parallel computation, as proposed by Brent et al [Brent82a].

Then, in Chapter 3 we look at the characteristics of systems with concurrent computation capabilities. The algorithmic model, the methodology for the design which is used throughout the rest of the analysis, and the cost and performance measures for the evaluation of different alternatives are introduced here. In this chapter we also study how the concurrent approaches apply to the computation of the SVD, through the analysis of the time required to perform the decomposition in a system with P processors, each consisting of S pipeline stages, and also in the linear systolic array proposed by Brent et al [Brent82a]. This analysis is done in terms of the number of orthogonalizations required. A scheme to exchange columns, which allows Brent's ordering of orthogonalizations to be used in systems other than the linear systolic array, is also presented here.

In Chapter 4 we look into the hardware requirements for the orthogonalization process in the different architectures studied. We provide detailed information about those alternatives in terms of throughput achievable and hardware needed, permitting to identify and select suitable architectures according to these parameters. A procedure to select an architecture for the orthogonalization computation is introduced here.

Finally, in Chapter 5 we present design considerations for the total system and the individual components, according to the previous results. A procedure for the design of architectures for the SVD is presented, which is applied to the alternatives studied in implementations for matrices whose dimensions are 20 by 20 and 40 by 40. As a result, a multilevel pipelined architecture for a digital system to compute the SVD is proposed and evaluated in terms of the performance and cost measures defined before. This architecture is compared with the linear systolic array. Hardware requirements for each component in the system are outlined, as also the effects that VLSI could have in the implementation, with current and near future technologies.

CHAPTER 2
SVD THEOREM AND ALGORITHM

In this chapter we present the Singular Value Decomposition theorem and one algorithm suitable for concurrent computation, as described in [Brent82a]. Actually, there are several versions of the SVD theorem and also several algorithms, but Brent's approach is more adequate for the objectives pursued here.

2.1 SVD Theorem

Let A be a $m \times n$ real-valued matrix, where $m \geq n$. Then, A can always be decomposed as [Nash79, Brent82a]

$$A_{(m \times n)} = U_{(m \times n)} \Sigma_{(n \times n)} V^t{}_{(n \times n)} \tag{2.1}$$

where Σ is a $n \times n$ non-negative diagonal matrix, $V = \left[v_1, v_2, ..., v_n \right]$ is a $n \times n$ orthogonal matrix and $U = \left[u_1, u_2, ..., u_n \right]$ is a $m \times n$ matrix with orthonormal columns. That is,

$$\Sigma = diag\,(\sigma_i) \tag{2.2}$$

such that

$$\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq .... \geq \sigma_r > 0 \tag{2.3}$$

$$\sigma_{r+1} = ... = \sigma_n = 0$$

and

$$V^t V = I_n \tag{2.4}$$

$$U^t U = I_n \tag{2.5}$$

where r is the rank of the matrix A, V^t and U^t are the transpose matrices of U and V, respectively and v_i, u_i are the column vectors of V and U, respectively.

The σ_i, $i=1,2,...,r$ are the **singular values** of A, while U and V are matrices of **left singular vectors** and **right singular vectors** of A, respectively.

A variation of equation (2.1) which is also found frequently in the literature is expressed as [Brent83]

$$A_{(m\,x\,n)} = U_{(m\,x\,m)}\,\Sigma_{(m\,x\,n)}\,V^t{}_{(n\,x\,n)} \tag{2.6}$$

Here, the dimensions of the component matrices have been changed. Now, U is a $m\ x\ m$ orthogonal matrix, Σ is a $m\ x\ n$ matrix with non-negative elements on the main diagonal and zeroes everywhere else, and V is an $n\ x\ n$ orthogonal matrix. This expression is characterized by a non-square matrix Σ.

As (2.1) instead of (2.6) has been used in the studies discussed here, the first of these expressions will be used for the analysis.

2.2 Hestenes' Method According to Brent and Luk

From the several algorithms available to compute the singular value decomposition [Finn82a], the one of interest here is Hestenes' one-sided orthogonalization method [Heste58] and its adaptation for parallel computation [Brent82a]. As Brent and Luk research has received the most attention lately, their version of Hestenes' method will be presented here.

The aim of the one-sided algorithm is to find an orthogonal matrix $\hat{V}$ such that

$$A\,V = B = \left[b_1, b_2, \cdots, b_n\right] \tag{2.7}$$

with the columns of B orthogonal. Hence, the inner product satisfies

$$<b_i, b_j> = b_i^t\ b_j = \sigma_i^2\ \delta_{ij} \tag{2.8}$$

where δ_{ij} is the Kronecker delta ($\delta_{ij} = 1$ when $i = j$, and 0 otherwise). The singular values of the matrix A (i.e. the σ_i values) may be considered as forming an $n\ x\ n$ diagonal matrix so that B may be written

$$B = U\,\Sigma \quad with \quad U^t\,U = I_r \tag{2.9}$$

where $r \leq n$ is the rank of A. Consequently

$$A = U\,\Sigma\,V^t \tag{2.10}$$

Thus, if V can be constructed then (2.10) is the Singular Value Decomposition of A.

Hestenes suggested that the orthogonal matrix V should be constructed as a sequence of plane rotations [Brent82a] $Q_1, Q_2, \ldots\ldots$ such that

$$A^{k+1} = A^k Q_k \tag{2.11}$$

and

$$A^s = B$$

where A^{k+1} is the rotated matrix at the $(k+1)th$ iteration and s is the number of iterations necessary for convergence.

At any rotation only two columns of the current A^k matrix are modified. Therefore, equation (2.11) may be expressed as

$$\left[a_i^{k+1}, a_j^{k+1}\right] = \left[a_i^k, a_j^k\right] \begin{bmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{bmatrix} \tag{2.12}$$

where a_i^k, a_j^k are the $i-th$ and $j-th$ columns of the matrix A at the $k-th$ iteration and ϕ is the angle of rotation. Thus:

$$a_i^{k+1} = a_i^k \cos\phi - a_j^k \sin\phi \tag{2.13}$$

$$a_j^{k+1} = a_i^k \sin\phi + a_j^k \cos\phi$$

The problem is to choose a rotation angle ϕ to make a_i^{k+1} and a_j^{k+1} orthogonal. To do this, Brent and Luk used the formulas given by Rutishauser [Rutis66] as follows. Let

$$\alpha_i^k = < a_i^k, a_i^k > \tag{2.14a}$$

$$\alpha_j^k = < a_j^k, a_j^k > \tag{2.14b}$$

and

$$\gamma_{ij}^k = < a_i^k, a_j^k > \tag{2.14c}$$

If $\gamma_{ij}^k = 0$ then $\phi = 0$; otherwise

$$\psi = \frac{\alpha_j^k - \alpha_i^k}{2\,\gamma_{ij}^k} \tag{2.15}$$

$$t = \frac{sign(\psi)}{|\psi| + \sqrt{1+\psi^2}} \tag{2.16}$$

$$\cos\phi = \frac{1}{\sqrt{1+t^2}} \tag{2.17}$$

$$\sin\phi = t\,\cos\phi \tag{2.18}$$

The rotation angle always satisfies $|\phi| \leq \frac{\pi}{4}$ which guarantees convergence if the columns are rotated in the classical order (as in the traditional Jacobi algorithm) of (1,2), (1,3), ..., (1,n), (2,3), (2,4), ..., (2,n), (3,4), ..., (3,n), ..., (n−1,n) [Brent82a].

The process above computes equation (2.11) which produces matrix B as in equation (2.7), but the actual desired result is the decomposition stated in (2.10). Applying equation (2.9), matrix Σ is obtained by computing the singular values of B (as the square-root of the norm of the columns) and matrix U is obtained dividing each column of B by its corresponding singular value. However, V requires to accumulate the plane rotations in one matrix. This accumulation can be achieved by letting $V^1 = I$, where I is an identity matrix, and

$$V^{k+1} = V^k\,Q_k \tag{2.19}$$

Then the $(k+1)th$ rotation will affect only two columns of V^k. Therefore, it is possible to combine equations (2.12) and (2.19) into

$$\begin{bmatrix} {a_i}^{k+1} & {a_j}^{k+1} \\ {v_i}^{k+1} & {v_j}^{k+1} \end{bmatrix} = \begin{bmatrix} {a_i}^{k} & {a_j}^{k} \\ {v_i}^{k} & {v_j}^{k} \end{bmatrix} \begin{bmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{bmatrix} \tag{2.20}$$

i.e. both matrices A^k and V^k are updated simultaneously [Brent82a].

Additionally, equation (2.15) states that at each rotation the columns norms ${\alpha_i}^k$, ${\alpha_j}^k$ are required to compute the rotation angle. Although possible to obtain those values every time by performing the inner product, this overhead may be avoided by updating the current values after cosϕ and sinϕ have been computed [Finn82a]. Therefore:

$$\begin{bmatrix} {\alpha_i}^{k+1} \\ {\alpha_j}^{k+1} \end{bmatrix} = \begin{bmatrix} \cos^2\phi & \sin^2\phi \\ \sin^2\phi & \cos^2\phi \end{bmatrix} \begin{bmatrix} {\alpha_i}^{k} \\ {\alpha_j}^{k} \end{bmatrix} + \begin{bmatrix} -2{\gamma_{ij}}^k \cos\phi \sin\phi \\ +2{\gamma_{ij}}^k \cos\phi \sin\phi \end{bmatrix} \tag{2.21}$$

Finally, some mechanism has to be defined to detect the end of the computation; such condition arises when, in one complete sequence of rotations (i.e. from (1,2) to (n−1,n)), no pair of columns is rotated because they are already orthogonal (the orthogonality of the columns is defined by an error tolerance, as is usually done in digital computations which have to compare values). A counter is used for this purpose, which is set to the maximum number of rotations at the beginning of each of

the sequences above. Every time an orthogonalization is skipped this counter is decremented; if at the end of a the sequence the counter value is zero, it means that all columns are orthogonal and the decomposition has been reached.

The only remaining steps are to compute the matrix Σ (i.e. the singular values) by applying equation (2.9), and to compute matrix U (the left singular vectors) dividing each column of the resulting matrix A by its corresponding singular value.

With rotations performed serially (i.e. with one processor) in the classical Jacobi order, the following pseudo-code shows this algorithm to compute the SVD.

set error tolerance *tol*

$\alpha_i{}^1 = < a_i , a_i > \qquad i = 1,2,..,n$

$A^1 = A , V^1 = I_n$

do $k = 1,2,....$

$count = n(n-1)/2$

do $i = 1,2,..,n-1$

do $j = i+1,...,n$

$\gamma = < a_i{}^k , a_j{}^k >$

if $\gamma < tol$ then $count = count - 1$

else

$$\psi = \frac{\alpha_j{}^k - \alpha_i{}^k}{2\gamma}$$

$$t = \frac{sgn(\psi)}{|\psi| + \sqrt{1+\psi^2}}$$

$$\cos\phi = \frac{1}{\sqrt{1+t^2}}$$

$$\sin\phi = t\ \cos\phi$$

$$\begin{bmatrix} a_i{}^{k+1} & a_j{}^{k+1} \\ v_i{}^{k+1} & v_j{}^{k+1} \end{bmatrix} = \begin{bmatrix} a_i{}^k & a_j{}^k \\ v_i{}^k & v_j{}^k \end{bmatrix} \begin{bmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{bmatrix}$$

$$\begin{bmatrix} \alpha_i{}^{k+1} \\ \alpha_j{}^{k+1} \end{bmatrix} = \begin{bmatrix} \cos^2\phi & \sin^2\phi \\ \sin^2\phi & \cos^2\phi \end{bmatrix} \begin{bmatrix} \alpha_i{}^k \\ \alpha_j{}^k \end{bmatrix} + \begin{bmatrix} -2\gamma_{ij}{}^k \cos\phi \sin\phi \\ +2\gamma_{ij}{}^k \cos\phi \sin\phi \end{bmatrix}$$

endif

end j

end i

if $count = 0$ then end iterations

end k

do $i = 1, 2, ..., n$

$\sigma_i = \sqrt{<a_i^s, a_i^s>}$

$$u_i = \frac{a_i^s}{\sigma_i}$$

end i

The last step in the pseudo-code above, namely the computation of Σ and U, is not compute-bound as the rest of the algorithm. Consequently, it is assumed that such operations are performed in a host processor and no dedicated hardware is introduced for them.

In order to perform the rotations in parallel, Brent and Luk [Brent82a] proposed a new scheme; in it, the orthogonalization process is the same above and all columns pairs are also rotated just once in a sequence of $n(n-1)/2$ rotations (called a "sweep"), but in an ordering different from the classical one. For the case of a matrix with 8 columns, this ordering would be

step 1:	(1,2), (3,4), (5,6), (7,8)
step 2:	(1,4), (2,6), (3,8), (5,7)
step 3:	(1,6), (4,8), (2,7), (3,5)
step 4:	(1,8), (6,7), (4,5), (2,3)
step 5:	(1,7), (8,5), (6,3), (4,2)
step 6:	(1,5), (7,3), (8,2), (6,4)
step 7:	(1,3), (5,2), (7,4), (8,6)

The characteristics of this approach are:

- up to $n/2$ orthogonalizations (a step above) can be computed concurrently, because they are independent
- input data for any step depends on the outcome of the previous step

- if all orthogonalizations are done concurrently and their results are available simultaneously, a new set of $n/2$ computations (i.e. a new step) can start immediately after the previous one is done (there are no dependencies among steps).

To take advantage of these facts, Brent and Luk proposed a scheme with $n/2$ parallel processors, where each one of them performs a complete orthogonalization of a column-pair; data is exchanged among neighbor processors before a new orthogonalization starts, with a rather simple mechanism. Figure 2.1 depicts the data exchange process in this method.

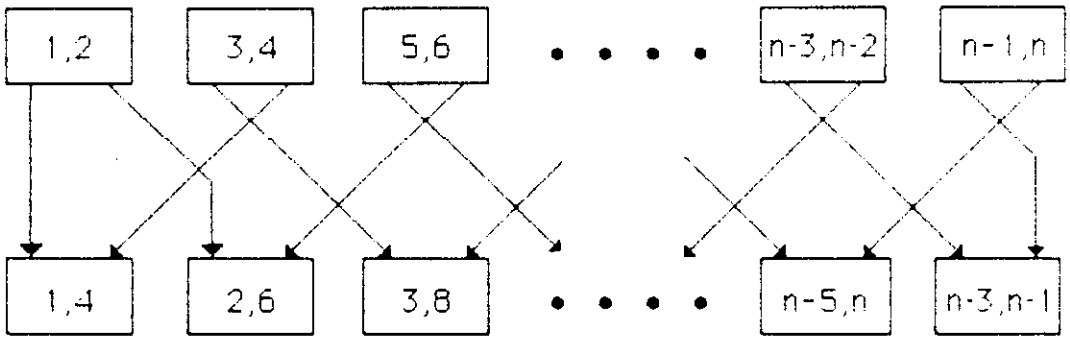

Figure 2.1 - Columns Exchange Process

Very little is known about the convergence properties of Brent and Luk's scheme [Brent82a]. To enforce convergence, they adopted a threshold approach by associating with each sweep a threshold value th; when making the transformations of that sweep, rotations are omitted if

$$\frac{< a_i^k , a_j^k >}{\alpha_i^k \, \alpha_j^k} < th \qquad (2.22)$$

that is, columns which are already orthogonal to each other, or their normalized inner product is small, are not rotated further. Their method enjoys ultimate quadratic convergence and numerical experience suggests that six to ten sweeps are required to compute the decomposition [Brent82a].

As each processor is in charge of rotating a column-pair, it must perform the following operations:

i. Computation of Inner Product among both columns

ii. Computation of Angle for Rotation

iii. Rotation of four columns; that is, compute a_i^{k+1}, a_j^{k+1}, v_i^{k+1}, v_j^{k+1}

iv. Columns Norms update; compute α_i^{k+1}, α_j^{k+1}

v. Exchange of columns for next orthogonalization between neighbor processors

An initial step is required to compute the norms of the columns of the original matrix A and to initialize matrices A and V. These operations are outside the loop in the pseudo-code shown above for the serial approach.

Now that the SVD algorithm has been described, the next step in the analysis is to look at different implementation alternatives for it, particularly those using the concurrent computation capabilities available. But first, it is convenient to formalize the analysis by introducing an algorithmic model for the computation and a methodology for the design. These tools can give insight into the characteristics of the algorithm and its suitability for particular implementations. Then, it is convenient to look into the throughput achievable for the orthogonalization process in different architectures. Those are the subjects of the next chapter.

CHAPTER 3
ARCHITECTURES FOR THE SVD

In this chapter we present the basic characteristics of architectures for algorithms with concurrent computation capabilities and we discuss how such characteristics apply to the design of a digital system for the SVD. An algorithmic model and a methodology for the design and evaluation of such architectures is presented and these techniques are applied in the design of a structure to compute the SVD. In doing so, the suitability of the SVD algorithm for concurrent computation is studied and a scheme to solve the data dependencies in it is introduced.

3.1 Architectures with Concurrent Computation Capabilities

Parallelism and pipelining are two well-recognized methods, which can be used separately or in combination, to improve the speed of execution of digital systems. In this section their characteristics are briefly reviewed and a methodology to design a system which uses them is described. Since the objective is to apply this methodology to the processor for the singular value decomposition, those aspects that are relevant to this case are emphasized.

The topic at hand can be considered from two complementary and interrelated angles. One aspect is the **design of algorithms** that are suitable for parallel and pipelined implementations, and the other is to **design the system** for a given algorithm. The first has been researched extensively, both in its general theory and in the search for algorithms for specific applications. This aspect is not considered further since a specific SVD algorithm will be used, to which only minor tuning can be applied to make the implementation more effective.

3.1.1 Algorithmic Model and Methodology

Algorithm Representation

To design the system for a given algorithm it is necessary to have the algorithm described in a suitable form. A **graphical description** is used here, in which nodes correspond to subcomputations and arcs describe the precedences between these subcomputations. Several models have been used to describe more precisely the precedences, which can include conditionals and loops [Estri78, Patil72, Peter81].

For the purposes here, it is sufficient to use AND-OR conditions as presented in Figure 3.1.

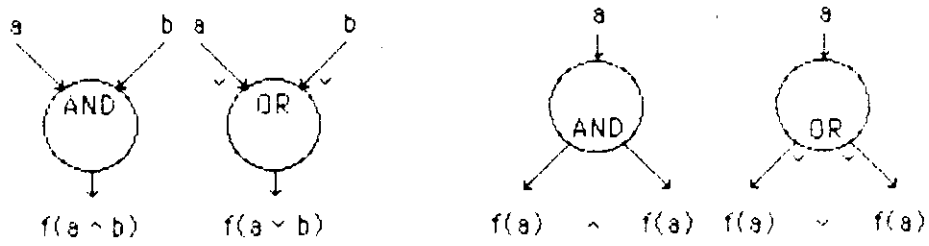

Figure 3.1 - Graphical Description Elements

The primitive subcomputations used in the algorithm can have any level of complexity. Depending on this level (also called the **granularity** of the algorithm), implementations of different degrees of parallelism/pipelining can be obtained. It is clear that larger concurrency can be obtained for finer granularity, which would indicate that it is always convenient to consider the representation with the finest granularity. However, this can lead to algorithms with a very large number of nodes, making the design of the system complex and unstructured. In such cases many different problems would have to be faced simultaneously, such as communications between the nodes, the control of the sequencing, the synchronization of nodes, and their individual design; the complexity of all these issues is directly related to the number of nodes in the algorithm.

Therefore, it is convenient to resort to a more structured top-down design, in which one begins with an algorithm consisting of a relatively small number of nodes and then refines each of the nodes into subalgorithms. This top-down approach has the limitations that it is possible to loose some potential concurrency and that the characteristics of each of the nodes (execution time and number of operation units needed) are not known until the node has been implemented. The alternative of first implementing the nodes and then going up one level (bottom-up approach) is not totally satisfactory either, because the use of parallelism and pipelining in the implementation of the node depends on how critical its execution time is in the overall algorithm. Consequently, the design process consists of several iterations until a satisfactory solution is found. Each of these iterations would be bottom-up and would consist in modifying the implementation of those nodes that have been identified as critical in the previous iteration, and then going up to the next level. At a particular level in a design iteration the algorithm is specified by a **graph whose nodes are indivisible.**

Number of Instances

An important parameter that influences the design methodology presented is the **number of times** M that the algorithm is executed on independent data. In fact, the pipelined implementation is effective only in cases in which this number is significant with respect to the number of stages in the system. On the other hand, if the algorithm is computed just once only parallelism is effective to reduce the computation time. Since the SVD process is composed of repeated invocation of orthogonalizations, the emphasis here is on implementations for significant number of instances.

The simplest model is to assume that all instances are independent. However, this does not capture the characteristics of the SVD algorithm and therefore is not adequate for comparisons of implementations of this algorithm. A better model for this case is to assume that the M instances are divided into **groups of** r **instances** each and that **instance** p **of group** q **is dependent on instance** p **of group** q-1, as illustrated in Figure 3.2.

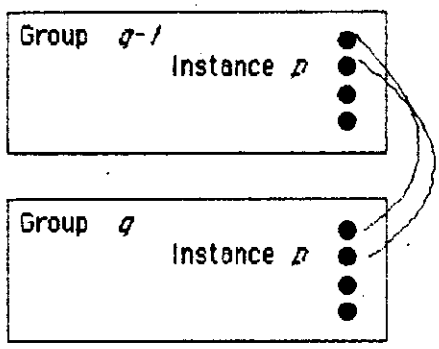

Figure 3.2 - Dependences among Instances

Types of Operation Units used in the Implementation

With regards to the alternative implementations, it is also important to specify the set of operation units that can be used. Two extreme possibilities exist, namely to require a different type of operation unit for each node (with no sharing among nodes) or to require just one type of operation unit that can perform any of the nodes and, therefore, can be shared among nodes. The effectiveness of the alternative implementations is completely different depending on which of these possibilities is chosen. The analysis here considers that **just one type of operation unit** is used (since this is the case for the SVD implementation described later). However, in a second level of refinement the operation units themselves are composed of suboperations, each of which is performed by a different suboperation unit. Consequently, this case will be

considered briefly later.

Model of the Algorithm and the Implementation

From the previous considerations we can formalize the following model:

• The algorithm is described by a directed graph in which the nodes are indivisible for that level of the implementation.

• Just one type of operation unit is used to execute all nodes. Node i is specified by its execution time t_i and by the number of operation units required N_i. More in general, the node is specified by $t_i(j)$ corresponding to its time of execution in an implementation with j operation units (denoted as (time/units)). This implies that for each node there might be more than one alternative implementation, with different characteristics.

For realizable implementations it is necessary that for $k > j$,

$$t_i(j) \geq t_i(k) \geq \frac{j}{k} t_i(j) \tag{3.1}$$

that is, an increase in the number of operation units reduces the computation time of the node at best proportionally to the number of units.

In many cases, because of the algorithm characteristics, there might be implementations for a given node which are not convenient. These correspond to cases where the use of some specific number of operation units, say h, does not reduce the computation time with respect to another implementation with fewer units. That is, the time $t_i(h)$ is equal to the time of the implementation that uses a number of operation units g such that $g < h$. Consequently, some idle operation units exist. The corresponding function $t_i(j)$ has the form shown in Figure 3.3.

• The algorithm is executed for M instances. These instances are divided into groups of r, and their dependencies are described by Figure 3.2.

Performance, Cost Measures, and Design Objective

Alternative implementations have to be compared using performance and cost measures. The following measures are used here:

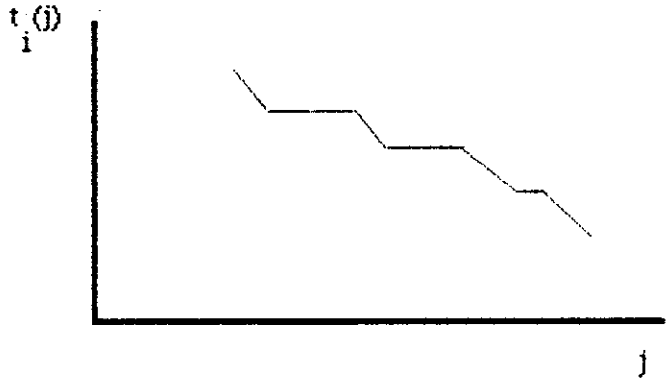

Figure 3.3 - Node Time vs. Operation Units

• Time of execution of the computation t (for M instances).

• Number of operation units N.

• Speedup $SU = t_{ref} / t$. The reference implementation (whose time of executions is t_{ref}) depends on the particular comparison being made. A particular reference is the **completely-sequential implementation**, which uses only one operation unit that is shared among all nodes.

• Efficiency $E = \frac{t_{ref}\ N_{ref}}{t\ N}$.

• Hardware Utilization $HU = \frac{1}{N\ t} \sum_j N_j\ t_j$, where t_j is the time that the units N_j are used.

Note that if the reference implementation is the completely sequential, the speedup of any alternative is less or equal to N and its efficiency is less or equal to 1.

In terms of these measures, the main objective of the design can be described as selecting the implementation that for a given speedup (or computation time) has the largest efficiency (or uses the minimum number of operation units). Of course, other factors are also important in the selection of the best implementation. These factors include interconnection complexity, data bandwidth requirements and expansibility, among others.

3.1.2 Characteristics of Replicated, Parallel, and Pipelined Systems

The characteristics, performance and cost measures of implementations of the algorithmic model are considered now, in which the concepts of replication, parallelism, and pipelining are applied. We first analyze the use of only one of these approaches in a given system and provide a comparison of the resulting measures. Then we look at systems which use a combination of approaches.

Sequential Implementations

The sequential implementations are considered first, since they form the basis for some of the others. **An implementation of an algorithm is sequential if only one node (subcomputation) is executed at a time.** To obtain this type of implementation the graph of the algorithm can be transformed by adding precedences to obtain a total ordering of the nodes.

Note that this definition, in the context of the model, permits the use of several operation units in a sequential implementation, since individual nodes can use more than one unit and the nodes are indivisible. That is, there are a series of sequential implementations using different number of operation units. We call $t_{seq}(j)$ the execution time of the sequential implementation that uses j operation units, and attach similar qualifiers to the other measures. The implementation that uses just one operation unit (i.e. the completely sequential implementation) plays a special role as reference and as the basis for some implementations.

The following expressions describe the measures for the sequential implementations. The execution time for the M instances is

$$t_{seq}(j) = M \sum_i t_i(j)$$

The speedup with respect to the completely-sequential implementation is

$$SU_{seq}(j) = \frac{t_{seq}(1)}{t_{seq}(j)}$$

The efficiency with respect to the completely-sequential implementation is

$$E_{seq}(j) = \frac{t_{seq}(1)}{t_{seq}(j)\, j}$$

Replicated Systems

For the purposes here, **a replicated implementation of an algorithm performs several instances of the computation simultaneously, using identical and separate hardware resources (processors) for each instance.** Consequently, the approach is effective only when several independent instances have to be computed. Because of the dependencies between instances assumed in the model, the number of instances executing simultaneously should be less or equal to r, the number of instances of a group.

If the hardware required to process an instance (a processor) is replicated P times, the execution time of the M instances is

$$t_{rep}(j,P) = \lceil M/P \rceil (t_{seq}(j)/M) = \frac{\lceil M/P \rceil t_{seq}(j)}{M} \quad \text{for } P \leq r$$

since the instances are performed in sets of P, excepting the last set which might be smaller than P. Note that the last instances in a group may be computed simultaneously with the first ones in the following group.

In particular, if $P = r$ then all instances in a group are processed at once and

$$t_{rep}(j,r) = \frac{t_{seq}(j)}{r}$$

The speedup and efficiency with respect to the completely-sequential implementation are given now. The speedup is

$$SU_{rep}(j,P) = \frac{t_{seq}(1)}{t_{rep}(j,P)} = \frac{t_{seq}(j)}{t_{rep}(j,P)} \frac{t_{seq}(1)}{t_{seq}(j)} = \frac{M}{\lceil M/P \rceil} SU_{seq}(j)$$

and the efficiency,

$$E_{rep}(j,P) = \frac{t_{seq}(1)}{t_{rep}(j,P)\, jP} = \frac{t_{seq}(j)\, j}{t_{rep}(j,P)\, jP} \frac{t_{seq}(1)}{t_{seq}(j)\, j} = \frac{M/P}{\lceil M/P \rceil} E_{seq}(j)$$

For the important case in which $M \gg r$, these expressions become

$$SU_{rep}(j,P) = P\ SU_{seq}(j)$$

and

$$E_{rep}(j,P) = E_{seq}(j)$$

Consequently, the maximum speedup is $r\ SU_{seq}(j)$. This is obtained with r replications and the efficiency for this case is the same as for the sequential implementation which is being replicated. That is,

$$SU_{rep}(j,r) = r\ SU_{seq}(j)$$

and

$$E_{rep}(j,r) = E_{seq}(j)$$

Parallel Systems

For the purposes here, **a parallel implementation of an algorithm uses the parallelism present in the graph to perform independent nodes concurrently.** Consequently, the time of execution of the sequential implementation might be reduced. This reduction may require additional operation units.

The speedup depends on the characteristics of the graph and on the scheduling. To obtain the maximum speedup for a given number of operation units an optimal schedule has to be devised. In general, the determination of this schedule requires an exhaustive search, so several suboptimal heuristics and upper and lower bounds have been developed [Gonza77, Ramam72a].

The following bounds are useful for the comparisons developed later. A lower bound on the execution time with j operation units is

$$t_{par}(j) \geq \sum_{crit.path} t_i(j)$$

Note that the critical path (not only its length) might vary with j.

Consequently, an upper bound on the speedup (with respect to the completely sequential case) is

$$SU_{par}(j) \leq \frac{\sum_{i} t_i(1)}{\sum_{crit.path} t_i(j)}$$

Because of relation (3.1), this speedup is bounded by

$$SU_{par}(j) \leq j$$

and the efficiency by

$$E_{par}(j) \leq 1$$

Without performing the actual scheduling it is not possible to know whether the parallel implementation (with j operation units) is better than the corresponding sequential one.

In this analysis it has been assumed that the time of execution is composed solely by the execution time of the subcomputations. In a practical implementation, there might be data transmission times and access to other shared elements, such as datapaths and memories. This might add significantly to the total execution time. In such cases, it is necessary to include these times in the evaluation and the shared resources in the scheduling algorithm.

Pipelined Systems

A pipelined implementation of an algorithm is characterized by the fact that several instances of the computation (on independent data) are executed simultaneously. For this, **the algorithm is divided into stages**, and the system is implemented so that **different instances are executed simultaneously at different stages.** An extensive literature exists on pipelined implementations. A reasonably complete treatment is given in [Kogge81].

Since only the use of pipelining (without the parallelism of the graph) is considered for the moment, the algorithm to use should be sequential. Furthermore, since each of the nodes of the graph is viewed at this point as indivisible, a stage is composed of one node or of a set of consecutive nodes. This partitioning is done so that the resulting stages have a (approximately) uniform delay since this is necessary for an adequate flow through the pipeline. Additional delays might have to be added to some stages to make the delay uniform. Also, the partitioning into some specific number of stages might not be possible.

If the resulting system has S stages, up to S instances can execute simultaneously. Therefore, the number of stages is limited by the number of independent instances. For the computation model considered here (instances divided into groups and dependency between corresponding instances of consecutive groups) the pipeline is kept full if

$$S \leq r$$

In terms of the number of stages S and the stage delay t_S, the time to execute M instances is

$$t_{pipe} = [S + (M-1)]\, t_S$$

The speedup of a pipelined implementation, with respect to the completely sequential case, is

$$SU_{pipe} = \frac{t_{seq}(1)}{[S + (M-1)]\, t_S}$$

This can be written as

$$SU_{pipe} = \frac{S}{S + (M-1)} \frac{t_{seq}(1)}{S\, t_S}$$

To compute the actual speedup it is necessary to determine t_S. To do this, the sequential implementation has to be partitioned. Assuming that the implementation which uses j operation units is utilized for the partitioning and that this can be done perfectly (i.e. into stages of uniform delay), then

$$t_S(j) = \frac{t_{seq}(j)}{M\, S}$$

and the speedup is

$$SU_{pipe}(j,S) = \frac{M\, S}{S + (M-1)} \frac{t_{seq}(1)}{t_{seq}(j)} = \frac{M\, S}{S + (M-1)} SU_{seq}(j)$$

To compute the efficiency we need to determine the number of operation units. This number is larger than that of the corresponding sequential implementation since now the units are not shared among stages. Consequently, the total number is the sum of the units used in each stage. The simplest possibility is to use the same number of units per stage, say j, which results in an efficiency with respect to the completely-sequential case of

$$E_{pipe}(j,S) = \frac{M}{S + (M-1)} E_{seq}(j)$$

The efficiency of the sequential implementation is reduced as a result of the startup time of the pipeline.

Of special interest is the case in which $M \gg S$. In such case, the speedup tends to

$$SU_{pipe}(j,S) = S \; SU_{seq}(j)$$

and the efficiency to

$$E_{pipe}(j,S) = E_{seq}(j)$$

These expressions are identical to those obtained for replicated systems.

However, **implementations with larger efficiencies** might be obtained if the stages of the pipeline use different number of operation units. This is the case when there is at least one stage that is formed of nodes for which

$$t_i(h) = t_i(j) \qquad \text{for } h < j$$

In such case h operation units (instead of j) can be used for that stage without changing its delay. This results in a reduction of the total number of operation units and, therefore, in an increase of the efficiency. This can be extended to cases in which it is possible to reduce the number of units for several stages. To make best use of this possibility it is convenient to group the nodes into stages so that all nodes of a stage "require" the same number of operation units (of course these nodes also have to be consecutive in the graph).

The pipelined implementation requires staging registers between stages. The addition to the cost and to the execution time that these registers produce is assumed to be negligible. This is true if the cost and delay of the operation units are much larger. Also, the control of the system becomes somewhat more complex since each stage has to be controlled independently and the delays of the stages have to be made equal.

