Computational Structures for Robotic Computations

C.S.G. Lee and P.R. Chang
Purdue University
West Lafayette, IN 47907

Abstract

~~This paper addresses~~ The computational problem of inverse kinematics and inverse dynamics of robot manipulators by taking advantage of parallelism and pipelining architectures. For the computation of inverse kinematic position solution, a maximum pipelined CORDIC architecture has been designed based on a functional decomposition of the closed-form joint equations. For the inverse dynamics computation, an efficient p-fold parallel algorithm to overcome the recurrence problem of the Newton-Euler equations of motion to achieve the time lower bound of $O(\lceil \log_2 n \rceil)$ has also been developed.

1. Introduction

Robot manipulators are highly nonlinear systems, and their motion control is usually specified in terms of the path traveled by the manipulator hand in Cartesian coordinates. To perform a simple kinematic path control, the controller is required to compute accurately the joint angles of the manipulator along the desired Cartesian path at an adequate and acceptable rate. To perform a dynamic path-tracking control, one must repeatedly compute the required generalized forces, from an appropriate manipulator dynamics model, using the measured data of displacements and velocities of all the joints, and the accelerations computed from some justifiable formulae or approximations, to drive all the joint motors. In order to achieve fast convergence of the control algorithm, a sampling rate of no less than 60 *Hz* is preferable because the mechanical resonant frequency of most industrial manipulators is around 5–10 *Hz*. The above kinematic and dynamic path control reveals a basic characteristic and common problem in robotic manipulator control — intensive computations with a high level of data dependency. Despite their impressive speed, conventional general-purpose uniprocessor computers can not efficiently handle the kinematic and dynamic path control computations at the required computation rate because their architectures limit them to a mostly serial approach to computation, and therefore limit their usefulness for robotic computational problems. This paper addresses these intensive robotic computational problems by taking advantage of parallelism and pipelining architectures.

Considering that most industrial robots have simple geometry, the kinematic path control requires the computation of the solution of joint angles which can be obtained by various techniques. The inverse transform technique [1] yields a set of explicit, closed-form, non-iterative joint angle equations which involve multiplications, additions, square root, and transcendental function operations. Based on an actual implementation on a multiprocessor system† [2,3] having a circuit to synchronize the CPUs and software scheduling for computing the joint solution, the best reported computation time was 3.8 *ms* for a six-link manipulator versus 20 *ms* running on a uniprocessor system. If we use a CORDIC (COordinate Rotation DIgital Computer) architecture [4], the computation time reduces to 40 μs, a speed-up factor of 500††!

For the dynamic path-tracking control, there are a number of ways to compute the generalized forces/torques applied to the joint motors [5], among which the computation of joint torques from the Newton-Euler (NE) equations of motion is the most efficient and has been shown to possess the time lower bound of $O(n)$ running in uniprocessor computers [6,7], where n is the number of degrees of freedom (DOF) of the manipulator. Based on the study of Luh, Walker, and Paul [7], it requires $(150n - 48)$ multiplications and $(131n - 48)$ additions per trajectory set point for a manipulator with rotary joints. It is unlikely that further substantial improvements in computational efficiency can be achieved, since the recursive NE equations are efficiently computing the minimum information needed to compute the generalized forces/torques: angular velocity, liner and angular acceleration, and joint forces and torques. For a Stanford robot arm (a total of 308 multiplications and 254 additions is required to compute the joint torques [8]), this amounts to 25 *ms* processing time on a uniprocessor system and 5.69 *ms* running on an experimental multiprocessor system with 7 processors [9]. If we use the parallel algorithm with 6 processors as proposed in this paper, this reduces the computation from 852 multiplications and 738 additions running on a uniprocessor to 197 multiplications and 183 additions for a PUMA

This work was supported in part by the National Science Foundation Engineering Research Center Grant CDR8500022. Any opinions, findings, and conclusions or recommendations expressed in this article are those of the author and do not necessarily reflect the views of the funding agency.

† The multiprocessor system consists of a MC6809 CPU and seven Z80 CPUs. Each Z80 is accompanied by two 9511 APUs, local memory, and I/O interfaces.

†† A speed-up factor is defined as the ratio of the computational time of a task running on a uniprocessor system to the computational time of the same task running on the proposed architecture (i.e., $20\ ms / 40\mu s = 500$).

robot (due to different kinematic structure of PUMA and Stanford robots, direct comparison on processing time is invalid) [10].

This paper discusses the development of a maximum pipelined CORDIC architecture for the computation of inverse kinematic position solution to achieve the pipelined time of 40 μs and an efficient p-fold parallel algorithm to achieve the time lower bound of computing the joint torques. The CORDIC architecture was designed based on a functional decomposition of the closed-form joint equations. Delay buffers are necessary to balance the pipelined CORDIC architecture to achieve maximum pipelining. The buffer assignment problem is solved by the integer linear programming technique. The efficient p-fold parallel algorithm can be best described as consisting of p-parallel blocks with pipelined elements within each parallel block to achieve the time lower bound of $O(\lceil \log_2 n \rceil)$ of computing the joint torques based on the Newton-Euler equations of motion, where n is the number of degrees of freedom of the manipulator. The algorithm can be implemented with a group of microprocessors without complex intercommunication among processors and bussing of data. A modified inverse shuffle scheme is suggested for connecting the processors together with efficient intercommunications.

2. Inverse Kinematic Position Computation

The general kinematic problem of a 6-DOF robot arm concerns the problem of finding the generalized coordinates $\mathbf{q} = [q_1, q_2, \cdots, q_6]^T$, together with the vector of their generalized velocities and the vector of their generalized accelerations in the n-dimensional space such that the characteristics of the motion of the free end, the hand, coincide with the pre-specified Cartesian trajectory. This inverse problem has earned considerable attention because of its importance in relating the Cartesian trajectory of the hand to the corresponding joint-variable trajectory of the manipulator. This paper focuses only on the inverse kinematic position solution.

In solving the inverse kinematic position problem, we are always interested in obtaining a closed-form solution (i.e. an algebraic equation relating the given manipulator hand position and orientation to one of the unknown joint displacements), which yields all the possible solutions in a fixed computation time. Fortunately, most industrial robots have simple geometry and exhibit closed-form joint solution. Utilizing the inverse transform technique [1], the joint angle equations of a six-link manipulator with simple geometry reveal the computation of a large set of elementary operations: real number multiplications, additions, divisions, square roots, trigonometric functions and their inverse. However, these elementary operations, in general, cannot be efficiently computed in general-purpose uniprocessor computers. In order to obtain a fixed computation time for the joint angle solution, time-consuming transcendental functions (sine, cosine, and arc tangent) are implemented as table look-up at the expense of the solution accuracy. The CORDIC algorithms [11-14] are the natural candidates for efficiently computing these elementary operations. They represent an efficient way to compute a variety of functions related to coordinate transformations with iterative procedures involving only shift-and-add operations at each step. Thus, cordic processing elements are extremely simple and quite compact to realize [14] and the interconnection of CORDIC processors to exploit the great potential of pipelining and multiprocessing provides a cost-effective solution for computing the inverse kinematic position solution.

