FAST COMPUTATION ALGORITHM FOR ROBOT DYNAMICS AND ITS IMPLEMENTATION

Han-Pang Huang Dyi-Rong Duh
Robotics Laboratory
Department of Mechanical Engineering
National Taiwan University
Taipei, Taiwan 10764, China

This paper proposes an architecture of parallel processing to increase the computational speed of the matrix/vector operation for robot dynamics. The system is formed by nine processor elements with X-bus and Y-bus. This array processor is able to achieve zero data transfer time by using data sharing on the bus. The multiplication of 3×3 rotation matrices requires only three multiplications and two additions. The vector inner products and vector additions can be performed in one multiplication and two additions (or one addition), respectively. It will be shown that the proposed architecture is more economic and more efficient than other existing approaches. Furthermore, the above architecture has been implemented on three IBM-PC boards by using Intel 8088 CPU.

1. Introduction

The robot dynamic equation requires a large amount of elementary operations, such as additions, subtractions and multiplications. In particular, the complexity of the robot dynamics computation is due to many vector and matrix operations. In general, these operations are inevitable and cannot be efficiently computed by a general-purpose computer with a uni-processor. In order to achieve fast computation, two approaches can be adopted: (1) use of efficient algorithm for the robot dynamics, such as Newton-Euler [1,2]; (2) use of the high-speed computer. The second approach can be achieved by either improving the manufacturing technology of computer chips so that the processor can operates at high clock rate, or using multiple processors, or using both. This paper aims to develop a parallel processing architecture for the robot dynamics based on the efficient Newton-Euler formulation.

The parallel processing technique can be fallen into pipelined architecture and multiple processors. Lathrop [8], Lee and Chang [9] proposed a systolic piplined architecture for the computation of the robot dynamics. They are conceptually implemented on VLSI devices. In their approaches, although only 15 multiplications and 38 additions of the initial delay time and 1 multiplication and 2 additions of the pipelined time are required for a six degree-of-freedom robot, a lot of processors, e.g. 530, and delay buffers should be used. Thus, it is very expansive and impratical for current technology. In addition, their architectures are unique for a specific robot and make them less flexible. Also, the architecture proposed by Lathrop [8] and Lee [9] does not consider the time delay due to data transfer and the computation for the trigonometric functions. Alternatively, Kazanzides [7] proposed a SIERA architecture in terms of multiple processors. Basically, SIERA is a trade-off between the loosely-coupled architecture and tightly-coupled architecture. It also takes into account the flexibility of applications. However, the number of required processors and the number of links are still considerably large.

In order to overcome the above drawbacks, this paper develops a parallel processor to increase the computation speed of the matrix and vector operations with limited amount of processor elements and great flexibility. It will be shown that the multiplication of two n by n matrices can be computed in O(n) time (e.g., the multiplication of two 3 by 3 rotation matrices only requires 3 multiplications and 3 additions) and without local data transfer delay. The proposed architecture is implemented on three PC boards by using nine Intel 8088 CPU to demonstrate its feasibility. The robot dynamics of the PUMA 560 are also performed on this architecture.

2. Hardware Architecture

Robot dynamics characterize the motion behavior of the robot. In general, the robot dynamics can be classified into direct dynamics and inverse dynamics. Given generalized torque/ force find the robot motion, i.e. the position, velocity and acceleration, is referred to as direct dynamics. On the other hand, given the robot motion find the generalized torque/force is referred to as inverse dynamics. We will focus on the inverse dynamics problem by using the efficient Newton-Euler algorithm.

The efficient Newton-Euler algorithm includes trigonometric, matrix and vector operations. The major computational load arises from matrix and vector operations. These operations are: addition of vectors, scalar product of vectors, cross product of vectors, multiplication of vector and matrix, addition of matrices, multiplication of matrices, and multiplication of vector and scalar. In particular, the computation time of matrix multiplication is considerable. For a uni-processor, i.e., SISD (Single Instruction Single Data) architecture, the multiplication of two n×n matrices is described as

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj} \quad \text{for } 0 \le i \le n-1, \quad 0 \le j \le n-1$$

or equivalently in pseudo code as [4]

```
for i←0 to n-1 do
for j←0 to n-1 do begin
c_ij ← 0; //initialization//
for k←0 to n-1 do
c_ij ← c_ij + a_ik b_kj; //scalar operation//
end;
```

The complexity of this method is $O(n^3)$, but only one processor is required. Alternatively, an array processor can be used to perform the same operation with less computational complexity. An array processor is composed of several similar processors to perform operations simultaneously. It is also called a SIMD (Single Instruction and Multiple Data) computer. The structure of an array processor includes a control unit (CU), a control unit memory (CUM), processor elements (PEs), processor element memories (PEMs), and communication network. CU is dedicated to process program. It fetches instructions from CUM and passes data to PEMs for PEs. The data transfer between PEMs is through communication network. Theoretically, an array processor made of n PEs can achieve n times computational speed in contrast to a uni-processor. However, the n times upper bound is usually restricted by data transfer time.