We are concerned here only with simple pipelines in which all instances use the pipeline in the same fashion, that is the data goes through the same path and the delay of the stages is data independent. More complex situations are discussed in [Kogge81]. Also presented there is the case in which stages are used several times by the same instance. We do not consider these cases since they do not occur in the implementation of the SVD.

3.1.3 Comparison between Replicated, Parallel, and Pipelined Implementations.

The use of these techniques is compared now in terms of the measures discussed before and of the design objective of obtaining a given speedup with the highest efficiency. At this point, we assume that only one of the techniques is applied in a particular implementation. This is not realistic but gives insight into the considerations, discussed later, used to determine an adequate combination.

To illustrate the choices that can be made regarding implementation, we consider the algorithm described by the graph in Figure 3.4, which is executed for $M = 104$ instances and with dependences in groups of $r = 8$. In this graph, each node has several alternative implementations with different number of operation units, as indicated by the descriptors x/y next to each node (x: computation time, y: number of units).

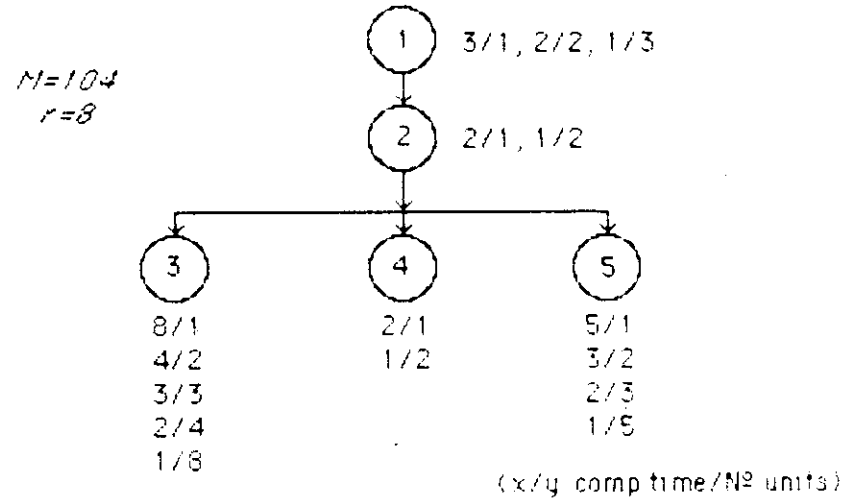

Figure 3.4 - Algorithm with Concurrent Computation Capabilities

Note that this algorithm is characterized by diversity in the computation times for the nodes and by some implementations for nodes which are not advantageous since they do not reduce the time (i.e. node 3, case 2/5 with respect to 2/4). The computation time for the completely sequential implementation, used later as a reference, is $20M$.

Implementation			$M \gg r$	
	Speedup	Efficiency	Speedup	Efficiency
Replicated	$\frac{M}{\lceil M/P \rceil} SU_{seq}(j)$	$\frac{M/P}{\lceil M/P \rceil} E_{seq}(j)$	$P\ SU_{seq}(j)$	$E_{seq}(j)$
Parallel	depends on graph and scheduling	depends on graph and scheduling	$SU_{par}(j) \leq j$	$E_{par}(j) \leq 1$
Pipelined	$\frac{M\ S}{S + (M-1)} SU_{seq}(j)$	$\frac{M}{S + (M-1)} E_{seq}(j)$	$S\ SU_{seq}(j)$	$E_{seq}(j)$

Table 3.1 - Speedup and Efficiency for Systems with Concurrent Capabilities

Table 3.1 presents the expressions for speedup and efficiency for the three cases described before. Since for a given speedup the alternative with the largest efficiency is desired, from the expressions the following situations are identified:

a.- For speedups up to r, replication of the completely-sequential implementation is the best (since it has an efficiency of 1). For M large ($M \gg r$), pipelining can have similar characteristics if perfect pipelining is achieved, but it might not be possible to do so for some values of S since there might be no suitable partition.

This situation is illustrated in Table 3.2 for the algorithm presented in Figure 3.4. The table shows the speedup and efficiency for both approaches, replication and pipelining of the completely sequential implementation, and also the nodes assigned to each stage in the pipelined case. As expected, replication alone has speedup P and efficiency 1, while pipelining is less effective for this example since perfect pipelining is not achieved. Furthermore, pipelining with more than three stages is not advantageous since the throughput does not increase, as a result of the long computation time of node 3. Consequently, only implementations with up to three stages are considered.

P or S	$SU_{rep}(1,P)$	$E_{rep}(1,P)$	$SU_{pipe}(1,S)$	$E_{pipe}(1,S)$	Nodes per stage
1	1	1	1	1	all nodes
2	2	1	1.98	0.99	1 2 5 | 3 4
3	3	1	2.45	0.82	1 2 4 | 3 | 5
4	4	1			
5	5	1			
8	8	1			

Table 3.2 - Replication and Pipelining of Completely Sequential Implementation

b.- For larger values of speedup it is necessary to select, among the following alternatives, the one that produces the highest efficiency. The choice depends on the characteristics of the graph.

- Replication of the sequential implementation that uses j operation units. The number of operation units is Pj and the maximum speedup that can be obtained under maximum efficiency conditions is rj. However, the efficiency is usually less than 1 because of the degraded efficiency of the corresponding sequential implementation.

  Table 3.3 depicts this alternative for the algorithm above, for different values of j and P, up to the maximum replication $P = r = 8$ defined by the dependences between groups of instances. For every value of P, the efficiency is constant and equal to the efficiency of the corresponding sequential implementation.

j	$SU_{rep}(j,1)$	$SU_{rep}(j,2)$		$SU_{rep}(j,8)$	$E_{rep}(j,P)$
2	1.82	3.64	--	14.56	0.91
3	2.50	5.00	--	20.00	0.83
4	2.86	5.72	--	22.88	0.71
5	3.33	6.66	--	26.64	0.66
8	4.00	8.00	--	32.00	0.50

Table 3.3 - Replication of Sequential Implementations

- Pipelining the sequential implementation that uses j operation units. This produces an efficiency equal to that of the sequential implementation if j operation units are used in all stages, and a better efficiency otherwise. Consequently, this approach would be preferred to replication whenever the efficiency of the pipelined case is high because of the reduction of the number of units required by some stages.

  Table 3.4 shows what is achievable with this approach in the algorithm used as example. The table also indicates the number of units saved in each case and the partitioning of the algorithm into stages. For each stage, the nodes in it and the particular implementation used for each node are given (i.e. 1(2/3): node 1, with 3 units and time 2).

  As depicted in the table, the partitioning of the graph does not always give perfect pipelining and therefore the efficiency is less than the maximum. Furthermore, a larger number of stages and units per stage does not necessarily increases the speedup and efficiency; only some implementations are advantageous.

j	S	$SU_{pipe}(j,S)$	$E_{pipe}(j,S)$	N_{saved}	Nodes per Stage
2	2	3.30	0.83	0	1(2/2) 2(1/2) 5(3/2) | 3(4/2) 4(1/2)
	3	4.90	0.82	0	1(2/2) 2(1/2) 4(1/2) | 3(4/2) | 5(3/2)
3	2	4.95	0.82	0	1(1/3) 2(1/2) 5(2/3) | 3(3/3) 4(1/2)
	3	6.54	0.82	1	1(1/3) 2(1/2) 4(1/2) | 3(3/3) | 5(3/2)
4	2	4.95	0.83	2	1(2/2) 2(1/2) 4(1/2) | 3(2/4) 5(2/3)
	3	6.54	0.82	4	1(2/2) 2(1/2) | 3(3/3) | 4(1/2) 5(2/3)
	4	9.71	0.88	5	1(2/2) | 2(1/2) 4(1/2) | 3(2/4) | 5(2/3)
5	2	6.60	0.83	2	1(1/3) 2(1/2) 4(1/2) | 3(2/4) 5(1/5)
	3	9.80	0.82	3	1(1/3) 2(1/2) | 3(2/4) | 4(1/2) 5(2/5)
8	2	6.60	0.83	8	1(1/3) 2(1/2) 4(1/2) | 3(2/4) 5(1/5)
	3	9.80	0.82	12	1(1/3) 2(1/2) | 3(2/4) | 4(1/2) 5(1/5)
	5	19.20	0.96	20	1(1/3) | 2(1/2) | 3(1/8) | 4(1/2) | 5(1/5)

Table 3.4 - Pipelining of Sequential Implementations

- Using the parallelism of the graph is the only of the three approaches that can be effective if there is only one instance. The speedup that is obtained depends on the graph and on the scheduling. The efficiency also depends on these factors, and is less than one in most cases.

  The use of this approach in the case of multiple instances is suitable if it provides the desired speedup at an efficiency that is larger than any of the other cases.

  This scheme is described in Table 3.5 for the graph above. A scheduling was performed for each value of j, which resulted in some implementations where nodes are executed in parallel (marked with *), The table also indicates the implementation chosen for each node.

j	$SU_{par}(j)$	$E_{par}(j)$	Implementation of Nodes					
			1	2	3	4	5	
2	1.82	0.91	2/2	1/2	4/2	1/2	3/2	
3	2.50	0.83	1/3	1/2	3/3	1/2	2/3	
4	3.33	0.83	1/3	1/2	2/4	2/1*	2/3*	$t_s = 2$
5	4.00	0.80	1/3	1/2	2/4*	2/1*	1/5	$t_s = 2$
8	5.00	0.63	1/3	1/2	1/8	1/2*	1/5*	$t_s = 1$

* : *nodes in parallel*

Table 3.5 - Implementations Using Parallelism of the Graph

The values for speedup from the different implementation alternatives discussed up to now are summarized in Figure 3.5. Note that continuous curves are shown but only discrete points exist. In the pipelined case, only those implementations which are convenient have been plotted (i.e. those which give higher speedup with a larger number of units).

From the figure it can be inferred that, for the algorithm given in the graph in Figure 3.4, replication of the sequential implementations with one and two units offer the best efficiency. For larger speedups than what is achievable with such schemes, pipelining offers better efficiency. This corresponds to the maximally pipelined implementations. Further speedup increases than the maximum obtained with pipelining are possible through replication of sequential implementations with three or more units, but with lower efficiency.

The next step is to study how these characteristics are affected when more than one of the concurrency techniques is exploited simultaneously. Such issue is discussed now.

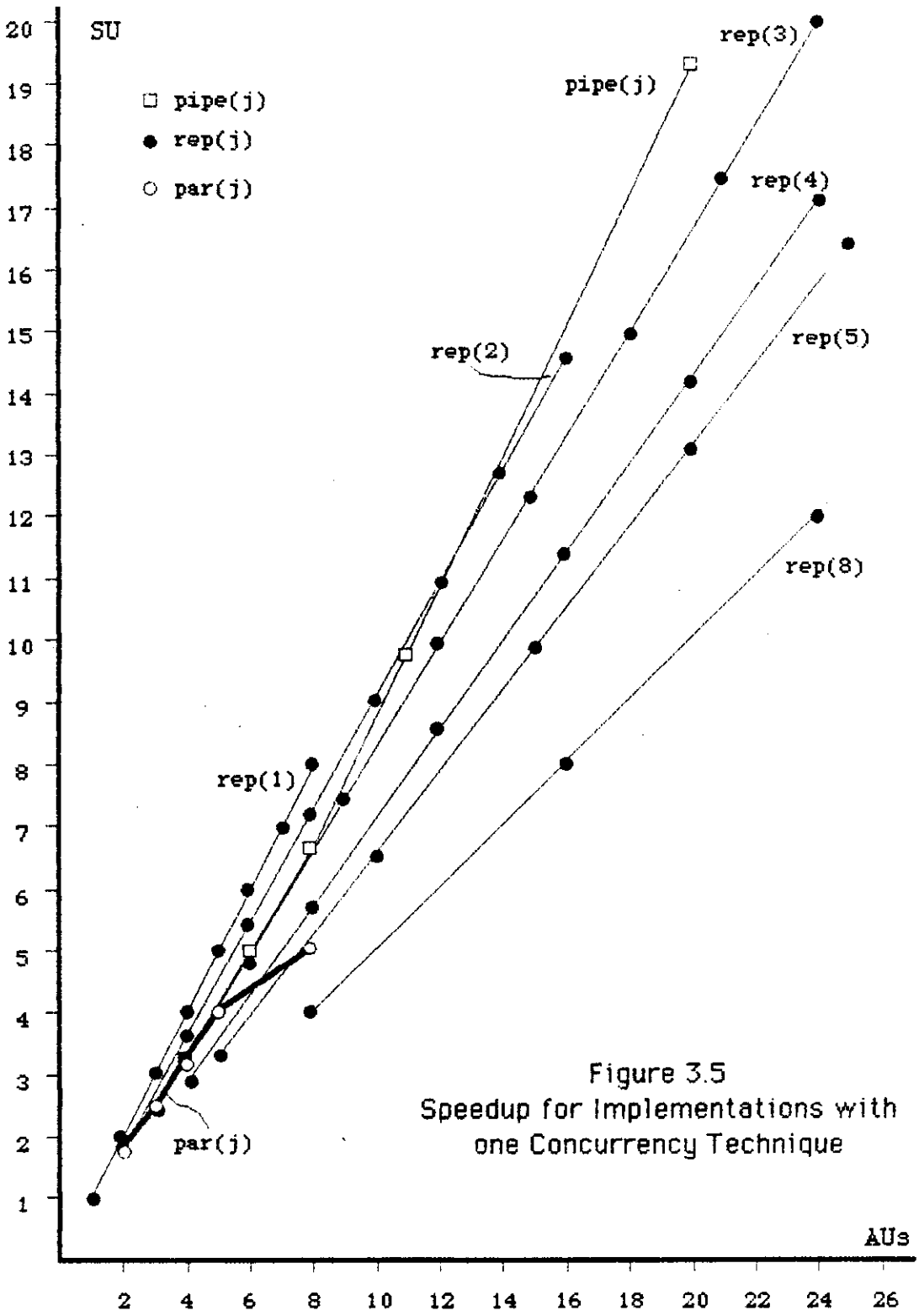

Figure 3.5
Speedup for Implementations with one Concurrency Technique

Combination of approaches

In an implementation it is possible to combine two or all three of the approaches discussed previously. The characteristics of these combinations are described now and the corresponding performance and cost measures are evaluated.

Replication and pipelining

In this case **the pipelined processor is replicated.** Because of the necessity of having enough independent instances, the total number of processors and stages combined is limited so that

$$PS \leq r$$

The following two situations arise:

- Replication of the pipelined implementation that uses one operation unit (completely-sequential case) in each stage. This alternative produces a speedup of PS and has efficiency 1 for large M and values of S that result in perfect pipelining, or lower efficiency otherwise. Under perfect pipelining conditions, it is equivalent in speedup and efficiency to the implementation that uses only replication. Therefore, it is only effective when the number of processors required for the replicated implementation (up to r) is large and produces realization problems, such as interconnection among the processors.

  For the example above, this approach corresponds to replications of the pipelined implementations in Table 3.2, as long as $P\,S \leq r$. The possible implementations are shown in Table 3.6. For each case, the efficiency is the same as the corresponding single pipelined processor, and the speedup is a multiple also of the corresponding pipelined implementation.

- Replication of the pipelined implementations that use more than one operation unit in some stages. These implementations increase the speedup of the single pipelined processor and maintain its efficiency. They are therefore adequate for higher speedups than that available with one (pipelined) processor.

S	$P=1$	$P=2$	$P=3$	$P=4$	$E_{rep/pipe}(S,P)$
2	1.98	3.96	5.94	7.92	0.99
3	2.45	4.9	-	-	0.82

Table 3.6 - Replication of Completely-Sequential Pipelined Processor

For the algorithm above, this scheme corresponds to replication of the alternatives in Table 3.4, as long as $P\,S \leq r$. The possible implementations are shown in Table 3.7

S	j_{tot}	$P=1$	$P=2$	$P=3$	$P=4$	$E_{rep/pipe}(S,P)$
2	4	3.30	6.60	9.90	13.20	0.83
3	8	6.54	13.08	-	-	0.82
4	11	9.71	19.42	-	-	0.88
5	20	19.20	-	-	-	0.96

Table 3.7 - Replication of Pipelined Processor with more than one Unit

Replication and graph parallelism

In this implementation **a processor which uses graph parallelism is replicated.** It provides an increase of speedup with an efficiency equal to that of the implementation using only graph parallelism. Usually this efficiency will be significantly smaller than 1. Consequently, this scheme is only effective if the speedup cannot be achieved using a more efficient technique.

For the example above, this approach is the replication of the entries in Table 3.5. The corresponding speedups are increased and the efficiency of the implementation without replication is preserved. Note that this efficiency is low for larger number of units. This approach provides a wide range of speedup values. The maximum speedup achievable in the example with this scheme is

$$SU_{rep\,/par}(P = 8, j = 8) = 40.0$$

and the corresponding efficiency is

$$E_{rep\,/par}(P = 8, j = 8) = 0.63$$

Pipelining and graph parallelism

In this case **one or more of the stages of the pipeline uses graph parallelism.** The partitioning into stages has to be modified (with respect to the implementation using pipelining only) to obtain stages of equal delay.

This scheme might be effective in increasing the efficiency of the pipeline since it can help to get stages of equal delay. The number of stages is defined by the graph of the algorithm. Nodes may be moved (preserving the dependences, of course) to achieve stages of (approximately) equal time. The smallest stage time possible is defined by the node with the longest computation time.

For the example considered above, Table 3.8 shows implementation alternatives for different stage times. Only two and three stages exist, given the characteristics of the graph. This table indicates the nodes in each stage and their corresponding implementations, with the same notation as Table 3.4; it also indicates which nodes in each stage are computed in parallel (marked with *) or sequentially (those not marked).

S	t_S	j_{tot}	$SU_{p/p}(j,S)$	$E_{p/p}(j,S)$	Nodes per Stage
2	7	3	2.83	0.94	1(3/1) 2(2/1) 4(2/1) | 3(4/2) 5(3/2)
3	5	4	3.92	0.98	1(3/1) 2(2/1) | 3(4/2) 4(1/2) | 5(5/1)
2	4	6	4.95	0.83	1(2/2) 2(1/2) 4(1/2) | 3(2/4) 5(2/3)
2	3	8	6.60	0.83	1(1/3) 2(1/2) 4(1/2) | 3(3/3)* 5(3/2)*
2	2	11	9.90	0.90	1(1/3) 2(1/2) | 3(2/4)* 4(2/1)* 5(2/3)*
3	1	20	19.61	0.98	1(1/3) | 2(1/2) | 3(1/8)* 4(1/2)* 5(1/5)*

Table 3.8 - Pipelining with Parallelism from the Graph

All three approaches

A possible application of this alternative is to replicate the processor obtained by the use of a combination of pipelining and graph parallelism.

For the analysis of its effectiveness the same considerations apply as those discussed in the section on pipelining and graph parallelism and on replication and pipelining.

For the example under discussion, the pipelined implementations in Table 3.8 are replicated as long as $P\ S \leq r$. Table 3.9 shows the possible implementations. The system with the highest speedup obtained with this approach has the following parameters:

$$SU(S = 3, P = 2, j_{tot} = 20) = 39.22$$

and

$$E(S = 3, P = 2, j_{tot} = 20) = 0.98$$

This implementation of the algorithm offers a high speedup with high efficiency.

t_S	S	j	$P=1$	$P=2$	$P=3$	$P=4$	E_{all}
7	2	3	2.83	5.66	8.49	11.32	0.94
5	3	4	3.92	7.84	-	-	0.98
4	2	6	4.95	9.90	14.85	19.80	0.83
3	2	8	6.60	13.20	19.80	26.40	0.83
2	2	11	9.90	19.80	29.70	39.60	0.90
1	3	20	19.61	39.22	-	-	0.98

Table 3.9 - Implementations with All Techniques

The results obtained in implementations with more than one concurrency technique for the example considered are summarized in Figure 3.6. From this figure it is concluded that, for this algorithm, a combination af all techniques is better than the other approaches, excepting in one point (at j=20). Furthermore, it can be noticed that due to the discrete nature of each curve, some implementations of replicated processors(i.e. *rep /graph* (2), *rep /graph* (3)) are also convenient.

Figure 3.7 shows the largest speedup achievable for different number of operation units, using either one concurrency technique or a combination of them. This graph also shows the curve of optimal efficiency (i.e. $E=1$). This figure shows that the selection of a particular implementation depends heavily on the characteristics of the algorithm and the number of units available.

3.1.4 Pipelined Operation Units

As mentioned before, the algorithmic model considered here uses the same type of operation unit for all nodes. Consequently, pipelining the processor requires the replication of the operation units, at least one per stage. This results in implementations with efficiencies that are at most equal to 1.

In contrast to this there are algorithms that use a **different type of operation unit** for each of its nodes. In such case, the sequential implementation cannot share operation units among nodes and therefore requires at least one operation unit for each node. Pipelining this implementation is done by adding staging registers but not operation units. Consequently, the efficiency of the pipelined implementation is

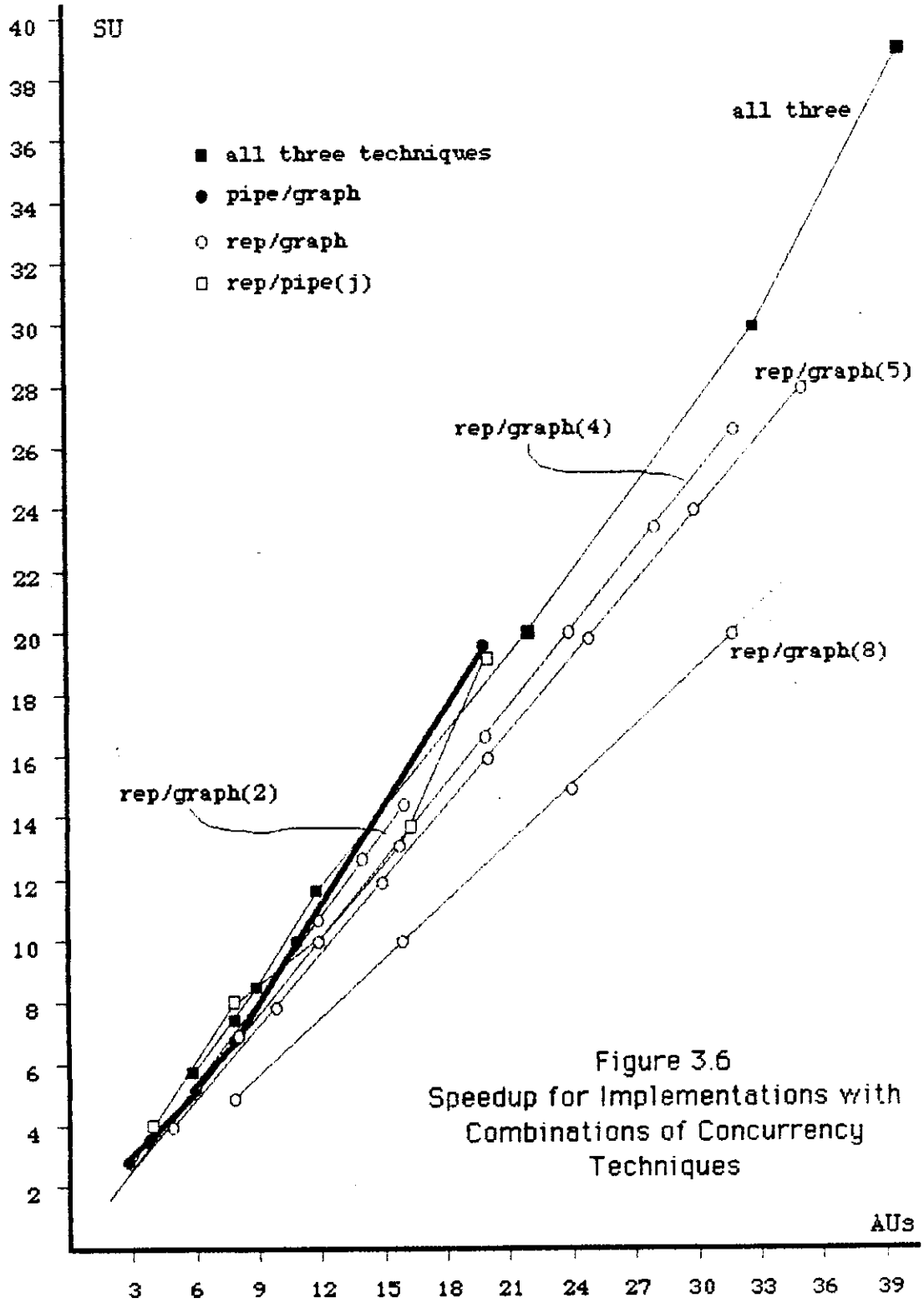

Figure 3.6
Speedup for Implementations with Combinations of Concurrency Techniques

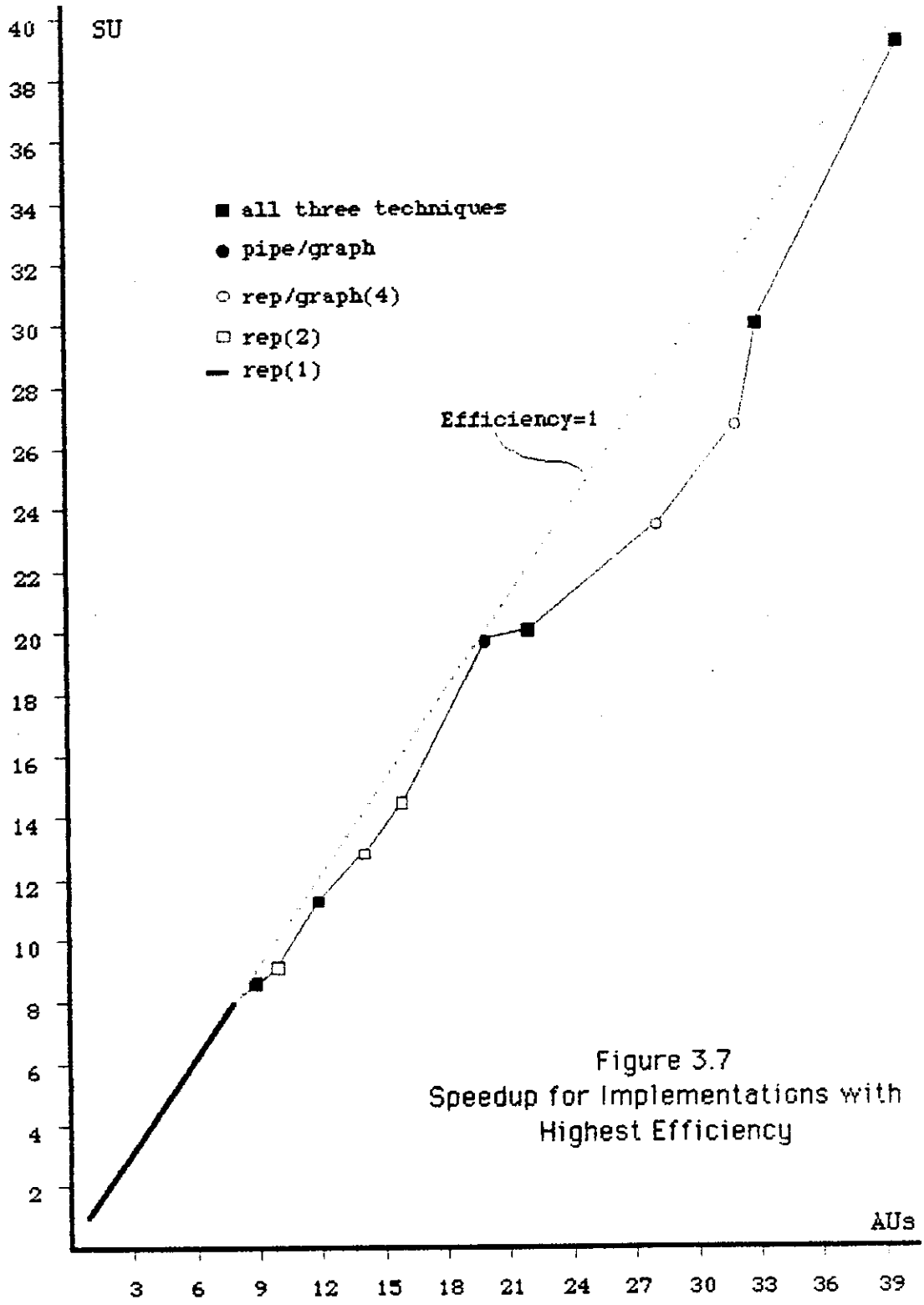

Figure 3.7
Speedup for Implementations with Highest Efficiency

larger than 1 since there is a speedup without an increase in the number of operation units. For a number of instances that is much larger than the number of stages, the speedup and the efficiency are both equal to the number of stages. Note that this higher efficiency comes from the increased number of operation units required for the sequential implementation and not from a reduction in the number of units in the pipelined case.

Moreover, for this algorithmic model, the implementation by replication still has an efficiency of at most 1. Therefore, in such case pipelining is clearly superior to replication.

This type of pipelining is used in the implementation of the SVD for the realization of the operation units themselves. This assumes, of course, that the nodes in which these operation units are used require the execution of more than one operation per instance.

Summary

This section has presented the characteristics of algorithms with concurrent computation capabilities and how these can be utilized in an implementation, using the different techniques suitable for such applications. Performance and cost measures were defined and general characteristics in the evaluation of different alternatives for algorithms was presented. These approaches are now applied to the design of a digital system to compute the SVD, where similar analysis and evaluations will be performed for the particular architectures devised.

3.2 SVD Computation Time and Design Options

As described before, it is necessary to know the basic characteristics of an algorithm and its computation time in a sequential implementation to serve as a reference system against which alternatives using concurrency are compared. This issue is studied first for the SVD algorithm and then the analysis moves into concurrent schemes for it.

3.2.1 SVD Computation Time

It was shown in the previous chapter that the SVD computation according to Brent's version of Hestenes' method consists of performing successive orthogonalizations in a predetermined order called a sweep, in an iterative manner. Due to the convergence properties of the method [Brent82a], a worst case situation of ten sweeps is normally assumed; each sweep consists of $\frac{n}{2}(n-1)$ column orthogonalizations. Therefore, the SVD computation time t_D can be stated in terms of the number of orthogonalizations to be performed as

$$t_D = 10\frac{n}{2}(n-1)\,t_O = 5\,n\,(n-1)\,t_O \tag{3.2}$$

where t_O is the time for each orthogonalization. This expression indicates that the SVD computation corresponds to the repeated invocation of a particular algorithm, in this case the orthogonalization process.

A description of the SVD algorithm using the top-down decomposition methodology stated in section 3.1 is given in Figures 3.8 and 3.9. Figure 3.8d shows a dependence graph for the orthogonalization process, as described in Chapter 2. The figure illustrates the different subfunctions involved in such process and their relationships. The columns-exchange subfunction has been excluded from the orthogonalization graph because it can be done at the same time as the data transfers, and data transfers may be done concurrently with the computations, in no extra time.

Figure 3.9 shows dependence graphs for each of the subfunctions in the orthogonalization process, providing a second level of detail. These graphs allow to infer information regarding the computation time for the orthogonalization process, the concurrency available for the computation and how to exploit it.

