Characterization of Robotics Parallel Algorithms and Mapping onto a Reconfigurable SIMD Machine

C. S. G. Lee and C. T. Lin

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

ABSTRACT

The kinematics, dynamics, Jacobian, and their corresponding inverse computations are six essential problems in the control of robot manipulators. Efficient parallel algorithms for these computations are discussed and analyzed. Their characteristics are identified and a scheme on the mapping of these algorithms to a reconfigurable parallel architecture is presented. Based on the characteristics including type of parallelism, degree of parallelism, uniformity of the operations, fundamental operations, data dependencies, and communication requirement, it is shown that most of the algorithms for robotic computations possess highly regular properties and some common structures, especially the linear recursive structure. Moreover, they are well-suited to be implemented on a single-instruction-stream multiple-data-stream (SIMD) computer with reconfigurable interconnection network. The model of a reconfigurable dual network SIMD machine with internal direct feedback is introduced. A systematic procedure to map these computations to the proposed machine is presented. A new scheduling problem for SIMD machines is investigated and a heuristic algorithm, called neighborhood scheduling, that reorders the processing sequence of subtasks to reduce the communication time is described. Mapping results of a benchmark algorithm are illustrated and discussed.

1. Introduction

Robot manipulators are highly nonlinear systems and their dynamic performance is directly dependent on the efficiency of the kinematic and dynamic models, the control schemes/algorithms, and the computer architecture for computing the control schemes. In general, robot manipulators are usually servoed in the joint-variable space while the objects to be manipulated are usually expressed in the world (or Cartesian) coordinate system. In order to control the position and orientation of the manipulator end-effector, the robot controller is required to compute, at a sufficient rate, such tasks as coordinate transformation between the joint-variable space and the Cartesian space, generalized forces/torques to drive the joint motors, the manipulator inertia matrix for model-based control schemes, and the Jacobian matrix which relates the joint velocity in the joint-variable space to the Cartesian space. These are the basic robotic computations for the control of robot manipulators. They are equivalent to the computations of kinematics, dynamics, Jacobian, and their corresponding inverses. These six basic robotics computations are required at various stages of robot arm control and computer simulation of robot motion, and reveal a basic characteristic and common problem in robot manipulator control — *intensive computations with a high level of data dependency*. They have become major computational bottlenecks in the control of robot manipulators. Despite their impressive speed, conventional general-purpose uniprocessor computers cannot efficiently handle the kinematics and dynamics computations at the required computation rate because their architectures limit them to a mostly serial approach to computation. Furthermore, less efficient, serial computational algorithms must be used to compute these robotics computations on a uniprocessor computer. Consequently, the quest for real-time robot arm control and motion simulation rests on the study and development of parallel algorithms of lower computational complexity with faster computational structures. The ultimate goal is to achieve an *order-of-magnitude* and/or an *order-of-complexity* improvement in computational efficiency in these robotics computations by taking advantage of parallelism, pipelining, and architectures.

A common feature of today's research on robotic computational problems is that a specific problem, mostly the inverse dynamics or the inverse kinematics, is studied at a time, and usually an algorithmically-specialized architecture or processor is developed for that particular algorithm. Obviously, this specialized architecture can make the most use of the parallel properties of the algorithm. However, most advanced robot control schemes always require to solve a combination of some or all of the six basic robotic computations. One solution for this problem is to wire these specialized architectures or processors together. This method is inflexible because the combination of these components is dedicated to a particular control scheme and cannot be used efficiently for another scheme. Another solution is to connect the architectures or processors to a bus as peripherals of a general-purpose computer. This is more flexible, but the bus becomes a bottleneck and time is wasted in data movements between different computational processes. Another possible solution is focussed on partitioning the original algorithm/task into a set of subtasks with precedence relationship and then developing efficient scheduling algorithms to map these

subtasks onto a general-purpose multiprocessor system. This solution is much more flexible because most computational algorithms can be represented by directed task graphs. However, this approach may result in ignoring some inherent parallelism in robotics algorithms.

In this paper, we shall address these robotic computational problems, and major effort is focussed on finding a scheme which provides the flexibility needed to solve robotic computational problems on the same architecture while maintaining high efficiency by taking into account the inherent parallelism of robotics algorithms. To exploit the inherent parallelism of these robotics algorithms, our approach is first to characterize the set of parallel robotic algorithms based on the six specified characteristics and features, including type of parallelism, degree of parallelism, uniformity of the operations, fundamental operations, data dependency, and communication requirement. Our analysis shows that machines operating in the single-instruction-stream multiple-data-stream (SIMD) mode are the most efficient and suitable for our robotic algorithms. By fully considering the common characteristics and inherent parallelism of the robotics algorithms, a prototype of a medium-grained, reconfigurable, dual-network, SIMD machine with internal direct feedback has been designed for the computation of these kinematic and dynamic computational tasks. A systematic mapping procedure has been developed for scheduling these robotic computational tasks onto the proposed SIMD machine. This procedure builds a task table which contains the subtask assignment from the original parallel algorithm. Then a simplified task table and an input table are produced through the notation simplification. These two tables are then used as inputs to the neighborhood scheduling algorithm which reorders the processing sequence of the subtasks into a rescheduled task table to reduce the communication time. Finally, the subtasks in this rescheduled task table are mapped onto the proposed SIMD machine and a control table which describes the control sequence in the machine is produced. A benchmark algorithm which contains the characteristics of the six basic robotic computations has been implemented on the proposed SIMD machine, and the mapping results are included for discussion.