Consider the above matrix multiplication example. The SIMD-LA (Linear array) structure [3,4] is of computational complexity $O(n^2)$ and n processors are used. The total data transfer is 2n(n-1). The SIMD-MC (Mesh-Connected SIMD model, 2 dimensions) structure [4] requires n^2 processors and $2n^2$ links. The total data transfer is up to $n^2 + n - 2$. But, its complexity is only O(n). The SIMD-CC (Cube-Connected SIMD model) structure [3,4] requires n^3 processors with complexity of $O(log_2\ n)$. The multiplication is performed only once. However, it requires $5log_2 n$ data transfer, $4log_2 n$ bit functions, $4log_2 n$ bit-complement functions, and $\frac{3}{2}n^3 log_2 n$ links. It is clear that these array processor structures can greatly reduce the computational complexity at the expense of increasing data transfer and processor links.

In order to design an efficient array processor, the compromise between computational complexity, the number of processors, the data transfer and the processor links should be taken into account. We propose a new array processor structure, as shown in Fig.1. The architecture looks like a two dimensional array processor; however, there are no communication networks and common memories in the structure. All processor elements transfer their data through X buses and Y buses. Data on each X bus or Y bus can be simultaneously shared by n processor elements; but only one processor among these n processors can have outgoing data. It will be shown that the multiplication of two n by n matrices can be computed

in O(n) time (e.g., the multiplication of two 3 by 3 rotation matrices only requires 3 multiplications and 3 additions) and without local data transfer delay. Since the matrices and vectors considered in the robot dynamics are 3×3 or 3×1, n is equal to 3 in the following development. The proposed structure only requires n^2 processors. Each processor element can be a 32 bit (or other) processor. Its structure is shown in Fig.2. AR register is used to store operand A (matrix, vector or scalar); BR register is used to store operand B; and CR register is used to store the result. CMR is the command register. It stores the executable commands. Since each PE has its own command register, each PE can perform different operations simultaneously. In addition, each PE has a floating-point adder and a floating-point multiplier. Thus, the addition and multiplication can be concurrently processed.

The floating-point adder in each PE is shown in Fig.3. The operation of the adder is as follows: (1) Compare the size of two exponents; the smaller exponent is shifted right so that the two exponents are equal. (2) Add the fraction parts of the two numbers; the resultant exponent is set to the exponent obtained from step 1. (3) Shift the resultant exponent until the MSB is equal to 1; add the number of shifted bits (positive for right shift and negative for left shift) to the exponent.

The floating-point multiplier in each PE is given in Fig.4. The operation of the multiplier is as follows: (1) Add the exponent parts and subtract the fraction parts of the two numbers. (2) Shift the resultant fraction part until the MSB is equal to 1. (3) Add the number of shifted bits (positive for right shift and negative for left shift) to the resultant exponent.

The proposed array processor can be implemented on a single chip by using current semiconductor technology. Although the systolic pipelined structure proposed by Lathrop [8] and Lee [9] could be implemented by using advanced semiconductor technology in the future, it suffers from less flexibility. For example, the structure for solving the inverse kinematics of the PUMA arm [9] is applicable only for PUMA arm. Whereas our structure can be applied to different robots.

3. Algorithms

The Newton-Euler formulation of the robot dynamics mainly involves the operations of addition of vectors, scalar product of vectors, cross product of vectors, multiplication of vector and matrix, addition of matrices, multiplication of matrices, and multiplication of vector and scalar. We use the proposed array processor to perform these operations. The algorithm of each operation is described in the following.