3.2.2 SVD Computation Time in a Completely-Sequential Implementation

If the SVD algorithm described in Figures 3.8 and 3.9 is computed in a completely-sequential implementation, that is one where the arithmetic operations are performed using only one arithmetic unit such as those commercially available (i.e. general-purpose floating-point co-processors or other similar device), the SVD computation time is dependent on the orthogonalization time; in turn, this last one is the sum of the times of its different subfunctions. For an $m \times n$ matrix, the orthogonalization subfunctions times are

Inner Product:	t_{ip}	$= m\ _{mult} + (m-1)\ _{add}$
Computation of Angle:	t_θ	$= 3\ _{mult} + 2\ _{div} + 2\ _{add} + 2\ _{sq.root}$

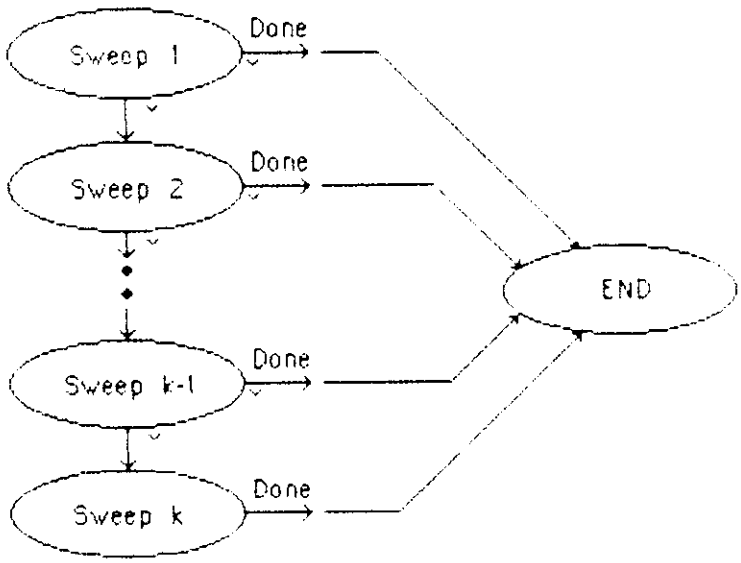

Figure 3 8a - SVD Dependence Graph

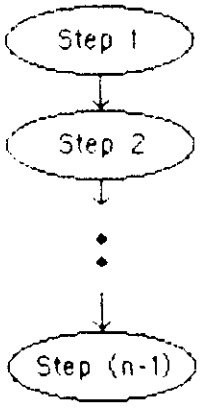

Figure 3 8b
Sweep Dependence
Graph

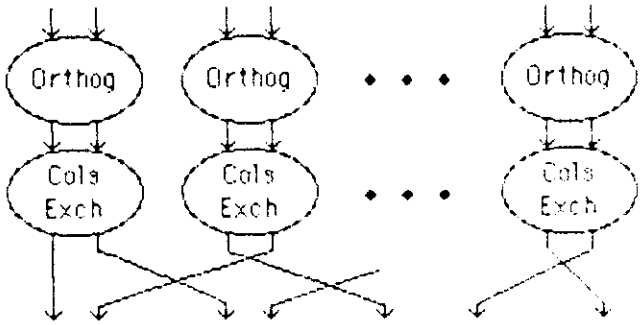

Figure 3 8c
Step Dependence Graph

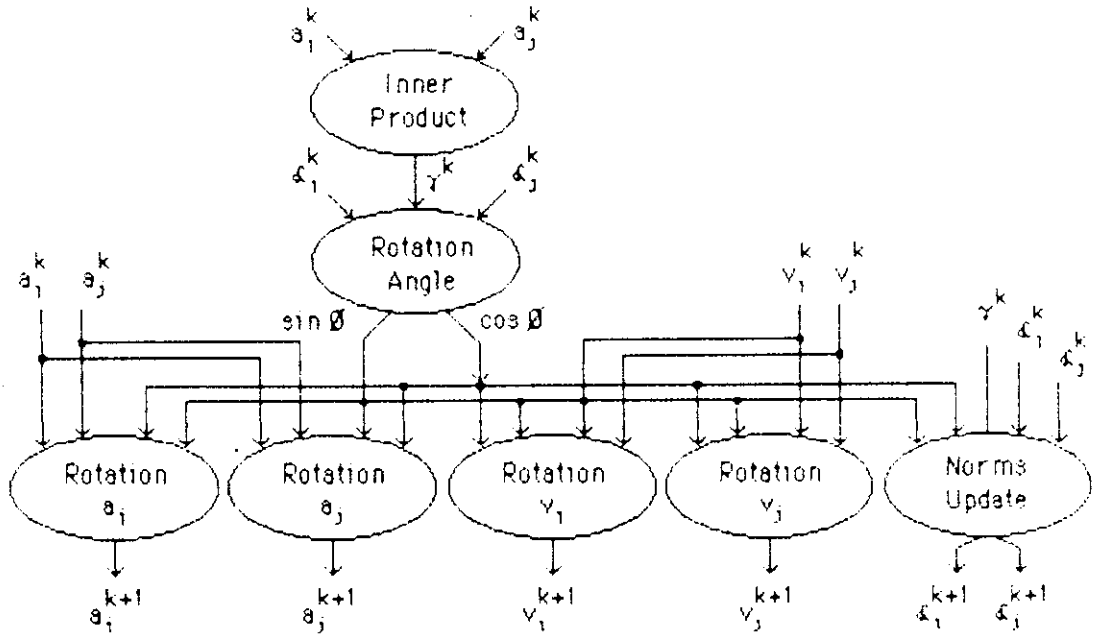

Figure 3.8d - Dependence Graph for the Orthogonalization Process

Columns Rotation: $t_r = 4\,(2\,m_{mult} + m_{add})$

Columns Norm Update: $t_{nu} = 8_{mult} + 4_{add}$

which results in a orthogonalization time of

$$t_O = t_{ip} + t_\theta + t_{nu} + t_r$$
$$= (9m+11)_{mult} + (5m+5)_{add} + 2_{div} + 2_{sq.root}$$

In the equations above, t_x is the time for the operations of subfunction x. These expressions are given in terms of the number of operations required.

To get a simple expression, the following assumptions are made:

- the time to perform a multiplication or an addition is the same
- division and square-root are performed in the equivalent of 9 and 12 multiply times respectively

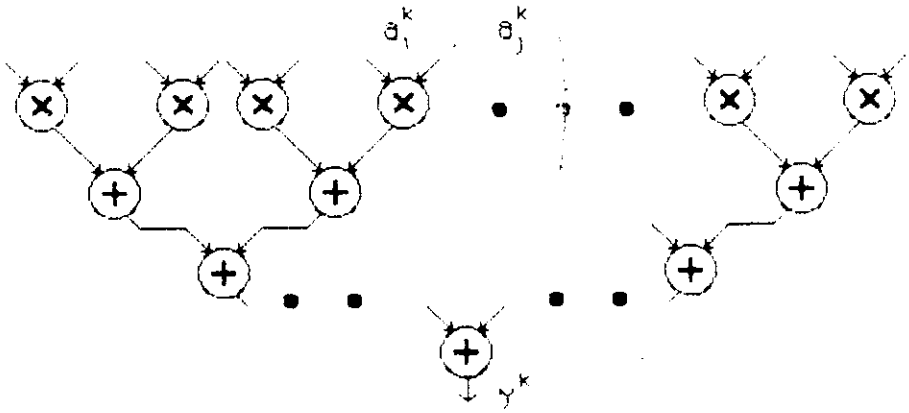

Figure 3.9a - Inner Product Dependence Graph

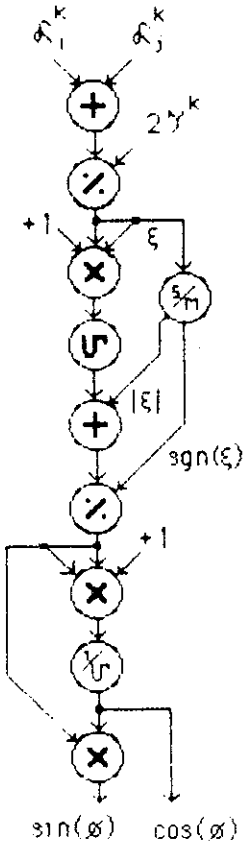

Figure 3.9b
Rotation Angle
Dependence Graph

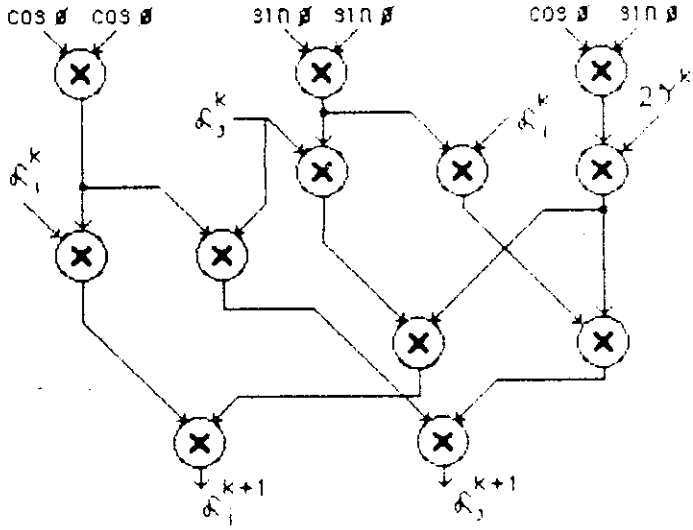

Figure 3.9c - Norms Update
Dependence Graph

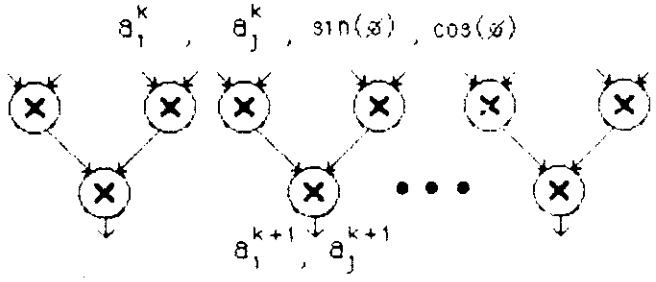

Figure 3.9d - Columns Rotation
Dependence Graph

These assumptions will be discussed later and suitable reasons for them will be provided.

Therefore, in a completely-sequential implementation the orthogonalization computation time in terms of multiply (or add) times becomes

$$t_O = 14m + 58 \quad [ops] \tag{3.3}$$

Substituting (3.3) in (3.2), the SVD computation time becomes

$$t_D = 5n\,(n-1)\,(14m+58) \quad [ops] \tag{3.4}$$

$$\approx 70n\,(n-1)\,(m+4) \quad [ops]$$

This expression and the underlying hardware (one arithmetic unit) will be used as the reference system to evaluate the performance of the architectures discussed later.

3.2.3 Suitability of SVD Algorithm for Concurrent Computation

A completely-sequential implementation for the SVD algorithm will result in a rather slow process, as the computation time above suggest; for instance, 20 by 20 and 40 by 40 matrices imply a decomposition time defined by 638,400 and 4,804,800 floating-point operations, respectively. If a lower computation time is desired, it is possible to exploit the inherent concurrency that exists in the orthogonalization process and the fact that this process is performed many times on independent data.

From the discussion in the previous subsection, considering the dependence graphs in Figures 3.8 and 3.9 and the number of instances the orthogonalization is computed, we infer that the algorithm has characteristics which make it attractive for the different concurrent implementations described before, using one or more of those techniques at several levels. Each technique by itself provides feasible alternatives. It is possible to replicate a sequential implementation of the algorithm, to use the parallelism of the graph or to combine both. There is also possibility for a pipelined design; the orthogonalization process may be partitioned into stages and each one of those stages can operate over a different pair of columns. Furthermore, it is also possible to combine all these different approaches at once.

The concurrency for any scheme derives from the fact that pairs of columns can be processed independently. The existing dependencies are solved at the columns exchange process, before new orthogonalizations start. However, there is a limit to this independence which places a restriction on the maximum concurrency possible in

the computation of the SVD. As described in Chapter 2, at most $n/2$ orthogonalizations may be in execution at any given time. Any successive one requires the result of some of those $n/2$ pairs of columns. This implies that the instances of the computation are divided in groups of size $r=n/2$, with dependences between consecutive groups.

However, any new orthogonalization depends on the outcome of two previous specific ones (two columns are required); that is, actually the **dependences are on two instances of the previous group instead of one** as was assumed in the algorithmic model. This issue is discussed in the next section.

The linear systolic array [Brent82a] has been proposed for the SVD; in it, there are $n/2$ processors and each one of them is used to perform the orthogonalization of one column pair. Figure 3.10 shows its structure; it corresponds to replication of a sequential implementation of the algorithm. Data dependencies are solved implicitly at columns exchange time, because all columns are available to start a new orthogonalization. Additional processors are useless, as there is no more concurrency to exploit through this approach.

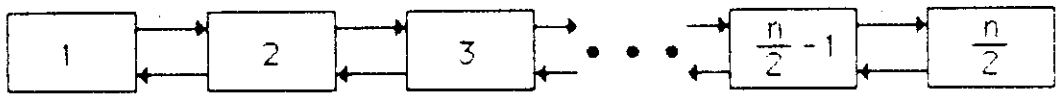

Figure 3.10 - Fully Replicated Architecture
(Linear Systolic Array)

A pipelined-only system is also an alternative for this computation. It will be shown later that it is possible to have up to $(n/2-1)$ stages in a pipelined architecture; more stages are useless, due to the data dependencies in the algorithm. Figure 3.11 shows a scheme for this system.

Between this two extremes, namely fully-replicated and pipelined-only systems, there is a whole set of possible options by combining the different techniques. It has been stated that their combination might provide better efficiency, so it is convenient to look into it. The algorithm is attractive for both types of designs and also for combined versions, that is one with P replicated processors each having S stages, where the stages might also exploit further concurrency. In such case, a "processor" is the hardware required to compute an instance of the problem, or in other words, to perform a complete orthogonalization on a pair of columns. This is the topic of the next section; the fully-replicated and pipelined-only versions are actually just particular cases of the general scheme presented there.

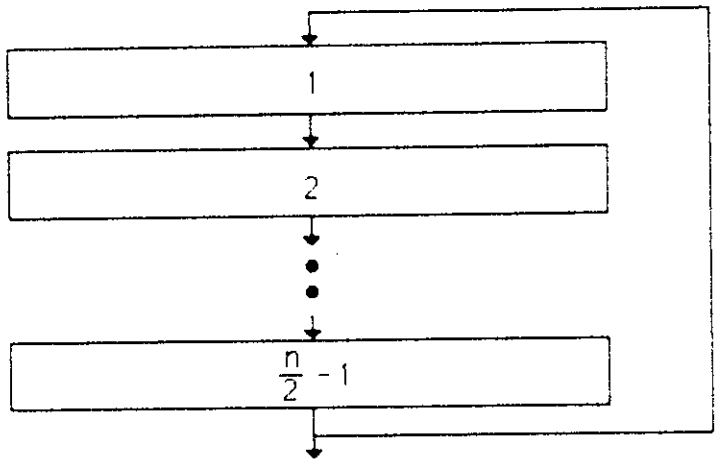

Figure 3.11 - Pipelined Architecture

3.3 A Replicated/Pipelined System for the SVD

Pipelining a fully replicated system to compute the singular value decomposition (i.e. one with $n/2$ processors) is not adequate for the same reasons that make more replicated processors useless: the throughput would not increase as there is no more concurrency to exploit. Recall that the computation is performed at most in groups of $r = n/2$ instances. If the $n/2$ processors are pipelined, then only one stage in each processor would be in use at any time and the others stages would be idle; consequently, the throughput would remain the same.

However, the orthogonalization algorithm is a sequential one. Therefore, less replication of pipelined units might be a reasonable architecture, which could have the advantages of both replication and pipelining. Consequently, it is of interest to study the throughput achievable in a system with a given number of replicated processors and stages per processor. This structure is called here a P/S system, which stands for a system with P processors and S stages each, as illustrated in Figure 3.12.

However, one problem exists in such scheme: the data dependencies introduced by the columns-exchange process. This issue is studied before the throughput characteristics are discussed.

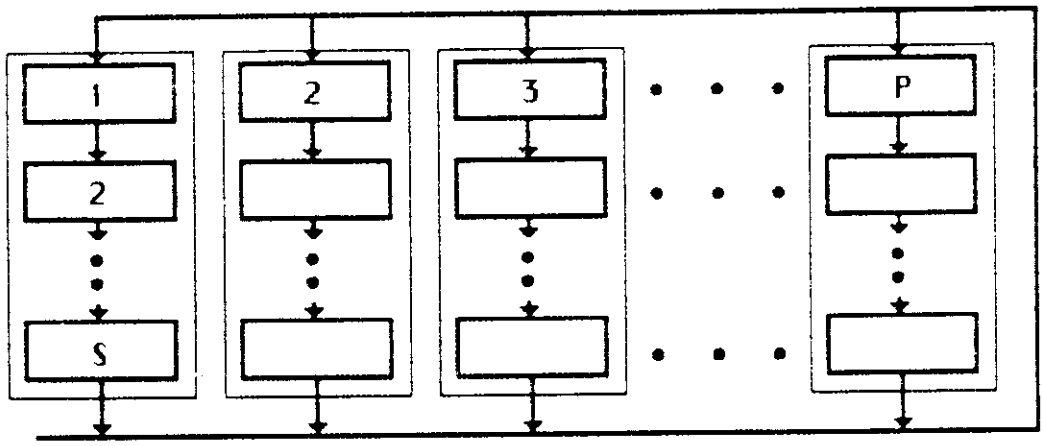

Figure 3.12 - Architecture of *P/S* System

3.3.1 Columns Exchange Process in a Parallel/Pipelined System for the SVD

The algorithmic model described in section (3.1) considers that the M instances of the computation are divided into groups of r instances and that there exists a dependence between the same instance in two consecutive groups. As a result of that, a system which uses replication and pipelining is required to satisfy the relation $PS \leq r$.

In the computation of the SVD each instance corresponds to an orthogonalization, which uses two columns of the matrix. The number of instances M corresponds to the number of these orthogonalizations required for convergence, as inferred from equation (3.2). The group size r corresponds to the number of successive independent orthogonalizations, which is the number of orthogonalizations in one step in a sweep; in turn, this number is defined by the number of columns in the matrix such that $r = n/2$.

However, the SVD algorithm is characterized by the fact that each new instance depends on the outcome of two instances in the previous group instead of only one as the model assumed, and the dependence is not identical for every instance. Therefore, it is necessary to study this issue and modify the relation $PS \leq r$ or equivalently $PS \leq n/2$ accordingly.

In the P/S system, at any given time there are $2PS$ columns being processed (two at each stage in each processor); assuming that $P < n/2$ and that the number of columns n is a multiple of P, then these columns are orthogonalized in sets of $2P$ as shown in Figure 3.13.

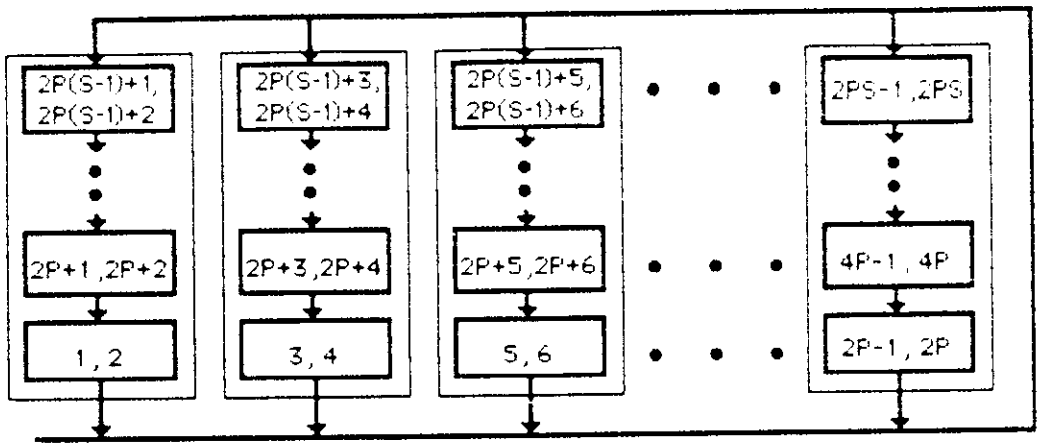

Figure 3.13 - Columns within *P/S* Architecture at a given time

If the columns are exchanged as described in Chapter 2 [Brent82a], the following alternative situations arise regarding this process and the data dependencies for each step in a sweep (the outputs of the system are the outputs of the last stage in each processor):

a.- The outputs of the system correspond to the $2P$ leftmost columns of the matrix, i.e. $a_1, a_2, \ldots a_{2P}$. To exchange these columns before performing another orthogonalization one extra column is required, namely a_{2P+2}, as shown below (the first row represents the columns placement within the processors before the exchange, while the second after it is done).

Proc. 1		Proc. 2			Proc. $P-1$		Proc. P	
a_1	a_2	a_3	a_4	--	a_{2P-3}	a_{2P-2}	a_{2P-1}	a_{2P}
a_1	a_4	a_2	a_6	--	a_{2P-5}	a_{2P}	a_{2P-3}	a_{2P+2}

Column a_{2P-1} is not selected at this exchange time while column a_{2P+2} is the extra one required. This column a_{2P+2} belongs to the next set of columns being orthogonalized. Notice that column a_1 keeps its position in processor 1, while all other columns change processor.

b.- The outputs of the system correspond to $2P$ internal columns of the matrix, i.e. they do not contain any of the $2P$ leftmost or $2P$ rightmost columns. Exchanging these requires one column from the previous orthogonalized group and one from the next group, as shown below (assuming this is the second group of $2P$ columns):

Proc. 1		Proc. 2			Proc. $P-1$		Proc. P	
a_{2P+1}	a_{2P+2}	a_{2P+3}	a_{2P+4}	--	a_{4P-3}	a_{4P-2}	a_{4P-1}	a_{4P}
a_{2P-1}	a_{2P+4}	a_{2P+1}	a_{2P+6}	--	a_{4P-5}	a_{4P}	a_{4P-3}	a_{4P+2}

Column a_{4P-1} is not selected at this exchange operation while columns a_{2P-1} and a_{4P+2} are the extra ones required. Column a_{2P-1} belongs to the previous set of columns orthogonalized, while column a_{4P+2} belongs to the next set of columns being orthogonalized. All columns change processor.

c.- The outputs of the system correspond to the $2P$ rightmost columns of the matrix. To exchange these columns, only one column from the previous group is required, as shown below. Therefore, this group does not have dependencies because the data required is already available.

Proc. 1		Proc. 2			Proc. $P-1$		Proc. P	
a_{n-2P+1}	a_{n-2P+2}	a_{n-2P+3}	a_{n-2P+4}	--	a_{n-3}	a_{n-2}	a_{n-1}	a_n
a_{n-2P-2}	a_{n-2P+4}	a_{n-2P+1}	a_{n-2P+6}	--	a_{n-5}	a_n	a_{n-3}	a_{n-1}

Column a_{n-2P-2} is the one needed at this exchange and it belongs to the previous group of columns orthogonalized. All columns change processor, except a_{n-1} which exchanges position within processor P.

At any given time, the columns-exchange function has to provide P new columns pairs to start that many new orthogonalizations and keep all processors busy. Considering case (b) in the discussion above, because it is the most stringent one, it follows that with the P columns pairs input to the columns-exchange function at any given time this function can only produce $(P-1)$ new pairs and one of those pairs uses a column from the previous set of columns orthogonalized. To produce the remaining column pair, one column from the next set of columns being orthogonalized is required and consequently the generation of such pair should be delayed until the corresponding column becomes available. But, as stated above, the system requires that P pairs be produced at each exchange time. Therefore, to keep all processors busy, this missing pair must be produced with data available to the exchange function in advance to the particular exchange time.

This implies that the P/S system, which has $2PS$ columns under processing at any time, will always have data available to process if the number of columns in the matrix satisfies the relationship

$$n \geq 2PS + 2 \tag{3.5}$$

because, besides the columns within the system, at least one extra pair must always be available to solve the dependencies at exchange time.

Notice that equation (3.5) modifies the corresponding expression for the algorithmic model given in section 3.1 regarding the number of instances required to keep the replicated pipelined processors busy, but the group size regarding dependences in the computation is still $r = n/2$. In terms of r equation (3.5) becomes

$$PS + 1 \leq r \tag{3.5b}$$

As an example, $n = 26$ satisfies this relation for a $P = 4/S = 3$ system. In such case, columns are processed as shown in Figure 3.14 (where the double bar separates different steps in a sweep).

t	Processors Input								Processors Output							
0	1	2	3	4	5	6	7	8								
1	9	10	11	12	13	14	15	16								
2	17	18	19	20	21	22	23	24								
3	25	26	1	4	2	6	3	8	1	2	3	4	5	6	7	8
4	5	10	7	12	9	14	11	16	9	10	11	12	13	14	15	16
5	13	18	15	20	17	22	19	24	17	18	19	20	21	22	23	24
6	21	26	23	25	1	6	4	8	25	26	1	4	2	6	3	8

Figure 3.14 - Starting Orthogonalization Times in $P = 4$, $S = 3$, $n = 26$ System

In this figure, step 1 of the first sweep starts at t=0, when four column-pairs begin the orthogonalization process; other column sets are input at t=1 and t=2. The situation at t=3 depicts a representative example of the discussion above: the first set of four columns is available for exchange (at the output of the processors) and therefore to start a new step, but it can only produce three new pairs; the missing pair is obtained using column-pair (25-26), which corresponds to the last pair from the previous step which has not been orthogonalized yet. As a result of this, four pairs are produced at this exchange time. Similar situations exist for every exchange time thereafter, keeping all pipelined processors fully busy.

If $n > 26$ in the example, then when the first group of columns is ready for the exchange operations (i.e. at t=3 above) there are columns from the previous step still waiting to be processed; therefore, those columns are used first and there are no limitations in the data input to the processors as a result of the data dependencies in the exchange process.

This analysis indicates that the exchange columns function must be able to store some columns internally to achieve its purpose, as it has to combine data from different set of columns which become available at different times, the amount of internal storage and its actual contents will be discussed later.

Notice that no requirement on the data bandwidth for the columns-exchange process has been stated so far. However, this issue is important and is discussed now.

In the linear systolic array [Brent82a] the data exchange process for a matrix with $n = 40$ is as shown in Figure 3.15; this figure depicts the input ordering of columns in the first row and the corresponding output ordering in the second. In this scheme, all exchanges are with neighbor processors only so it is very regular, simple and reduces data communications bandwidth requirements, attractive characteristics for hardware implementation. To obtain the same properties in systems with fewer replicated processors, it is necessary to assign a special order to the columns within the processors.

1	2	3	4	5	6	7	8	-	-	33	34	35	36	37	38	39	40
1	4	2	6	3	8	5	10	-	-	31	36	33	38	35	40	37	39

Figure 3.15 - Columns Exchange for Matrix with $n = 40$ in Linear Systolic Array

Figure 3.16 shows this special ordering for a four-processor system and $n = 40$, which is assumed to have a number of stages such that the relation $n \geq 2PS+2$ is satisfied (that is, there are enough columns to keep all processors fully busy and solve all dependences). Each row in the left side of this figure represents the sets of columns input to the columns-exchange function; these columns are output from the replicated processors in the same order. The resulting ordering after the exchange process must be as indicated in the right side of the same figure.

Columns Exchange Input								Columns Exchange Output							
P1		P2		P3		P4		P1		P2		P3		P4	
2	1	3	4	5	6	7	8	4	1	2	6	3	8	5	10
16	15	14	13	12	11	10	9	18	13	16	11	14	9	12	7
17	18	19	20	21	22	23	23	15	20	17	22	19	24	21	26
32	31	30	29	28	27	26	25	34	29	32	27	30	25	28	23
33	34	35	36	37	38	39	40	31	36	33	38	35	40	37	39

Figure 3.16 - Columns Ordering in $P = 4$, $n = 40$ System

In this scheme, column 1 does not change its position within processor 1, while all other columns change either processor or position within the same processor. As a result of column 1 not changing its position, its new companion (i.e. column 4) requires a special transfer; all other columns change their location in the same manner, as follows:

- the left columns of P1, P2 and P3 go to the left columns of P2, P3 and P4, respectively
- the right columns of P2, P3, P4 go to the right columns of P1, P2 and P3, respectively
- the right column of P1 goes to the left column of P1, while the left column of P4 goes to the right column of P4

Figure 3.17 depicts these exchanges. Notice that all data transfers are between neighbor processors, in the same way as in the linear systolic array [Brent82a]. The dashed lines represent the connections required to exchange columns 1 and 4, while the solid lines are for all other exchanges.

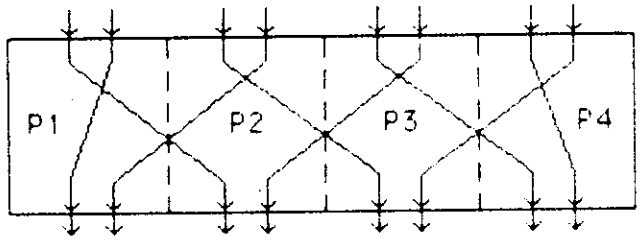

Figure 3.17 - Columns Exchange Process in $P=4$ System

The important feature of this scheme is that its behavior is the same for any number of replicated processors.

The remaining issue to be solved relates to when columns are selected for exchange and how many columns must be stored at any time in the exchange logic to solve the data dependencies in the process. Figure 3.18 illustrates this situation using again the example with 4-processors and $n = 40$, but including the time when columns are input to and output from the exchange function. Again, it is assumed that the number of stages is such that there are columns available to be orthogonalized independent from the columns-exchange output.

t	Columns Exchange Input								Columns Exchange Output							
	P1		P2		P3		P4		P1		P2		P3		P4	
0	2	1	3	4	5	6	7	8								
1	16	15	14	13	12	11	10	9	4	1	2	6	3	8	5	10
2	17	18	19	20	21	22	23	24	18	13	16	11	14	9	12	7
3	32	31	30	29	28	27	26	25	15	20	17	22	19	24	21	26
4	33	34	35	36	37	38	39	40	34	29	32	27	30	25	28	23
5	4	1	2	6	3	8	5	10	31	36	33	38	35	40	37	39
6	18	13	16	11	14	9	12	7	6	1	4	8	2	10	3	12
7	15	20	17	22	19	24	21	26	20	11	18	9	16	7	14	5

Figure 3.18 - Columns Exchange Process for $P = 4$, $n = 40$ System

In this figure, at t=0 no columns are exchanged since there is not enough data to produce the four pairs required. At each successive exchange time, a set of columns is produced using data from the set of columns currently at the input of the exchange logic plus the last two sets of inputs; actually, one column is used from the set currently at the input, $(2P - 2)$ columns from the previous input, and one column from the set two time periods before. The only exception is the first exchange for each step in a sweep, which uses the data from the set currently at the input and the previous one only.

Therefore, this scheme requires the exchange logic in each processor to store the last two sets of columns it has orthogonalized; that data plus the set currently at the input of this logic are required to solve all data dependencies in the exchange process.

Summary

This section has shown that, as in a linear systolic array, a P/S system can also have a columns-exchange process that is simple, regular, and with nearest neighbor communications, attractive characteristics for hardware implementation. The only requirements for such exchange process are to store the columns of the matrix into the processors in a particular order, and to provide the columns-exchange logic with memory to store the last two sets of columns orthogonalized.

3.3.2 Throughput Characteristics of a Replicated/Pipelined System for the SVD

Given then that $n \geq 2PS+2$, (i.e. equation (3.5) is satisfied) a replicated / pipelined system for the SVD is always completely busy. Under these circumstances, the time for the entire decomposition can be expressed by

$$t_D = \left[S + \left[\frac{N_O}{P} - 1 \right] \right] t_S \tag{3.6}$$

where

t_D	:	decomposition time
t_S	:	pipeline stage time
N_O	:	number of orthogonalizations to perform
P	:	number of replicated processors
S	:	number of stages per processor

The stage time is related to the time for a complete orthogonalization. Ideally, it should be possible to partition the orthogonalization process into S stages of equal time, as it would provide the highest throughput for that number of stages. If that is the case, then

$$t_S = \frac{t_O}{S} \tag{3.7}$$

where t_O is the time of each orthogonalization.

Recalling that $N_O = 10\frac{n}{2}(n-1)$ and substituting (3.7) in (3.6), the total time for the decomposition is obtained as

$$t_D = \left[\frac{10n(n-1)}{2PS} + (1 - \frac{1}{S}) \right] t_O \tag{3.8}$$

$$= \left[\frac{5n(n-1)}{PS} + (1 - \frac{1}{S}) \right] t_O$$

From this expression it follows that, if it is possible to partition the orthogonalization process into stages of equal length, the total SVD time is related to the product PS (the number of replicated processors and pipeline stages), as long as condition (3.5) is satisfied. The term $(1-\frac{1}{S})t_O$ corresponds to the pipeline latency and is

negligible when compared to the other term in equation (3.8).

This means that, with $n \geq 2PS+2$, replicated-only (i.e. $S=1$), pipelined-only (i.e. $P=1$) or combined designs, all with the same value for the term PS, have the same computation time for the SVD in terms of the orthogonalization time. This is exactly the same conclusion stated in section 3.1, when the characteristics of replication and pipelining were studied for an algorithm which captured the properties of the SVD computation.

The obvious question is how the computation time is affected when equation (3.5) does not hold. If the number of computing units (i.e all stages in all processors) is such that $n < 2PS + 2$, then there will be times when stages are idle because there is no new data to process. An extreme example is $n/2$ processors and S stages: after the $n/2$ processors start performing the corresponding orthogonalizations, no more data is available until the entire pipeline is completely traversed by the data; in this case $2PS + 2 = nS + 2 > n$, for any number of stages.

In those situations, orthogonalizations will be started until no more columns are available. The system will continue performing the computations for those sets of data but some stages of the pipeline will be idle until data starts flowing out of the last stage, columns are exchanged, and fed back into the pipeline inputs. This is equivalent to extending the matrix dimensions such that equation (3.5) holds, i.e. $n = 2PS + 2$; as a result of it, the problem size is extended.

Therefore, once the limiting condition given by equation (3.5) has been reached, it doesn't matter how much hardware is introduced in the form of more pipeline stages or replicated processors; the time to compute the decomposition is always the same and the extra hardware is idle part of that time.

An example for the situation described is shown in Figure 3.19, where the starting times of the orthogonalizations are given for $P=4$, $S=3$ and $n=20$. However, instead of extending the matrix dimensions as described above, idle times have been moved around to start the orthogonalizations as soon as possible; furthermore, to simplify the example, the ordering of the columns within the processors is not the particular one presented before but a simple sequential one. Under these conditions, the orthogonalizations are started in cycles such as the one shown in Figure 3.19. For the example considered, the cycle takes thirteen time periods and then repeats itself.

t	Processors Input								Processors Output							
0	1	2	3	4	5	6	7	8								
1	9	10	11	12	13	14	15	16								
2	17	18	19	20	--	--	--	--								
3	1	4	2	6	3	8	--	--	1	2	3	4	5	6	7	8
4	5	10	7	12	9	14	11	16	9	10	11	12	13	14	15	16
5	13	18	15	20	17	19	--	--	17	18	19	20	--	--	--	--
6	1	6	4	8	--	--	--	--	1	4	2	6	3	8	--	--
7	2	10	3	12	5	14	7	16	5	10	7	12	9	14	11	16
8	9	18	11	20	13	19	15	17	13	18	15	20	17	19	--	--
9	1	8	--	--	--	--	--	--	1	6	4	8	--	--	--	--
10	6	10	4	12	2	14	3	16	2	10	3	12	5	14	7	16
11	5	18	7	20	9	19	11	17	9	18	11	20	13	19	15	17
12	13	15	--	--	--	--	--	--	1	8	--	--	--	--	--	--
13	1	10	8	12	6	14	4	16	6	10	4	12	2	14	3	16
14	2	18	3	20	5	19	7	17	5	18	7	20	9	19	11	17
15	9	15	11	13	--	--	--	--	13	15	--	--	--	--	--	--

Figure 3.19 - Starting Orthogonalization Times in P=4, S=3, n=20 System

Including the case for $n < 2PS + 2$ described above, equation (3.8) becomes

$$t_D = \begin{cases} \left[\frac{5n(n-1)}{PS} + (1-\frac{1}{S})\right] t_O & \text{if } n \geq 2PS+2 \\ \left[\frac{10n(n-1)}{(n-2)} + (1-\frac{1}{S})\right] t_O \approx \left[10n + (1-\frac{1}{S})\right] t_O & \text{if } n < 2PS+2,\ P < \frac{n}{2} \end{cases} \tag{3.9}$$

The last expression in this equation is obtained from the first one by substituting $n = 2PS+2$. This is an upper bound for t_D, because the columns introduced do not need to be orthogonalized with the real columns of the matrix and equation (3.9) assumes they are. It can be shown that if these extra columns do not take part of the orthogonalization process (i.e. they are not orthogonalized with the real columns) but are just idle times introduced for the stages, the starting times for orthogonalizations produce cycles as in Figure 3.19; these cycles generate an average orthogonalization time $t^*_O = \frac{2(PS+1)}{n}$.