2. Characteristics of Parallel Algorithms

A key factor to the design of a parallel architecture for a group of algorithms is the understanding of their architectural requirements, and this requires us to identify the characteristics of these algorithms. This identification is usually helpful because the algorithms from a given application area such as robotics often possess an identifiable structure. In order to examine the characteristics of the six basic robotics parallel algorithms, a set of features which have the greatest effects on the execution of parallel algorithms is defined for robotics application [1].

- **Type of parallelism.** Two levels of parallelism can be identified.
  - (a) *Job-level parallelism.* The original algorithm is reformulated to a parallel processable form. In this level, the variables carrying the same kind of information but with different indices (e.g., for different links or joints of a manipulator) are processed parallelly. Due to the nature of the robot's serial link structure, variables representing the same physical meaning are defined for each link such as joint velocities, joint accelerations, and joint torques. Usually, the same class of variables are produced through an identical computational procedure but with different set of data. This property is called uniformity of operations as defined below. So the job-level parallelism will often be amenable to the SIMD implementation and usually the required number of processors depends on the number of degrees of freedom of the manipulator (i.e., one processor for each joint).
  - (b) *Task-level parallelism.* The original algorithm is decomposed into multiple subtasks. While the computation within a subtask is serial, the number of subtasks that can be processed concurrently is maximized by using some scheduling techniques. Obviously, this implies multiple-instruction-stream multiple-data-stream (MIMD) operations. Furthermore, for this level of parallelism, a subtask usually performs the same computation for different set of data, and hence the operation can be pipelined. An advantage of this task-level parallelism is that the required number of processors is independent of the number of degrees of freedom of the manipulator.
- **Degree of parallelism (Granularity).** Three levels of granularity are distinguished. In the *large grain granularity*, the parallelism is performed at the algorithmic level. That is, only the parallelism between different segments or subtasks is considered. For the *medium grain granularity*, the concurrency is considered at the operation level and the parallelism is performed based on some basic mathematical operations such as vector cross product and matrix-vector multiplication. If we consider the implementation of parallelism within the basic arithmetic operations, then the *fine grain granularity* is achieved. Different degrees of parallelism often imply different synchronization requirements. The finer the granularity is, the more frequent synchronization is required.
- **Uniformity of operations.** A robotics algorithm is said to possess uniformity of operations if the required computations for some set of variables, especially the joint variables, are uniform. An algorithm with operation uniformity can be implemented on an SIMD machine with higher efficiency.

- **Fundamental operations.** Algorithms in an application area usually perform similar mathematical operations. The identification of basic operations performed in the algorithm will dictate the processor capabilities needed.
- **Data dependency.** Three kinds of data dependency are classified for robotics algorithms: *local neighborhood dependency, special type dependency,* and *global dependency.* The local dependency means that the required operands in an operation come from its neighborhood; for example, from the results of last operation or using the same operands of last operation. The special type dependency is defined for some special equation or problem. There are some special types of data dependency that are peculiar and inherent to the robotics algorithms. Among them, the homogeneous (or hetero-homogeneous) linear recursive type of dependency which describes the data dependency in a homogeneous (or hetero-homogeneous) linear recursive equation appears most frequently. This linear recurrence structure plays a major role in the robotics algorithms because the variables of a joint are usually related to the corresponding variables of its adjacent joint due to the robot's serial link structure. Other special types of data dependency are defined for some well-known problems; for example, system of linear equations and Column-Sweeping algorithm for a triangular linear system. The global dependency means that the results of some operations may be required by other operations or equations that may appear in other places of the algorithm. Since few algorithms possess absolutely one kind of data dependency, we can just identify whether an algorithm is local data dependency oriented or not. The data dependency in an algorithm usually dictates memory organization, data allocation, and communication requirements.
- **Communication requirement.** The communication requirement decides the required interconnection type between processor and processor or between processor and memory. Three types of interconnection are considered: *one-to-one* connection, *permutation* and *broadcast* connections. Of course, the exact required interconnection type for each computation in an algorithm depends on many factors such as task assignment of each processor, data allocation in the memories, and data dependency of each computation. Hence, the exact required interconnection type can only be decided at the time of the algorithm-architecture mapping process. In examining the features of robotics parallel algorithms, only rough connection requirements can be observed.

3. Characterization of Basic Robotics Parallel Algorithms

Based on the above set of features, each of the six basic robotics algorithms have been carefully examined and analyzed to find the common features and characteristics among them [2]. Only the final results are presented here, which are useful for better understanding of the robotics computations and for designing a suitable parallel architecture for their computations.