Matrix Multiplication

The multiplication of two n×n matrices, A and B, is still an n×n matrix C. Namely,

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix} \times \begin{bmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,n-1} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n-1,0} & b_{n-1,1} & \cdots & b_{n-1,n-1} \end{bmatrix} = \begin{bmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,n-1} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,n-1} \\ \vdots & \vdots & & \vdots \\ c_{n-1,0} & c_{n-1,1} & \cdots & c_{n-1,n-1} \end{bmatrix}$$

The memory allocation of this operation is given in Fig.5. All matrix elements are equally placed in the local memory of each processor. Note that the memory allocation is exactly the same as the element location of the matrices. The elements of matrices A and B are stored in registers AR and BR, respectively. The result is stored in register CR.

The algorithm of the matrix multiplication is as follows.

par for i←0 to n-1 do
par for j←0 to n-1 do
$c_{ij} \leftarrow 0$;
for k←0 to n-1 do
par for i←0 to n-1 do
par for j←0 to n-1 do
$c_{ij} \leftarrow c_{ij} + a_{ik} b_{kj}$;

where par denotes parallel processing. The procedures of the parallel processing are shown in Table 1. The data appeared on X bus and Y bus are given in Table 2. At the kth operation, the data on the X_i bus, a_{ik}, can be simultaneously used by processors $PE_{i,0}, \cdots, PE_{i,n-1}$; and the data on the Y_j bus, b_{kj}, can be simultaneously used by processors $PE_{0,j}, \cdots, PE_{n-1,j}$. Thus, the multiplication of two n×n matrices exists no time delay for data transfer. That is why we call it "zero time data transfer." Obviously, the computational complexity of the proposed structure is O(n). The structure requires n^2 processors and 2n buses. Since the adder and multiplier in each processor element can be processed in a pipelined way, only n+1 flops computational procedures are required, as shown in Table 3. For n=3, the matrix multiplication only takes 3 multiplications and 3 additions. If the pipeline is used, only 4 operations are required.

Vector Addition

The memory allocation for vector addition is given in Fig.6. The algorithm is as follows.

par for j←0 to 2 do
$c_j \leftarrow a_j + b_j$;

The array processor can perform addition for three different sets of vectors simultaneously but only one addition is required.

Scalar Product of Vectors

The memory allocation for scalar product of vectors is given in Fig.7. The algorithm is as follows.

par for j←0 to 2 do
$c_j \leftarrow a_j \times b_j$;
for j←1 to 2 do
$c_0 \leftarrow c_0 + c_j$;

The array processor can perform scalar product for three different sets of vectors simultaneously but only one multiplications and one addition are required.

Cross Product of Vectors

The memory allocation for cross product of vectors is given in Fig.8. Although the allocation is not in order, the required time for the data transfer from host to the register of the matrix processor does not increase. The algorithm is as follows.

par begin
$c_{00} \leftarrow a_1 \times b_2$;
$c_{01} \leftarrow a_2 \times b_1$;
$c_{10} \leftarrow a_2 \times b_0$;
$c_{11} \leftarrow a_0 \times b_2$;
$c_{20} \leftarrow a_0 \times b_1$;
$c_{21} \leftarrow a_1 \times b_0$;
end;
par for i←0 to 2 do
$c_{i0} \leftarrow c_{i0} - ci1$;

After execution, the values of c_0, c_1, c_2 are stored in c_{00}, c_{01}, c_{02}, respectively. One multiplication and one subtraction (it can be achieved by changing the sign bit) are required for this operation.

Vector Multiplying Matrix

The memory allocation for vector multiplying matrix is given in Fig.9. The algorithm is as follows.

par for i←0 to 2 do
par for j←0 to 2 do
$c_{ij} \leftarrow a_i \times b_{ij}$;
for i←1 to 2 do
par for j←0 to 2 do
$c_{0j} \leftarrow c_{0j} + c_{ij}$;

After execution, the values of c_0, c_1, c_2 are stored in c_{00}, c_{01}, c_{02}, respectively. One multiplication and two additions are required for this operation.

Matrix Multiplying Vector

The memory allocation for matrix multiplying vector is given in Fig.10. The algorithm is as follows.

par for i←0 to 2 do
par for j←0 to 2 do
$c_{ij} \leftarrow a_{ij} \times b_j$;
for j←1 to 2 do
par for i←0 to 2 do
$c_{i0} \leftarrow c_{i0} + c_{ij}$;

After execution, the values of c_0, c_1, c_2 are stored in c_{00}, c_{01}, c_{02}, respectively. One multiplication and two additions are required for this operation.