2.1. CORDIC Algorithms and Processors

In conventional uniprocessor computers, computation of elementary functions such as square roots, sine, cosine, hyperbolic sine and cosine and their inverse consumes a considerable amount of effort than multiplication operation. These elementary functions can be efficiently computed by the cordic algorithms which can be described by a single set of iterative equations parametrized by a quantity m $(= -1, 0, 1)$ which determines the type of rotations. To establish connections between CORDIC and rotation-based algorithms, let the angle of rotation θ be decomposed into a sum of n sub-angles $\{d_i;\ i = 0, n-1\}$

$$\theta = \sum_{i=0}^{n-1} u_i d_i \tag{1}$$

where the sign u_i (± 1) is chosen based on the direction of rotation. Similarly, the plane rotation matrix $R(\theta)$

$$R(\theta) = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \tag{2.a}$$

or hyperbolic rotation matrix $R(\theta)$

$$R(\theta) = \begin{bmatrix} \cosh\theta & \sinh\theta \\ -\sinh\theta & \cosh\theta \end{bmatrix} \tag{2.b}$$

can also be decomposed into a product of sub-angle rotation matrices

$$R(\theta) = \prod_{i=0}^{n-1} R(d_i) \tag{3}$$

Thus, a single rotation of θ angle can be replaced by n smaller rotations with d_i angle each. In the cordic algorithms, d_i is chosen such that

$$d_i = \begin{cases} \tan^{-1}(2^{-s(i)}) & , m = 1 \text{ (circular)} \\ 2^{-s(i)} & , m = 0 \text{ (linear)} \\ \tanh^{-1}(2^{-s(i)}) & , m = -1 \text{ (hyperbolic)} \end{cases} \tag{4}$$

where m determines the type of rotations and $\{s(i); i = 0, n-1\}$ is a non-decreasing integer sequence. Using d_i from (4), $\mathbf{R}(d_i)$ can be written as

$$\mathbf{R}(d_i) = p_i \begin{bmatrix} 1 & -m u_i 2^{-s(i)} \\ u_i 2^{-s(i)} & 1 \end{bmatrix} \tag{5}$$

where p_i is a scaling factor and equals to $(1 + m\,2^{-2s(i)})^{-1/2}$. Let $\mathbf{R}^N(\theta)$ and $\mathbf{R}^N(d_i)$ be the normalised form of $R(\theta)$ and $\mathbf{R}(d_i)$, respectively, then from (3), we have

$$\mathbf{R}(\theta) = \prod_{i=0}^{n-1} p_i \prod_{i=0}^{n-1} \mathbf{R}^N(d_i) = k_m \prod_{i=0}^{n-1} \mathbf{R}^N(d_i) = k_m\, \mathbf{R}^N(\theta) \tag{6.a}$$

where

$$k_m = \prod_{i=0}^{n-1} p_i = \prod_{i=0}^{n-1} (1 + m\,2^{-2s(i)})^{-1/2} \quad ; \quad \mathbf{R}^N(\theta) = \prod_{i=0}^{n-1} \begin{bmatrix} 1 & -m u_i 2^{-s(i)} \\ u_i\, 2^{-s(i)} & 1 \end{bmatrix} \tag{6.b}$$

Usually, k_m is a machine constant and $k_m \simeq 0.6072$ (for $m = 1$) or 1.00 (for $m = 0$) or 1.205 (for $m = -1$), when $n \geq 10$ [12, 15]. The normalised rotation matrix of (6.b) indicates that each small rotation can be realised with one simple shift-and-add operation. Hence, the computation of a trigonometric function can be accomplished with n shift-and-add operations, which is comparable to conventional multiplications. This makes a CORDIC ALU a very appealing alternative to the traditional ALU for implementing the elementary functions. In general, the normalised CORDIC algorithm can be written as follows:

FOR $i = 0, 1, \cdots, n-1$, DO

$$\begin{bmatrix} x_{i+1}^N \\ y_{i+1}^N \end{bmatrix} = \begin{bmatrix} 1 & -m\, u_i\, 2^{-s(i)} \\ u_i\, 2^{-s(i)} & 1 \end{bmatrix} \begin{bmatrix} x_i^N \\ y_i^N \end{bmatrix} \tag{7.a}$$

$$z_{i+1}^N = z_i^N + u_i\, d_i \tag{7.b}$$

where $x_0^N = x_0$, $y_0^N = y_0$, m determines the type of rotation, d_i is chosen as in (4), and the auxiliary variable z_i^N is introduced to accumulate the rotation after each iteration. And the corresponding "unnormalised" CORDIC algorithm is described as:

FOR $i = 0, 1, \cdots, n-1$, DO

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = p_i \begin{bmatrix} 1 & -m\, u_i\, 2^{-s(i)} \\ u_i\, 2^{-s(i)} & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{8.a}$$

$$z_{i+1} = z_i + u_i d_i \tag{8.b}$$

where $x_0 = x_0$ and $y_0 = y_0$. It can be shown that z_i and z_i^N will accumulate the angle of the total rotation and have the same value after n iterations. However, the end results of (x_n, y_n) from the iterations of (8.a) and the end results of (x_n^N, y_n^N) from the iterations of (7.a) are related according to

$$x_n = k_m\, x_n^N \quad ; \quad y_n = k_m\, y_n^N \tag{9}$$

Consequently, one may evaluate x_n^N and y_n^N by using only the shift-and-add operations in (8.a), then realise x_n and y_n by other simple methods such as ROM look-up tables and regular combinatorial logic, etc. Fortunately, it is possible to find a simple way to normalise the scale factor k_m using the same shift-and-add hardware [14, 15]. The supplementary operations that are used to force the scale factor k_m to converge toward unity can be either performed after all the operations of (7.a) are terminated, that is,

$$x_{i+1}^c = (1 + \gamma_i 2^{-i})\, x_i^c \quad ; \quad y_{i+1}^c = (1 + \gamma_i\, 2^{-i}) y_i^c \tag{10}$$

where $x_0^c = x_n^N$, $y_0^c = y_n^N$, and $0 \leq i \leq n-1$, or interleaved with the operations of (7.a), that is,

$$x_i^c = (1 + \gamma_i\, 2^{-i})\, x_i^N \quad ; \quad y_i^c = (1 + \gamma_i\, 2^{-i})\, y_i^N \tag{11}$$

where $0 \leq i \leq n-1$. The parameter γ_i in (10) or (11) may be -1 or 0 or 1 depending on the value of i and the type of rotations (i.e. m) [14, 15].

Haviland et al. [14] realised the CORDIC algorithm on a CMOS chip and showed that the processing time of the CORDIC chip is 40 μs. They also suggested $n = 13$ as the minimum cycle time of a two-byte (24-bit) fixed point operation. However, in practice, they used $n = 24$. For a conventional CORDIC module, it requires 5 shift-and-add modules to compute one CORDIC iteration and one normalization iteration in parallel (that is, 3 shift-and-add modules for (7.a) and (7.b), and 2 shift-and-add modules for (11)). The desired output can be obtained in 24 iterations ($n = 24$). Thus, 24 iterations of 5 shift-and-add modules computing in parallel will be enough to realise CORDIC algorithms. This indicates that the CORDIC processing time is no slower than the time for a serial multiplier computing two 24-bit operands.

Figure 1 summarises the elementary functions that can be obtained from the CORDIC processor when m is set to -1, 0, or 1. In this figure, a CORDIC processor is depicted as a box with three inputs x_0, y_0, z_0 which are the initial values of x_i, y_i, and z_i in (8), as well as three outputs that correspond to the final values of x_n, y_n, and z_n in (8). Thus, the outputs x_n, y_n, z_n are the desired elementary functions, when m is appropriately set to -1, 0, or 1. These CORDIC processors will be configured and connected, based on a functional decomposition of the joint angle equations, to arrive at an efficient architecture for the computation of inverse kinematic position solution.

2.2. CORDIC Architecture for Inverse Kinematic Position Computation

The design philosophy is to examine the inverse kinematic position solution for its computational flow and data dependencies in order to functionally decompose the computations into a set of CORDIC computational modules (CCMs) with an objective that each CCM will be realizable by a CORDIC processor.* This functional decomposition is not unique and can be best represented by a directed task graph with a finite set of nodes denoting CCMs and a corresponding finite set of communication edges denoting operands movements. An examination of the inverse kinematic position equations in Appendix A shows a limited amount of parallelism with a large amount of sequentialism in the flow of computations and data dependencies. This serial nature of the computational flow lends itself to a pipelined CORDIC processors implementation [18]. The decomposition of the inverse kinematic position equations is done by looking at the equations which can be computed as elementary functions by CORDIC processors as listed in Figure 1.