If t^*_O is used instead of t_O then the second expression in equation (3.9) becomes

$$\begin{aligned} t_D &= \left[\frac{10(n-1)(PS+1)}{PS} + (1-\frac{1}{S})\right] t_O \quad \text{if } n < 2PS+2,\ P < \frac{n}{2} \\ &\approx \left[10(n-1) + (1-\frac{1}{S})\right] t_O \end{aligned} \tag{3.10}$$

For either expression ((3.9) or (3.10)), when equation (3.5) does not hold the SVD time is approximately constant, independent of the values for P and S (ignoring again the latency of the pipeline).

The case $P = \frac{n}{2}$ has been excluded from the equations above, as it is a special one. In such situation, all columns are orthogonalized simultaneously and the dependences in the exchange process are solved implicitly. This corresponds to the linear systolic array, which has already been shown as unable to use pipelining to increase throughput. The decomposition time in this case is given by the total number of orthogonalizations divided by the number of processors, which is

$$t_D^{**} = 10(n-1)\, t_O \tag{3.11}$$

This is the same value obtained in (3.10), excluding the latency of the pipeline.

Figure 3.20 shows a graph of the throughput for an SVD system in terms of the orthogonalization time, for different values of n and $1 \le S < n/2$, $1 \le P < n/2$ and as a function of the number of processing units PS. A special entry is provided for $P = \frac{n}{2}$, for the reasons stated above.

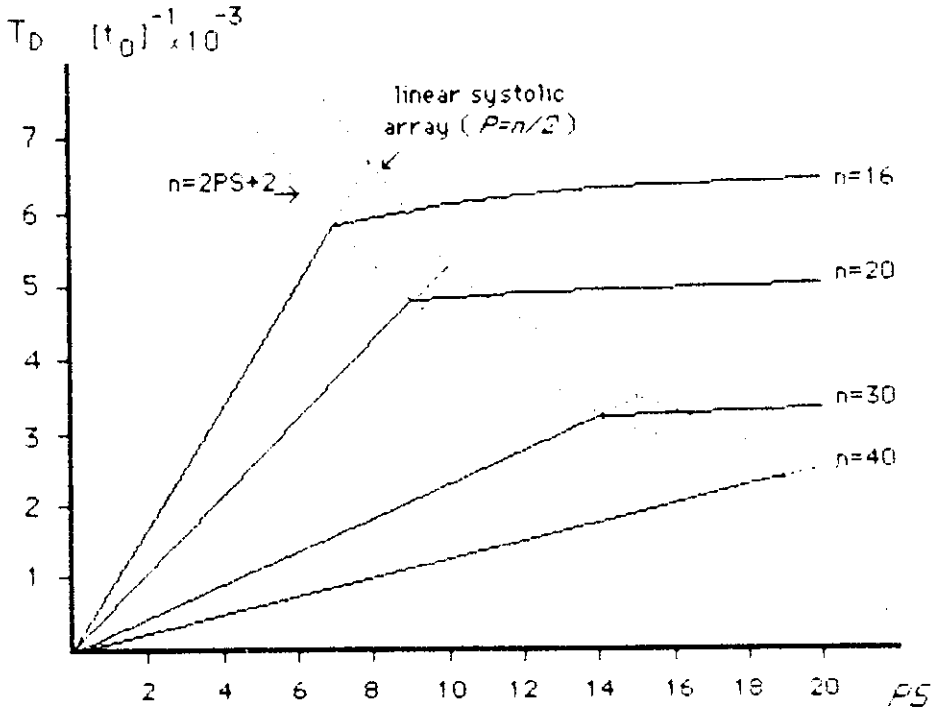

Figure 3.20 - Throughput Characteristics for P/S System

This graph shows the gain obtained with more processing power in the system, up to the limiting condition given by equation (3.5). It also shows that, after the linear systolic array, the next fastest architecture is related to the number of columns in the matrix. For instance, with $n = 16$ the best choice is $PS = 7$, which can only be achieved in a system with one processor and seven stages; with $n = 20$, $PS = 9$ allows two options: $P = 3 / S = 3$ or $P = 9 / S = 1$.

These results could be expected, according to the characteristics for replicated and pipelined systems presented at the beginning of this chapter. The P/S system described so far corresponds to the case of replication of a pipelined implementation of the orthogonalization algorithm. The analysis here has shown that this replication is possible, because data dependencies can be solved with the columns exchange scheme proposed.

It was also stated that exploiting the parallelism of the graph for an algorithm might be convenient, particularly in implementations which replicate a pipelined processor as it is the case here. With that approach, the orthogonalization throughput and its hardware requirements are directly related to how the concurrency in the graph is

exploited. Therefore, conclusions about the most adequate scheme in terms of actual speed and resources required for the SVD can only be obtained after the orthogonalization characteristics are studied. That is the subject of the next chapter.

CHAPTER 4
ORTHOGONALIZATION THROUGHPUT

The previous chapter has shown that the SVD throughput is proportional to the orthogonalization throughput, becoming necessary to study this last one. This throughput is directly related to the amount of arithmetic hardware available to compute each of the subfunctions and how those resources are used, because it is possible to exploit the concurrency in the orthogonalization algorithm. To address these issues, this chapter studies the orthogonalization subfunctions. The goal is to provide the most efficient implementation, with highest throughput, for a given amount of hardware. Only arithmetic hardware is considered here, because more units of this kind are needed than other hardware resources (the algorithm is compute-bound), and consequently they represent the highest cost in an implementation.

The concurrency possible during the orthogonalization process and the precedences between the different subcomputations have been exhibited in the corresponding dependence graphs, which are shown again in Figure 4.1.

Of the different techniques described in Chapter 3 to increase throughput in a digital system, an approach combining those techniques is more adequate for the orthogonalization process. Two issues indicate that a scheme combining replication, pipelining and parallelism of the graph is more convenient:

- the orthogonalization algorithm is a sequential one with different requirements at different steps in the computation, as seen from the dependence graphs
- the computation has to be performed many times with different data.

According to the methodology described before, the subfunctions in the orthogonalization algorithm are studied first to determine their computation time and their characteristics towards a pipeline for the orthogonalization (i.e. number of operation units required for each and their individual throughput). Each stage in such pipeline is comprised of one or more subcomputation nodes, which are considered indivisible.

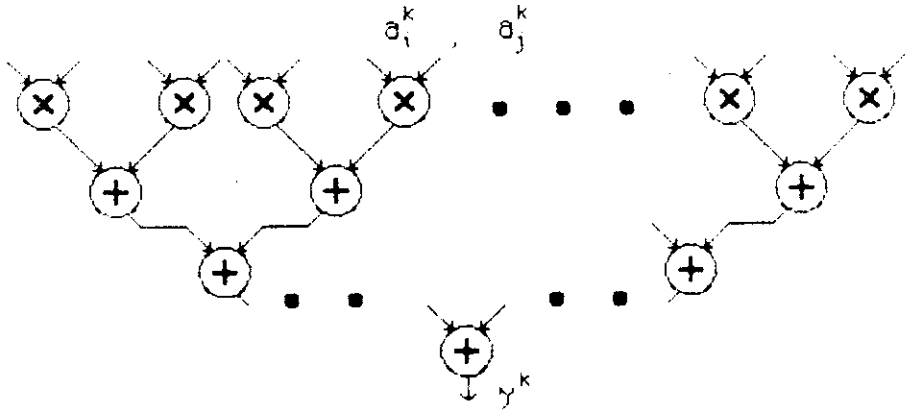

Figure 4.1a - Inner Product Dependence Graph

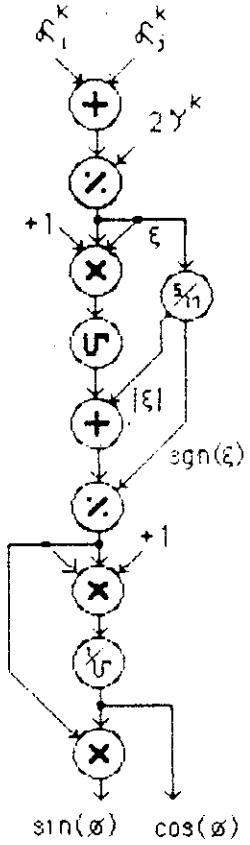

Figure 4.1b
Rotation Angle
Dependence Graph

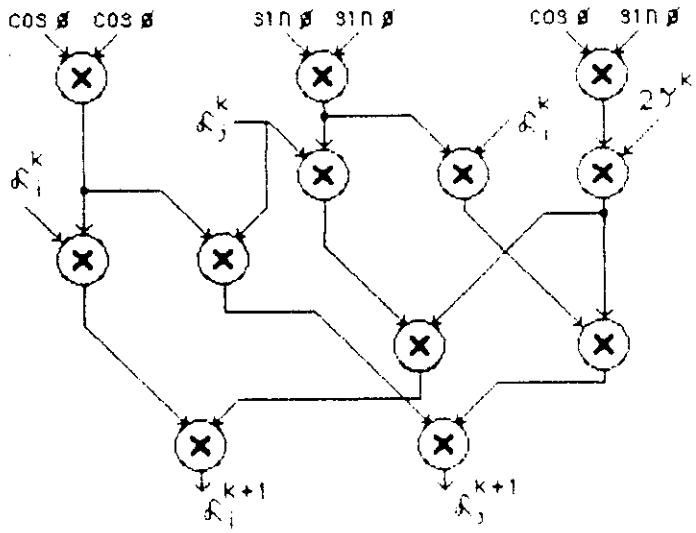

Figure 4.1c - Norms Update
Dependence Graph

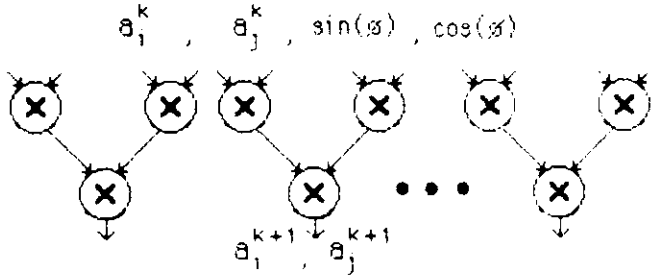

Figure 4.1d - Columns Rotation
Dependence Graph

A second step consists in studying the subfunctions, looking for possible implementations of further concurrency within the subcomputations. This might result in new levels of concurrency, internal to the nodes. Additionally, depending on the nodes characteristics, it might be convenient to partition certain nodes into less complex ones to have finer granularity in the orthogonalization algorithm. However, the need for such refinements should arise in an evaluation of an initial architecture, and their implementations should be part of a bottom-up iteration in the design process.

For this analysis, the hardware is implemented around arithmetic units (AU) capable of performing one floating-point multiplication or addition at a time. It could perform any of the two operations or be dedicated just to a particular one. Current technology developments have made it possible to reach comparable speed for floating-point addition and multiplication (for 32-bit values around 100 nsec is a feasible figure nowadays for both operations) [AMD84, Uya84]. Therefore, the analysis is done in terms of the number of multiplications and additions, assumed of equal computation time.

4.1 Computation Time and Throughput for the Orthogonalization Subfunctions

Expressions for the computation time and throughput for each orthogonalization subfunction are obtained now, in terms of the number of operations in each and as a function of the number of arithmetic units required. The results reported here actually represent two steps in the top-down methodology, because nodes identified as critical in a first design attempt have been studied further.

4.1.1 Columns Rotations

The computation in the columns rotation subfunction is described by the corresponding dependence graph in Figure 4.1d. Such graph shows that there is no dependence between the different operations involved besides what exists in the multiplication / multiplication / addition process. Unless the full parallelism available is used, the two sequential steps in this computation are performed over subsets of the input data, in a pipelined fashion. The fully parallel option is not studied, because its cost in terms of the number of arithmetic units required is high ($12m$ arithmetic units, as inferred from the analysis below).

This subfunction is best implemented around G multiplier / multiplier / adder units (M/M/A), as they take advantage of the concurrency in the node. Therefore, the architecture for the node is the same as the dependence graph shown in Figure 4.1d, but with G parallel units only. It becomes one stage of the orthogonalization pipeline, with an internal pipeline of two stages: a level of multipliers and a level of adders.

The amount of hardware resources is $3G$ AUs, as each M/M/A unit requires three AUs.

The throughput of the node is given by the time required to rotate the four columns (two from each matrix A and V) with m elements each. The computation process consists in taking the $4m$ elements, dividing them into groups of G and processing those groups sequentially. The time to perform this computation is

$$t_r = \left\lceil \frac{4m}{G} \right\rceil + 1 \tag{4.1}$$

The first term in equation (4.1) represents the time for the $4m$ elements to traverse the multipliers in the G units, while the second is due to the time required to perform the last addition.

The corresponding throughput is

$$T_r = \frac{1}{\left\lceil \frac{4m}{G} \right\rceil} \tag{4.2}$$

because there is no need to wait for one computation to finish completely (i.e. perform the last addition) before the next one can start. The throughput for the internal pipeline is 1, i.e. the time between two successive data inputs is one operation.

4.1.2 Norms Update

This subfunction is a serial set of operations, with some parallelism at each step. Letting NU represent the number of AUs for it, from the dependence graph in Figure 4.1c the time to compute this subfunction is

$$t_{nu} = \begin{cases} 12 & \text{if } NU = 1 \\ 5 & \text{if } NU = 3 \\ 4 & \text{if } NU \geq 5 \end{cases} \tag{4.3}$$

The throughput of this node is low, as can be inferred from equation (4.3). To increase the throughput of this subfunction, a refinement step was applied to it which showed that it can be divided into any number of nodes up to the total number of operations in it (assuming that precedences are added to the graph to make it as sequential as necessary). These new nodes become nodes in the orthogonalization algorithm and therefore they become stages in the orthogonalization pipeline. The

resulting computation time depends on the partition into the new nodes and the number of AUs in them.

Assuming that only one AU per node is used, then the partitioning is applied to a completely-sequential implementation for the node and $t_{nu} = 12$. Therefore, if S_{nu} is the number of stages for this subcomputation in the orthogonalization pipeline (or in other words the number of nodes created), the throughput is

$$T_{nu} = \frac{1}{\left\lceil \frac{12}{S_{nu}} \right\rceil} \tag{4.4}$$

because each new node must correspond to an integer number of operations; the hardware requirements are S_{nu} AUs.

4.1.3 Computation of Angle for Rotation

This subfunction is just a sequence of operations, as can be inferred from the dependence graph in Figure 4.1b. Since division and square root operations are required in it and as the other subcomputations in the orthogonalization process are based on fast AUs, it seems proper to use the same high-speed hardware to compute these two operations through a multiplicative approach. It will be shown later that for the data format considered here, it is possible to perform them in the equivalent to 9 and 12 multiply times, respectively. This approach reduces hardware diversity while it provides adequate computation time for both operations.

Under this assumption and from the dependence graph, the time to compute the angle for the rotation becomes:

$$t_{\theta} = 47 \tag{4.5}$$

The throughput for this subfunction is also low and it can be improved by applying the same refinement technique used for the norms update node, namely divide this subfunction into any number of stages or nodes up to the total number of operations, with one AU per node. If S_{θ} is the number of nodes or stages created, then the throughput T_{θ} is

$$T_{\theta} = \frac{1}{\left\lceil \frac{47}{S_{\theta}} \right\rceil} \tag{4.6}$$

and the hardware requirements are S_{θ} AUs.

The norms update and angle computation subfunctions have similar properties regarding their capabilities to be partitioned into new nodes, which become stages of the orthogonalization pipeline. If both subfunctions are combined into one, the total granularity of the resulting node is higher, with the same serial dependence. Therefore, grouping them provides better resolution for the partition into stages or refinement process.

Let the two subfunctions combined have the following parameters:

$S_{\theta/nu}$	:	number of stages in the resulting subfunction
$t_{\theta/nu}$	:	computation time for the resulting subfunction
$T_{\theta/nu}$	:	throughput for the resulting subfunction stages

The computation time for the two subfunctions combined is the sum of the individual subfunctions values so that

$$t_{\theta/nu} = 12 + 47 = 59 \tag{4.7}$$

The hardware requirements are one AU per node or stage with a total of $S_{\theta/nu}$ AUs and the throughput for the new nodes is

$$T_{\theta/nu} = \frac{1}{\left\lceil \frac{59}{S_{\theta/nu}} \right\rceil} \tag{4.8}$$

This expression corresponds to the throughput obtained when the operations are assigned to new nodes or stages, such that each stage has an integer number of operations. For most choices of the number of stages the partitioning is not perfect and delays have to be added to some stages so that all of them have equal latency. Also, for $S_{\theta/nu}$ large (i.e. $S_{\theta/nu} > 10$) the partition into certain number of stages does not increase the throughput, again due to the need to have an integer number of operations per node and because the resolution in the partitioning is smaller (i.e. for both $S_{\theta/nu}$=10 and $S_{\theta/nu}$=11 the throughput is 0.166 or, in other words, there are six operations in each stage). This drawback is stronger for larger number of stages.

4.1.4 Inner Product

The inner product computation, with respect to the dependencies, is a mixture of the characteristics described for the other subfunctions: multiplications are independent but additions are not. There are several approaches for this unit's architecture [Cimin81, Swart78], particularly when all the concurrency in the dependence graph is not exploited since it might be too expensive. Three schemes are presented here, which take advantage of the concurrent capabilities in the computation.

Inner Product Computation with Multiplier/Adder Units

This scheme applies pipelining to the subfunction by dividing the input data into several subsets and performs the accumulation of each subset in Multiplier / Adder (M/A) units; after this process is done, the values obtained are added together using either the same adders in the M/A units or different ones. Figure 4.2 shows a dependence graph for this scheme. Pipelining is used for the addition part while parallelism is possible by having several M/A units and combining their results as needed.

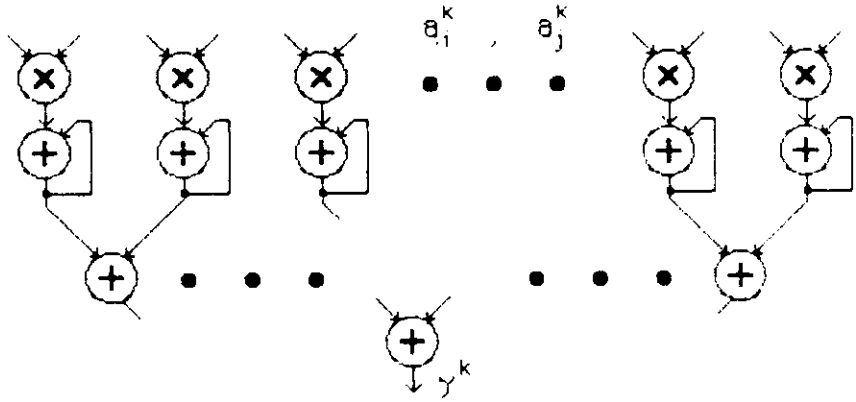

Figure 4.2 - Dependence Graph for Inner Product with M/A Units

If the same adders in the M/A units are used to reduce the partial summands after all multiplications have been performed, then no extra hardware is needed besides the M/A units themselves. In such case initially there is an internal pipeline of two stages with stage time of one operation and later only additions are performed in a tree reduction fashion, but using the same adders.

If there are $I \leq m/2$ M/A units, the execution time is:

$$t_{ip\,1} = \left\lceil \frac{m}{I} \right\rceil + \left\lceil \log_2 I \right\rceil + 1 \qquad \text{if } I \leq \frac{m}{2} \tag{4.9}$$

This expression is obtained as follows:

- $\lceil m/I \rceil + 1$ represents the time to compute m multiplications in I units plus the accumulation of those products
- $\lceil \log_2 I \rceil$ represents the time to reduce the outputs of the I units to one.

For $I > m/2$, the hardware utilization is low. More than half the columns elements are used in the first multiply time, so in a second only some multipliers would be required. Furthermore, the adders needed at later steps are less than $I/2$. Although the throughput would be higher, the efficiency would be low and that is undesirable for the purposes pursued here. Therefore, this variation of the scheme is not convenient and will not be considered further.

For the orthogonalization pipeline, this scheme is one stage with throughput

$$T_{ip\,1} = \frac{1}{\left\lceil \frac{m}{I} \right\rceil + \lceil \log_2 I \rceil} \qquad \text{if } I \leq \frac{m}{2} \tag{4.10}$$

as a new inner product can start while the last accumulation of the previous one is being computed. The hardware requirements are $2I$ AUs.

Inner Product with M/A Units and Extra Adders

If extra adders are provided to perform the reduction of summands (instead of using the ones in the M/A units after multiplying the last subset of input data), it is possible to have two stages in the orthogonalization pipeline: the first one performs the accumulation of partial results in the M/A units, while the second one reduces those values to one. The computation time for these stages depends on the number of extra adders, the number of M/A units and the number of rows in the matrix. The adequate choice is to have $\lceil I/2 \rceil$ extra adders such that the structure of this step is as shown in Figure 4.3, where the same extra adders are used for the successive reduction steps. This approach is equivalent to modifying the node by dividing it into two different ones. These two nodes become stages of the orthogonalization pipeline.

As in the previous case, this scheme is convenient only if $I \leq m/2$. Additionally, a necessary condition for efficiency is $\lceil m/I \rceil + 1 \geq \lceil \log_2 I \rceil$, otherwise there would be idle times for the M/A units. With this approach, the time to compute the inner product is given by

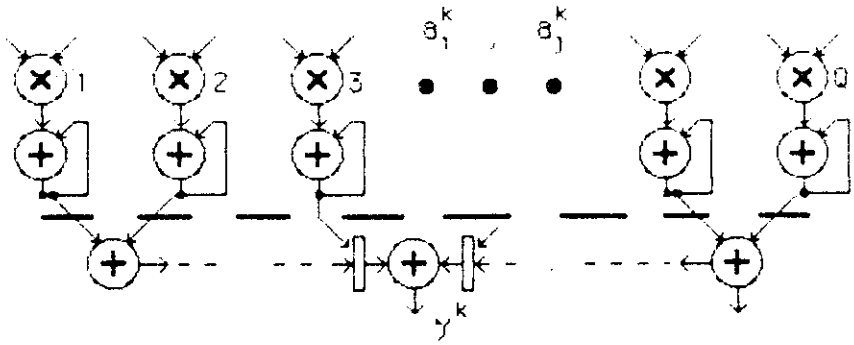

Figure 4.3 - Inner Product with M/A Units and Extra Adders

$$t_{ip\,2} = \left\lceil \frac{m}{I} \right\rceil + \left\lceil \log_2 I \right\rceil + 1 \tag{4.11}$$

which is the same as $t_{ip\,1}$. However, this scheme allows two stages in the orthogonalization pipeline, with $2I$ M/A units in the first stage and $\lceil I/2 \rceil$ adders in the second (a total of $\lceil 5I/2 \rceil$ AUs), providing a throughput for this subcomputation of

$$T_{ip\,2} = \frac{1}{\left\lceil \frac{m}{I} \right\rceil + 1} \tag{4.12}$$

which is defined by the time to perform all multiplications and the corresponding accumulations in the M/A units. Note that the first stage has an internal pipeline of one operation stage-time, while the second shares the same units for the different reduction steps (i.e. it exploits the maximum parallelism of the graph for its computation).

Inner Product Computation with Tree Structure

This scheme is based on a first level of multipliers and then as many levels of adders as required to perform the reduction of the summands generated by the multipliers. This approach has the same structure as the data dependency in the inner product computation. However, implementing the entire tree might be too expensive given the number of AUs required (m multipliers and $m-1$ adders). Alternatives with less parallelism are considered where the number of multipliers is less than m, the computation process is divided into groups which are processed sequentially and a final accumulator is included, as shown in Figure 4.4. Because of its structure, this scheme leads itself to an internal pipeline for this subcomputation.

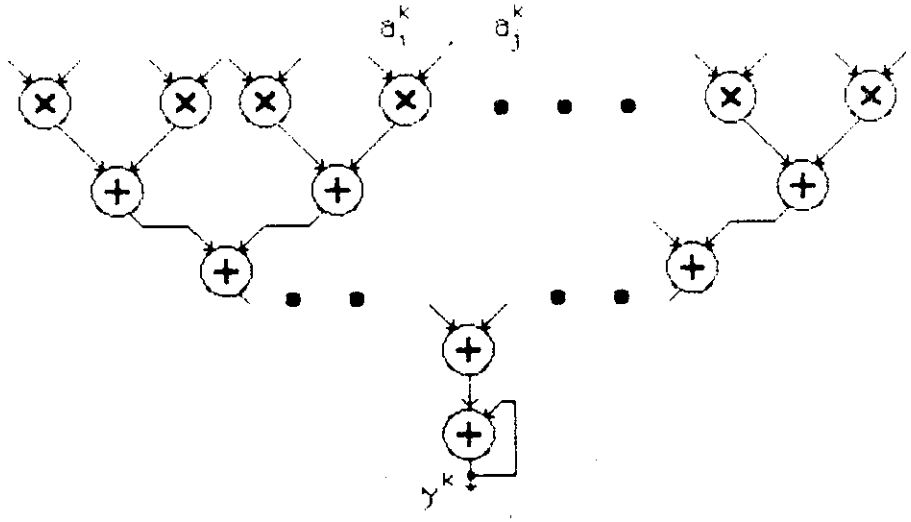

Figure 4.4 - Inner Product with Tree Structure

Assuming there are F multipliers, the height of the tree L including the multipliers is

$$L = 2 + \left\lceil \log_2 F \right\rceil \tag{4.13}$$

This expression is obtained from one level for the multipliers, $\left\lceil \log_2 F \right\rceil$ for the tree reduction of F summands, and one last level to accumulate the partial results. Each level in the tree corresponds to one operation.

The time to compute the inner product is

$$t_{ip\,3} = \left\lceil \frac{m}{F} \right\rceil + (L - 1) \qquad L \geq 2 \tag{4.14}$$

where $\left\lceil \frac{m}{F} \right\rceil$ represents the time for all the data elements to traverse the multipliers and $(L-1)$ is the time required to traverse the adders while accumulating the final result.

The total hardware required is $2F$ units: F multipliers, $F-1$ adders to reduce the multipliers output to one, and an extra adder to accumulate previous partial results if $F < m$. There are at least two units (one multiplier and one adder) and the tree height is at least 2.

This scheme also appears as one stage in the orthogonalization pipeline; its throughput is

$$T_{ip3} = \frac{1}{\left\lceil \frac{m}{F} \right\rceil} \tag{4.15}$$

as there is no need to wait for the last set of data in one inner product to traverse the entire tree before a new one can start. The tree is actually an internal pipeline of L stages and stage time of one operation.

4.2 Comparison of Throughputs for the Orthogonalization Subfunctions

The previous expressions for the different subfunctions in the orthogonalization process are summarized in Table 4.1, in terms of throughput and hardware resources required. They are also shown in Figures 4.5 and 4.6 for systems with m=20 and m=40 respectively, with different number of stages for angle computation and norms update subfunctions combined (θ/nu). Although these graphs are shown as continuous lines, each one of them represents discrete sets of points such that only their trend and the actual points are meaningful.

An important characteristic of the throughput in the orthogonalization subcomputations is readily visible from the figures: their values are quite different for similar hardware resources. Also, any attempt to match them requires a fairly large amount of hardware for the rotations subfunction. Some modification to the architecture described so far has to be devised to improve those characteristics, if at all possible, as discussed later.

Another issue worth noticing is that, of the three different schemes presented for inner product computation, the tree structure provides the best throughput for the same amount of hardware. This result is not surprising, because a tree naturally pipelines the different operations, without introducing delays or idle times. (1) Later analysis will only use this alternative for the inner product node.

With the consideration above regarding the structure for the inner product unit, the architecture devised for the orthogonalization process is a pipeline with $S_{\theta/nu}+2$ stages. Its characteristics in terms of throughput and hardware requirements are discussed in the next section.

(1) At the time of this research, a paper written by Smith and Tong [Smith85] was published which also states that the tree scheme is the best alternative for a fast inner product unit.

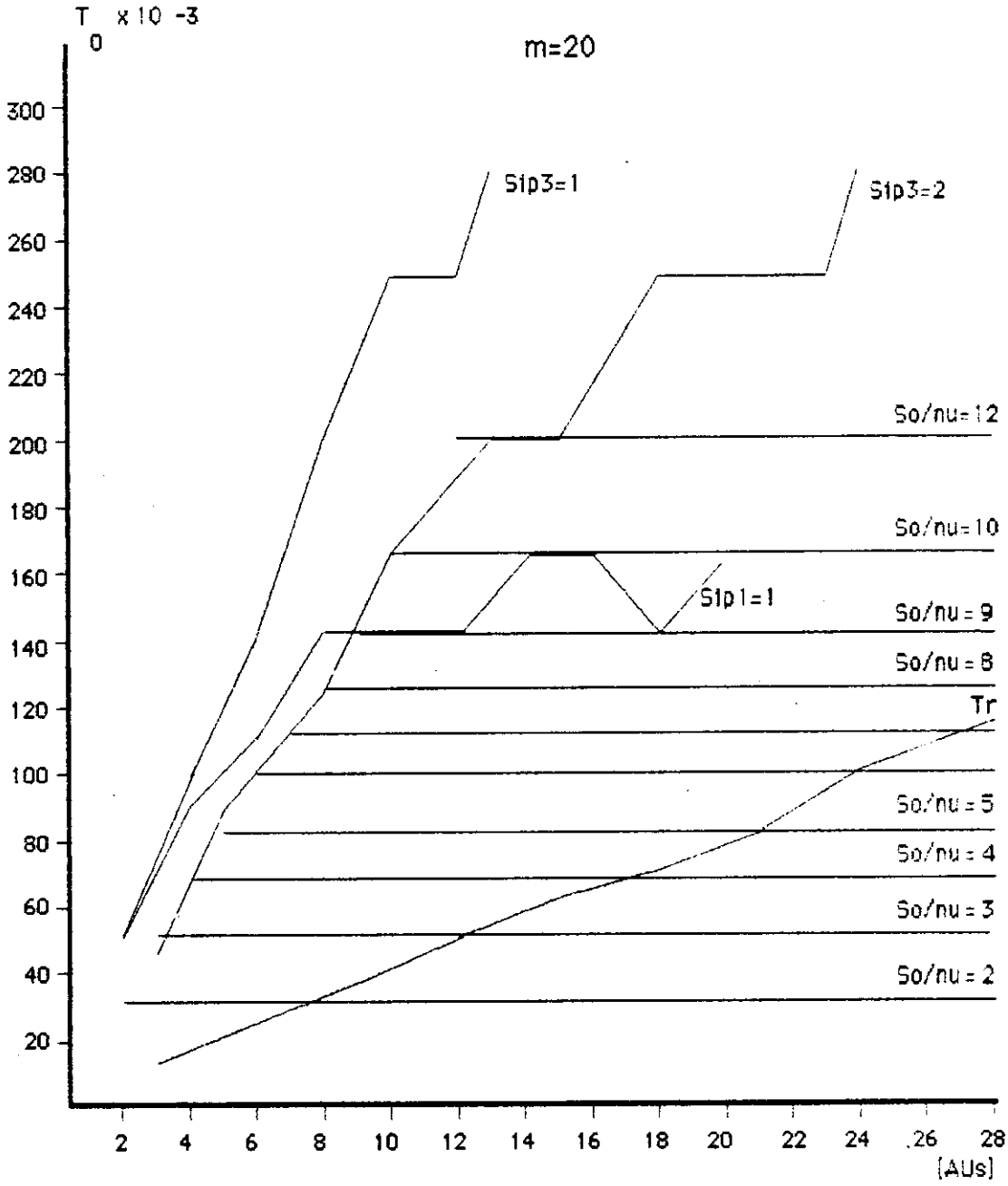

Figure 4.5 - Orthogonalization Steps Throughputs for m=20

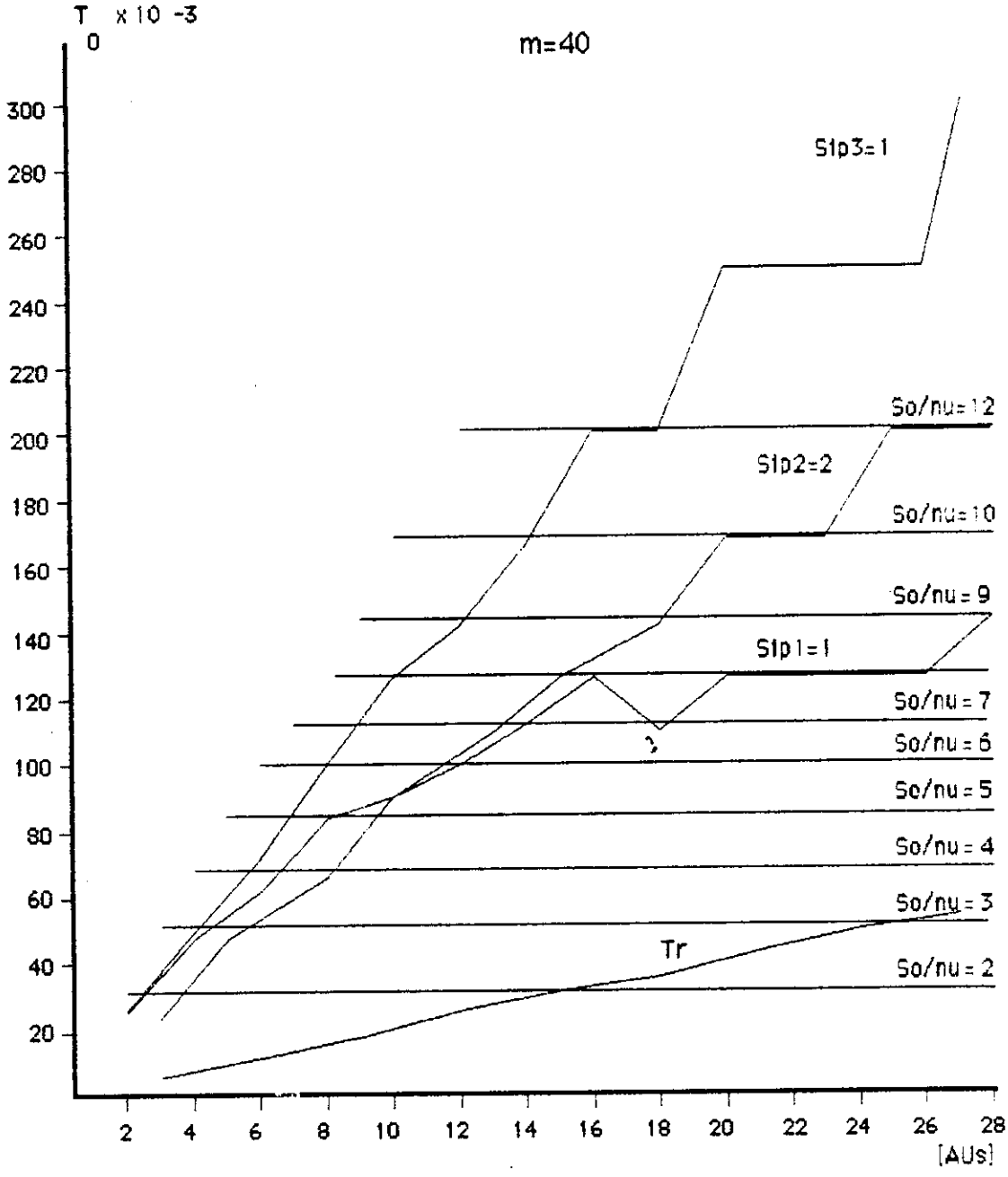

Figure 4.6 - Orthogonalization Steps Throughputs for m=40

Subfunction	Throughput $[op^{-1}]$	Stages	Hardware Required [in AUs]	
Rotations	$T_r = \frac{1}{\lceil 4m/G \rceil}$	1	$H_r = 3G$	
θ/nu	$T_{\theta/nu} = \frac{1}{\left\lceil \frac{59}{S_{\theta/nu}} \right\rceil}$	$S_{\theta/nu}$	$H_{\theta/nu} = S_{\theta/nu}$	One AU per stage
Inner Product	$T_{ip1} = \frac{1}{\lceil m/I \rceil + \lceil \log_2 I \rceil}$	1	$H_{ip1} = 2I$	I M/A units only
	$T_{ip2} = \frac{1}{\lceil m/I \rceil + 1}$	1	$H_{ip2} = \left\lceil \frac{5}{2} I \right\rceil$	I M/A units plus $\lceil I/2 \rceil$ adders
	$T_{ip3} = \frac{1}{\lceil m/F \rceil}$	1	$H_{ip3} = 2F$	Tree structure, $L \geq 2$; F multipliers and F adders

Table 4.1 - Throughput for the Orthogonalization Subfunctions

4.3 Design Procedure for the Orthogonalization Hardware

The sets of curves obtained in the last section can be used as follows for the design of a SVD processor with the largest efficiency for a given speed-up:

i. for a given number of columns in the matrix, an adequate combination for P and S is chosen according to equations (3.5) or Figure 3.20

ii. for the chosen number of stages S, the plot for $S_{\theta/nu} = S - 2$ is selected in Figures 4.5 or 4.6, which gives the throughput for the orthogonalization with that number of stages. This number of stages also corresponds to the number of arithmetic units for the θ/nu subfunction

iii. the hardware requirements and throughput characteristics for the other subfunctions are obtained by finding the intersection between the chosen θ/nu plot with the other subfunctions curves, and moving from there to valid points on those curves.