Matrix Addition

The memory allocation for matrix addition is given in Fig.11. The algorithm is as follows.

par for i←0 to 2 do
par for j←0 to 2 do
$c_{ij} \leftarrow a_{ij} + b_{ij}$;

Only one addition is required for this operation.

Vector Multiplying Scalar

The memory allocation for matrix addition is given in Fig.12. The algorithm is as follows.

par for i←0 to 2 do
$c_i \leftarrow a_i + b$;

Only one multiplication is required for this operation.

4. Implementation And Example

In order to demonstrate the feasibility of the proposed array processor, the Intel 8088 CPU is chosen. The reason for choosing 8088 is that 8088 is very cheap and it requires the least external components. Although the data bus of 8088 is only 8 bits, it can process 16-bit data internally. For the robot application, nine 8088 CPUs are used to construct the array processor. The array processor is implemented on three PC boards and each board contains three 8088/8MHz CPUs. The array processor can be plugged into IBM compatible PC. The relevant interconnection circuitry and driver are also developed. All programs are written in C language. The picture of the array processor is shown in Fig.13. Table 4 is the comparison of computation time between with and without the proposed array processor. The values in the table are the average values of 10000 runs. Also the values include the time for data read/write.

The array processor is then applied to the computation of the robot dynamics. For a six degree-of-freedom PUMA robot, the Newton-Euler formulation requires only 306 additions and 162 multiplications for the proposed array processor in contrast to 1314 additions and 1242 multiplications for a uni-processor. The speed improvement ratio is equal to (1314+1242)/(306+162)=5.46. If we use it to compute the forward kinematics of the PUMA robot, the speed improvement ratio is equal to (234+216)/(36+24)=7.5. Clearly, if all the operations are matrix operations, then the array processor shows the best performance.

5. Conclusion

This paper has proposed an array processor so that the multiplication of two 3×3 matrices only requires 3 multiplications and 2 additions. The array processor is useful for the robot applications. Although the implementation is based on Intel 8088 processors, the 16-bit or 32-bit processor (e.g. TMS 32020, Intel 80386, 80486) can be used to construct the array processor. The performance of the array processor is expected to considerably increase.

References

[1]. K.S. Fu, R.C. Gonzalez and C.S.G. Lee, *Robotics Control, Sensing, Vision, and Intellignece,* New York: McGraw-Hill, 1987.

[2]. J.M. Hollerbach, "A Recursive Lagrangian Formulation of Manipulator Dynamics and a Comparative Study of Dynamics Formulation Complexity, "IEEE, Intl. Conf. on Robotics & Automation, pp.730-736, 1980.

[3]. E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms,* Computer Science Press Inc., 1978.

[4]. K. Hwang and F.A. Briggs, *Computer Architecture and Parallel Processing,* New York: McGraw-Hill, 1984.

[5]. Intel, *iAPX 286 Hardware Reference Manual,* 1983.

[6]. Intel, *iAPX 286 Programmer's Reference Manual Including The iAPX 286 Numeric Supplement,* 1985.

[7]. P. Kazanzides, H. Wasti and W. A. Wolovich, "A Multiprocessor System for Real Time Robotic Control: Design and Application," IEEE, Intl. Conf. on Robotics & Automation, pp. 1903-1908, 1987.

[8]. R.H. Lathrop, "Parallelism in Manipulator Dynamics," IEEE Intl. Conf. on Robotics & Automation, pp.772-778, 1985.

[9]. C.S.G. Lee and P.R. Chang, "Efficient Parallel Algorithm for Robot Inverse Dynamics Computation," IEEE Trans. Sys., Man, and Cybern., Vol. 16, No.4, pp.532-542, 1986.

[10]. C.S.G. Lee, C.L. Chen and E.S.H. Hou, "Efficient Scheduling Algorithms for Robot Inverse Dynamics Computation on A Multiprocessor System," IEEE Intl. Conf. on Robotics & Automation, pp. 1146-1151, 1988.

[11]. J.Y.S. Luh, M.W. Walker and R. P.C. Paul,"On-Line Computational Scheme for Mechanical Manipulators," J. of Dynamic Systems, Measurement, and Control, Vol. 102, pp. 120-127, June 1980.

[12]. Motorola Inc., *WC68020 32-Bit Microprocessor User's Manual,* 2nd edition, Prentice-Hall, 1985.

[13]. M.J. Quinn, *Designing Efficient Algorithms For Parallel Computers,* Ch.1-3, Ch.6, McGraw-Hill, 1987.