The task of computing the inverse kinematic position of a PUMA robot arm, based on the equations in Appendix A, can be decomposed into 25 subtasks in which each subtask corresponds to a CCM and can be realised by a CORDIC processor. For example, θ_1 can be computed by the following 3 subtasks:

Subtask 1: $z1_a = r = (p_x^2 + p_y^2)^{1/2}$

$$z1_a = \tan^{-1}\left(\frac{p_y}{p_x}\right) = -\tan^{-1}\left(\frac{-p_y}{p_x}\right)$$

CORDIC Processor: $\text{CIRC2} = \begin{cases} x_0 = p_x \\ y_0 = -p_y \\ z_0 = 0 \end{cases}$

Subtask 2: $z2_a = \sqrt{r^2 - d_3^2} = \sqrt{z1_a^2 - d_3^2}$

CORDIC Processor: $\text{HYPE2} = \begin{cases} x_0 = z1_a \\ y_0 = d_3 \\ z_0 = 0 \end{cases}$

Subtask 3: $z3_a = \theta_1 = z1_a - \tan^{-1}\left(\frac{d_3}{z2_a}\right)$

CORDIC Processor: $\text{CIRC2} = \begin{cases} x_0 = z2_a \\ y_0 = d_3 \\ z_0 = z1_a \end{cases}$

The computational flow of these 25 tasks together with the input data can be represented by the directed acyclic data dependency graph (ADDG) with switching nodes and parallel edges as shown in Figure 2 and the details about the decomposition of the inverse kinematic position solution into CCMs can be found in [4]. In Figure 2, each computational node, indicated by a circle, represents a CORDIC computational module, and each switching node, indicated by a dot, performs no computations but just switches data to various CCMs. The operands or data move along the edges. A major bottleneck in achieving maximum throughput or maximum pipelining in Figure 2 is the different arrival time of the input data at the multi-input CCMs (e.g. nodes T18 and T22 in Figure 2). The computations of multi-input CCMs can not be initiated until all the input data have arrived. This different arrival time of input data lengthens the pipelined time. Thus, the ADDG is said to be *unbalanced* and fails to achieve maximum pipelining. Several techniques [17]-[19] have been suggested to remedy this data arrival problem by inserting appropriate number of buffers (or delays) in some of the paths from the input node z to the multi-input CCMs to "balance" the ADDG and achieve maximum pipelining. This buffer assignment problem for balancing the ADDG can be reduced to an integer linear optimization problem. Detailed formulation of the optimal buffer assignment problem as an integer linear optimization problem can be found in [4]. After solving the buffer assignment problem, realization of the balanced ADDG results in a maximum pipelined CORDIC architecture. For a PUMA robot arm, the architecture consists of 25 CORDIC processors and 141 buffer stages with 4 tapped-delay-line-buffers [4]. The initial time delay of the pipeline is equal to 18 stage latency (or 720 μs), where the stage latency of a CORDIC processor is assumed to be 40 μs [14]. The pipelined time of the CORDIC architecture equals to one stage latency or 40 μs. The realization of the maximum pipelined CORDIC architecture is shown in Figure 3.

3. Inverse Dynamics Computation

The general inverse dynamic problem for an n-link manipulator can be stated as follow: given the joint positions and velocities $\{q_j(t)\ ,\ \dot{q}_j(t)\ \}_{j=1}^{n}$ which describe the state of the manipulator at time t, together with the joint accelerations $\{\ddot{q}_j(t)\}_{j=1}^{n}$ which are desired at the point, solve the dynamic equations of motion for the joint torques $\{\tau_j(t)\}_{j=1}^{n}$ as follows:

$$\tau(t) = f\ (q(t),\ \dot{q}(t),\ \ddot{q}(t)) \tag{12}$$

where $\tau(t) = [\tau_1,\ \tau_2,\ldots,\tau_n]^T$, $q(t) = [q_1,\ q_2,\ldots,q_n]^T$, $\dot{q}(t) = [\dot{q}_1,\ \dot{q}_2,\ldots,\dot{q}_n]^T$, $\ddot{q}(t) = [\ddot{q}_1,\ \ddot{q}_2,\ldots,\ddot{q}_n]^T$, $f(\cdot)$ is an nx1 nonlinear vector function and superscript T denotes transpose operation on matrices and vectors.

At present, much attention has been focused on the computational issues of the inverse dynamics based on the Newton-Euler (NE) formulation, resulting in various multiprocessor-based control systems [8,21-24]. The recursive structure of the NE equations of motion is obviously well suited to standard single-instruction-stream and single-data-stream (SISD) computers. It is, however, not an efficient parallel processing for new single-instruction-stream and multiple-data-stream (SIMD) computers that are capable of performing many simultaneous operations. Our approach in designing efficient algorithms for computing the robot inverse dynamics is to look at the computational complexity of the problem first. In particular, we need to know what is the limitation of speeding up the computation of the inverse dynamics while running on p processors, where $1 \le p \le n$. That is, we would like to establish a time lower bound for the inverse dynamics computation problem so that several efficient computational schemes can be compared and contrasted. Then efficient algorithms achieving the time lower bound can be designed for the computation of the inverse dynamics. The following notations and lemma will be used to derive the time lower bound of the inverse dynamic problem.

Notations:

(1) *Linear arithmetic expression* is any well-formed string composed of four arithmetic operators $(+,-,\times,/)$ or, for convenience, two operators +(or −), × (or /), left and right parentheses, and atoms, which are constants or variables. We denote a linear arithmetic expression E of m distinct atoms by $E<m>$, e.g. $E<4> : a + b - c\ /\ d$.

(2) $T_p\ [f_1(\cdot)\ ,\ f_2(\cdot)\ ,\ \ldots\ ,\ f_n(\cdot)]$ = Minimum computing time needed to evaluate $[\ f_1(\cdot)\ ,\ f_2(\cdot)\ ,\ \ldots\ ,\ f_n(\cdot)\ ]$ using p processors.

Lemma 1: The time lower bound of $T_p\ [E<m>]$ [25]. The shortest parallel time to evaluate a linear arithmetic expression $E<m>$ using p processors is bounded below by and equal to $O(\lceil m/p \rceil + \lceil \log_2 p \rceil)$, that is,

$$T_p\ [E<m>] \ge O(\lceil m/p \rceil + \lceil \log_2 p \rceil)$$

Theorem 1: The shortest parallel time to evaluate the joint torques $\{\tau_j(t)\}_{j=1}^{n}$ in equation (12) using p processors is bounded below by $O(k_1\ \lceil n/p \rceil + k_2\ \lceil \log_2 p \rceil)$, where k_1 and k_2 are specified constants, that is,

$$T_p\ [\tau_1,\tau_2,\ldots,\tau_n] \ge O(k_1 \lceil n/p \rceil + k_2 \lceil \log_2 p \rceil) \tag{13}$$

The proof of Theorem 1 can be found in [10]. Two extreme cases follow from Theorem 1:

(a) If $p = 1$, then the shortest computing time $T_p\ [\tau_1,\tau_2,\ldots,\tau_n]$ is not lower than $O(n)$. Thus, the NE formulation is the most efficient algorithm of evaluating the inverse dynamics running in uniprocessor computers.

(b) If $p = n$, then the shortest parallel computing time $T_p\ [\tau_1,\tau_2,\ldots,\tau_n]$ is not lower than $O(\lceil \log_2 n \rceil)$.

Theorem 1 indicates that an efficient algorithm running on p processors may not achieve the same time order as $O(k_1 \lceil n/p \rceil + k_2 \lceil \log_2 p \rceil)$. However, if a parallel algorithm possesses the time lower bound, then it must be the most efficient algorithm of evaluating the inverse dynamics. Theorem 1 also indicates that, although NE formulation is very efficient for computing the inverse dynamics, a better solution is to find an efficient parallel algorithm, running on p processors, that possesses a time order of $O(k_1 \lceil n/p \rceil + k_2 \lceil \log_2 p \rceil)$. A parallel algorithm running on an SIMD machine and achieving the time lower bound is discussed next.