Inverse Dynamics Problem. Among various methods for computing the inverse dynamics problem, the one based on the Newton-Euler (NE) equations of motion is the most efficient [3]. Since this method has been shown to possess the time lower bound of $O(n)$ running on uniprocessor computers, where n is the number of degrees-of-freedom of the manipulator, further substantial improvements in computational efficiency appear unlikely. Nevertheless, some improvements could be achieved by taking advantage of particular computation structures [4], customized algorithms/architectures for specific manipulators [5], parallel computations [6,7], and scheduling algorithms for multiprocessor systems [8-11].

Forward Dynamics Problem. Among various methods for solving the forward dynamics problem [12-14], the composite rigid-body method [12], based on the computation of the NE equations of motion, is widely used to develop efficient parallel algorithms [14-16]. The composite rigid-body method is suitable for parallel processing because efficient parallel algorithms for the inverse dynamics computation have been well developed and can be used to speed up the computation time.

Forward Kinematics Problem. Using the Denavit-Hartenberg matrix representation for establishing the link coordinate frames [17,18], the solution to the forward kinematics problem is the successive multiplication of the 4×4 homogeneous link transformation matrices for an n-link manipulator

$$\mathbf{T} = \mathbf{A}_0^1 \mathbf{A}_1^2 \mathbf{A}_2^3 \cdots \mathbf{A}_{i-1}^i \cdots \mathbf{A}_{n-1}^n \tag{1}$$

where $\mathbf{A}_{i-1}^i$ is the D-H link transformation matrix which relates the ith coordinate frame to the $(i-1)$th coordinate frame [17,18]. The above successive matrix multiplication equation can be reformulated in a homogeneous linear recursive form

$$\mathbf{T}_0^1 = \mathbf{A}_0^1 \quad \text{and} \quad \mathbf{T}_0^i = \mathbf{T}_0^{i-1} \mathbf{A}_{i-1}^i \quad \text{for } i = 2, \cdots, n, \tag{2}$$

from which the configuration of all the coordinate frames can be obtained at the time lower bound [7,19,20].

Forward Jacobian Problem. Existing methods in computing the Jacobian are mostly confined to uniprocessor computers. In particular, Orin/Schrader [21], and Yeung/Lee [22] exploited the linear recurrence characteristics of the Jacobian equations. These methods differed from each other only by a different selection of the reference

coordinate frame for computation. The reference coordinate frame is selected such that all the vectors and matrices and the Jacobian computed are referred to that reference coordinate system. They all have the computational order of $O(n)$ for an n-jointed manipulator.

Inverse Jacobian Problem. The inverse Jacobian algorithms for a general manipulator can be divided into two categories. One is to calculate the inverse or the generalized inverse Jacobian explicitly [23]. The other is to consider the inverse Jacobian problem as a system of linear equations and solve the joint rate from the Cartesian velocity implicitly [24]. For practical purposes, the latter approach is easier to be parallelized due to the use of some standard techniques to solve a system of linear equations such as the Gaussian elimination method.

Inverse Kinematics Problem. In general, the inverse kinematic position solution can be obtained by various techniques [18], among which the inverse transform [25] and the iterative method [26] are widely discussed. The inverse transform technique yields a set of explicit, non-iterative joint angle equations which involve multiplications, additions, square root, and transcendental function operations. The iterative methods can obtain robot independent joint solution, but they usually have some disadvantages: more computations than the closed-form solution, variable computation time and, more important, convergence problem, especially in the singular and degenerate cases. We shall examine the characteristics of the inverse transform technique and the iterative methods.

The equations for closed-form solution appear highly non-uniform [27]. To achieve higher parallelism for the inverse kinematics problem, the iterative method provides a better approach, since nearly every presented iterative method contains the computations of forward kinematics, forward Jacobian, and inverse Jacobian [26], which have been shown to be highly parallelized.

If we consider these six basic robotics computations as a set of tasks that we need to compute for the control of robot manipulators, then we need to find their common features and characteristics so that a parallel architecture can be designed to efficiently compute these tasks. The characteristics of the six basic robotics algorithms are tabulated in Table 1 and it shows that these algorithms do possess some important common features and characteristics. This is especially true for the inverse dynamics, the forward dynamics, the forward kinematics, and the forward Jacobian computations for the following three reasons. First, they are all suitable to be parallelized at the job-level and the parallelization can be performed at the large, medium, and fine grain granularities simultaneously, although different granularities are emphasized in each individual algorithm. Second, their operations are all uniform for the variables corresponding to each joint, and the most important fundamental operation is the matrix-vector operation. Finally, the strongest common feature is that they are all in homogeneous linear recursive form, for which the recursive doubling technique can be applied to achieve the time lower bound of $O(\lceil \log_2 n \rceil)$. The communication requirement indicates that one-to-one and some regular or irregular permutation capabilities are required for these four computational problems and the broadcast capability is necessary for the forward dynamics and the forward Jacobian algorithms. This indicates that some efficient, versatile network is required in the parallel architecture for their computations.