For example, if m=20 and the selected value for S is 6, then from Figure 4.5 the curve for $S_{\theta/nu} = 4$ gives a throughput of 0.067 $[op^{-1}]$; proper points for the other stages are

Subfunction	Hardware	Throughput	
Inner Product:	2 AUs	$T_{ip} = 0.050$	$[op^{-1}]$
	4 AUs	$T_{ip} = 0.100$	$[op^{-1}]$
Rotations:	18 AUs	$T_r = 0.072$	$[op^{-1}]$

The final selection should be done using these values and the actual throughput for the entire orthogonalization is the minimum for all stages. For instance, choosing four AUs for the inner product node gives a final throughput defined by the θ/nu step, i.e. 0.067 $[op^{-1}]$. With these values, recalling from equation (3.2) that the orthogonalization algorithm is executed $5n(n-1)$ times (instances) and assuming n=20, then the performance measures are

M =	1900 instances
t_{seq} =	$M\ t_O = M\ (14m + 58) = 642{,}200\ [ops]$
t_{pipe} =	$(S + M - 1)\, t_S = 1905\, x\ 15 = 28575\ [ops]$
N_{pipe} =	24 $[AUs]$
$SU_{pipe}(S, N)$ =	$SU_{pipe}(6{,}24) = 22.5$
$E_{pipe}(S, N)$ =	$SU_{pipe}(6{,}24) = 0.94$

where t_{seq} is the computation time in a completely sequential implementation and t_{pipe} is the time in the pipelined system.

Alternatively, some reduction in the number of AUs is possible by choosing only two AUs in the inner product for a final throughput of 0.050 $[op^{-1}]$, although such selection has a lower efficiency as a result of the extra throughput available in the other subfunctions but not used.

This approach can be used to provide the highest throughput for the orthogonalization computation and correspondingly for the entire SVD by selecting the largest number of stages possible, but this might be too expensive for a given application. It is also possible to obtain lower cost alternatives from the graphs. In such cases, one can assign a certain amount of hardware for the rotations subcomputation (which is the one that requires the largest amount) and determine from the graphs intersecting points with the other subfunctions, using the same procedure as above. Alternatively, a desired throughput for the computation might be used as the starting point and from it obtain the hardware requirements for the different subcomputations, again using the graphs as before.

While these represent valid design alternatives for a SVD processor, the cost issue necessarily has to be considered. The obvious question is whether these or similar throughput characteristics can be achieved, but at a lower cost. The next section looks into that issue, again by exploiting concurrency in the computation. Since the results of that section modify substantially the data obtained here regarding throughput and hardware requirements for an implementation, the SVD throughput as a function of the number of AUs for the cases described above is not presented.

4.4 Pipelined Arithmetic Units in the Orthogonalization Process

The most critical of the orthogonalization subfunctions are the θ/nu and the rotations computations since they have the lower throughput characteristics, particularly the last one. As described above, the θ/nu node throughput is improved by decomposing it into several new nodes which become stages of the orthogonalization pipeline.

The same approach applied to increase the throughput in the θ/nu node is not convenient for the rotations subfunction, because its characteristics are not adequate for such an attempt. The reasons for this are

i. it only has two sequential steps (multiplications and then additions), so that as a maximum there would be two nodes

ii. the amount of data processed in this subfunction is high ($4m$ elements). If the subfunction is partitioned into new nodes, these new nodes operate on different instances of the computation (i.e. rotation of different column pairs) and produce many results which would have to be passed from one stage to the next. Therefore, many staging registers would be required to store the intermediate results for the operations under computation.

In searching for solutions to the limitations of the rotations subfunction, one has to look at the characteristic of this subcomputation and the implementation for it described so far, where replication has been used by duplicating the M/M/A units (which have some degree of internal pipelining and parallelism). Although this approach is using both pipelining and parallelism, pipelining has been restricted by the use of non-pipelined arithmetic units and therefore limited to only two stages in total. This is not necessarily a requirement, because such units can also be pipelined.

Pipelining floating point multiplications and additions is possible and has been extensively studied, as discussed later. In such case, the total hardware in a pipelined unit is the same as for a non-pipelined one except for the addition of staging registers. Therefore, the efficiency obtained when using pipelined AUs is larger than what is obtained using parallelism, because the amount of extra hardware introduced for the same speed-up is smaller (i.e. doubling the throughput of one M/M/A unit requires 3 additional AUs when replication is used, while pipelining each of the elements in the M/M/A unit with two stages produces the same speed-up at the cost of the staging registers per AU only). This situation was briefly described in section 3.1 and now it is discussed in detail.

Pipelining the AUs is advantageous as a result of the type of computation performed at the rotations step, i.e. independent successive operations. This approach can also be applied to the inner product computation, although some special considerations are in order there due to the feedback in the accumulation process.

Pipelined arithmetic units seems to be a rather natural solution for the desire to improve the throughput of the orthogonalization steps, with less hardware. This is merely an extension of the architecture described so far, because the best design for the inner product is already a pipelined tree, while the rotation step has been implemented with pipelined M/M/A units. The suggestion now is to enlarge this feature by allowing pipelined operators, therefore increasing the number of internal stages in each subfunction. This corresponds to a new iteration in the design methodology, this time applied to the individual operators.

The following analysis studies how the throughput is affected when the AUs at each subfunction are pipelined. New expressions are obtained in terms of the operation time of regular (non-pipelined) AUs, as functions of the number of pipelined AUs and the number of stages in them.

4.4.1 Rotation Subfunction

For the analysis here it will be assumed that the number of stages in both multipliers and adders is the same, though this is not necessarily a restriction (the actual parameter of interest is the total number of stages in the M/M/A units).

Given the structure of this subfunction, its throughput is incremented proportionally to the number of stages in the AUs which is restricted only by their implementation properties. Letting s_r represent the number of stages in the AUs used in the rotation process, the throughput becomes

$$T^*_r = \frac{s_r}{\left\lceil \frac{4m}{G} \right\rceil} \tag{4.16}$$

and the computation time is

$$t^*_r = \frac{\left\lceil \frac{4m}{G} \right\rceil}{s_r} + (2\,s_r - 1) \tag{4.17}$$

The hardware requirements are $3G$ pipelined AUs.

4.4.2 Inner Product with Tree Structure

This subfunction involves a tree of adders to reduce the multipliers outputs to one, including the accumulation of successive partial results. If the arithmetic operators are pipelined, the accumulation becomes a problem because of the feedback path. Kogge [Kogge81] presents a solution for this situation, whose scheme is shown in Figure 4.7. This structure allows the use of pipelined units of s stages (assuming s is a power of 2), by decomposing the accumulation process in the addition of groups of s elements performed two at a time. In such approach, $\log_2 s$ cascaded pipelined adders compute the addition of each group independently and then partial results are accumulated in one additional adder, also of s stages. If s is not a power of 2, the structure is the same shown but the feedback path in the last adder must include a delay block of $(2^q - s)$ where $q = \left\lceil \log_2 s \right\rceil$.

Therefore, the one-level accumulation in the non-pipelined scheme is converted into $\left(\left\lceil \log_2 s_{ip} \right\rceil + 1 \right)$ levels of pipelined adders, where s_{ip} is the number of stages in the AUs in the inner product step (it will be assumed again that this number is the same for both adders and multipliers).

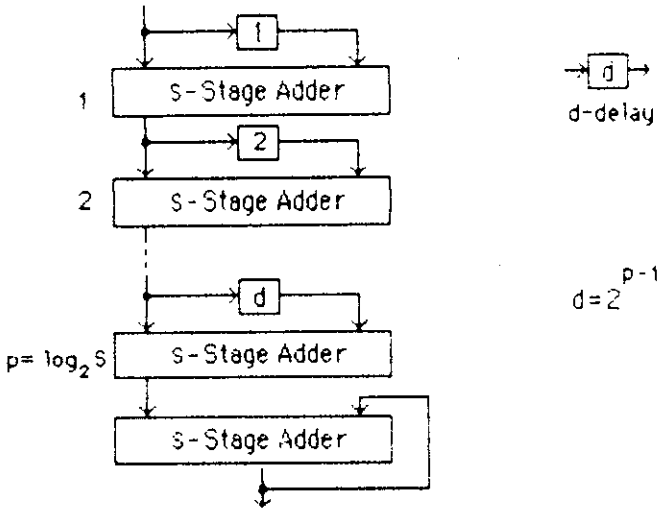

Figure 4 7 - Pipelined Accumulator

However, the total number of summands must be a multiple of $2^{\lceil \log_2 s_{ip} \rceil}$ because the computation is performed in groups of this size. A matrix whose number of rows does not satisfy this condition is either extended by introducing rows of zeroes, or those zeroes are generated by a controller in the inner product unit itself. This restriction may be stated as

$$m_{ip} = b\ 2^{\lceil \log_2 s_{ip} \rceil} \quad , \quad b \quad integer \tag{4.18}$$

where m_{ip} is the number of rows for the inner product computation.

In terms of number of internal AU stages, the new inner product tree height is

$$L^* = \left(\left\lceil \log_2 F \right\rceil + 1 \right) s_{ip} + \left(1 + \left\lceil \log_2 s_{ip} \right\rceil \right) s_{ip} \tag{4.19}$$

where the first term represents one level of pipelined multipliers and $\lceil \log_2 F \rceil$ levels of pipelined adders to reduce the summands to one, while the last term corresponds to the accumulation process. This expression may be rewritten as

$$L^* = \left(\left\lceil \log_2 F \right\rceil + \left\lceil \log_2 s_{ip} \right\rceil + 2 \right) s_{ip} \tag{4.20}$$

The time to compute an inner product, in terms of the operation time of non-pipelined AUs, now is

$$t^*_{ip} = \frac{\left\lceil \frac{m_{ip}}{F} \right\rceil + (L^* - 1)}{s_{ip}} \tag{4.21}$$

and the new throughput is

$$T^*_{ip} = \frac{s_{ip}}{\left\lceil \frac{m_{ip}}{F} \right\rceil} \tag{4.22}$$

The total hardware for this subfunction now is $2F + \left\lceil \log_2 s_{ip} \right\rceil$ pipelined AUs.

Consequently, in both the rotations and the inner product subfunctions the speed up is proportional to the number of stages in the AUs. The penalty in extra hardware requirements is internal to the AUs and consists only of the registers required between the stages. Considerations about the maximum number of stages and clock frequency for these units will be discussed later.

4.4.3 Rotation Angle and Norms Update

As seen in Figure 4.1b, the angle computation includes two divisions and two square-root operations which are being implemented through a multiplicative approach. It will be shown later that this multiplicative scheme has two parallel multiplications in its dependence graph which can also be pipelined. One pipelined AU allows to compute division in 6 multiply times and square-root in 9, as will also be shown later; two pipeline stages are enough for these two operations. Then, adding the times for the operations in the graph in Figure 4.1b, the angle computation is performed in the equivalent of 35 multiply times.

The norms update subfunction can also take advantage of pipelined AUs and reduce its computation time by allowing the parallelism existing in it to be computed through pipelining. One device with three stages allows to compute this subfunction in the equivalent to 5 multiply times, as it can be inferred from the corresponding dependence graph.

Therefore, the θ and *nu* subfunctions combined are computed in the equivalent of 40 multiplication times, with only one pipelined AU with 3 stages (note that only two stages are actually needed for the angle part). In other words

$$t^*_{\theta/nu} = 40 \quad , \quad 1\,AU, 3\,internal\ stages \tag{4.23}$$

This subfunction can also become several stages in the orthogonalization pipeline, same as in the case with non-pipelined AUs described before. Each stage performs part of the computation with its own arithmetic unit. Letting $S_{\theta/nu}$ represent the number of stages, the corresponding throughput is given by

$$T^*_{\theta/nu} = \frac{1}{\left\lceil \frac{40}{S_{\theta/nu}} \right\rceil} \tag{4.24}$$

and the hardware requirements are $S_{\theta/nu}$ AUs.

4.5 Comparison of Throughputs with Pipelined Arithmetic Units

The new expressions obtained above are summarized in Table 4.2. The corresponding throughput graphs are shown in Figure 4.8 and 4.9, for m=20 and m=40 as in the case for non-pipelined AUs.

Step	Throughput $[op^{-1}]$	Hardware Required [in AUs]
Rotations	$T^*_r = \frac{s_r}{\lceil 4m/G \rceil}$	$H_r = 3G$
Angle/Norms Update	$T^*_{\theta/nu} = \frac{1}{\left\lceil \frac{40}{S_{\theta/nu}} \right\rceil}$	$H_{\theta/nu} = S_{\theta/nu}$
Inner Product	$T^*_{ip} = \frac{s_{ip}}{\lceil m_{ip}/F \rceil}$	$H_{ip} = 2F + \lceil \log_2 s_{ip} \rceil$ $m_{ip} = b\ 2^{\lceil \log_2 s_{ip} \rceil} \geq m$

Table 4.2 - Throughput of the Orthogonalization Subfunctions with Pipelined AUs

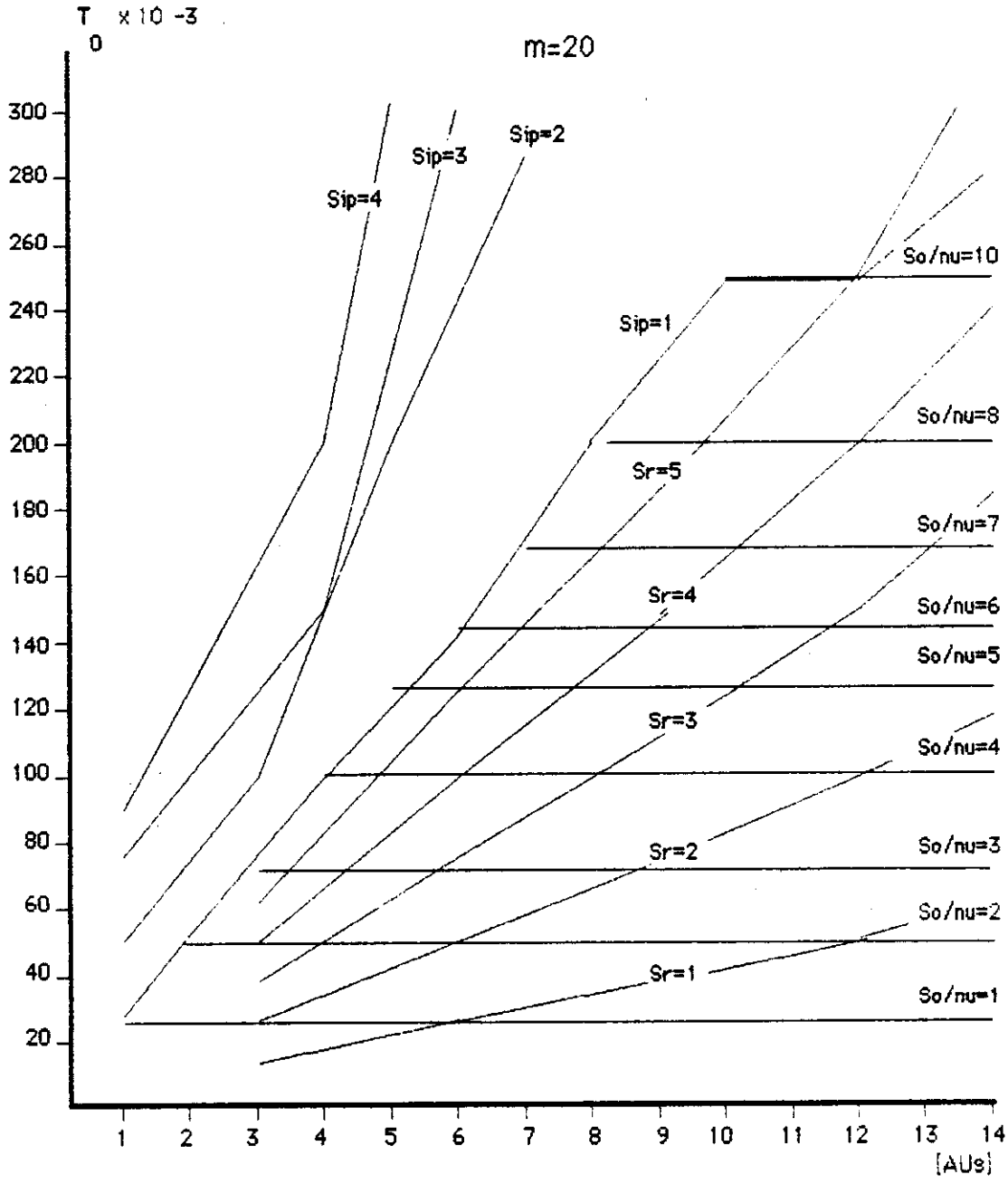

Figure 4.8 - Orthogonalization Subfunctions Throughputs with Pipelined AUs for m=20

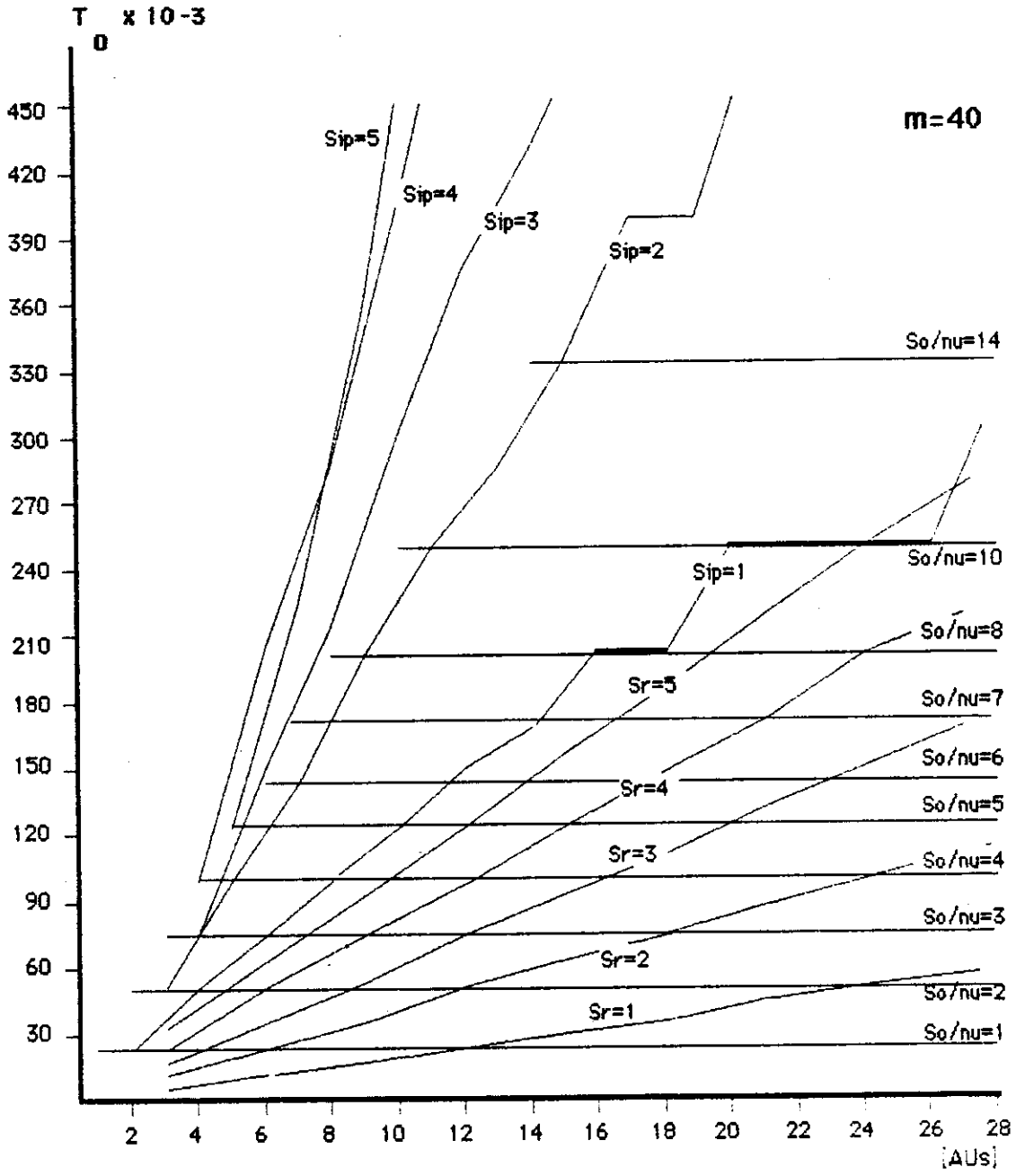

Figure 4.9 - Orthogonalization Subfunctions Throughputs with Pipelined AUS for m=40

Some features of the plots are:

- the values for s_r and s_{ip} range from 1 to 5; for these, both values of m satisfy equation (4.18), except m=20, s_{ip}=5. The maximum number of stages per AU chosen is discussed later and proper reasons for it are given

- the entries for one AU in the inner product plots were obtained independently, because the tree assumes a minimum of two levels and two AUs

- as stated before, some values of $S_{\theta/nu}$ are not advantageous because they do not increase the throughput (i.e. $S_{\theta/nu}$=9 with respect to $S_{\theta/nu}$=8).

For a given number of AUs, now there are more intersecting points between the curves for the different subfunctions than with non-pipelined AUs, which implies more possible alternatives for the design of the SVD processor. Furthermore, similar throughputs to the cases with non-pipelined AUs are achieved with less hardware, as might have been expected. These intersections include high cost options, those with higher throughput, and others less expensive.

As an example of the gains achieved with pipelined AUs, from Figure 4.8 we infer that it is possible to obtain a similar throughput to the case presented with non-pipelined units (i.e. $T_O = 0.063[ops^{-1}]$) but with only seven units instead of 24 ($S_{\theta/nu} = H_{\theta/nu} = 3$; $H_r = 3$, $s_r = 5$; $H_{ip} = 1$, $s_{ip} = 3$). Therefore, the same speedup in the computation achieved there (i.e. $SU = 22.5$) now implies an efficiency $E \simeq 3.2$. Indeed, this is an attractive gain. The upper bound for the speedup improvement is the number of stages in the AUs.

In the next chapter, we study the throughput of the whole computation for alternatives with different cost. In doing so, we provide insight towards the selection of specific implementation alternatives.

Summary

In this chapter, we have studied the characteristics of the orthogonalization subfunctions in terms of throughput and arithmetic hardware requirements. It has been shown that significant gains are attainable with an architecture for the orthogonalization algorithm which exploits pipelining up to the maximum possible, using this technique at several levels. When pipelining reaches its limits (imposed either by implementation issues or by the algorithm) then parallelism of the graph is used, creating a system which combines pipelining and parallelism. Therefore, the architecture devised for the orthogonalization computation is a multilevel pipelined system, with

parallelism at certain stages.

The resulting architecture is not surprising if one considers the issues discussed in section 3.1, which stated that using pipelining and graph parallelism might be effective (as is the case here) in increasing the efficiency of a pipeline since it helps to get stages of equal delay. Actual efficiency characteristics for particular implementations will be discussed later.

A methodology has been presented to design the multilevel pipeline, which uses the data for each subfunction obtained in this chapter. Such methodology allows to obtain the highest throughput system (which is also the most expensive one), but also allows to design lower cost alternatives with lower throughput.

The next step in the design of a SVD processor consists to use the information obtained here to actually implement it, which requires to consider also other issues such as memories, control hardware, etc. That is the subject of the next chapter.

CHAPTER 5
DESIGN OF A DIGITAL SYSTEM FOR THE SVD

The expressions derived in the previous chapter allowed to obtain architectures and the corresponding throughputs for the different subfunctions in the orthogonalization process. This information is used now to study alternatives for a digital system to compute the SVD. First, the procedure used here for the design of such system is described and then suitable architectures are chosen for schemes with twenty and forty columns. These systems are evaluated in terms of the performance measures defined in Chapter 3 and also compared with linear systolic array architectures. Further details on the hardware for the chosen architectures are also discussed.

5.1 Design and Evaluation of Architectures for the SVD

The design procedure used here for a system to compute the SVD starts with the information regarding the throughputs of the orthogonalization subfunctions and the architectures for these subcomputations. Such architectures are inferred from the corresponding graphs obtained in the previous chapter, with a procedure as described in section 4.3. However, instead of using that approach to obtain only one alternative, hardware requirements and throughput for the orthogonalization process implemented with different number of stages are obtained. For each number of stages considered, the orthogonalization plots are searched for configurations with the largest efficiency for each of its subfunctions.

The architectures resulting from the procedure described above are combined with equation (3.9) or equivalently with Figure 3.20 (these give the SVD throughput in terms of the orthogonalization throughput), and used to obtain the characteristics for systems with one or more parallel processors and with different number of orthogonalization stages (P/S systems). The procedure at this step consists in determining from equation (3.9) the parameters corresponding to each P/S system considered, and computing the throughput and hardware requirements for the P/S systems from those parameters and the orthogonalization hardware requirements obtained before.

The characteristics obtained for the systems studied are summarized in tables, where each row corresponds to configurations with a given number of stages and different number of parallel processors. These values are also plotted as curves for different number of parallel processors, with the number of stages as a parameter.

Through a simple comparison, the resulting graphs allow to determine the best architectures for the SVD computation in terms of highest throughput for given hardware resources.

Alternatives with distinct cost characteristics are chosen from the graphs obtained above. The resulting systems are evaluated further using the performance and cost measures defined for these purposes in Chapter 3. With this evaluation, it is possible to state conclusions regarding the most adequate alternative for the implementation of a digital system for the SVD.

5.1.1 Alternatives for 20-by-20 and 40-by-40 Matrices

Following the procedure outlined above, digital systems to compute the SVD for matrices with $m=n=20$ and $m=n=40$ are studied now. The corresponding data is shown in Table 5.1 and Table 5.2. The information for these tables is obtained from the graphs for the orthogonalization in Chapter 4 and the combination of those values with equation (3.9) as described above. Although possible to include entries for every number of stages, some of these are not considered because their efficiency characteristics are not adequate (i.e. 9 stages for the θ/nu subfunction has the same throughput as 8 stages, as shown before).

The tables give the hardware requirements and the throughput achieved in systems with up to three parallel processors. Further replication of processors is not considered, as it provides little throughput improvement with high hardware cost. Some entries in the tables are not included for the same reason (they correspond to cases where $n \ll 2PS+2$). Tables for linear systolic arrays which use the same techniques to improve throughput (i.e. pipelined AUs) are also included. (1)

These values are plotted in Figure 5.1 and Figure 5.2. The graphs have excluded some points from the tables, because they are not attractive alternatives (they do not offer significant gains with respect to other points, or equivalently little extra hardware added to them provide large throughput gains).

Figures 5.1 and 5.2 indicate that, for throughputs higher than what is achievable with replication of a completely sequential implementation for the algorithm, one pipelined processor is the most adequate alternative for a SVD processor. It provides the highest throughput for a given number of AUs, higher than what is obtained with more parallel processors or with the linear systolic array architecture. The characteristics of such systems are evaluated in detail in the next section.

(1) The orthogonalization time in the systolic array is the sum of the times for the different subfunctions, as given by equations (4.17), (4.21) and (4.23)

These results are not surprising if one considers the characteristics of the computation for the SVD algorithm as described in the algorithmic model in Chapter 3. It was shown there that for algorithms as the SVD it is more efficient a scheme combining all three approaches to exploit concurrency in a digital system, through replication of a pipelined processor which exploits the parallelism in the graph for the algorithm. The results obtained here are simply an example of those assertions.