The recursive NE equations of motion are very efficient in evaluating the inverse dynamics whether they are formulated in the base coordinate frame [6] or the link coordinate frames [7]. The clear advantage of referencing both the dynamics and kinematics to the link coordinates is to obviate a great deal of coordinate transformations and to allow the link inertia tensor to be fixed in each link coordinate frame, which results in a much faster computation in a uniprocessor computer. However, the recursive structure of this formulation is in an inhomogeneous linear recursive form, e.g. $\omega_i = a_i \omega_{i-1} + b_i$, where $a_i = {}^{i}R_{i-1}$ (a 3×3 rotation matrix) and $b_i = {}^{i}R_{i-1}\ z_0 \dot{q}_i$ †, which requires more calculations and arrangements for parallel processing than the homogeneous linear recursive form. On the other hand, the NE formulation in the base coordinates can be re-arranged and transformed into a homogeneous linear recurrence form, e.g. $\omega_i = \omega_{i-1} + \dot{q}_i\ s_{i-1}$, which is more suitable for parallel processing on an SIMD computer, yielding a much shorter computing time.

Once the NE equations of motion are formulated in the base coordinates in a homogeneous linear recurrence form, then a parallel algorithm, called recursive doubling [16,17,26-27], can be utilised to compute the kinematics in the forward equations and the dynamics (or torques) in the backward equations [5]. The homogeneous linear recurrence problem of size $(n+1)$ can be described as follows: given $x(0) = a(0) \neq$ identity and $a(i)$, $1 \le i \le n$, find $x(1)$, $x(2)$, ... ,$x(n)$ by an algorithm running on an SIMD computer of n processors, where

† a_i and b_i are used here as variables.

$$z(i) = z(i-1) * a(i) \tag{14}$$

and "$*$" denotes an associative operator, which can be a matrix product or addition, a vector dot product or addition, etc. If $a(0)$ is equal to an identity, then the linear recurrence of size $(n+1)$ can be reduced to the case of size n by taking a shift operation as follows: $a(0) \leftarrow a(1)$, $a(1) \leftarrow a(2)$, ..., $a(n-1) \leftarrow a(n)$, where "$\leftarrow$" denotes a replacing operation. The recursive doubling algorithm basically solves the homogeneous linear recurrence form in (14) by splitting the computations of a serial associative operation. For an arbitrary $z(n)$, the generalization of the idea allows $(n+1)/2$ parallel operations at the first splitting, $(n+1)/4$ at the second, ..., and $(n+1)/2^k$ at the k th until $\lceil \log_2(n+1) \rceil$ splittings, then $z(n)$ is computed with one final operation. Similarly, it can be shown that $z(i)$ can be computed in $\lceil \log_2(i+1) \rceil$ splittings, where $1 \leq i \leq n$. In other words, with the recursive doubling algorithm, $z(1), z(2), \ldots, z(n)$ can be computed concurrently no later than the time step $\lceil \log_2(n+1) \rceil$.

The procedure in computing the inverse dynamics from the NE equations of motion formulated in the base coordinates is to re-arrange the kinematic equations and the dynamic equations in a homogeneous linear recursive form, and the following input parameters relating the formulation must be given or evaluated in advance:

(a) The 3×3 rotation matrices ${}^{i-1}\mathbf{R}_i$, $i = 1,2,\ldots,n$, which indicates the orientation of link i coordinates referenced to link $(i-1)$ coordinates, need to be evaluated in advance.

(b) ${}^i\mathbf{p}_i^*$ denotes the origin of link i coordinate frame from the origin of link $(i-1)$ coordinate frame expressed with respect to link i coordinates; ${}^i\mathbf{s}_i$ denotes the location of the center of mass of link i from the origin of link i coordinate frame expressed with respect to link i coordinates; and ${}^i\mathbf{J}_i$ denotes the inertia matrix of link i about its center of mass expressed with respect to link i coordinates, must be given in advance. Note that ${}^i\mathbf{p}_i^*$, ${}^i\mathbf{s}_i$, and ${}^i\mathbf{J}_i$ are constants when referred to their own link coordinates.

(c) λ_i is a joint indicator which specifies link i is rotational or translational as follow:

$$\lambda_i = \begin{cases} 0 & , \text{ if link } i \text{ is rotational} \\ 1 & , \text{ if link } i \text{ is translational} \end{cases}$$

(d) Let $\omega_0 = \dot{\omega}_0 = 0$, $\ddot{\mathbf{p}}_0 = [g_x, g_y, g_z]^T$ and $|\mathbf{g}| = 9.80621 \text{m/s}^2$. If external force $\mathbf{f}_e$ and external moment $\mathbf{n}_e$ are exerting on link n, then $\mathbf{f}_{n+1} = \mathbf{f}_e$, $\mathbf{n}_{n+1} = \mathbf{n}_e$; otherwise, $\mathbf{f}_{n+1} = \mathbf{n}_{n+1} = 0$.

The procedure of evaluating the NE equations of motion as a linear recurrence problem is then given below (note b_i are used here as variables):

STEP 1. Compute the rotation matrix ${}^0\mathbf{R}_i$ with respect to the base coordinates for $i = 1,\ldots,n$

$$ {}^0\mathbf{R}_i = {}^0\mathbf{R}_{i-1} * {}^{i-1}\mathbf{R}_i \tag{15}$$

STEP 2. Compute $\mathbf{p}_i^*$, $\mathbf{s}_i$, and $\mathbf{s}_i$ for $i = 1,2,\ldots,n$

$$\mathbf{s}_i = {}^0\mathbf{R}_i * \mathbf{s}_0 \quad , \quad \mathbf{s}_0 = [0,0,1]^T \quad ; \quad \mathbf{p}_i^* = {}^0\mathbf{R}_i \, {}^i\mathbf{p}_i^* \quad ; \quad \mathbf{s}_i = {}^0\mathbf{R}_i \, {}^i\mathbf{s}_i \tag{16}$$

The evaluation of $\mathbf{s}_i$ only involves taking the third column of ${}^0\mathbf{R}_i$.

STEP 3. Compute

$$b_i = \mathbf{s}_{i-1} \dot{q}_i \, (1 - \lambda_i) \tag{17}$$

and

$$\omega_i = \omega_{i-1} + b_i \tag{18}$$

STEP 4. Compute

$$b_i = (\mathbf{s}_{i-1} \ddot{q}_i + \omega_{i-1} \times \mathbf{s}_{i-1} \dot{q}_i) \, (1 - \lambda_i) \tag{19}$$

and

$$\dot{\omega}_i = \dot{\omega}_{i-1} + b_i \tag{20}$$

STEP 5. Compute

$$b_i = \dot{\omega}_i \times \mathbf{p}_i^* + \omega_i \times (\omega_i \times \mathbf{p}_i^*) + (\mathbf{s}_{i-1} \ddot{q}_i + 2 \, \omega_i \times (\mathbf{s}_{i-1} \dot{q}_i)) \, \lambda_i \tag{21}$$

and

$$\ddot{\mathbf{p}}_i = \ddot{\mathbf{p}}_{i-1} + b_i \tag{22}$$

STEP 6. Compute

$$\ddot{\mathbf{r}}_i = \dot{\omega}_i \times \mathbf{s}_i + \omega_i \times (\omega_i \times \mathbf{s}_i) + \ddot{\mathbf{p}}_i \tag{23}$$

STEP 7. Compute

$$\mathbf{F}_i = m_i \ddot{\mathbf{r}}_i \tag{24}$$

STEP 8. Compute

$$\mathbf{N}_i = \mathbf{J}_i \, \dot{\omega}_i + \omega_i \times (\mathbf{J}_i \, \omega_i) \tag{25}$$

For the sake of saving the calculations of evaluating $J_i = {}^0R_i\ {}^iJ_i\ {}^iR_0$, which Luh et al. [7] showed that the computation was quite complicated, (25) is modified to

$$ {}^i\omega_i = {}^iR_0\,\omega_i = ({}^0R_i)^T\,\omega_i \quad ; \quad {}^i\dot{\omega}_i = {}^iR_0\,\dot{\omega}_i = ({}^0R_i)^T\,\dot{\omega}_i \tag{26}$$

$$ {}^iN_i = {}^iJ_i\ {}^i\dot{\omega}_i + {}^i\omega_i \times ({}^iJ_i\ {}^i\omega_i) \quad ; \quad N_i = {}^0R_i\ {}^iN_i \tag{27}$$