The inverse Jacobian and the inverse kinematics computations may seem less common to the above four algorithms. However, if less efficient methods to solve these two problems are chosen individually, then these two algorithms may possess some common features to the other four algorithms, and a common parallel architecture can be designed to match all these common characteristics for their computations. From previous discussions, we found that either the direct method or the iterative method for the inverse Jacobian is a proper candidate for parallel processing, while the direct method is more efficient with somewhat complex data dependencies. For the inverse kinematics problem, only the iterative method possesses regular properties similar to the other four computations.

With all the characteristics listed in Table 1, we shall next examine how to reformulate and parallelize these robotics algorithms from their original serial algorithms by complying to their common features [2]. The parallelization process is performed at the job level; that is, we try to express the original algorithms as a sequence of serial steps (jobs). Each individual step is accomplished through the cooperation of all the processors and for each step, the operations of each processor are almost identical by using one of their common features: the uniformity of operations. Hence, each step can be considered to be a single instruction in a serial program. Two different steps (or jobs) are identified after the parallelization process: *single* steps and *macro* steps. The notion of "single instruction" and "subroutine" of a serial program can be used to distinguish between these two steps. A single step corresponds to a single instruction in a serial program, while a macro step corresponds to a subroutine in a serial program. The macro steps require more complex parallel computations for all the processors, for example, the homogeneous linear recursive equation, the hetero-homogeneous linear recursive equation, and the system of linear equations are all macro steps. These macro steps are identified by their completeness and repeatity. The completeness means that the step can be treated as an individual problem. The technique to process these macro steps parallelly needs special consideration and the algorithm to solve these steps is so well-structured that finer decomposition is not helpful or even impossible, for example, the parallel recursive doubling technique for solving the homogeneous linear recursive equation, or the parallel Cholesky factorization technique for solving the system linear

equations with a symmetric-positive-definite square matrix. The repeatity means that the problem which can be solved in the step is so important and common that it appears repetitively at many other places; for example, many equations of robotics algorithms are in homogeneous linear recursive form, then the procedure for parallelly solving this problem can be applied to all these places. The method to parallelize each of these macro steps is designed separately.

Instead of computing all the six basic robotics algorithms, we synthesize a benchmark algorithm (see Table 2) which represents the general structure of the basic robotics parallel algorithms. This benchmark algorithm consists of six serial steps, and each step needs the cooperation of n processors. This benchmark algorithm will be used to demonstrate the whole process of mapping the "serial type" parallel algorithms onto a proposed parallel architecture in the following sections.

4. Design of Algorithmically-Specialized Parallel Architecture

In this section, an appropriate parallel architecture with the attributes that best match the common features of the six basic robotics parallel algorithms is designed. The important parallel architecture attributes include the type of machine (e.g., SIMD or MIMD mode), number of processors, synchronization requirement, processor capabilities, memory organization, and network requirement. Each of these attributes is affected by one or more features of the six basic robotics algorithms discussed in section 3. Detailed consideration for the design of this machine can be found in [2]. With all these requirements and attributes, the appropriate parallel architecture is a *reconfigurable, dual-network, SIMD* (DN-SIMD) machine for the computation of robotic algorithms.

The structure of the proposed DN-SIMD machine, as shown in Fig. 1, consists of multiple processing elements, two reconfigurable interconnection networks (RIN1 and RIN2), a set of global data registers (GDRs), three data buffers including register output buffer, PE output buffer and input data buffer (IDB), and a set of multiplexers. All of these are coordinated by a central control unit (CU) which is not shown in Fig. 1. The functions of each element are briefly described here.

1. *Processing Element* (PE). There are n identical PEs. Each PE is essentially an arithmetic logic unit (ALU) with attached working registers (see Fig. 1). All the ALUs perform the same programmable function synchronously in a lock-step fashion under the command of the CU. Some of the PEs can be masked (disabled) for some computation period, while other unmasked or enabled PEs perform computations. Each PE has two input working registers (IWRs) which are used to store two operands for each computation, and one output working register (OWR) which is used to store the current result of each computation. The operands in the IWRs are kept there until they are replaced. Thus, they can be used repetitively if one or two operands are common for a series of continuous computations. An inner loop connection within a PE is designed, which connects the OWR to one of the two IWRs. This provides an immediate inner-PE forwarding path such that the current result can be used as an operand for the next computation immediately.
2. *Global data registers* (GDRs). There are n groups of data registers which correspond to the n global memory modules. In each computation period, the registers with the same relative position in each group can be accessed under the control of the CU. The result of each computation from each PE will be stored in the GDRs only when either the result is the final output or the result will be used in later computations but not the immediate following one, which can make use of the internal forwarding path for data exchange among PEs or inner loop within PEs.
3. *Reconfigurable interconnection networks.* There are two sets of identical interconnection networks: RIN1 and RIN2. They are assumed to have full connectivity including one-to-one, permutation, and broadcast capabilities (e.g., the crossbar network). The RIN1 connects the GDRs to the PEs. This provides the paths for sending required operands to the appropriate PEs. The RIN2 makes the connection from the outputs of PEs to the inputs of PEs; this provides the direct paths for internal forwarding data exchange among PEs. It should be noted that, if necessary, the output of PE i can be stored into its corresponding memory module i. This is not affected by the RIN2.
4. *Data buffers.* There are three sets of data buffers. The register output buffer allows the "current computation" and the "RIN1 reconfiguration and operand fetch for the next computation" be processed at the same time. The PE output buffer allows the "current computation" and the "RIN2 reconfiguration and output data storing" be processed simultaneously. The input data buffer (IDB) is the buffer for operands directly from external input data.
5. *Multiplexer.* The n multiplexers in advance of the n PEs are used to select proper operands to enter PEs from three possible sources: GDR, IDB, and IFD exchange. They are also under the control of the control unit.