S	s_{ip}	H_{ip}	s_r	H_r	$H_{\theta/nu}$	H_O	T_S	t_O	$T_D x\, 10^{-6}$		
									$P=1$	$P=2$	$P=3$
		$[AUs]$		$[AUs]$		$[AUs]$	$[op^{-1}]$	$[op]$	$[op^{-1}]$	$[op^{-1}]$	$[op^{-1}]$
3	1	1	2	3	1	5	0.025	120	13.2	26.3	39.5
4	2	1	4	3	2	6	0.050	80	26.4	52.6	60.7
5	3	1	3	6	3	10	0.072	70	37.6	68.3	70.4
6	5	1	4	6	4	11	0.100	60	52.7	81.0	83.2
7	3	4	5	6	5	15	0.125	56	65.7	87.6	89.6
8	3	4	4	9	6	19	0.143	56	75.2	88.4	90.2
9	4	4	5	9	7	20	0.167	54	87.7	92.4	94.0
10	4	4	4	12	8	24	0.200	50	95.6	100.2	102.0
12	5	5	5	12	10	27	0.250	48	101.3	105.2	
16	4	6	5	18	14	38	0.333	48	103.1	106.3	

s_{ip} : *stages in inner product AUs*
H_{ip} : *number of AUs in inner product*
s_r : *stages in rotations AUs*
H_r : *number of AUs in rotations unit*
S : *number of stages per processor*
$S = S_{\theta/nu} + 2$
$S_{\theta/nu}$: *stages in θ/nu subfunctions*
$H_{\theta/nu}$: *number of AUs in θ/nu subfunctions*
$H_{\theta/nu} = S_{\theta/nu}$
H_O : *number of AUs per processor*
T_S : *orthogonalization throughput*
t_O : *orthogonalization time*
T_D : *decomposition throughput*

Table 5.1.1 - Characteristics of Parallel/Pipelined System for $m=n=20$

s_{sa}	H_p	H	t_{ip}	$t_{\theta/nu}$	t_r	t_O	$T_D x 10^{-6}$
			$[op]$	$[op]$	$[op]$	$[op]$	$[op^{-1}]$
1	1	10	39	58	240	337	15.6
1	2	20	20	41	120	181	29.1
1	3	30	20	40	80	140	37.3
1	6	60	7	39	40	86	61.2
2	1	10	20	42	120	182	28.9
2	3	30	10	41	40	91	57.8
2	6	60	5	39	20	64	82.2
3	1	10	14	40	81	135	39.0
3	3	30	14	40	27	81	65.8
3	6	60	7	39	14	60	87.7
4	1	10	11	40	60	111	47.4
4	3	30	11	40	20	71	75.2
4	6	60	5	39	10	54	97.5
5	1	10	10	40	48	98	54.0
5	2	20	7	40	24	71	75.6
5	3	30	7	40	16	63	82.0
5	6	60	5	39	8	52	101.2

s_{sa} : *stages in systolic array AUs*
H_p : *number of AUs per processing element*
H : *total number of AUs*

Table 5.1.2 - Characteristics of Linear Systolic Array for $m=n=20$

S	s_{ip}	H_{ip}	s_r	H_r	$H_{\theta/nu}$	H_O	T_s	t_O	$T_D x 10^{-6}$		
									$P=1$	$P=2$	$P=3$
		$[AUs]$		$[AUs]$		$[AUs]$	$[op^{-1}]$	$[op]$	$[op^{-1}]$	$[op^{-1}]$	$[op^{-1}]$
3	2	1	4	3	1	5	0.025	120	3.2	6.4	9.6
4	4	1	4	6	2	9	0.050	80	6.4	12.8	19.2
5	3	4	4	9	3	16	0.072	70	9.1	18.3	27.4
6	4	4	4	12	4	20	0.100	60	12.8	25.6	38.5
7	5	5	5	12	5	22	0.125	56	16.1	32.1	43.4
8	3	6	5	15	6	27	0.143	56	18.4	36.8	43.9
9	4	6	5	18	7	31	0.167	54	21.3	42.6	45.7
10	4	6	5	21	8	35	0.200	50	25.6	48.8	49.6
12	5	7	5	24	10	41	0.250	48	32.1	51.2	
16	5	9	5	33	14	56	0.333	48	42.7	51.8	
22	5	11	5	48	20	79	0.500	44	55.7	57.0	
42	5	19	5	96	40	155	1.000	42	59.6		

s_{ip} : *stages in inner product AUs*
H_{ip} : *number of AUs in inner product*
s_r : *stages in rotations AUs*
H_r : *number of AUs in rotations unit*
S : *number of stages per processor*
$S = S_{\theta/nu} + 2$
$S_{\theta/nu}$: *stages in θ/nu subfunctions*
$H_{\theta/nu}$: *number of AUs in θ/nu subfunctions*
$H_{\theta/nu} = S_{\theta/nu}$
H_O : *number of AUs per processor*
T_S : *orthogonalization throughput*
t_O : *orthogonalization time*
T_D : *decomposition throughput*

Table 5.2.1 - Characteristics of Parallel/Pipelined System for $m=n=40$

s_{sa}	H_p	H	t_{ip}	$t_{\theta/nu}$	t_r	t_O	$T_D x 10^{-6}$
			$[op]$	$[op]$	$[op]$	$[op]$	$[op^{-1}]$
1	1	20	79	58	480	617	4.2
1	3	60	40	40	160	240	10.6
1	6	120	14	39	80	133	19.3
1	9	180	10	39	56	105	24.4
2	1	20	40	42	240	322	7.9
2	3	60	20	40	80	140	18.2
2	6	120	10	39	40	89	28.8
2	9	180	7	39	28	74	34.7
3	1	20	27	40	160	227	11.0
3	3	60	27	39	56	122	21.0
3	6	120	7	39	28	74	34.7
3	9	180	5	39	19	63	40.7
4	1	20	21	40	120	181	14.1
4	3	60	21	39	40	100	25.6
4	6	120	5	39	20	64	40.1
4	9	180	4	39	14	57	44.2
5	1	20	18	40	96	154	16.6
5	3	60	18	39	32	89	28.7
5	6	120	5	39	16	60	42.8
5	9	180	3	39	11	53	48.4

s_{sa} : *stages in systolic array AUs*
H_p : *number of AUs per processing element*
H : *total number of AUs*

Table 5.2.2 - Characteristics of Linear Systolic Array for $m=n=40$

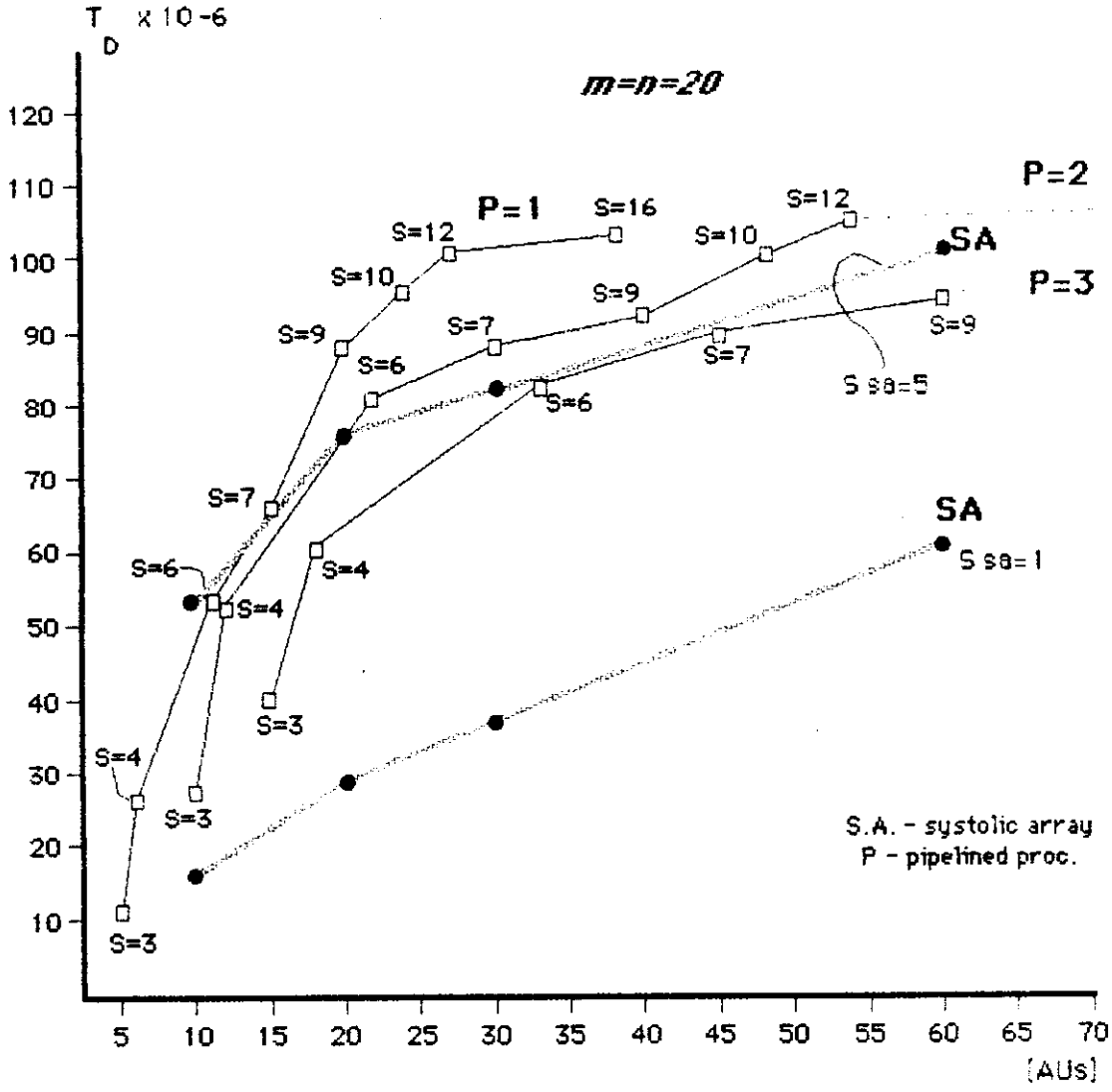

Figure 5.1 - Decomposition Throughput for m=n=20 Systems

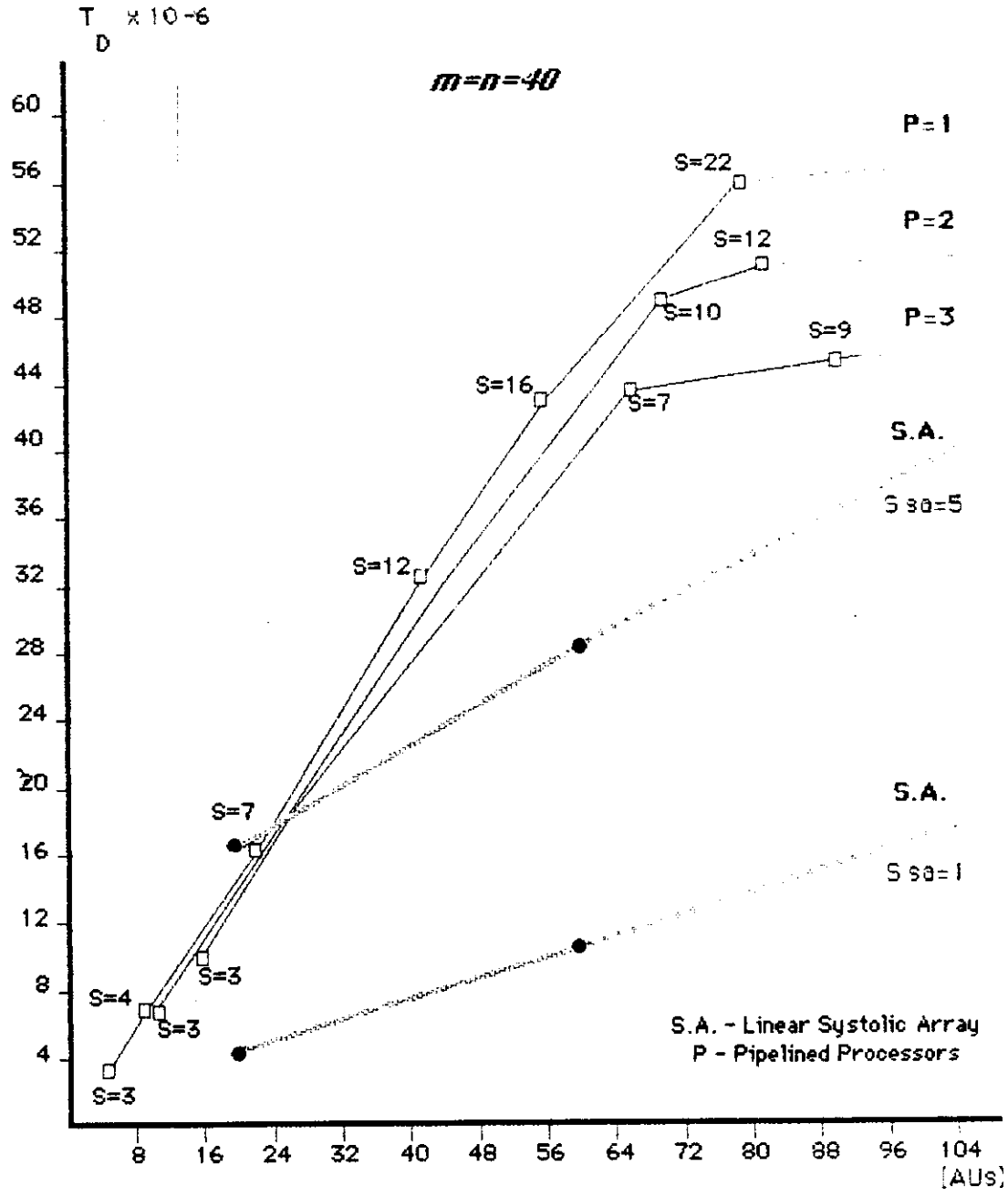

Figure 5.2 - Decomposition Throughput for m=n=40 Systems

5.1.2 Architectures for the SVD with Highest Throughput

For the cases studied here, namely matrices with twenty and forty columns, the suitable choices for designs with the highest throughput are

$m=n=20$ case: one processor, 12 stages in total
27 AUs with 5 stages each
$t_D \approx 10.000[ops]$

$m=n=40$ case: two processors, 12 stages each
82 AUs with 5 stages each
$t_D \approx 19600[ops]$

In the case for $m=n=40$, the selection above indicates two processors instead of one as the previous analysis had stated. Actually the best alternative uses one processor as expected, with 22 stages, which can be inferred from Figure 5.2. The throughput for such system is higher than what is obtained for the two-processors alternative (although not much) and it requires less hardware (three AUs less). However, it will be shown later that the two-processors scheme selected above can be obtained as an extension of the architecture chosen for the $m=n=20$ case, becoming a good example of expansibility for the system. Therefore, it has been selected for those characteristics.

The largest systems in terms of processors and stages such that the criteria imposed by equation (3.5) is satisfied (i.e. the number of columns is greater than $2PS+2$) have nine stages in the orthogonalization process. With that number of stages there are no idle times for the stages as a result of exhausting the columns at the input. But in such case the θ/nu subfunction does not allow to achieve perfect pipelining. This subfunction is composed of forty operations which can not be equally divided into certain numbers of stages, particularly seven as it would be the case. Therefore, some inefficiency would exist from this source since delays would have to be inserted in some stages.

Consequently, as the systems selected above have twelve stages, they do not satisfy the criteria imposed by equation (3.5) and columns will be exhausted at the processors input, producing idle times for the arithmetic hardware. However, this number implies ten stages for the θ/nu subfunction and now perfect pipelining is possible for the orthogonalization process, as can be inferred from equation (4.24) Therefore, in spite of the inefficiency from having $n < 2PS+2$, throughput gains obtained from pipelining into stages of equal length make this scheme attractive.

Choices beyond the points selected above offer small improvements in throughput, as the idle times due to both few columns and non-perfect pipelining become more significant.

The systems obtained above and also the linear systolic arrays with a similar amount of arithmetic hardware are evaluated now, in terms of the performance measures defined previously. The reference system for these purposes is the sequential implementation with one AU whose computation time was given by equation (3.4), which is repeated here

$$t_{seq}(m \times n) = 70\, n\, (n-1)\, (m+4) \qquad [ops] \tag{5.1}$$

Therefore

$$\begin{aligned} t_{seq}(40 \times 40) &= 4{,}804{,}800 \quad [ops] \\ t_{seq}(20 \times 20) &= 638{,}400 \quad [ops] \end{aligned} \tag{5.2}$$

and

$$N_{seq}(40 \times 40) = N_{seq}(20 \times 20) = 1 \quad [AU] \tag{5.3}$$

The corresponding performance measures for the indicated systems are given in Table 5.3. In this table the entries for the linear systolic array with m=n=40 have been obtained from the graph in Figure 5.2 rather than from an actual design alternative, because such scheme with an amount of hardware similar to the P/S system is not adequate (it implies four AUs per processor and that is not convenient as the AUs can not be used efficiently in the rotations subcomputation). It is included only to allow the comparison with the pipelined processors system. The hardware utilization for this alternative has not been computed for the same reason.

			$m=n=20$		$m=n=40$	
			P/S System	Linear Array	P/S System	Linear Array
Computation Time	t_D	[ops]	10,000	12,196	19608	29412
Arithmetic Hardware	N	[AUs]	27	30	82	80
Speed-up	SU		63.84	52.35	245.04	163.36
Efficiency	E		2.36	1.745	3.00	2.04
Hardware Utilization	HU		0.77	0.58	0.80	-

Table 5.3 - Performance Measures for Highest Throughput Systems

The values for throughput and hardware requirements are obtained directly from the graphs in Figures 5.1 and 5.2. Computation time is the inverse of throughput. The remaining performance measures are obtained applying the expressions defined for them in section 3.1.

The computation of hardware utilization requires further explanation. For both P/S systems perfect pipelining has been achieved and the throughputs of all stages are identical, which implies that the hardware is fully matched and no utilization degradation arises from throughput differences among the stages. However, the condition $n \geq 2PS + 2$ is not satisfied as stated before and idle times exist. In those cases, the ratio $it = \frac{(2PS+2) - n}{2PS + 2}$ gives the number of times data is not available for computation and therefore it represents the idle time for the hardware. Then $HU = 1 - it$ and such expression has been used in Table 5.3 for the P/S systems.

The linear systolic array hardware utilization for $m=n=20$ is computed differently. We may assume that the inner product and the rotations subcomputations use the arithmetic units fully (actually, there is some utilization degradation because orthogonalizations are not pipelined and during both the start-up time and the final steps in these subcomputations the internal stages in the AUs can not be used fully). In contrast, the θ/nu subcomputation does not use all the arithmetic hardware because

this subfunction is performed in 40 operations time with only one pipelined AU. The array selected above has three AUs per processor; therefore, two of those AUs are idle while the θ/nu subcomputation is being performed. Actually, the utilization is even lower if those AUs have five stages, because most operations in the θ/nu node are performed with only one or two stages in the AUs.

Neglecting the utilization degradation as a result of not using all the stages in the AUs, the hardware utilization is obtained using the definition for it as follows

$$HU_{syst.array}(20\ x\ 20) = \frac{1}{N\ t_O}\left[H_{ip}\ t_{ip} + H_{\theta/nu}\ t_{\theta/nu} + H_r\ t_r\right]$$

$$= \frac{1}{30\ x\ 63}\left[30\ x\ 7 + 10\ x\ 40 + 30\ x\ 16\right]$$
$$= 0.58$$

where H_x is the number of AUs in subcomputation x.

Note that the efficiency values in Table 5.3 are larger than 1. This is due to the utilization of pipelined arithmetic units in the concurrent computation systems, while the reference one uses just one non-pipelined AU (the cost of pipelining the AUs has been neglected). Systems with better hardware utilization are discussed later.

Table 5.3 together with Figures 5.1 and 5.2 indicate that, for highest throughput in the SVD computation, pipelining is more effective than replication because it provides better efficiency. As stated before, these results are not surprising due to the characteristics of the orthogonalization process: the last subfunction in it demands many arithmetic resources while the first one needs a small amount and the middle subfunctions require only one AU. A parallel processor doubles the throughput but it also doubles the hardware required. If the replicated processors have more than one AU each, they have some inherent inefficiency because the θ/nu subfunction cannot use the extra arithmetic units. This inefficiency is carried to the replicated processors.

On the other hand, the hardware requirements in the pipelined case are just what is needed at each stage, such that the efficiency is always close to its maximum. Furthermore, for larger throughput in the computation, the θ/nu subfunction is modified by introducing more stages for it and the amount of hardware required to do that is small; the throughputs of the other subfunctions are increased by increasing the degree of pipelining within the AUs, which has a lower cost than extra units.

If all steps in the orthogonalization were of the same complexity in time and resources required, then parallelism or pipelining would be similar. But in the SVD case, it is possible to take advantage of the different requirements at each subcomputation by pipelining them as much as possible and only then adding the parallelism needed. This is exactly the same reasoning behind the conclusions stated in Chapter 3 regarding the efficiency of parallel, pipelined or combined approaches to the design of a digital system.

Once the limiting conditions regarding number of orthogonalization stages is reached and the stages have also been extended to their limit (i.e. AUs can not be pipelined further because of implementation restrictions such as clock frequency or area requirements in VLSI, and internal parallelism of the nodes has reached the algorithm or implementation limits), only then it is convenient to go to parallel processors to increase the throughput further.

Note that in the linear systolic array scheme it is possible to obtain a maximum throughput higher than what is achievable in the pipelined systems. However, such maximum requires a very large amount of arithmetic hardware, as the plots above indicate.

5.1.3 Architectures for Lower Cost Alternatives

For some applications the amount of hardware resources involved in the designs in the previous section might be too high. It is possible to have schemes with less hardware and lower throughput if proper points are chosen in Figures 5.1 or 5.2.

Two examples of lower cost alternatives for each matrix size considered in the previous section are evaluated now: one defined by the amount of arithmetic hardware required in the smallest linear systolic array (i.e. with one AU per processing element PE) and one intermediate solution between this minimum and the highest throughput case studied above. The corresponding data for these examples is obtained in the same way as for the highest throughput case, namely by reading the hardware requirements and throughput from Figure 5.1 or 5.2 and the other performance measures are obtained by applying the expressions for them. The results are shown in Table 5.4.

			$m=n=20$		$m=n=40$	
			P/S System	Linear Array	P/S System	Linear Array
Characteristics			$P=1, S=6$	1 AU per PE	$P=1, S=7$	1 AU per PE
Computation Time	t_D	[*ops*]	18976	18519	62112	60241
Arithmetic Hardware	N	[AUs]	11	10	22	20
Speedup	SU		33.64	34.47	77.36	79.76
Efficiency	E		3.06	3.45	3.52	4.00
Hardware Utilization	HU		1	1	1	1

Table 5.4 - Performance Measures for Lowest-Cost Alternative

As seen from Table 5.4, for the lowest cost linear systolic array (i.e. only one AU per processor) a difference arises with respect to the conclusions stated above. In such case the array does not have the drawbacks in efficiency described above for parallel systems, because it has only one AU which is used all the time. Furthermore, in this architecture data dependencies do not exist and parallelism is exploited with less restrictions. As a result, the linear array achieves larger throughput than the pipelined processor, when the amount of arithmetic hardware is small. But its AUs should be pipelined. This last fact has not been discussed in the literature up to now, although it is quite obvious.

These results were predicted in Chapter 3, where it was stated that up to a certain speedup the best approach was replication of a completely sequential implementation of an algorithm, with the limit imposed by the algorithm itself. This is exactly the case here, because the linear systolic array consists of replicating a simple processor $n/2$ times, which is the limit defined by the group size in the dependences among instances of the computation.

However, for systems with higher cost (i.e. more than one AU per processor) and larger throughput than the minimum one, the efficiency and hardware utilization of the linear systolic array are lower than the P/S system, implying that this last architecture is more adequate. This fact is illustrated in Table 5.5. Actually, the linear systolic array advantages exist only for the minimum cost system.

			$m=n=20$		$m=n=40$	
			P/S System	Linear Array	P/S System	Linear Array
Characteristics			$P=1, S=9$	2 AUs per PE	$P=1, S=16$	3 AUs per PE
Computation Time	t_D	[*ops*]	11403	13228	23420	34844
Arithmetic Hardware	N	[AUs]	20	20	56	60
Speedup	SU		56	48.26	205.16	137.89
Efficiency	E		2.8	2.41	3.66	2.3
Hardware Utilization	HU		1	0.72	0.97	0.71

Table 5.5 - Performance Measures for Medium Cost Alternatives

5.1.4 Effect of Matrix Dimension on Throughput Characteristics

Another important advantage of the pipelined scheme relates to its capability to solve problems of a different size than the one for which it was designed, without hardware modifications. Actually, the pipelined scheme can solve problems of any size (assuming the memory is large enough to hold the data). Larger matrix dimensions merely represent more data inputs to the system, although the throughput will decrease correspondingly.

In contrast, a linear systolic array is only able to compute decompositions for matrices with a fixed number of columns, unless hardware changes are introduced (i.e. adding more processing elements). Only variations in the number of rows in the matrix are possible, with throughput decrease evidently (assuming again that the memory to hold the data is large enough).

Figure 5.3 shows the throughput variations when the matrix dimensions are modified, for both P/S systems selected above and also for their systolic arrays counterparts. This figure shows that the decrease in throughput when the number of rows in the matrix changes is steeper for the pipelined processor system than for the fully parallel one. This result could be expected, because of the characteristics of the orthogonalization algorithm. In the pipelined case, when m increases the inner product and rotations subcomputations take longer, decreasing the total throughput of the system. While this extra time is needed by those subfunctions, the θ/nu stages do not require it and are temporarily idle, decreasing the total efficiency of the system. On the other hand, the systolic array only extends inner product and rotations computations times as needed.

The systolic array behavior is also less dependent than the pipelined scheme when the number of rows in the matrix is smaller than the number for which the system was designed, because in such cases the orthogonalization time decreases proportionally (again, inner product and rotations computations take only the time needed). The pipelined processor can not take advantage of such situation, because now the throughput is limited by the θ/nu subfunction and the others are idle part of the time. In this sense, the systolic array is less sensible to changes in the number of rows in the matrix.

However, the pipelined design accepts any variation in the value of n, the number of columns in the matrix; furthermore, fewer columns produce larger throughputs because there are less orthogonalizations to perform. In contrast, the systolic array can only solve problems of the size for which it was designed or with a smaller number of columns (with the same throughput). Actually, the systolic array scheme can solve problems whose size is a multiple of the size for which it was designed, if the columns ordering and exchange process proposed in section 3.2 is used. But in such case it becomes a P/S system with $S = 1$, whose throughput characteristics are lower than other alternatives with larger value of S as the previous analysis has concluded.

Therefore, the pipelined processor offers more flexibility to changes in the matrix dimensions than the systolic array. This flexibility is more important than the advantages outlined before regarding changes only in the number of rows. It can be expected that variations in the problem size affect both matrix dimensions similarly, rul-

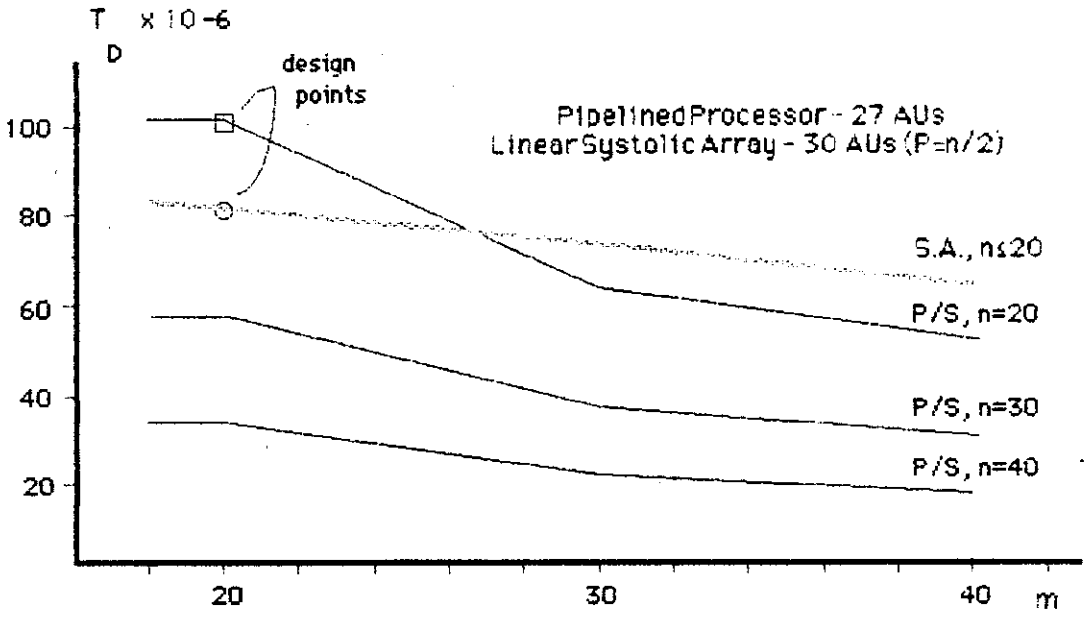

Figure 5.3a - Throughput Variation for P=1, S=12 System

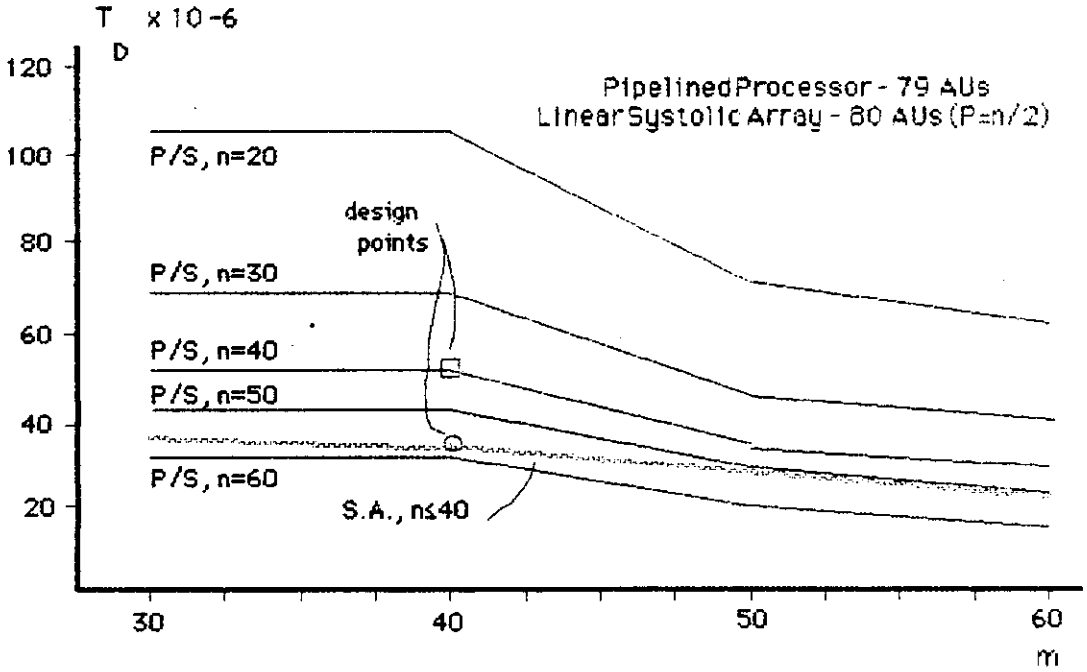

Figure 5.3b - Throughput Variation for P=2, S=22 System

ing out the usability of the systolic array for those applications.

Furthermore, the architecture of the P/S system is such that it is always possible to add another parallel processor to deal with larger matrices, with the same efficiency characteristics described above. The implementation of this expansibility will be described later.

Now that the general architecture, in terms of parallel processors and pipelined stages, to compute the SVD has been discussed, the next step is to study the actual hardware implementation. That is the subject of the remaining of this chapter.

5.2 Implementation of a Digital System for the SVD

Given the results obtained in the previous section, it is of interest now to look into some of the implementation issues in a digital system to compute the SVD whose characteristics are those just determined. As in any design, important factors at this time besides what is related to the functional characteristics of the system are, among others, modularity, expansibility, interconnection complexity, control complexity, and testability. These implementation issues are discussed now.

5.2.1 SVD System for a 20-Columns Matrix

Figure 5.4 shows the architecture for the $m=n=20$ case implemented according to the conclusions obtained before, namely one processor, twelve stages, and computation time for the SVD of 10.000 [*ops*]. This figure shows the computing hardware and the memories existing in the system. The control function is discussed later.

Components of the System

This system has the following components:

Inner Product Node: A unit with five AUs as shown in Figure 5.5, where each AU has an internal pipeline of five stages.

θ/nu Node: Pipelined unit with ten stages, where each stage has one AU internally pipelined (with three stages). Further details for this node are discussed later.

Columns Rotation Node: A unit with twelve AUs, where each AU is internally pipelined in five stages. The AUs are grouped into pairs of M/M/A units, as shown in Figure 5.6.

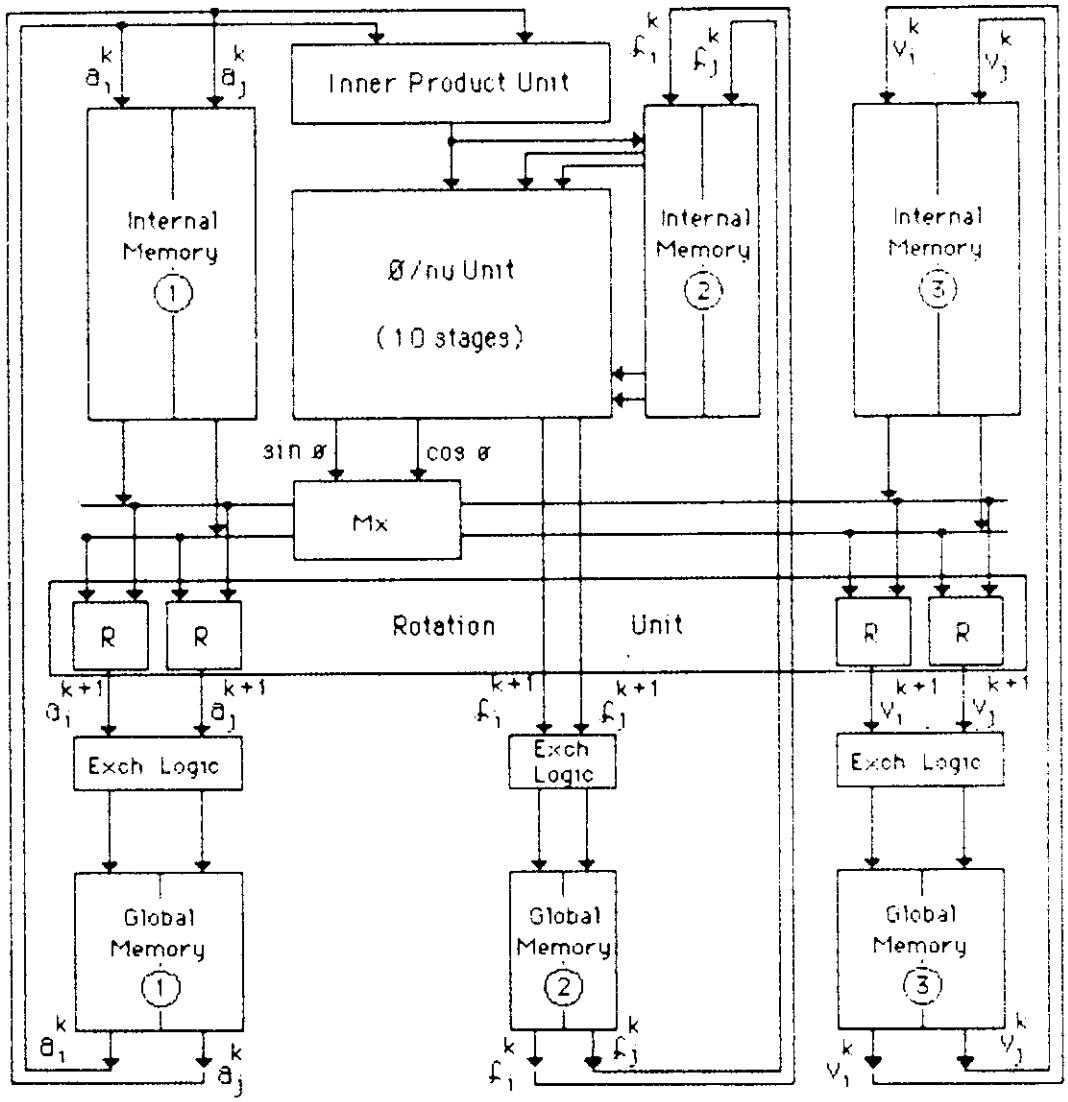

Figure 5.4 - Pipelined SVD Processor for m=n=20

Internal Memory: Memory to store the columns elements and norms while they are being used in the computation. It comprises three blocks: one to store A, one to store V (blocks 1 and 3 in Figure 5.4 respectively), and one to store the norms (block 2). Each block must hold the corresponding data for as many orthogonalizations as there are stages in the system (i.e. twelve in this case). Blocks 1 and 3 have the same capacity ($2m$ elements per stage with a total of $24m$ elements) as they have to store the entire columns for each orthogonalization in execution; block 2 only stores two values for each orthogonalization under computation (the norms of the corresponding columns, which implies 24 elements).