***STEP* 9.** Compute .

$$ f_i = f_{i+1} + F_i \tag{28}$$

***STEP* 10.** Compute

$$ b_i = N_i + (p_i^{\cdot\cdot} + s_i) \times F_i + p_i^{\cdot} \times f_{i+1} \tag{29}$$

and

$$ n_i = n_{i+1} + b_i \tag{30}$$

***STEP* 11.** Compute

$$ \tau_i = \begin{cases} (n_i)^T s_{i-1} & , \text{ if } \lambda_i = 0 \\ (f_i)^T s_{i-1} & , \text{ if } \lambda_i = 1 \end{cases} \tag{31}$$

Previously undefined terms, expressed in the base coordinates, are given as follows: m_i is the mass of link i, ω_i is the angular velocity of link i, $\dot{\omega}_i$ is the angular acceleration of link i, $\ddot{p}_i$ is the linear acceleration of link i, $\ddot{r}_i$ is the linear acceleration of the center of mass of link i, F_i is the total force exerted on link i at the center of mass, N_i is the total moment exerted on link i at the center of mass, f_i is the force exerted on link i by link $i-1$, n_i is the moment exerted on link i by link $i-1$, τ_i is the torque exerted by the actuator at joint i if rotational, force if translational, q_i is the joint variable of joint i (θ_i if rotational and d_i if translational).

Equation (15) shows that the evaluation of ${}^0R_i$ is a simple recursive matrix product form. Equations (18), (20), (22), (28), and (30) only involve simple recursive vector addition form. The other equations in the NE equations can be computed parallelly. Thus, the evaluation of the total computational complexity of the parallel algorithm for a PUMA robot arm can be derived as follows:

(a) The parallel evaluation of (15) using recursive doubling indicates $(27\lceil\log_2 n\rceil - 19)$ scalar multiplications and $(18\lceil\log_2 n\rceil - 14)$ scalar additions.

(b) Equations (18), (20), (22), (28), and (30) all have the same recursive vector addition form, the total parallel evaluation of these equations requires $(6\lceil\log_2 n\rceil + 9\lceil\log_2(n+1)\rceil)$ scalar additions.

(c) The parallel evaluation of the other equations in the NE formulation, e.g. $p_i^{\cdot} = {}^0R_i\ {}^ip_i^{\cdot}$, $s_i = {}^0R_i\ {}^is_i$, F_i, N_i, τ_i, and all the b_i of (17), (19), (21), and (29) can be calculated by simple parallel computations, yielding a constant computation of 135 scalar multiplications and 98 scalar additions.

Combining the results of (a), (b), and (c), the total computational complexity of the parallel algorithm applied to a PUMA robot arm is $(27\lceil\log_2 n\rceil + 116)$ scalar multiplications and $(24\lceil\log_2 n\rceil + 9\lceil\log_2(n+1)\rceil + 84)$ scalar additions. Note that it is of time order $O(\lceil\log_2 n\rceil)$ because we are using $p = n$ processors. If further reduction on the coefficients of $(\lceil\log_2 n\rceil)$ is desirable, this can be accomplished by using matrix multiplier chips. This would reduce the coefficients 27 and 18 in evaluating (15) as discussed in (a). If $n = 6$, then the complexity of the parallel NE algorithm is 197 multiplies (mults) and 183 additions (adds) as compared with the complexity of the NE algorithm running on a uniprocessor [7]: 852 mults and 738 adds. Moreover, even if n becomes large, say $n = 12$ (for redundant robots), then the number of multiplications and additions increases only by 27 and 33, respectively. Thus, we have shown that considerable savings in computation time can be achieved from embedding the inverse dynamic computation in a parallel algorithm, which has a time complexity of logarithmic in the number of joints, $O(\lceil\log_2 n\rceil)$.

3.1. An Efficient Parallel Algorithm With p-Fold Parallelism

Last section showed that the bottleneck of parallel computation of the inverse dynamics depends on solving the homogeneous linear recurrence of the N-E formulation. If the restriction that one microprocessor "handles" one joint is relaxed, it is desirable to obtain an efficient parallel algorithm which can greatly improve the evaluation of the linear recurrence using p processors. A parallel algorithm of evaluating the inverse dynamics with a restricted number of p processors has been developed to achieve the time lower bound of $O(k_1\lceil n/p\rceil + k_2\lceil\log_2 p\rceil)$. The proposed p-fold parallel algorithm can be best described as consisting of p-parallel blocks with pipelined elements within each parallel block. The results from the computations in the p blocks form a new homogeneous linear recurrence of size p, which again can be computed using the recursive doubling algorithm. The parallel algorithm with p-fold parallelism (PFP) is summarised and presented as follows:

Algorithm PFP (p-fold Parallelism). This algorithm divides the computations into p-parallel blocks of computations. The jth processor computes the elements in the jth block serially. The results from the the p-parallel blocks form a new homogeneous linear recurrence of size p, which can be computed by the recursive doubling algorithm.

P1. [Initialization.] Given $a(0) \neq$ identity, let $s = \lceil (n+1)/p \rceil$, $m = ps$, and set

$$M(i) = \begin{cases} a(i-1) & , \ 1 \leq i \leq n+1 \\ 0 & , \ n+1 \leq i \leq m \end{cases} \tag{32}$$

where s indicates the block size (number of elements in a block).

P2. [Divide into p blocks.] Divide $M(i)$, $1 \leq i \leq m$, into p blocks as follows:
jth block $= (M((j-1)s+1), M((j-1)s+2), \ldots, M(js))$ and let $N(1,j) = M((j-1)s+1)$, $1 \leq j \leq p$
where $N(i,j)$ indicates the ith element in the jth block.

P3. [Compute $N(i,j)$ in a DO loop.] The jth processor serially computes $N(i,j)$ in the jth block as:
For $i = 2$ step 1 until s Do

$$N(i,j) = N(i-1,j) * M((j-1)s+i) \ , \ 1 \leq j \leq p \tag{33}$$

END

It is seen that $x(1), x(2), \ldots, x(s-1)$ has been evaluated in the DO loop as well as $N(i,1)$, $2 \leq i \leq s$. That is, $x(1) = N(2,1)$, $x(2) = N(3,1)$, $\ldots$, $x(s-1) = N(s,1)$.

P4. [Form a new homogeneous linear recurrence of size p.] Let $\gamma(j) = N(s,j)$ and $y(j) = x(js-1)$, for $1 \leq j \leq p$ and referring to (14), (32), and (33), we have

$$\begin{aligned} y(j) = x(js-1) &= x((j-1)s-1) * a((j-1)s) * a((j-1)s+1) * \ldots * a(js-1) \\ &= y(j-1) * M((j-1)s+1) * M((j-1)s+2) * \ldots * M(js) = y(j-1) * N(s,j) \\ &= y(j-1) * \gamma(j) \end{aligned} \tag{34}$$

Equation (34) is a new homogeneous linear recurrence of size p which can be parallelly evaluated by the algorithm FOHRA, running in time proportional to $O(\lceil \log_2 p \rceil)$ and yielding the results $x(s-1) = y(1)$, $x(2s-1) = y(2)$, $\_$, $x(ps-1) = y(p)$. (Note that if $(n+1)$ is divisible by p, then $x(n) = y(p)$; otherwise, $x(ps-1) = y(p) = 0$).