With the functions of these elements described above, the basic mathematical operations performed by each PE of the DN-SIMD machine involve at most two operands,

$$T = A \circ B \tag{3}$$

where A and B are two arbitrary operands and they can be scalar, vector or matrix, and "$\circ$" indicates the operation performed by the PE. When either A or B is null, the computation only involves one operand such as the transpose of a matrix. The operands A or B may come from five different sources. They are GDRs through RIN1, IFD exchange through RIN2, IDB, IWR within PE, and OWR within PE through inner loop connection. The result T may be sent to two possible destinations: GDRs directly, or PEs through RIN2 via IFD exchange. The possible input operands and output result transfer path diagrams are illustrated in Figs. 2(a) and 2(b) respectively. In Fig. 2(a), we demonstrate all the possible source combinations except the case that the operands come from the IWR within the PE. We assume that the time to transfer one operand from the GDR or the IDB to a PE (i.e., operand fetching) is the same as the time to transfer the output result from a PE to the GDR (i.e., result storing) and equals to the computation time of one basic PE operation. This time interval is called a *cycle*. Since operands fetching, computation, and result storing can be performed simultaneously due to the data buffers designed in this system, a three-stage pipelined operation can be performed on our DN-SIMD machine. Since a computation usually needs two operands A and B, and if A and B come from different sources, then they can be transferred to a PE simultaneously in one period. In this case, the three-stage pipelined operation proceeds normally. However, if A and B come from the same source (e.g., GDR or IDB), then it will take 2 cycles to transfer them. This situation is called the *double transmission required* (DTR) computation. In this case, a delay period must be added to the pipeline operation to synchronize the operation. This DTR computation obviously will slow down the system speed. Hence, we need to minimize the number of DTR computations in a computational task.

5. Mapping of Parallel Robotic Algorithms onto the Dual-Network SIMD Machine

Since our DN-SIMD machine was designed to best match the common characteristics of the six basic robotics parallel algorithms, the scheduling of their computations in our system is more straightforward with less difficulties as compared with other general mapping problems. Based on this characteristics matching, a systematic and efficient mapping procedure is developed to map the parallel robotic algorithms onto the proposed medium-grained DN-SIMD machine.

The proposed mapping procedure consists of three stages [2]. In the first stage, each of the single steps of these parallel robotic algorithms is further decomposed to a set of "subtasks" and each subtask possesses the basic mathematical form of consisting at most two operands. On the other hand, each of the macro steps in these algorithms is viewed as a subtask and is not decomposed at this stage. The first stage results in a series of parallel subtasks. In the second stage, these subtasks are reordered to reduce the number of DTR operations through a *neighborhood scheduling* algorithm. The reordered subtasks will be mapped onto the DN-SIMD machine directly in the third stage. In the final stage, the actual implementation of the macro steps in the parallel algorithms on the DN-SIMD machine is performed. Using the benchmark algorithm in Table 2 as an example, the details of these three stages of our mapping procedure are discussed in the following subsections.

5.1. Subtask Assignment

Since the proposed DN-SIMD is a medium-grained machine and is synchronized at each basic mathematical operation, each parallel algorithm must be decomposed into a series of subtasks. Each subtask is either in the basic mathematical form which involves at most two operands or in a well-defined macro step. Although this functional decomposition can be easily performed on the single steps, it is not the case for the macro steps, in which the data dependencies are so complex that the decomposition based on basic computational unit is not obviously feasible. So the macro step will be viewed as a single subtask in this stage. Consider the decomposition of the following equation

$$K = L \times (C + E) + G \times C \,. \tag{4}$$

Here we use three temporary variables, T_1, T_2, and T_3 to rewrite Eq. (4) into four simple equations in the basic mathematical form:

$$T_1 = C + E \;,\; T_2 = L \times T_1 \;,\; T_3 = G \times C \;, \text{ and } K = T_2 + T_3 \tag{5}$$

This same technique is applied to our decomposition process for single steps. For clarity, the benchmark algorithm is used as an example to demonstrate the technique. The decomposition result and the original algorithm are shown in Table 2. Here, two sets of variables are introduced: T_i's represent the immediate results (temporary variables) or the final outputs. If T_i is a macro subtask, then it is specially denoted as $\overline{T}_i$. I_i's represent the external input variables; that is, the variables that do not come from the outputs of other computations.