[14]. Texas Instruments Inc., *TMS32020 User's Guide,* 1986.

[15]. M.W. Walker and D.E. Orin, "Efficient Dynamic Computer Simulation of Robotic Mechanisms," J. of Dynamic Systems, Measurement, and Control, Vol.104, pp.141-147, September 1982.

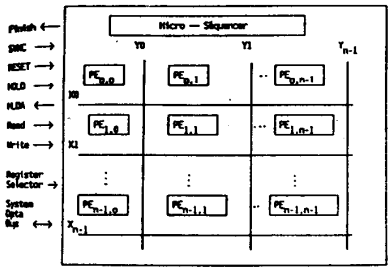

Fig.1 Structure of the proposed array processor

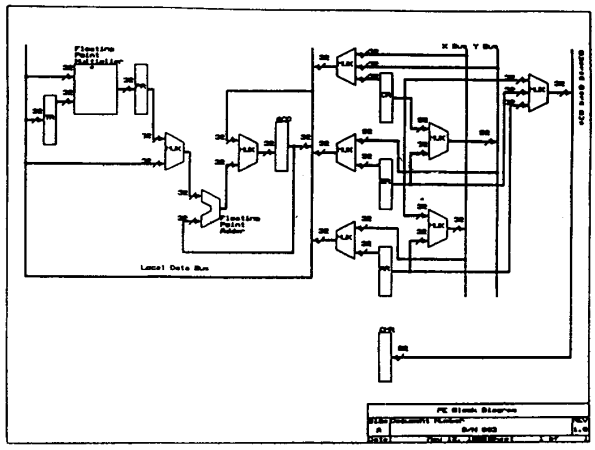

Fig.2 Internal structure of PE

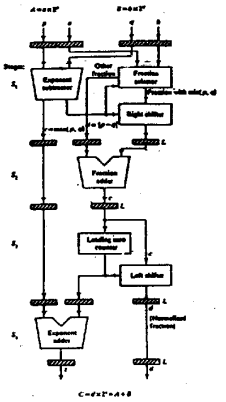

Fig.3 Floating-point adder [4]