P5. [Compute intermediate $x(i)$ in equation (35).] Without loss of generality, assuming that $(n+1)$ is not divisible by p, then there are $n-p-s+3$ intermediate terms of $x(i)$ that need to be determined. They are:

$$x(js+i) \quad , \ 0 \leq i \leq s-2, \ 1 \leq j \leq p-2 \quad \text{and} \quad x((p-1)s+i) \quad , \ 0 \leq i \leq n-(p-1)s \tag{35}$$

Referring to (7), (33), (34), and (35), thereby giving

$$\begin{aligned} x(js+i) = x(js-1) * a(js) * a(js+1) * \ldots * a(js+i) &= y(j) * M(js+1) * M(js+2) * \ldots * M(js+i+1) \\ &= y(j) * N(i+1,j) \quad , \ 0 \leq i \leq s-2, i \leq j \leq p-2 \end{aligned} \tag{36}$$

and $x((p-1)s+i) = y(p-1) * N(i+1,p-1)$, $0 \leq i \leq n-(p-1)s$

where $N(i+1,j)$, $y(j)$ of (36) have been evaluated in steps P3 and P4, respectively. Equation (36) shows that if $(n-p-s+3)$ tasks are of the same evaluation, then the calculations of these equal tasks are suited to an SIMD computer of p processors. It is shown that parallel evaluation of (36) requires $\lceil (n-p-s+3)/p \rceil$ time steps (note that if $(n+1)$ is divisible by p, then $x(n)$ can be evaluated in step P4, yielding $(n-p-s+2)$ equal tasks in (36), thereby requiring $\lceil (n-p-s+2)/p \rceil$ time steps).

END PFP

It is seen that the total parallel computing time T_p of the homogeneous linear recurrence of size $(n+1)$ using p processors is:

$$T_p = \begin{cases} \lceil (n+1)/p \rceil + \lceil (n-p-s+3)/p \rceil + \lceil \log_2 p \rceil - 1, & \text{if } (n+1) \text{ is not divisible by } p. \\ \lceil (n+1)/p \rceil + \lceil (n-p-s+2)/p \rceil + \lceil \log_2 p \rceil - 1, & \text{if } (n+1) \text{ is divisible by } p. \end{cases}$$

Applying the above p-fold parallel algorithm to the N-E formulation for an n-link rotary manipulator, it is able to achieve the time lower bound of $O(k_1 \lceil n/p \rceil + k_2 \lceil \log_2 p \rceil)$.

The above n-fold parallel algorithm is suited to be run on an SIMD computer. A cascade structure can be used for connecting the PEs. An alternate structure is to position a network between the processors and memories. The interconnection pattern, called the "perfect shuffle [26, 27]," has the number of links between processors proportional to n. An attractive interconnection pattern, called "inverse perfect shuffle [26,27]," is suitable for the implementation of solving the homogeneous linear recurrence and can be obtained by reversing the arrows of the perfect shuffle. Details about this network connection for solving the homogeneous linear recurrence for computing the joint torques can be found in [10].

4. Conclusion

A maximum pipelined CORDIC architecture for computing the inverse kinematic position solution and an efficient parallel algorithm for computing the joint torques have been discussed. To achieve maximum throughput, delay buffers

are required to balance the pipeline. For a PUMA robot arm, the CORDIC architecture consists of 25 CORDIC processors and 141 buffer stages with 4 tapped-delay-line buffers. The initial time delay of the pipeline is equal to 720 μs and the pipelined time of the CORDIC architecture equals to one stage latency or 40 μs. The parallel algorithm for computing the joint torques has a time complexity of logarithmic in the number of joints. The interconnection networks for the processors have also been investigated to improve the utilization of communication and internal buffering between processors in an SIMD computer. Using the concepts of the proposed parallel algorithm, it would be possible to devise a VLSI chip capable of implementing the inverse dynamics computation at speed primarily bounded by the proposed parallel algorithm.

5. Appendix A: Inverse Kinematic Position Solution

The inverse kinematic position problem can be stated as: Given the position/orientation of the manipulator hand and the link/joint parameters, determine the joint angles so that the manipulator can be positioned as desired. That is, given

$$\mathbf{T} = \begin{bmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{n} & \mathbf{o} & \mathbf{a} & \mathbf{p} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

the joint angle equations are:

$$r = (p_x^2 + p_y^2)^{1/2} \tag{A-1}$$

$$\theta_1 = \tan^{-1}\left[\frac{p_y}{p_x}\right] - \tan^{-1}\left[\frac{d_3}{\pm\sqrt{r^2 - d_3^2}}\right] \tag{A-2}$$

$$f_{11p} = p_x C_1 + p_y S_1 \;;\; f_{11o} = o_x C_1 + o_y S_1 \;;\; f_{13p} = -p_x S_1 + p_y C_1 \;;\; f_{13o} = -o_x S_1 + o_y C_1 \tag{A-3}$$

$$f_{11a} = a_x C_1 + a_y S_1 \;;\; f_{13a} = -a_x S_1 + a_y C_1 \tag{A-4}$$

$$d = f_{11p}^2 + f_{12p}^2 - d_4^2 - a_3^2 - a_2^2 \;;\; e = 4a_2^2 a_3^2 + 4a_2^2 a_4^2 \tag{A-5}$$

$$\theta_3 = \tan^{-1}\left[\frac{a_3}{-d_4}\right] - \tan^{-1}\left[\frac{d}{\pm\sqrt{e - d^2}}\right] \tag{A-6}$$

$$\theta_{23} = \tan^{-1}\left[\frac{a_2(f_{11p}S_3 - p_z C_3) + d_4 f_{11p} - a_3 p_z}{a_2(f_{11p}C_3 + p_z S_3) + a_3 f_{11p} + d_4 p_z}\right] = -\tan^{-1}\left[\frac{(p_z C_3 - f_{11p}S_3) + (-\frac{d_4}{a_2})f_{11p} + (\frac{a_3}{a_2})p_z}{(p_z C_3 + f_{11p}S_3) + (\frac{a_3}{a_2})f_{11p} + (\frac{d_4}{a_2})p_z}\right] \tag{A-7}$$

$$\theta_2 = \theta_{23} - \theta_3 \tag{A-8}$$

$$\theta_4 = \tan^{-1}\left[\frac{-S_1 a_x + C_1 a_y}{C_{23}(C_1 a_x + S_1 a_y) - S_{23} a_z}\right] = \tan^{-1}\left[\frac{f_{13a}}{C_{23} f_{11a} - S_{23} a_z}\right] \tag{A-9}$$

$$\theta_5 = \tan^{-1}\left[\frac{C_4[C_{23}(C_1 a_x + S_1 a_y) - S_{23} a_z] + S_4[-S_1 a_x + C_1 a_y]}{S_{23}(C_1 a_x + S_1 a_y) + C_{23} a_z}\right] = \tan^{-1}\left[\frac{C_4(C_{23} f_{11a} - S_{23} a_z) + S_4 f_{13a}}{S_{23} f_{11a} + C_{23} a_z}\right] \tag{A-10}$$

$$\theta_6 = \tan^{-1}\left[\frac{-C_5[C_4(C_{23} f_{11o} - S_{23} o_z) + S_4 f_{13o}] + S_5(S_{23} f_{11o} + C_{23} o_z)}{-S_4(C_{23} f_{11o} - S_{23} o_z) + C_4 f_{13o}}\right] \tag{A-11}$$

where $(-\frac{d_4}{a_2})$, $(\frac{d_4}{a_2})$, $(\frac{a_3}{a_2})$ are constants, $C_i \equiv \cos\theta_i$, $S_i \equiv \sin\theta_i$, $C_{ij} \equiv \cos(\theta_i + \theta_j)$, and $S_{ij} \equiv \sin(\theta_i + \theta_j)$.

6. References

1. Paul, R.P., Shimano, B.E., and Mayer, G., "Kinematic Control Equations for Simple Manipulators," *IEEE Trans. Syst. Man, Cybern.*, Vol. SMC-11, No. 6, pp. 456-460, 1981.
2. Kametani, M. and Watanabe, T., "Hardware and Software of a Multiprocessor System Applied for Robot Control," *1984 Proc. of Industrial Electronic Conf.*, pp. 749-758.