To ease the subtask scheduling in the second stage, notation simplification is performed on the above task table to produce a *simplified task table* as shown in Table 3. In this table, two arrays are defined: TB[i] contains the

identification of subtasks T_i's and OP[i] represents the corresponding operation for subtask TB[i]. Each element of OP[i] is either a macro subtask or in the form of $A \circ B$, where A and B may be T_i ($\overline{T_i}$) or I_i. Moreover, the superscript on A or B indicates the difference between the index i of the result, $T_{j_1}[i]$, and the index k or l of its operand $T_{j_2}[k]$ or $T_{j_2}[l]$, where $T_{j_1}[i] = T_{j_2}[k] \circ T_{j_3}[l]$. For example, the subtask $T_1[i] = T_2[i+2] \circ T_3[i-1]$ is denoted as $T_1 = T_2^2 \circ T_3^{-1}$. If their indices are equal, that is, $i = k$ or $i = l$, then the superscript is omitted. For example, the subtask $T_4[i] = T_5[i] \circ T_6[i]$ is denoted simply as $T_4 = T_5 \circ T_6$. The simplified task table is the final result of this stage and will be used as the input for the next stage.

5.2. Subtask Scheduling

To schedule the subtasks for computation, we first observe all the possible operand sources and their combinations for each computation. The operand may be one of the four possible types denoted as S_I, S_{OI}, S_T, and S_{OT} which correspond to four kinds of different sources. S_I denotes the operand from the IDB and it needs one period of transmission time. S_{OI} denotes the operand which is fetched by the previous computation (subtask) from the IDB and is still in the IWR within the PE, so no transmission is required for this operand. S_T denotes the operand from the GDR and this operand requires one cycle of transmission time via the network RIN1. S_{OT} denotes the operand from other sources including the following three possibilities: (i) The operand which is fetched by the previous computation from the GDR and is still in the IWR within the PE, so no transmission time is required; (ii) Current computation result through the inner loop; (iii) Current computation result through the internal forwarding path with data exchange provided by the network RIN2. The transmission time for the last two cases is ignored when compared to the system cycle time. Using these notations, all the possible combinations of operand sources including the situation of only one operand are listed below:

(S_I', S_I'')	(S_I, S_T)	(S_{OT}, S_T)	(S_{OI}, S_{OI})	(S_I)
(S_I, S_{OI})	$S_{OI}, S_{OT})$	(S_T', S_T'')	(S_{OT}, S_{OT})	(S_{OI})
(S_I, S_{OT})	(S_{OI}, S_T)	(S_I, S_I)	(S_T, S_T)	(S_T) , (S_{OT})

where the prime superscripts are used to distinguish different operands from the same kind of source. Among these situations, the combinations (S_I', S_I'') and (S_T', S_T'') are DTR operations and require two cycles to transmit two operands through the same transmission path. It is possible to eliminate DTR operations, if we reorder the processing sequence without violating the constraint of precedence relation. That is, in these two situations, one operand S_T' (or S_T'') can become the type S_{OT}, or S_I'(or S_I'') can become the type S_{OI}. Then, the DTR operation phenomena can be avoided and the unnecessary transmission can also be avoided for the efficient use of the same data repetitively and instantly.

A neighborhood scheduling algorithm for scheduling and reordering the execution of these subtasks to minimize the total number of DTR operations has been developed and is considered here.

Definition 1. For two subtasks in the kth and lth rows of the simplified task table, $TB[k]$ and $TB[l]$, assume $OP[k] = A \circ B$ and $OP[l] = C \circ D$, where A, B, C, and D are operands, each with one of these possible types: $\{I_j, T_j, T_j^i\}$. Then the subtask $TB[k]$ is called a *neighborhood* of $TB[l]$ if all the following conditions are satisfied:

(i) $k < l$,

(ii) $C = TB[k]$ or $C = TB^i[k]$ or $C = A$ or $C = B$ or
$D = TB[k]$ or $D = TB^i[k]$ or $D = A$ or $D = B$.

From the above definition, we know that if subtask $TB[l]$ has a previous subtask $TB[k]$ as its neighborhood ($k < l$) and moreover, if these two subtasks are next to each other; i.e., $l = k + 1$, then at least one operand of subtask $TB[l]$ comes directly from the result or operand of subtask $TB[k]$ without accessing the GDR or the IDB. This obviously will save the communication time to access global memories, and the subtask $TB[l]$ will never be a DTR subtask, thus minimizing the number of DTR subtasks.