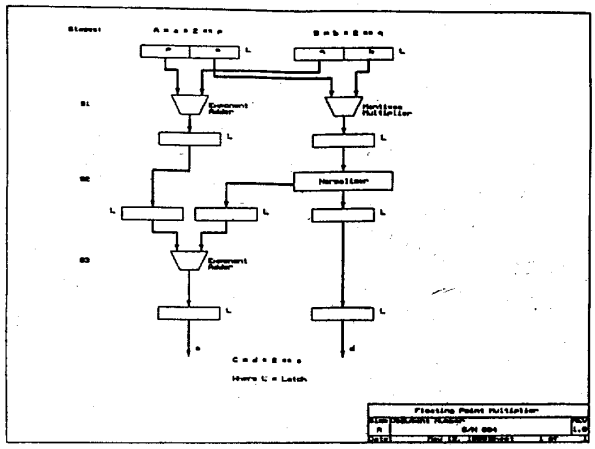

Fig.4 Floating-point multiplier

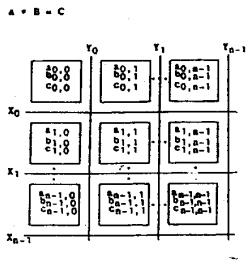

Fig.5 Memory allocation for matrix multiplication

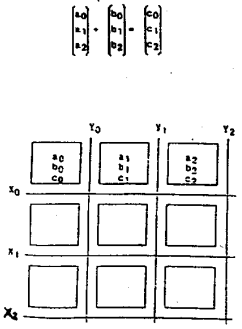

Fig.6 Memory allocation for vector addition

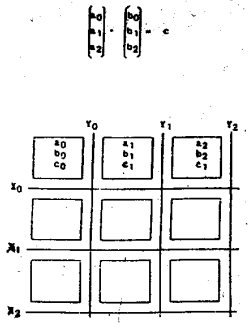

Fig.7 Memory allocation for scalar product

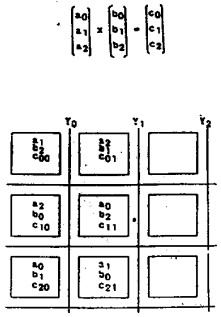

Fig.8 Memory allocation for cross product

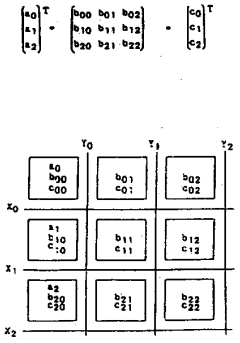

Fig.9 Memory allocation for vector multiplying matrix

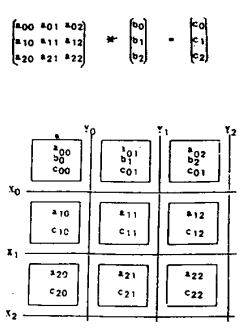

Fig.10 Memory allocation for matrix multiplying vector

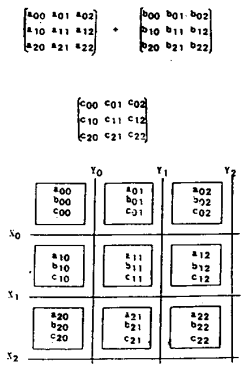

Fig.11 Memory allocation for matrix addition

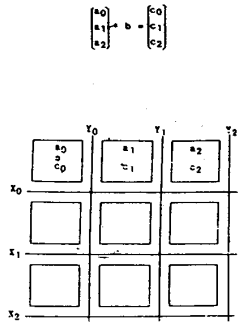

Fig.12 Memory allocation for vector multiplying scalar

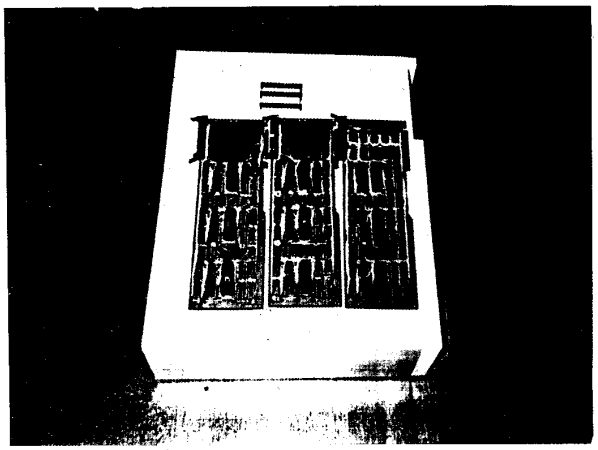

Fig.13 Picture for the proposed array processor

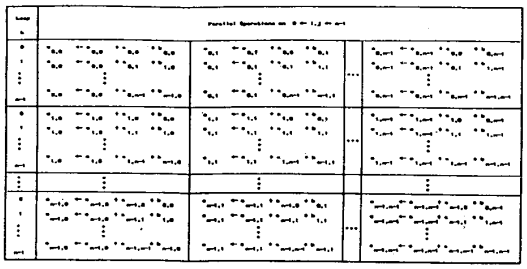

Table 1. Procedures for parallel processing

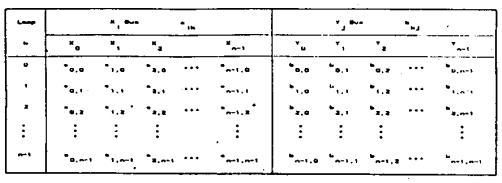

Table 2. Data on X-bus and Y-bus

```
loop
 k
 0  PR ← a(i,0) * b(0,j)
 1  PR ← a(i,1) * b(1,j)      c(i,j) ← c(i,j) + PR
 2  PR ← a(i,2) * b(2,j)      c(i,j) ← c(i,j) + PR
                :
n-1 PR ← a(i,n-1) * b(n-1,j)  c(i,j) ← c(i,j) + PR
 n                            c(i,j) ← c(i,j) + PR
```

Table 3. Pipelined processing

	80286/6MHz		80286/10MHz	
opr	without	with	without	with
Vec+Vec	1.511	1.477	0.873	1.169
Vec*Vec	2.417	1.955	1.406	1.577
VecxVec	3.262	2.021	1.890	1.555
Vec*Mat	7.184	2.268	4.158	1.746
Mat*Vec	7.552	2.279	4.372	1.763
Mat+Mat	4.502	2.213	2.609	1.599
Mat*Mat	22.733	3.109	13.160	2.395
Vec*Sca	1.581	0.950	0.917	0.654
Sca*Sca	5.034	1.741	2.916	1.104

Table 4. Comparison between with and without array processor (ms)