3. Watanabe, T. et al., "Improvement of the Computing Time of Robot Manipulators Using a Multiprocessor," *Proc. of ASME Winter Annual Meeting*, Miami, Florida, 1985, pp. 13-22.

4. Lee, C.S.G. and Chang, P.R., "A Maximum Pipelined CORDIC Architecture for Inverse Kinematics Computation," Technical Report TR-EE-86-5, School of Electrical Engineering, Purdue University, January 1986.

5. Fu, K. S., Gonzales, R. C., and Lee, C.S.G., *Robotics: Control, Sensing, Vision, and Intelligence*, McGraw-Hill, September, 1986.

6. Orin, D.E., R.B. McGhee, M. Vukobratovic, and G. Hartoch, "Kinematic and Kinetic Analysis of Open-chain Linkages utilising Newton-Euler Methods," *Math. Biosc.*, Vol. 43, 1979, pp. 107-130.

7. Luh, J.Y.S., M.W. Walker, and R.P.C. Paul, "On-line Computational Scheme for Mechanical Manipulator," *Trans. of ASME, J. of Dynam. Syst., Meas. and Contrl.*, Vol. 102, pp. 69-76, June 1980.

8. Luh, J.Y.S. and C.S. Lin, "Scheduling of Parallel Computation for a Computer-controlled Mechanical Manipulator," *IEEE Trans. Syst. Man, and Cyber.*, Vol. SMC-12, No. 2, pp. 214-234, March/April 1982.

9. Kasahara, H. and Narita, S., "Parallel Processing of Robot Arm Control Computation on a Multiprocessor System," *IEEE J. of Robotics and Automation*, Vol. RA-1, No. 2, June 1985, pp 104-113.

10. Lee, C.S.G. and Chang, P.R., "Efficient Parallel Algorithm for Inverse Dynamics Computation," *EEE Trans. on Systems, Man, and Cybernetics, Vol. SMC-16, no. 4, July, 1986, pp. 532-542.*

11. Volder, J.E., "The CORDIC Trigonometric Computing Technique," *IRE Trans. Electronic Computers*, Vol. EC-8, No. 3, Sept. 1959, pp. 330-334.

12. Walther, J.S., "A Unified Algorithm for Elementary Functions," *AFIPS Conf. Proc.*, Vol. 38, 1971, pp. 379-385.

13. Ahmed, H. M., J. M. Delosme and M. Morf, "Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing," *IEEE Computer*, Vol. 15, No. 1, pp. 65-82, Jan. 1982.

14. Haviland, G. L. and A. A. Tuszynski, "A CORDIC Arithmetic Processor Chip," *IEEE Trans. Comput.*, Vol. C-29, No. 2, pp. 68-78, Feb. 1980.

15. Dewide, P. et al., "Parallel and Pipelined VLSI Implementation of Signal Processing Algorithms," in *VLSI and Modern Signal Processing*, S. Y. Kung, H. J. Whitehouse, T. Kailath, (eds.), Prentice-Hall, Inc., Englewood Cliffs, NJ, pp. 257-276.

16. Kogge, P.M., "Parallel Solution of Recurrence Problems," *IBM J. Res. Develop.*, Vol. 18, pp. 138-148, Mar. 1974.

17. Kogge, P.M. and Stone, H.S., "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE Trans. on Comput.*, Vol. C-22, pp. 789-793, Aug. 1973.

18. Kung, H.T. and Lam, M., "Wafer-Scale Integration and Two-level Pipelined Implementation of Systolic Arrays," *J. of Parallel and Distributed Computing*, Vol. 1, No. 1, Sept. 1984, pp. 32-63.

19. Dennis, J. B. and R. G. Gao, "Maximum Pipelining of Array Operations on Static Data Flow Machine," *Proc. of 1983 Int'l. Conf. on Parallel Processing*, pp. 331-334, Aug. 1983.

20. Leiserson, C.E. and Saxe, J. B., "Optimising Synchronous Systems," *J. VLSI and Computer Systems*, Vol. 1, 1983, pp. 41-68.

21. Lee, C.S.G., Mudge, T.N., and Turney, J.L., "Hierarchical Control Structure using Special Purpose Processors for the Control of Robot Arms," *Proc. 1982 Pattern Recognition and Image Processing Conf.*, Las Vegas, Nevada, June 14-17, 1982, pp 634-640.

22. Lathrop, L.H., "Parallelism in Manipulator Dynamics," M.I.T. Artificial Intelligence Tech. Rep. No. 754, Dec. 1983.

23. Nigam, R. and C. S. G. Lee, "A Multiprocessor-Based Controller for the Control of Mechanical Manipulators," *IEEE J. of Robotics and Automation*, Vol. 1, No. 4, Dec. 1985, pp. 173-182.

24. Orin, D.E., "Pipelined Approach to Inverse Plant Plus Jacobian Control of Robot manipulators," *Proc. 1984 IEEE Int'l Conf. on Robotics and Automation*, Atlanta, GA, pp. 169-175, March 1984.

25. Horowitz, E. and Sahni, S. *Fundamentals of Computer Algorithms*, Computer Science Press Inc., 1978, pp. 488-494.

26. Stone, H.S., *Introduction to Computer Architecture*, Science Research Associate Inc., 1975, pp. 319-373.

27. Stone, H.S., "Parallel Processing with Perfect Shuffle," *IEEE Trans. on Comput.*, Vol. C-20, pp. 153-161, Feb. 1971.