Definition 2. A subtask in the kth row of the simplified task table $TB[k]$ is called a *double transmission required* (DTR) subtask if the following two conditions are satisfied:

(i) Its operand is one of these types:
$OP[k] = TB[m] \circ TB[n]$ for some m, $n < k$ and $m \neq n$.
$OP[k] = TB^i[m] \circ TB[n]$ for some m, $n < k$ and $m \neq n$.
$OP[k] = TB[m] \circ TB^j[n]$ for some m, $n < k$ and $m \neq n$.
$OP[k] = TB^i[m] \circ TB^j[n]$ for some m, $n < k$ and $m \neq n$.
$OP[k] = I[m] \circ I[n]$ for $m \neq n$.

(ii) $k = 1$ or $TB[k-1]$ is not a neighborhood of $TB[k]$ for $k > 1$.

Notice that for $OP[k] = TB[m] \circ I[n]$ and $OP[k] = TB^i[m] \circ I[n]$, the subtask $OP[k]$ is not a DTR subtask because its two operands can be transmitted simultaneously through two different set of connection lines. Moreover, a subtask involves only one operand is obviously a non-DTR subtask. For example, in the simplified task table of the benchmark algorithm, subtasks $T_7, T_9, T_{12}, T_{13}, T_{15}$, and T_{17} are all DTR subtasks as indicated in Table 3.

From the above definition, whether a subtask is a DTR subtask depends on its "position" in the simplified task table. A DTR subtask can become a non-DTR subtask if it is moved to the place exactly behind its neighborhood. Since it is possible that the movement of a DTR subtask may introduce another new DTR subtask, this reordering process is desirable only when it complies with the precedence constraint of the original algorithm and the number of DTR subtasks in the reordered task table is less than that in the original table. This forms the *scheduling problem*; that is, to reorder the processing sequence of subtasks to reduce the number of DTR subtasks as far as possible without violating the precedence constraint of the original algorithm. This reordering process can be performed by the following efficient neighborhood scheduling algorithm.

Algorithm N-Scheduling (*Neighborhood Scheduling Algorithm*).
Input: Simplified Task Table with n rows (i.e., n subtasks).
Output: Reordered Task Table.

N1. [Main Loop] Check each subtask to see if it is a DTR subtask. If yes, try to change its position.
For $k = 1$ **step** 1 **until** n **do**

N2. [Check DTR]
Check if $TB[k]$ is a DTR subtask according to definition 2? If not, go to step N4.

N3. [Main Body] Try to change the position of a DTR subtask to make it into a non-DTR subtask.
If $OP[k] = (TB[m]$ or $TB^a[m]) \circ (TB[n]$ or $TB^b[n])$,

then let $i \leftarrow max(m, n)$;
else let $i \leftarrow 1$; {* $OP[k] = I[m] \circ I[n]$ *}
End {If}
While $i < k-1$ **do**

If $TB[i]$ is a neighborhood of $TB[k]$, **then**

If {$TB[i+1]$ is a DTR subtask} or {the insertion of $TB[k]$ between $TB[i]$ and $TB[i+1]$ will not make $TB[i+1]$ a DTR subtask},

then insert $TB[k]$ behind $TB[i]$ to make $TB[k]$ the new $(i+1)$th subtask;
go to step N4
End {If}
End {If}
Let $i \leftarrow i+1$;
End {While}

N4. Continue {main loop}
End {For}

END. {N-Scheduling}

As an example, the N-Scheduling algorithm is applied to the benchmark algorithm. The input is the simplified task table in Table 3, which has a total of 18 subtasks and six of them are DTR subtasks. After applying the N-Scheduling algorithm to this simplified task table, the reordered task table is produced as shown in Table 3, in which all the DTR subtasks in the simplified task table have been removed.

5.3. Mapping Procedure

The reordered task table produced by the N-Scheduling algorithm can be mapped onto the proposed DN-SIMD machine in a rather straightforward way because these subtasks are all single-step, simple subtasks. If the subtasks are macro steps, then their mapping requires further consideration. Our mapping procedure at this final stage consists of two phases. In the first phase, the subtasks including single steps and macro steps which are viewed as single steps temporarily are mapped onto the DN-SIMD machine in a row directly. The actual mapping of the macro steps is considered in the second phase. The output of the mapping procedure is a control table as shown in Table 4. This table consists of ten columns and indicates the exact movement of the central control unit. The first column represents the identification of subtasks appearing in processing order. It also represents the result of the corresponding subtask. The second column indicates the first operand; it may be T_j ($T_j^i, \bar{T}_j, \vec{T}_j$) or I_j for some i. The third column indicates the source of the first operand, and there are five possibilities: the GDR, the IDB, the IFD, the IWR and the OWR within the PE. The fourth column describes which network is used (RIN1 or RIN2) and the required connection type on it to transmit the first operand if necessary. Columns 5 to 7 contain the

same information as the previous three columns, but for the second operand if it exists. Column 8 indicates the operation performed in this subtask. Column 9 indicates the destination of the result; it may be the GDR, the IFD, or both. If the IFD is needed, the connection type of network RIN2 is specified. Column 10 contains some comment on this subtask. For a macro subtask, these columns possess somewhat different meanings. Columns 2-7 indicate the corresponding information for the initial conditions of the macro subtask (similar to the parameters for a subroutine in a serial program). Columns 9-10 indicate the corresponding information for the final result of the macro subtask (similar to the return values of a subroutine in a serial program).