Global Memory: Memory to store the columns elements and norms not being processed at a given time, with the same distribution as the internal memory. The size of these memory blocks is actually defined by expansibility considerations, which are discussed later.

Exchange Logic: Logic to perform the columns exchange process, which places the columns in the global memory in the order required for the next time they are used in the computation.

Multiplexer (Mx): 2-input, 4-output multiplexer to broadcast $\sin\phi$ and $\cos\phi$ to the M/M/A units.

Operation of the System

The operation of this system is as follows: a host processor loads the matrix A into the system. While this data is being received and stored in the corresponding memory (block 1 in both internal and global memory), the columns norms are computed and stored in the block of memory for the norms (block 2 also in both internal and global memory). Simultaneously, the matrix V is initialized internally by setting it to be a unitary matrix. After data has been transferred completely and everything has been initialized, the actual decomposition starts to be computed through the iterative orthogonalization process.

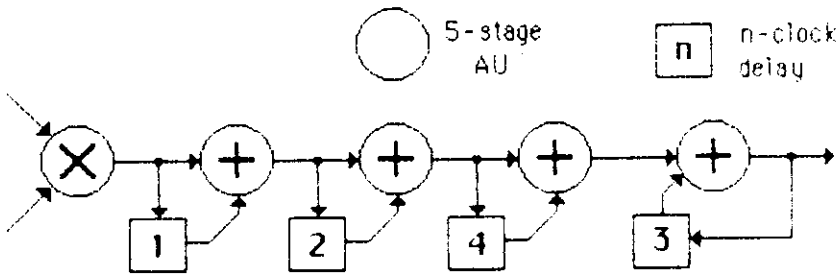

Figure 5.5 - Pipelined Inner Product Unit

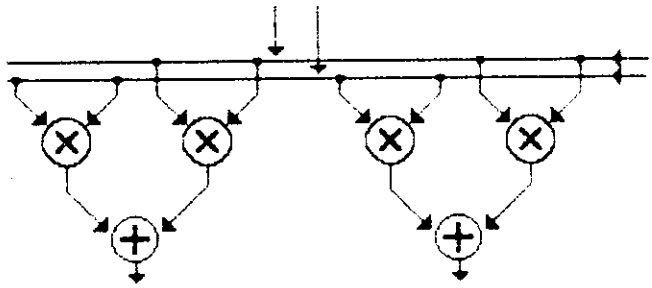

Figure 5.6 - Pipelined Rotation Units

Each individual orthogonalization consists of the following actions:

i. the elements of the columns-pairs to be rotated (of both A and V) and the norms of the columns of A are moved from the blocks in the global memory to the corresponding blocks in the internal memory. The columns of A are also input to the inner product unit.

ii. once the inner product is computed, the output of this unit and the columns norms are used to compute the rotation angle (actually $\sin\phi$, $\cos\phi$)

iii. the rotations angle values are broadcasted to all M/M/A units in the rotations stage

iv. the rotations unit also receives the columns elements from the internal memory, computes the updated ones, and transfers them to the columns exchange logic

v. the exchange logic receives the updated columns and transfers them to the global memory in the order required for later computations, after solving the

dependencies involved

All the process above is done in the twelve stages of the pipeline, so that twelve sets of columns are under processing at any time.

Data transfers

Data transfers take place at a rate of m for every orthogonalization stage time (m elements of each column have to be transferred through a data path one word wide). As a result of the operation described above, the following data movements occur at every transfer:

a. One element from each of the two columns of A whose inner product is being computed (i.e. a_i^k, a_j^k) are moved from the global memory into the inner product unit; they are also stored in the internal memory. The elements of columns of V corresponding to those above (i.e. v_i^k, v_j^k) are also transferred simultaneously from the global to the internal memory.

b. One element of each of the columns of A and V which were transferred to the internal memory $S-2$ stage-times before (i.e. $a^k{}_{i-t}$, $a^k{}_{j-t}$, $v^k{}_{i-t}$, $v^k{}_{j-t}$, $t=S-2$) are transferred to the rotation unit. These elements correspond to columns whose rotation angle has been computed and whose norms have also already been updated.

c. One element of each of the newly rotated columns (i.e. elements of $a^{k+1}{}_{i-t}$, $a^{k+1}{}_{j-t}$, $v^{k+1}{}_{i-t}$, $v^{k+1}{}_{j-t}$) are moved into the exchange logic.

d. The exchange logic output is transferred to the global memory. This output corresponds to elements from the newly rotated columns and from columns rotated in the last two stage times, as described in section 3.3.

Organization of the memory

From the description above we infer that it is necessary to read from a location and write to a different one at the same time, both in the global and in the internal memories. However, there is no possibility of conflict (contention) for the same memory location at any reference, as the columns accessed are entirely different. Furthermore, columns and columns elements are always accessed sequentially, which implies that each memory block can be implemented as two sequential-access banks (one for read and the other for write) which exchange their function after every column is transferred completely.

Data paths

The rotation of every column element requires the corresponding elements of both columns being computed as also $\sin\phi$ and $\cos\phi$; this might imply that four words must be provided at once to each M/M/A unit. However, $\sin\phi$, $\cos\phi$ are constant values for the rotation of an entire column pair, so they are transferred at the beginning and stored internally in the M/M/A units. This reduces the width of the data path into these units to only two words. Therefore, all data paths shown in Figure 5.4 are one word wide.

Multiplexer

A multiplexer is introduced to simplify the transfer of $\sin\phi$ and $\cos\phi$ to the rotations unit. With it, both values are simultaneously sent to all M/M/A units through two busses. Note that each M/M/A unit requires both $\sin\phi$ and $\cos\phi$, but those values are stored in different multipliers in each M/M/A pair. Therefore, both $\sin\phi$ and $\cos\phi$ are broadcasted once in each bus and the corresponding multipliers are selected at the proper times to latch such values.

After those values are transferred, the multiplexer isolates the M/M/A pairs such that thereafter they can operate on the columns of A and V independently and simultaneously.

Columns Exchange Logic

Logic is required to perform the columns exchange procedure described in Chapter 3. This unit involves some memory and an exchange network. Its characteristics are described in detail later.

5.2.2 Expansibility of the System for 20-Columns Matrices

The processor described above can compute not only the SVD of a 20-columns matrix as stated initially, but actually it can perform the decomposition of a matrix of any size given that its total memory is large enough to store all columns. However, larger matrices result in lower throughput. To process larger problems with small or no throughput degradation, this implementation can be extended as described now.

Assume it is desired to modify the system discussed above to compute the decomposition of matrices with 40 rows and 20 columns, with no throughput degradation. This increase in m represents an increase in the time for the inner product and rotations subcomputations, unless extra hardware is provided. The total number of

orthogonalizations is still the same, as it only depends on n .

To preserve the throughput of the rotations subfunction (which is the more stringent one), it is possible to add a slice of hardware identical to the existing at both sides of the processor (comprised of internal memory, rotation units, exchange columns logic and global memory), as shown in Figure 5.7. Doubling the arithmetic hardware in the rotations step allows to maintain the speed of the system by processing two elements of each column in parallel at every time. The extra memory is required to store the increased number of elements in each column.

Additionally, it is necessary to preserve the throughput of the inner product unit. This can be done by duplicating its computing power with two extra AUs added at the input of the existing hardware for this unit, as shown in Figure 5.8. This modification implies replacing the existing multiplier by an adder and placing two multipliers whose outputs feed the new adder. The data path width into the inner product unit is now four words.

It is also necessary to introduce an extra component which allows to broadcast the values $\sin\phi$ and $\cos\phi$ at the same time to both pairs of M/M/A units for each matrix A and V. This component is a switch which isolates each pair of rotation units during the rotation itself so that they can operate simultaneously on different data, and connects them when transferring the values of the angle for the rotations.

Further increases in the number of rows in the matrix may be treated similarly. Note that the number of data paths in such cases grows proportionally to the number of additional slices of hardware introduced.

5.2.3 SVD System for a 40-Columns Matrix

The design of a system for $m=n=40$ according to the results at the beginning of this chapter is very similar to the scheme just described. Alternatively, it is possible to arrive to such system by expanding the previous one. Recall that for matrices of that dimension, the choice was a two-processors system with twelve stages each. This system is shown in Figure 5.9, where each processor has the same architecture and operating characteristics as the one shown in Figure 5.7, but the columns exchange process now must combine the outputs from both of them. Each processor orthogonalizes different and independent pairs of columns. The corresponding exchange procedure was described in Chapter 3 and its implementation is discussed later.

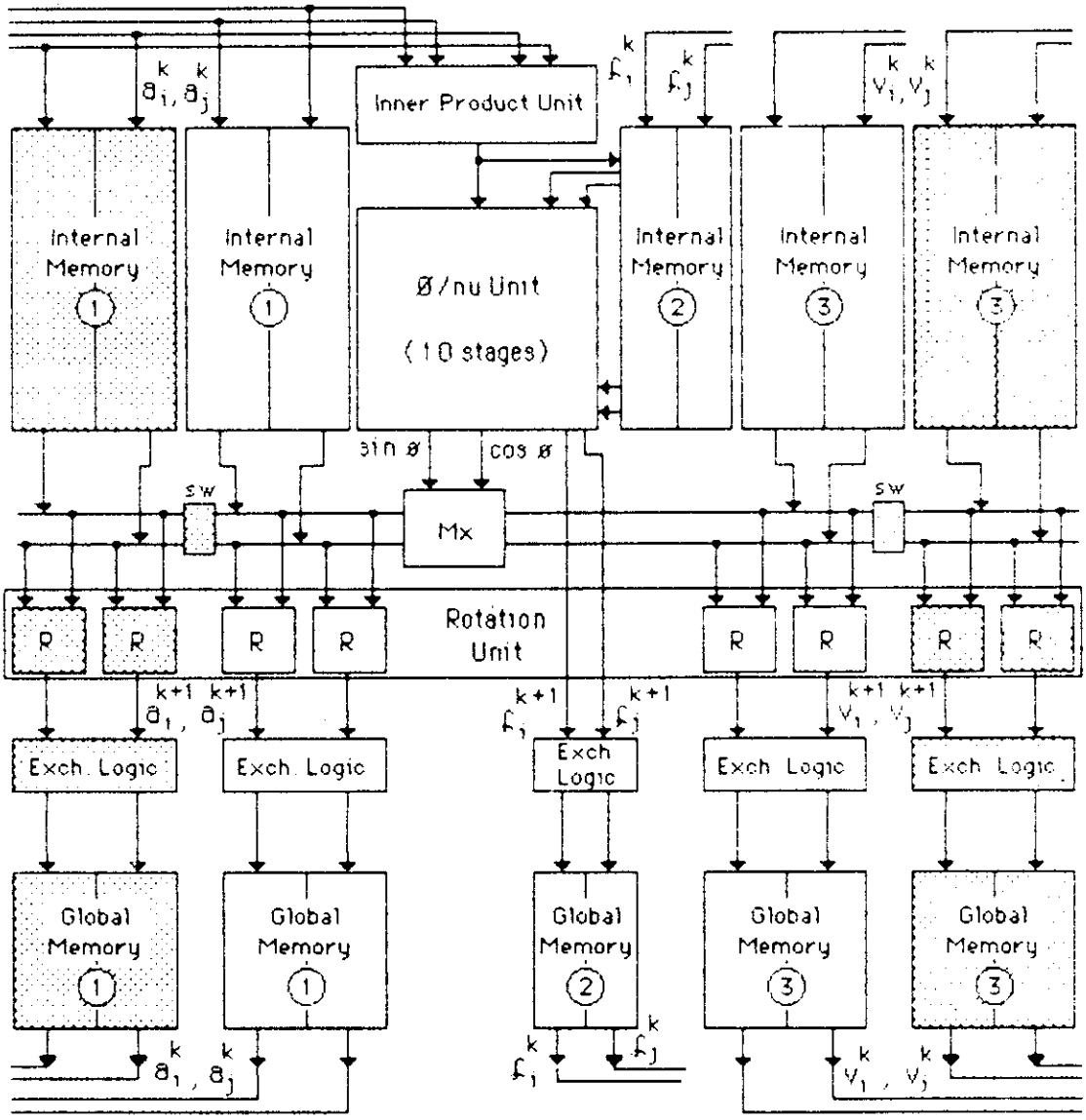

Figure 5.7 - Pipelined SUD Processor for m=40, n=20

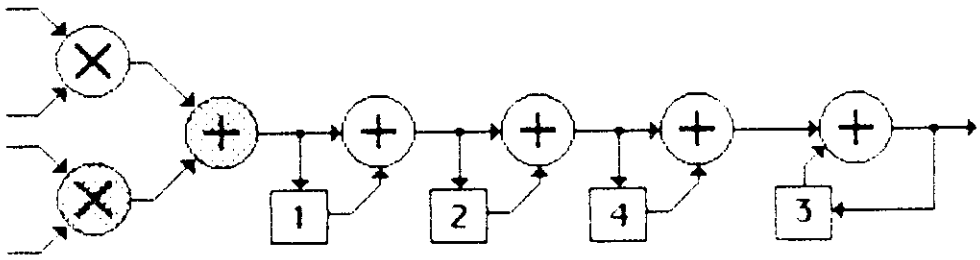

Figure 5.8 - Pipelined Inner Product Unit for m=40

Further increases in the number of columns in the matrix may also be allowed by introducing additional parallel processors. The performance characteristics of such systems have been discussed in Chapter 4. Note that the throughput for such expanded system would not be as good as what could be obtained in a system designed for a larger number of columns, that is with more stages in the orthogonalization pipeline.

Throughout the discussion here it has been shown that the pipelined scheme proposed has modularity and expansibility characteristics, in addition to its performance advantages. These issues make it an attractive alternative for the realization of a digital system to compute the SVD.

5.3 Implementation of the Columns-Exchange Process

The P/S system requires a columns-exchange process at the end of each orthogonalization such that columns will be stored in the global memory in the proper order for future rotations. The procedure used here was described in Chapter 3 for a system with P processors and an example was presented with four processors. Now the implementation for systems with one and two processors is described in more detail.

According to the discussion in Chapter 3, the exchange process for the first and last sets of columns in every step in a sweep is different from the remaining exchanges in the step. Furthermore, the exchange process requires the output of successive orthogonalizations to generate the new pair(s). It was stated that the exchange logic has to store the last two sets of orthogonalized columns; that data plus the set currently being orthogonalized allow to solve the dependencies.

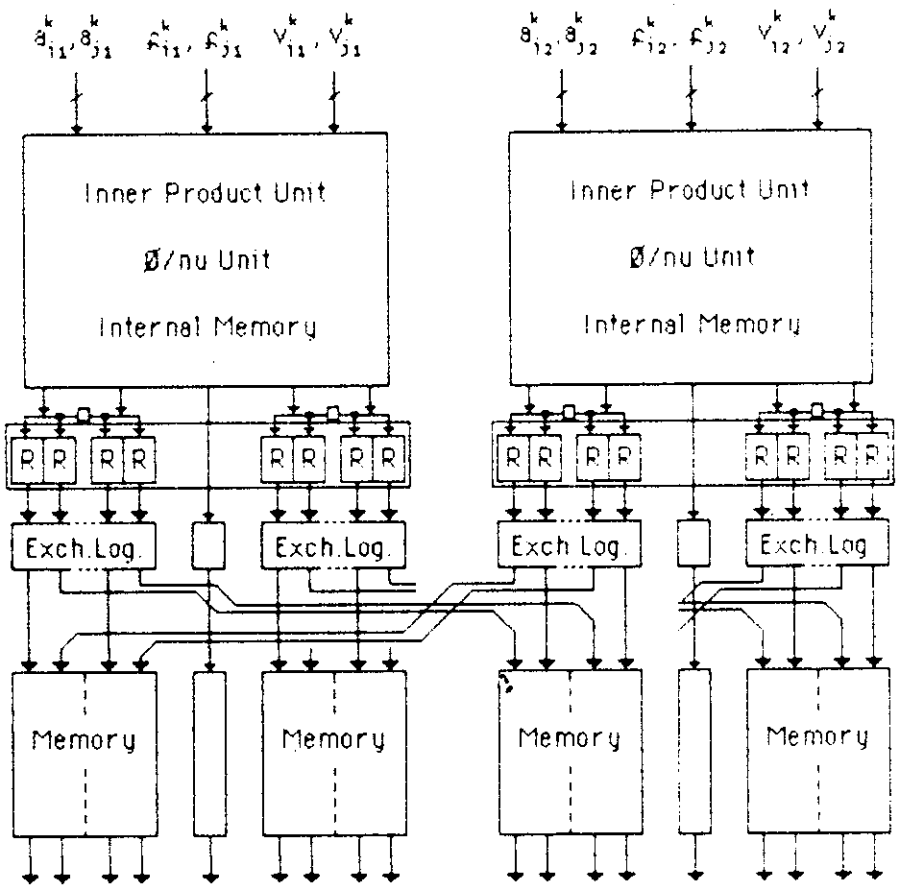

Figure 5.9 - SUD Processor for m=n=40

For the cases studied here, the ordering of the columns in the processors is as shown in Figure 5.10; this figure depicts the ordering before and after the exchange process.

Columns Before Exchange						Columns After Exchange					
$P = 1$		$P = 2$				$P = 1$		$P = 2$			
20	19	40	39	38	37	19	17	39	37	40	35
17	18	33	34	35	36	15	20	31	36	33	38
16	15	32	31	30	29	18	13	34	29	32	27
13	14	25	26	27	28	11	16	23	28	25	30
12	11	24	23	22	21	14	9	26	21	24	19
9	10	17	18	19	20	7	12	15	20	17	22
8	7	16	15	14	13	10	5	18	13	16	11
5	6	9	10	11	12	3	8	7	12	9	14
4	3	8	7	6	5	6	2	10	5	8	3
2	1	2	1	3	4	4	1	4	1	2	6

Figure 5.10 - Columns Ordering During Exchange Process

The corresponding networks to perform the exchange are shown in Figure 5.11. The dashed lines represent the connection existing for the first set of columns exchanged, while the solid lines are for all remaining ones. Implementation of this interconnection network is simple.

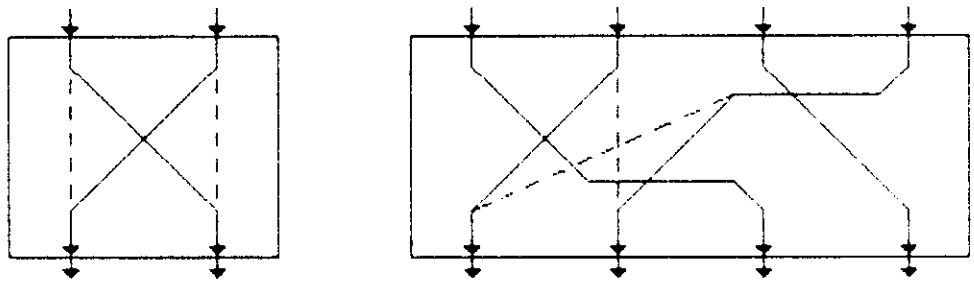

Figure 5.11 - Columns Exchange Network

The remaining issue relates to how columns are selected from the internal memory in the exchange logic and combined with the data from the set of columns being orthogonalized. All this data may be regarded as a 3-row queue which acts like a window into the data input to the columns exchange logic shown in Figure 5.10. Each of the internal cells of the queue holds a column of the matrix; these cells are addressed as indicated in Figure 5.12. Therefore, row 3 in the window (or queue) corresponds to the columns which are being rotated at a given time, row 2 to the ones rotated in the last orthogonalization and row 1 to the columns rotated two orthogonalizations before.

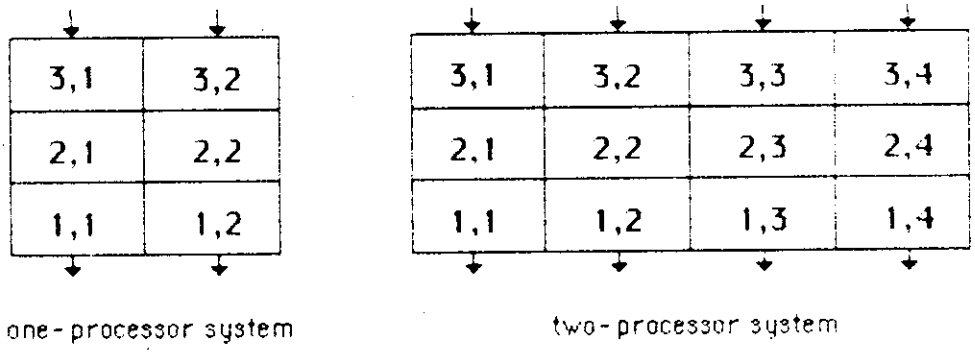

Figure 5.12 - Cells Addresses in Exchange Logic Queue

For the exchange process, this array or queue is used as follows: as soon as the first two sets of columns are available, i.e. only the two topmost rows of the queue have data, the first set of exchanged columns is output. Data moves down one row and a new column set gets in, providing the new output. For the $n = 20$ and $n = 40$ cases the cells read at every data transfer are given in Figure 5.13.

t	$P = 1$		$P = 2$			
10	11	22	21	22	13	24
9	31	12	21	12	33	24
8	11	32	21	32	13	24
7	31	12	21	12	33	24
6	11	32	21	32	13	24
5	31	12	21	12	33	24
4	11	32	21	32	13	24
3	31	12	21	12	33	24
2	11	32	21	32	13	24
1	31	22	21	22	33	24

Figure 5.13 - Addresses of Cells Read at Columns Exchange Time

There exists a high regularity in this process, as it might have been expected from its characteristics; the only exceptions are the first and last exchanges. Again, the implementation of this scheme is simple.

5.4 Division and Square-Root Algorithm and Implementation

It has been stated that, for high throughput, the SVD processor requires a fast multiplier unit. Given this condition, it becomes attractive to use a multiplicative approach to compute division and square-root, as they can take advantage of such fast hardware. While other approaches are also possible, this one is more convenient because it provides more flexibility in the design, as the multipliers involved can be shared with the other operations in the θ/nu subfunction. This is particularly adequate when the θ/nu subfunction is partitioned into stages such that division and square-root can eventually be partially performed in different stages. Actually, this corresponds to replacing those operations by several multiplications, producing a finer granularity in the algorithm to compute to angle for the rotation.

There are several schemes to perform division through a multiplicative approach. Among them, the most attractive one is Goldschmidt's method [Ander67] as it provides quadratic convergence. Ramamoorthy et al [Ramam72b] reviewed several high-speed multiplicative algorithms for square-root. They showed that there also exists a method for square-root based on the same principle as Golschmidt's division, with quadratic convergence, which shares basically the same hardware.

The basic idea in these methods is to successively multiply both numerator and denominator of a rational expression by proper factors, such that the denominator tends to 1 and the numerator to the corresponding result. Actually, the square-root method computes $Y/\sqrt{X}$ and $\sqrt{X}$ is obtained letting $Y=X$. However, the value $1/\sqrt{X}$ is also needed at the angle computation subfunction and it can be obtained letting $Y=1$ without extra hardware requirements.

Let $X_0 = X$ and $Y_0 = Y$. Then, the recurrences for $\frac{Y}{X}$ and $\frac{Y}{\sqrt{X}}$ are

Division

$$X_k = X\prod_{i=0}^{k-1} d_i \qquad X_{k+1} = X_k\, d_k$$

$$Y_k = Y\prod_{i=0}^{k-1} d_i \qquad Y_{k+1} = Y_k\, d_k$$

$$d_k = 2 - X_k$$

Square-Root

$$X_k = X\prod_{i=0}^{k-1} d_i^2 \qquad X_{k+1} = X_k\, d_k^2$$

$$Y_k = Y\prod_{i=0}^{k-1} d_i \qquad Y_{k+1} = Y_k\, d_k$$

$$d_k = 1 + \frac{x_k}{2}, \quad x_k = 1 - X_k$$

The quadratic convergence characteristic guarantees that, at every iteration, the number of leading ones in X_k is doubled. The factors d_k are easy to obtain with simple complement logic [Ramam72b]. In the square-root algorithm, the factor $d_k = 1 + x_k/2$ is obtained from $1 + x_k = 2 - X_k$, i.e. performing two's complement of X_k and then right shifting the fractional part one bit.

There are several possible improvements to the basic algorithm [Ander67]. A relevant one here is related to the selection of the first factor to guarantee that X_1 has several leading 1's; this is accomplished using a table lookup scheme instead of 2's complement logic. If X_1 has four leading 1's, then 4 iterations are necessary to achieve 32 bits of precision in the result, because every time the number of leading 1's is doubled; six initial leading 1's will produce 24 bits of precision in only three

iterations, adequate results for 32-bit floating point numbers.

Figure 5.14 shows the dependence graphs for the two operations according to this method, assuming four leading 1's after the table lookup operation. Due to the parallel and independent nature of multiplying numerator and denominator, pipelined multipliers with two stages are useful. Assuming that the table-lookup and the 2's complement operations are performed in half the multiplication time, division requires the equivalent of 6 multiply times and square-root the equivalent of 9.

Implementations of these two operations might consider the use of the same multipliers as in the other parts of the system, which for this analysis have been considered as having five stages. The additional stages in such AUs would not be used, as there is no more concurrency in these computations.

5.5 Characteristics of Pipelined Arithmetic Units

It has been shown that pipelined AUs are useful for higher throughput in the SVD computation. The analysis developed considered up to five stages in the arithmetic units, though the feasibility of this value has not been discussed so far. The characteristics of pipelined AUs are reviewed now in more detail, including the limits of pipelining with current technologies.

It is interesting to note that the latest research efforts in the area of VLSI design of multiplier units have been oriented towards fast devices, with multiply times of around 100 [nsec] for 32-bit floating-point values. Ho et al [Ho84] designed a 24-bit by 24-bit multiplier, which uses partial product generation in 4-bit nibbles followed by parallel pseudo-counters for product term reduction. Their device performs the computation in 70 [nsec] and is implemented using 1.5 micron CMOS technology. Iwamura et al [Iwamu84] presents a CMOS/SOS 16-bit by 16-bit parallel multiplier using a modified array technique, to achieve 27 [nsec] multiply time. Kaji et al [Kaji84] designed a 16-bit by 16-bit CMOS parallel multiplier using Booth's algorithm, Wallace tree, and a final conditional sum adder to achieve 45 [nsec] time. Uya et al [Uya84] have a 32-bit by 32-bit CMOS floating-point multiplier with 78.7 [nsec] multiply time, using modified Booth's algorithm, CSA array, and modified carry select adder for the final step. Advanced Micro Devices [AMD84] has the Am29300 family, which includes a 3-port 32-bit multiplier able to operate in a single cycle of less than 100 [nsec]; this device uses ECL circuitry internally, though its output is TTL.

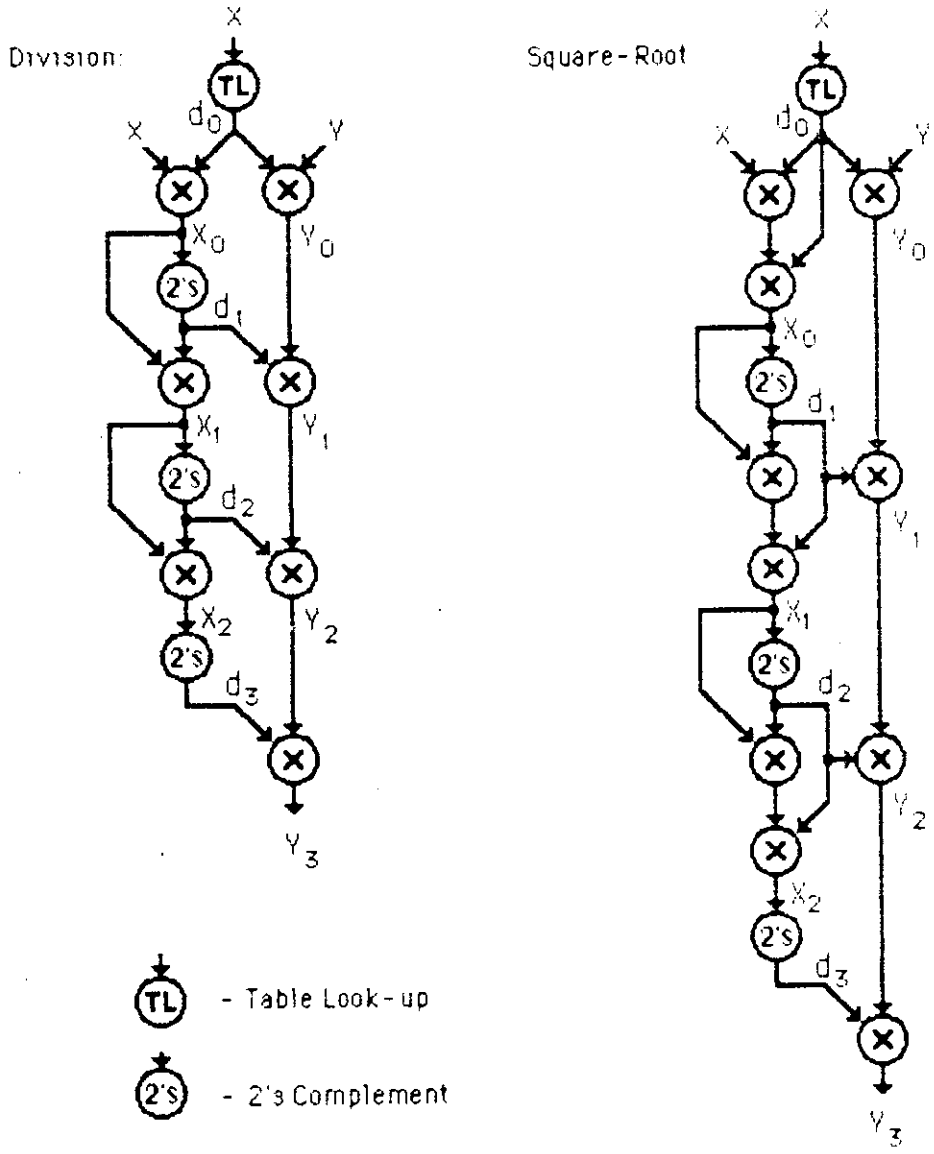

Figure 5.14 - Dependence Graphs for Division and Square-Root

However, some pipelined VLSI units have been also been implemented although their degree of pipelining is low. Welten and Lohstroh [Welte84] designed a 12-bit by 12-bit fully parallel multiplier/accumulator, able to operate at 50 Mhz in a two-stage pipeline or at 25 Mhz in non-pipelined mode. Woo et al [Woo84] discuss the Weitek WTL 1030 family, which has NMOS devices to perform addition and multiplication in a 3-stage pipeline of 200 [nsec] per stage, or in flow-through mode at 600 [nsec] per operation.

The reasons why not much effort has gone into pipelined VLSI AUs are unclear; it seems that user demands have pushed for faster units, with low latency, instead of requesting high throughput. Technologically, there are no real limitations to the design of devices with clock rates in the order of 20 [nsec], as discussed later.

Larger degree of pipelining in arithmetic units is found in vector computers. Some examples are mentioned here. Texas Instrument Advanced Scientific Computer TI-ASC has a multiconfigurable pipeline with eight stages for floating-point and fixed point operations [Siewi82, Kogge81], which runs at 60 [nsec]. The IBM 2938 Array Processor arithmetic unit is a four-stage multiply-add pipeline which operates at a staging rate of 200 [nsec] [Kogge81]. The CDC STAR-100 computer had two pipelines, both of them with a 40 [nsec] clock: a four-stage adder / eight-stage multiplier and a general purpose pipeline for floating-point operations [Kogge81]. The CRAY-1 computer has twelve functional units, all of them pipelined and ranging from two to fourteen stages; it includes a six-stages floating-point adder, a seven-stages multiplier and a fourteen stages reciprocal unit, all running at 12.5 [nsec] [Kogge81]. These implementations indicate that highly pipelined AUs are feasible and practical for certain applications, as is the case here.

Hallin and Flynn [Halli72] studied the effect of pipelining on system efficiency for two addition and three multiplication algorithms, with 48-bit fixed operands. They defined efficiency as $e = \frac{w}{t_s\, g}$ where w is the number of bits in the operand, t_s is the pipeline stage delay and g is the total number of gates including any used for latches. Their analysis was based on the use of Earle latches [Earle65]. They concluded that the most efficient algorithms among the ones studied were conditional-sum adder and adder-tree multiplication (with recoding algorithm), both maximally pipelined.