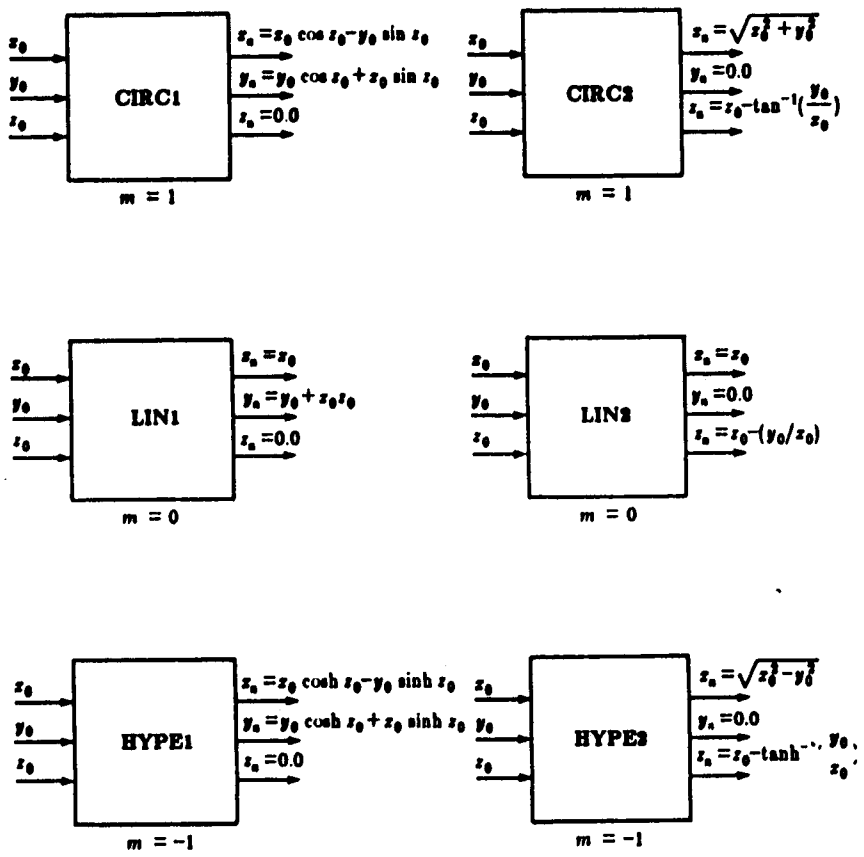


Figure 1 Elementary Functions Computed by CORDIC Processors

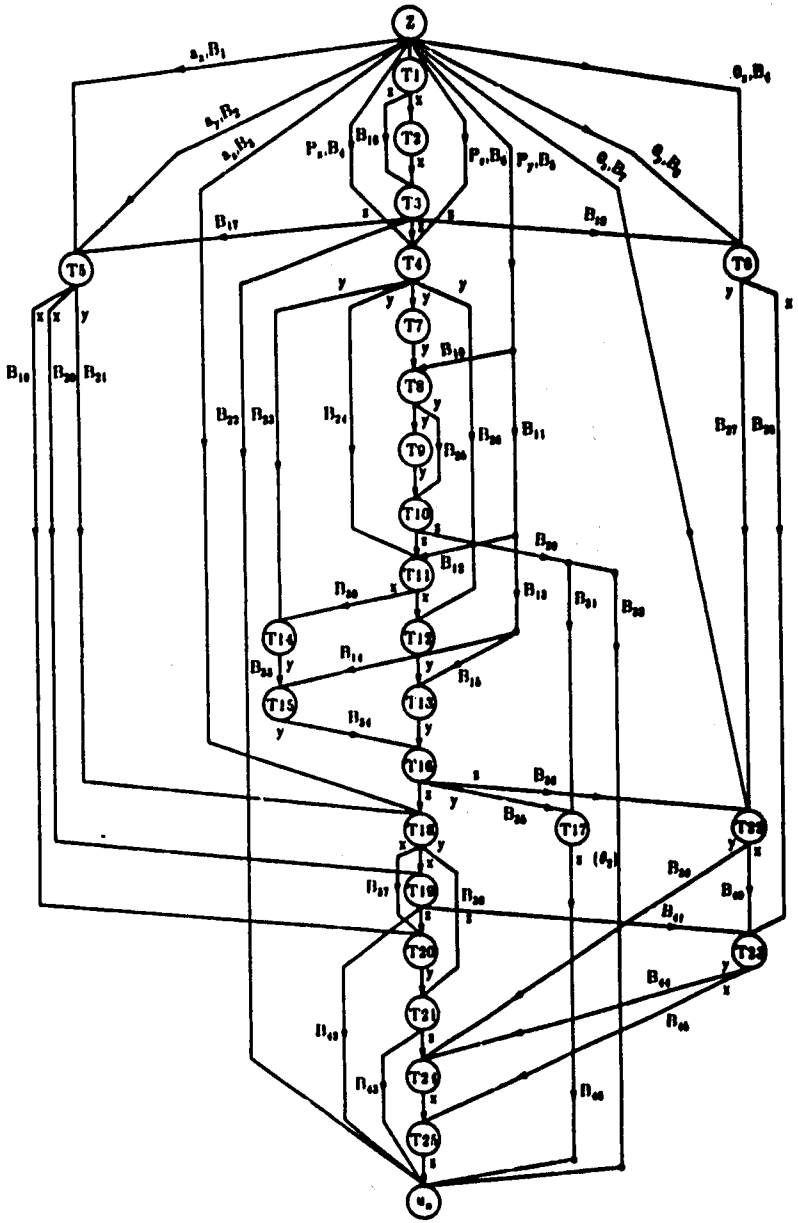
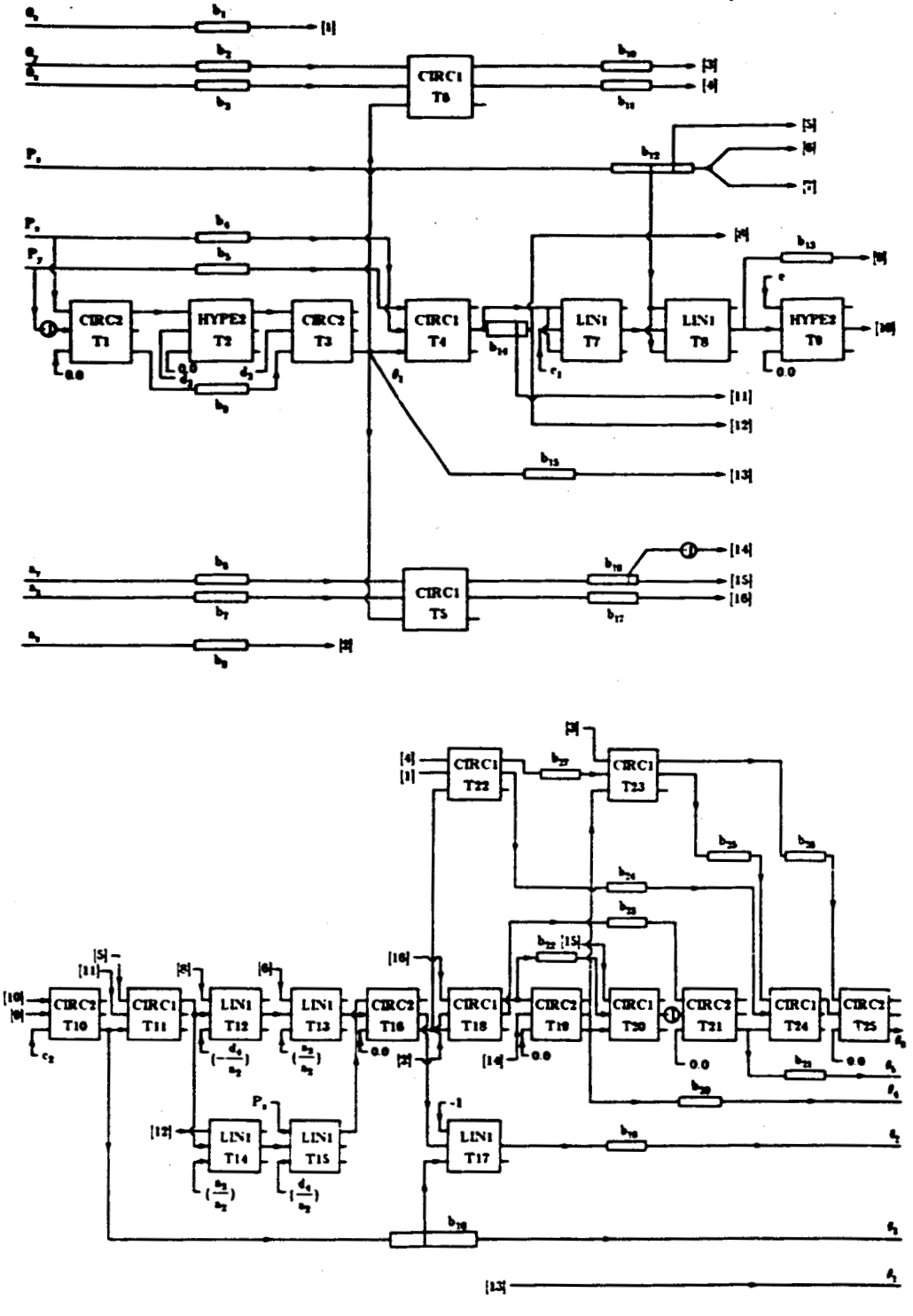


Figure 2 Task Graph for A PUMA Inverse Kinematic Position Solution



Buffers:

$|b_1| = 12, |b_2| = |b_3| = |b_4| = |b_5| = |b_6| = |b_7| = 3, |b_8| = 12,$
$|b_9| = 12, |b_{10}| = 10, |b_{11}| = 8, |b_{12}| : TDLB\text{–}(10,5,3,2,2),$
$|b_{13}| = 1, |b_{14}| : TDLB\text{–}(5,4,1,1), |b_{15}| = 15, |b_{16}| : TDLB\text{–}(10,9,1),$
$|b_{17}| = 8, |b_{18}| : TDLB\text{–}(10,4,6), |b_{19}| = 5, |b_{20}| = 4, |b_{21}| = 2,$
$|b_{22}| = 1, |b_{23}| = 2, |b_{24}| = 3, |b_{25}| = 1, |b_{26}| = 2, |b_{27}| = 1$

Constants:

$c_1 = -(d_4^2 + a_3^2 + a_2^2) \quad ; \; c_2 = \tan^{-1}(\frac{a_3}{-d_4})$

Figure 3. Realisation of Figure 2 with CORDIC Processors