Because of the underlying hardware technology used in their study, i.e. the Earle latch, each pipelined stage has to include at least four gates [Halli72, Kogge81]. Any smaller number of gates per stage would not increase the bandwidth.

Hallin and Flynn concluded that the 48-bit multiplier was 32 gate delays long: 4 gate delays for the pre-encoder, 14 for the adder tree and 14 for the carry-propagating adder [Halli72]. To use pipelining efficiently, the entire multiplier had to be considered as one unit and partitioned into stages depending on the gates delay only. That approach resulted in 8 stages for the maximally pipelined multiplier. For 32-bit floating-point values, the corresponding multiplier would have a delay of 30 gate delays: 4 gate delays for the encoder, 10 gate delays for the reduction of 12 addends, 12 gate delays for the final carry propagating addition, and 4 gate delays for post-normalization. (1) Therefore, with this methodology and technology, it is possible to have 7 stages for the multiplier unit (maximally pipelined). This assumes extra hardware in parallel to handle the exponents.

Hallin's fixed-point 48-bit conditional-sum adder has 3 stages and 12 gate delays in total; the unit is implemented with 12-bit conditional-sum sections. For 32-bit floating-point values, estimating again 4 gates delay each for pre-shifting and post-shifting, and considering 10 gate delays for the fractional part addition, the total reaches 18 gate delays. Therefore, 4 stages could be implemented in a maximally pipelined adder.

Hallin and Flynn's results allow to estimate the degree of pipelining possible to include in multiplier and adder units, using Earle latches technology. Consequently, the maximum of 5 stages per AU used in the analysis before is a reasonable amount, although it is not the real maximum limit.

Current VLSI designs of pipelined units do not use Earle latches, since a better alternative is to use dynamic circuits and registers: they are faster and require less area and power. The design methodology is based on register-to-register transfers through combinational logic, implemented with inverting logic and pass transistors, respectively. Latches are implemented by charge stored on the input-gate capacitance of the inverting logic stages, isolated by pass transistors controlled by clock signals. This approach reduces area and power requirements significantly [Mead80].

Ideally, only pass transistors should be used to implement the logic functions. As the inverting logic stages are connected through these pass transistors, inverter ratios of $k \approx 8$ are required; these inverters have a maximum delay of 8τ (where τ is the transit time of the input gate in the inverter). To minimize the delay per stage, the total delay through the pass transistors is made equal to the inverter delay. Therefore,

---

(1) Pre-shifting and post-shifting can be performed fast (one gate delay) using barrel-shifters [Mead80]. The remaining gate delays are for the detection of the amount of shifting. However, this approach is somewhat expensive in area. Alternatively, a normalization scheme proposed in [Uya84] performs this function in 5 gate delays, with less hardware.

the time available for logic functions in each stage is only 8τ, which allows a maximum path length of about 3 pass transistors [Mead80]. This amount of logic could implement a two level boolean expression (AND/OR) of medium complexity (about 3 terms with 3 to 4 literals each). For higher-complexity subsystems, a PLA structure in each stage could be used, as it provides similar timing characteristics [Mead80].

Mead and Conway state that a system of this type could run at a clock period of $\approx 100\tau$. Under these conditions, clocking periods of 20 [nsec] are achievable today in carefully structured integrated systems, where successive stages are in close physical proximity. This figure may increase by a factor of 2-3 in the near future, as the minimum feature size of transistors reduces to 1.5 microns [Mead80, Tyree85].

Arithmetic units implemented with this scheme can have larger number of stages than those obtained from Hallin and Flynn limits, as the amount of logic required at each stage to match the delay of the latches is less than the four gates stated before. Therefore, it is feasible to think about highly pipelined arithmetic units, with at least 5 stages each, running at 20 [nsec] clocking periods. These devices are critical to the success of the SVD processor. For instance, devices with the characteristics described above would allow to perform the decomposition of a 40 by 40 matrix in the equivalent of 19600 multiply times, which corresponds to $t_D \approx 1.96$ [msec]. Alternatively, 100 and 200 [nsec] clocking periods would lead to 9.8 [msec] and 19.6 [msec] decomposition times respectively.

Larger number of stages in the AUs would allow to reduce the number of units required for a given speedup, increasing the efficiency of the implementation, as indicated by the analysis performed in Chapter 4.

5.6 Control Logic Implementation Characteristics

The control function in the pipelined processor presented before has to be distributed throughout the different stages, given the particular characteristics of each stage. The controller in each of them is essentially a finite-state machine which may be implemented using a read-only-memory (where each ROM location contains the actual control signals for the different components within the stage), or a programmable-logic-array (PLA) which implements the combinational logic functions for the states. A general scheme for such device is shown in Figure 5.15.

The architecture of this unit is the same for all stages, with the only differences found in the actual contents and possible the width of the ROM (or the functions implemented and the width of the AND/OR planes in the PLA case).

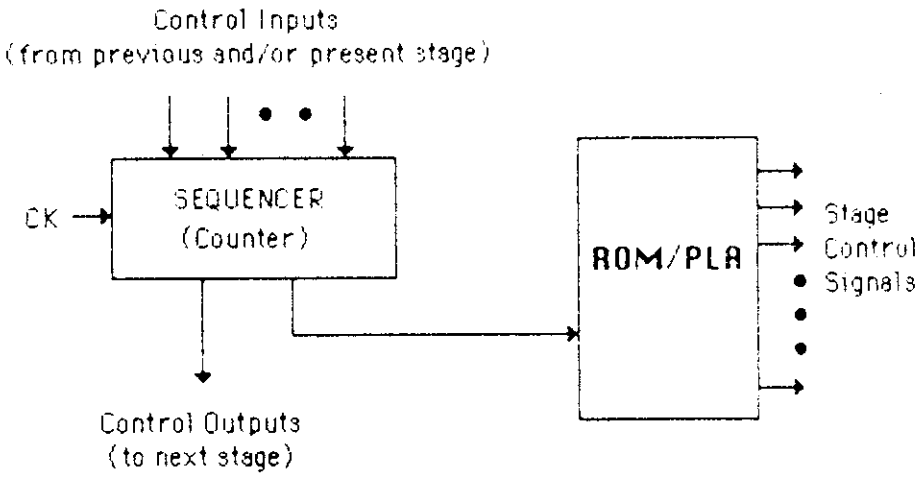

Figure 5.15 - Stage Controller

This scheme is feasible because the operations in each stage are a linear sequence of events (without branches). The only deviation from this sequential behavior is found when columns are tested to verify if they are already orthogonal, since in that case they do not have to be rotated. However, in such case the controller still needs to traverse a sequence of no-operations of the same length as normal processing, to keep the pipeline synchronization. This can be achieved by one status bit inhibiting the transmission of the control signals to the stage units.

Using devices designed for this application, it should be possible to implement the entire control function in each stage in only one VLSI device. The amount of logic required and/or the size of the ROM are small (for the P/S systems described before, the control function in each stage has to traverse a sequence of twenty states defined by the same number of arithmetic operations in pipelined units with five internal stages). The number of external connections depends in the control signals required in each stage; this issue requires further study.

5.7 Testability Considerations

Due to the complexity of the implementation, it is convenient to have some form of support for testing to allow to verify the operating condition of the system, either on-line or off-line. Some preliminary considerations regarding the feasibility of testing are described here. A more detailed study is out of the scope of this work, but it is adequate to point out some of the characteristics in the system devised which are useful for this task.

As inputs and outputs to the processor exist only at the ends of the orthogonalization hardware, the entire system may be regarded as one component. Therefore, the same principles applied to support testing of VLSI devices may be used here, which involves the need to include some special functions or hardware in the implementation to improve its testability.

As the system is comprised of many devices, two levels of testing may be considered:

i. Internal Testing of Devices. For this, each device should include the hardware required to allow for its own testing, providing an output to indicate its status.

ii. System Level Testing. The system includes the required hardware for testing purposes. This may be on-line (i.e. one out of a number of orthogonalizations is used for testing, for instance) or off-line (i.e. testing functions are performed at power-up or when the processor is idle).

Assuming a design based on scheme (ii), and given the pipelined characteristics of the processor, one approach is as follows:

a) Define the entire system as a set of register-to-register transfer blocks, that is, successive stages of storage registers with logic between them.

b) Provide external reading and writing to/from each of the storage locations.

The proposed SVD processor suits this approach with little extra hardware. Each of the stages has the described register-to-register transfer structure, being necessary only to allow their direct reading/writing. The methodology in this case is to test the storage locations first, for their ability to store data and control information. After this is done successfully, each block between registers may be tested separately [Mead80].

The logic at each internal block is rather complex and it might still require some additional considerations for testing purposes. However, the relevant issue is the need for access to the individual storage locations, as without it "testing rapidly becomes hopeless" [Mead80]. This requirement may be achieved in the proposed system using the busses already existing in the processor(s) and providing the proper connections with each storage register. The testing function may be assigned to a global controller, which works together with the local controllers in each stage to perform this task. Figure 5.16 illustrates the connections for this scheme.

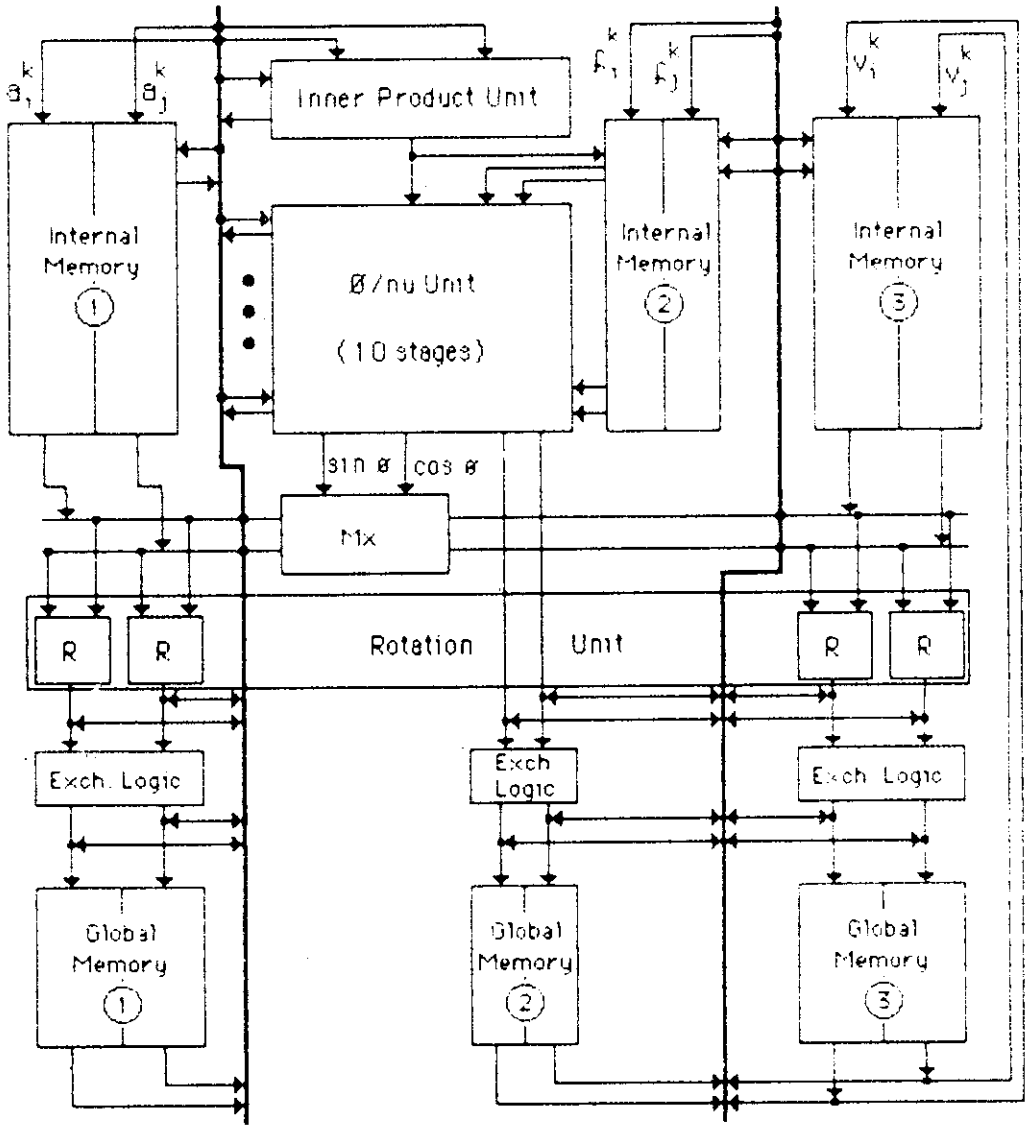

Figure 5.16 - Connections for Testing Support

In the bi-processor design, there is a simpler possibility for rapid testing of the operating condition of the entire processor. It consists in providing both processors with the same data and compare their results; if they do not match, then a deeper testing mode is entered. Some extra logic is required to compare those outputs. This testing can be done on-line (i.e. interspersed with actual computations).

Even more attractive is the fact that, in the bi-processor system, upon the detection of fault in one processor and its proper recognition, the system may continue to work with only one of them by reconfiguring its input/output from the global memory, though obviously the throughput will be reduced.

5.8 Custom Device Design in the Digital System for the SVD

A current realization of the digital system to compute the SVD would be based upon the use of one arithmetic unit per device, such as those currently available. Therefore, there would be many devices in the system leading to a board-level implementation. In addition to the arithmetic units, other devices are required to implement memories, control functions, and columns exchange logic.

However, the design of special purpose devices could reduce significantly the number of components in the system. Those special devices required have been suggested or mentioned throughout the analysis, and can be listed as follows:

i. Highly pipelined adders, multipliers, and adder/multiplier units.

ii. ROM-based or PLA-based controllers (finite-state-machines).

iii. Columns-exchange logic, as described in section 5.3.

iv. Sequential-access memories as two banks (one for read, one for write) which exchange their function, as described in section 5.2.1.

v. Highly pipelined multipliers specially designed for fast computations of products between a constant and successive numbers, as needed in the rotation unit.

Furthermore, potential density increase in VLSI technology could allow to integrate into one device subsystems of several of the devices above, with the corresponding advantages regarding total number of components, speed, and power consumption. Those subsystems are characterized for being self contained such that their integration does not increase significantly the number of external connections, as

is described next.

Besides what is achievable just from higher integration, there are parts in the proposed system which could exploit some particular characteristics of the computation in the SVD algorithm, such as the following:

• The input data to the inner product unit (which are floating-point values) is normally available before it is actually used (given that $n \geq 2PS + 2$). Therefore, it is possible to precompute the exponent of the result by adding the exponents of corresponding elements of each pair of columns and saving the largest one, before starting the inner product computation of the mantissa. Then, when data is transferred to the unit, it could be properly denormalized, perform the computation only with the mantissa, concatenate the previously obtained exponent and perform a final normalization step if necessary.

Even if data is not available in advance, this same approach may be implemented in a two-stage inner product unit where the first one precomputes the exponent of the result, and the second performs the actual inner product and final normalization for the mantissa.

• For the θ/nu subfunction it was assumed and shown that division and square-root for the data format considered could be performed in 6 and 9 multiply times respectively. The entire angle computation and norms update subfunctions were partitioned into stages based on those assumptions, using regular floating-point multipliers. With larger density in the devices, it might become possible to implement the entire multiplicative algorithm for each of these two operations in one device, reducing their computation times. Some properties of the algorithm are useful for such case. The scheme multiplies both numerator and denominator of a rational expression by a factor in the range (1,2). At every iteration there could be a limited overflow only in the numerator. If this condition is handled adequately, then it is possible to perform this iterative algorithm only with the mantissa, subtract exponents in parallel and compute the final normalization, reducing the time for these operations. Furthermore, the speed of this process could be increased more if it is considered that the first iterations do not require the full precision in the computation [Ander67].

• The rotations unit is the most compute-bound of the entire processor. As it is based in M/M/A units, the obvious approach is to integrate the three operators in one device. Full floating-point computations are required here and no precomputation seems possible, though part of the data (i.e. the columns elements) is available well in advance.

This subfunction needs the most attention, as larger throughput requires a large amount of hardware. To reduce VLSI area and be able to integrate several functions in one device, it is convenient to consider the suggestion to use highly pipelined multipliers taking advantage from the constant operand for successive operations. Special multiplication algorithms suitable for this application should be devised, but at least some gains are possible if the modified Booth's algorithm is used. A higher level of multiplier recoding can be used; the extra time and hardware required for the recoding would allow faster computation of the operations and savings of area, since there would be less addends to reduce.

Summary

This chapter has presented the implementation characteristics of a digital system to compute the SVD. It has been shown that the P/S architecture is convenient not only in terms of efficiency but it is also a regular structure, with expansibility properties for higher throughputs, and with capabilities to compute the decomposition of a matrix of any size without hardware modification. All these characteristics make this architecture a promising and attractive alternative for realization.

CHAPTER 6
CONCLUSIONS

In this thesis we have studied the singular value decomposition (SVD) algorithm according to Brent and Luk [Brent82a] version of Hestenes' method, and its characteristics for hardware implementation. The goal has been to search for an efficient implementation in terms of highest throughput for a given amount of hardware.

We first reviewed the developments in the field, particularly what has been researched regarding systolic arrays for this algorithm. We concluded that most of the reported research about the design of a digital system to compute the SVD has been focused on the suitability of Hestenes' method for concurrent computation, but no analysis has been presented regarding the actual computation time and hardware requirements for the schemes proposed (the systolic array architectures). We also questioned the suitability of those arrays for the algorithm under consideration, because none of those schemes really considers the particular properties existing at the lower levels of the algorithm.

To formalize the analysis, an algorithmic model for the computation was introduced in Chapter 3 and cost and performance measures were defined. The model used a directed graph as a description of the algorithm; nodes correspond to subcomputations and arcs to precedences among those subfunctions. The concepts of replication, pipelining and parallelism of a graph were discussed with respect to this model. The cost and performance measures used were computation time, hardware requirements, speedup with respect to a completely sequential implementation, efficiency and hardware utilization.

This model allowed to infer basic properties of architectures for these type of computations, concluding that replication of a pipelined processor which also exploits the parallelism of the graph of the algorithm was the alternative that offered the best characteristics for higher throughput. However, this model also showed that for speedup factors up to a certain value the best approach is replication of a completely sequential implementation of the algorithm; the limit is imposed by the properties of the algorithm. This model assumed only one type of operation units, as is the case for the SVD.

A methodology for the design of digital systems exploiting concurrency at several levels of the algorithm was also presented in Chapter 3. This methodology is essentially an iterative procedure consisting of top-down decomposition and bottom-up refinement of the nodes in the graph of the algorithm.

One pipelined processor or a system with replicated processors requires a mechanism to solve the data dependencies existing in the algorithm. The solution presented here is based on an adaptation of the parallel ordering for the orthogonalizations of the columns of the matrix proposed by Brent et al [Brent82a]. The scheme obtained has the same advantages as the original one regarding simplicity and communications only with nearest neighbors in implementations with multiple processors, but is not restricted to a particular number of processors and, therefore, is useful for a system with any number of replicated units up to the limits imposed by the algorithm. In contrast, Brent's scheme assumes the existence of $n/2$ processors.

In Chapters 3 and 4 we applied to the SVD algorithm the design methodology mentioned above, in a system with P replicated processors with S stages per processor; this scheme was called a P/S system. In this architecture, one pipelined processor and a linear systolic array are just particular examples. Chapter 3 discussed throughput characteristics for the system in terms of the time for an orthogonalization. It was shown that such throughput is proportional to the number of processors and to the stages in them, as the algorithmic model predicted. A slight modification to the model was required to account for the data dependencies, which are somewhat more complex than what the model considered; however, that modification did not change the results obtained.

In Chapter 4, implementations for the different subfunctions within the orthogonalization process were devised, discussing the throughput characteristics and hardware requirements for them. It was shown that those subfunctions have very different characteristics and the implementations for them should pursue distinct approaches, including internal pipelines and exploiting the parallelism in their corresponding graphs. It was also shown that the throughput characteristics are heavily dependent on the utilization of pipelined arithmetic units. A procedure for the design of the hardware for the orthogonalization process was presented, according to the results obtained.

Using the information obtained before, Chapter 5 presented a procedure for the design of architectures for the SVD based on the P/S system. The architectures devised were compared and evaluated in terms of efficiency, cost, and hardware utilization as defined in the performance measures, placing emphasis on the arithmetic hardware since this is the predominant requirement.

The procedure used in the design allowed to obtain not only implementations for high throughput but the most efficient architecture for a given throughput. In this sense, it was shown that for throughputs up to a certain value, the best approach is replication of a completely sequential implementation for the orthogonalization algorithm. The limit is imposed by the characteristics of the SVD computation, which exhibits dependences for groups of instances of the computation. The lowest cost linear systolic array corresponds to an implementation with that limiting number of replicated processors, with one arithmetic unit per processing element. However, for higher throughput than what is achieved with such systolic array, a multilevel pipelined approach offers better efficiency and hardware utilization than any other alternative. Such scheme exploits concurrency at several levels through pipelines and the use of the parallelism in the subcomputations.

Implementations for systems to compute the SVD of 20 by 20 and 40 by 40 matrices were devised. It was shown that those architectures have modularity and expansibility characteristics, in addition to their performance advantages. Therefore, they are attractive alternatives for realization. Furthermore, they are not restricted to compute the decomposition of matrices of the dimensions given, but they can be used for problems of any size as long as the memory in them is large enough to hold the data. If higher throughput than what is achieved with a particular multilevel pipeline implementation is desired, that implementation can be improved by introducing hardware structures identical to the existing ones.

It is also suggested that the implementation of the scheme proposed might be improved significantly (in terms of number of devices and interconnections among them) if units specially designed for the application are used. This includes the integration of certain modules in the computation or exploiting some characteristics of the implementation and the algorithm. In fact, this topic should be the focus of additional study: the research for VLSI implementation of devices suitable for the SVD system described here, namely dual-port dual-bank sequential access memories, highly pipelined arithmetic units, columns exchange logic and controllers for the stages of the pipeline.

The results obtained here can be extended to algorithms with similar characteristics to the SVD. Furthermore, the methodology presented can also be applied to algorithms with characteristics different than the SVD, and be a convenient tool for the analysis and evaluation of the most efficient architectures for such cases.

References

[Ahmed82] H. Ahmed, J. Delosme, and M. Morf, "Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing," *IEEE Computer* **15**, pp. 65-82 (January 1982).

[AMD84] Advanced Micro Devices AMD, *Am29300 Family Catalog*, 1984.

[Ander67] S.F. Anderson, J.G. Earlie, R.E. Goldschmidt, and D.M. Powers, "The IBM System/360 Model 91: Floating Point Execution Unit," *IBM Journal*, pp. 34-53 (January 1967).

[Andre76b] H. Andrews and C. Patterson, "Singular Value Decomposition (SVD) Image Coding," *IEEE Transactions on Communications* **24**, pp. 425-432 (1976).

[Andre76a] Harry C. Andrews and Claude L. Patterson, "Singular Value Decomposition and Digital Image Processing," *IEEE Transactions on Acoustics, Speech and Signal Processing* **ASSP-24**(1), pp. 26-53 (February 1976).

[Brent82a] R.P. Brent and F.T. Luk, "A Systolic Architecture for the Singular Value Decomposition," Tech. Rep. TR-82-522, Computer Science Department, Cornell University, (1982).

[Brent82b] R.P. Brent and F.T. Luk, "A Systolic Architecture for Almost Linear-Time Solution of the Symmetric Eigenvalue Problem," Tech. Rep. TR-82-525, Computer Science Department, Cornell University, (1982).

[Brent83] R.P. Brent, F.T. Luk, and C. Van Loan, "Computation of the Singular Value Decomposition Using Mesh-Connected Processors," Tech. Rep. TR-82-528, Computer Science Department, Cornell University, (1983).

[Busin69] Peter A. Businger and Gene H. Golub, "Singular Value Decomposition of a Complex Matrix," *Communications of the ACM* **12**(10), pp. 564-565 (October 1969).

[Chart62] B.A. Chartres, "Adaptation of the Jacobi Method for a Computer With Magnetic-Tape Backing Store," *The Computer Journal* **5**, pp. 51-60 (1962).

[Cimin81] L. Ciminiera and A. Serra, "Arithmetic Array for Fast Inner Product Evaluation," *Proceedings IEEE 5th. Symposium on Computer Arithmtic*, pp. 207-214 (1981).

[Earle65] J. Earle, "Latched Carry-Save Adder," *IBM Tech. Disclosure Bulletin* **7**(10), pp. 909-910 (March 1965).

[Estri78] G. Estrin, "A Methodology for the Design of Digital Systems - Supported by SARA at the Age of One," *AFIPS Conference Proceedings, NCC* **47**, pp. 313-324 (1978).

[Finn82b] Alan Finn and Christopher Pottle, "An Algorithm and Simulation Results for a Systolic Array Computation of the Singular Value Decomposition," *International Large Scale Systems Symposium*, pp. 93-97 (October 1982).

[Finn82a] Alan M. Finn, Franklin T. Luk, and Christopher Pottle, "Systolic Array Computation of the Singular Value Decomposition," *SPIE Real Time Signal Processing V* **341** , pp. 35-43 (1982).

[Forsy77] G. Forsythe, Malcolm, and M. Moler, pp. 218-235 in *Computer Methods for Mathematical Computations*, Prentice-Hall (1977).

[Golub70] G. Golub and C. Reinsch, "Singular Value Decomposition and Least Squares Solution," *Numerische Mathematik* **14**, pp. 403-420 (1970).

[Golub81] G. Golub, F. Luk, and M. Overton, "A Block Lanczos Method for Computing the Singular Values and Corresponding Singular Vectors of a Matrix," *ACM Transactions on Mathematical Software* **7**(2), pp. 149-169 (1981).

[Golub65] G.H. Golub and W. Kahan, "Calculating the Singular Values and Pseudo-Inverse of a Matrix," *SIAM Journal on Numerical Analysis* **2**, pp. 205-224 (1965).

[Golub77] Gene H. Golub and Franklin T. Luk, "Singular Value Decomposition: Applications and Computations," *Transactions of 22nd Conference of Army Mathematicians*, pp. 577-605 (May 1977).

[Gonza77] M. Gonzalez, "Deterministic Processor Scheduling," *Computer Surveys* **9**(3), pp. 173-204 (September 1977).

[Halli72] T. Hallin and M. Flynn, "Pipelining of Arithmetic Functions," *IEEE Transactions on Computer*, pp. 880-886 (August 1972).

[Helle83] H. Heller and I. Ipsen, "Systolic Networks for Orthogonal Decompositions," *SIAM J. Scient. and Stat. Comp.* **4**, pp. 261-269 (1983).

[Heste58] M.R. Hestenes, "Inversion of Matrices by Biorthogonalization and Related Results," *Journal SIAM*, pp. 51-90 (1958).

[Ho84] C. Ho, R. Jerdonek, S. Noujaim, and D. Schumacher, "A High Performance 1.5 Micron CMOS 24x24 Bit Multiplier," *IEEE Custom Integrated Circuits Conference*, pp. 30-33 (1984).

[Huang75] T. Huang and P. Narendra, "Image Restoration by SVD," *Applied Optics* **14**(9), pp. 2213-2215 (September 1975).

[Iwamu84] J. Iwamura, K. Suganuma, M. Kimura, and S. Taguchi, "A CMOS/SOS Multiplier," *Proceedings International Solid State Circuits Conference* **ISSCC-84**, pp. 92-93 (1984).

[Kaji84] Y. Kaji, N. Sugiyama, Y. Kitamura, S. Ohya, and M. Kikuchi, "A 45 ns 16x16 CMOS Multiplier," *Proceedings International Solid State Circuits Conference* **ISSCC-84**, pp. 84-85 (1984).

[Klema80] Virginia C. Klema and Alan J. Laub, "The Singular Value Decomposition: Its Computation and Some Applications," *IEEE Transactions on Automatic Control* **AC-25**(2), pp. 164-176 (April 1980).

[Kogge81] P. Kogge, in *The Architecture of Pipelined Computers*, McGraw-Hill (1981).

[Konst84] K. Konstantinides and K. Yao, "Applications of SVD to System Modeling in Signal Processing," *Proceedings International Conference on Acoustics, Speech and Signal Proceesing* **ICASSP-84**, pp. 5.7.1-5.7.4 (1984).

[Kung82] H. Kung, "Why Systolic Architectures?," *IEEE Computer* **15**(15), pp. 37-46 (1982).

[Luk80] Franklin T. Luk, "Computing the Singular Value Decomposition on the ILLIAC IV," *ACM Transactions on Mathematical Software* **6**(4), pp. 524-539 (December 1980).

[Mead80] C. Mead and L. Conway, *Introduction to VLSI Systems,* Addison-Wesley (1980).

[Miao84] N. Miao and Z. Chen, "Application of SVD to 2-D Spectral Estimation," *Proceedings International Conference on Acoustics, Speech and Signal Proceesing* **ICASSP-84**, pp. 4.2.1-4.2.4 (1984).

[Nash75] J.C. Nash, "A One-Sided Transformation Method for the Singular Value Decomposition and Algebraic Eigenproblem," *The Computer Journal* **18**(1), pp. 74-76 (1975).

[Nash76] J.C. Nash and L.P. Lefkovitch, "Principal Components and Regressions by Singular Value Decomposition on a Small Computer," *Applied Statistics* **25**(3), pp. 210-216 (1976).

[Nash79] J.C. Nash, in *Compact Numerical Methods for Computers*, J. Wiley and Sons (1979).

[Patil72] S. Patil and J. Denis, "The Description and Realization of Digital Systems," *Digest of Papers, COMPCON 72*, pp. 223-226 (1972).

[Peter81] J. Peterson, in *Petri Net Theory and Modelling of Systems*, Prentice Hall (1981).

[Ramam72a] C. Ramamoorthy, K. Chandy, and M. Gonzalez, "Optimal Scheduling Strategies in a Multiprocessor System," *IEEE Transactions on Computers* **C-21**(2), pp. 137-146 (February, 1972).

[Ramam72b] C.V. Ramamoorthy, J. Goodman, and K. Kim, "Some Properties of Iterative Square-Rooting Methods Using High-Speed Multiplication," *IEEE Transactions on Computers* **C-21**(8) (August 1972).

[Rutis66] H. Rutishauser, "The Jacobi Method for Real Symmetric Matrices," *Numerische Mathematik* **9**, pp. 1-10 (1966).

[Schre82] Robert Schreiber, "Systolic Arrays for Eigenvalue Computation," *SPIE Real Time Signal Processing V* **341** , pp. 27-34 (1982).

[Schre83] Robert Schreiber, "On the Systolic Arrays of Brent, Luk and Van Loan," *SPIE Real Time Signal Processing VI*, pp. 72-76 (1983).

[Shim81] Y. Shim and Z. Cho, "SVD PseudoInversion Image Reconstruction," *IEEE Transactions on Acoustics, Speech and Signal Processing* **ASSP-29**(4), pp. 904-909 (August 1981).

[Sibul84a] L. Sibul, "Application of SVD to Adaptive Beamforming," *Proceedings International Conference on Acoustics, Speech and Signal Proceesing* **ICASSP-84**, pp. 33.11.1-33.11.4 (1984).

[Sibul84b] L. Sibul and A. Fogelsanger, "Application of Coordinate Rotation Algorithm to SVD," *IEEE International Symposium on Circuits and Systems*, pp. 821-824 (1984).

[Siewi82] D. Siewiorek, *Computer Structures: Principles and Examples*, McGraw-Hill (1982).

[Smith85] S.P Smith and H.C. Tong, "Design of a Fast Inner Product Processor," *Proceedings 7th Symposium on Computer Arithmetic*, pp. 38-43 (1985).

[Speis80] J. Speiser, H. Whitehouse, and K. Bromley, "Signal Processing Applications for Systolic Arrays ," *14th Asilomar Conference on Circuits, Systems and Computers*, pp. 100-104 (1980).

[Speis83] J. Speiser and H. Whitehouse, "A Review of Signal Processing with Systolic Arrays," *SPIE Real Time Signal Processing VI* **431**, pp. 2-6 (1983).

[Sulli84] B. Sullivan and B. Lin, "Extrapolation of DiscreteTime Band-Limited Signals using SVD with Decimation," *Proceedings International Conference on Acoustics, Speech and Signal Proceesing* **ICASSP-84**, pp. 31.3.1-31.3.4 (1984).

[Swart78] E. Swartzlander, B. Gilbert, and I. Reed, "Inner Product Computers," *IEEE Transactions on Computers* **C-27**(1), pp. 21-31 (January, 1978).

[Syman83] J.J. Symanski, "Implementation of Matrix Operations on the Two-Dimensional Systolic Array Testbed," *SPIE Real Time Signal Processing VI* **341** , pp. 136-142 (1983).

[Tyree85] V. Tyree, *Personal communication*, MOSIS Project, ISI/USC; UCLA-Computer Science Dept. (1985).

[Uya84] M. Uya, K. Kaneko, and J. Yasui, "A CMOS Floating Point Multiplier," *Proceedings International Solid State Circuits Conference* **ISSCC-84**, pp. 90-91 (1984).

[Walth71] J. Walther, "A Unified Algorithm for Elementary Functions," *1971 Proceedings Joint Spring Computer Conference* **AFIPS/38**, pp. 379-385 (1971).

[Welte84] F. Welten and J. Lohstroh, "A 25/50 MHz Dual-Mode Parallel Multiplier/Accumulator," *IEEE International Solid State Circuits Conference* **ISSCC-84**, pp. 86-87 (February 1984).

[Woo84] B. Woo, L. Lion, and F. Ware, *A High-Speed 32 Bit Floating-Point Chip Set for Digital Signal Processing*, 1984.

[Zhou84] Y. Zhou, "SVD, Singular Vectors and the Discrete Prolate Spheroidal Sequences," *Proceedings International Conference on Acoustics, Speech and Signal Proceesing* **ICASSP-84**, pp. 37.6.1-37.6.4 (1984).