At the end of phase 1 of the mapping procedure, the control table of the benchmark algorithm is obtained as shown in Table 4. Since there are three macro subtasks in the control table, further mapping must be performed in phase 2. Among these macro subtasks, $\bar{T}_1$ and $\bar{T}_6$ are the HLR equations, and $\bar{T}_{11}$ is the HHLR equation. The mapping of HLR equations are demonstrated next.

The first-order homogeneous linear recurrence equation is defined as: Given $x(0) = a(0) =$ null, and $a(i)$, $1 \le i \le n$, find all the $x(i)$ for $1 \le i \le n$ from the following recursive equation

$$x(i) = x(i-1) \circ a(i) . \tag{6}$$

An efficient technique called the recursive doubling technique has been found to solve this recursive equation efficiently on an SIMD machine [7,19]. Using this technique, the parallel algorithm to solve Eq. (6) and the mapping diagram of this algorithm onto the proposed DN-SIMD machine are shown in Fig. 3. This diagram possesses the same information as a control table including the sources of operands, destination of result, network used and required connection types for each iteration. It takes an order of $O(\lceil \log_2(n+1) \rceil)$ iterations to produce the final results. Also notice that, in Fig. 3, we assume that the initial conditions $a(i)$'s come from the IDB. In fact, they may also come from the GDR depending on whether $a(i)$'s are external input variables or not. In that case, its mapping diagram is exactly the same except that the $a(i)$'s are from the GDR through the network RIN1 at the beginning. Similarly, the final results $x(i)$'s can be stored in the GDR or directly fedback to PEs depending on the necessity of the next subtask. Using the similar techniques, the mapping of HHLR equations can also be performed [2].

6. Conclusions

To design a global architecture for a set of parallel robotics algorithms, the characteristics of these algorithms are identified according to six fundamental features: degree of parallelism, uniformity of operations, fundamental operations, data dependency, and communication requirements. Considering the characteristics matching between the common features of the robotics algorithms and the architecture features. a medium-grained, DN-SIMD machine is designed. It consists of two sets of reconfigurable interconnection networks. One provides the communication between the PEs and the GDRs. The other provides the internal direct feedback paths among PEs to avoid unnecessary data storing and routing time. This machine performs three-stage pipelined operations and is synchronized at each basic mathematical calculation.

With the parallel robotics algorithms and the proposed DN-SIMD parallel machine, a systematic mapping procedure to schedule the subtasks of the parallel algorithms onto the parallel architecture is developed. This mapping procedure consists of three stages. At the first stage, mathematical decomposition is performed on the parallel algorithms to achieve a series of subtasks and each subtask is either in the basic mathematical form which involves at most two operands, or a well-structured macro subtask such as the linear recurrence equations. At the second stage, to shorten the communication time, the processing sequence of subtasks is reordered to minimize the total number of DTR subtasks using the Neighborhood Scheduling algorithm. At the final stage, the reordered subtasks are mapped onto the DN-SIMD machine. In this process, the single-step subtasks can be mapped directly, while the macro-step subtasks need further design and special technique such as the recursive doubling technique for solving the linear recurrence equations. A benchmark algorithm was used throughout as an example to illustrate the mapping procedure.

7. References

[1] L. H. Jamieson, "Characterizing Parallel Algorithms," in *The Characteristics of Parallel Algorithms*, L. H. Jamieson et al. (Eds.), The MIT Press, 1987.

[2] C. T. Lin, "Parallel Algorithms and Reconfigurable Architecture for Robotics Computations," MSEE Thesis, School of Electrical Engineering, Purdue University, West Lafayette, IN, August 1989.

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

[4] C. S. G. Lee, T. N. Mudge, and J. L. Turney, "Hierarchical Control Structure Using Special Purpose Processor for the Control of Robot Arm," *Proc. 1982 Conf. Patt. Recog. and Image Processing*, Las Vegas, Nevada, pp.

634-640, June 14-17, 1982.

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

[6] L. Lathrop, "Parallelism in Manipulator Dynamics," *Int'l J. of Robotics Res.*, Vol. 4, No. 2, pp. 80-102, Summer 1985.

[7] C. S. G. Lee and P. R. Chang, "Efficient Parallel Algorithm for Robot Inverse Dynamics Computation," *IEEE Trans. on Syst. Man. Cybern.*, Vol. SMC-16, No. 4, pp. 532-542, July/Aug. 1986.

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

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

[10] C. L. Chen, C. S. G. Lee, and E. S. H. Hou, "Efficient Scheduling Algorithms for Robot Inverse Dynamics Computation on a Multiprocessor System," *IEEE Trans. on Syst. Man. Cybern.*, Vol. SMC-18, No. 5, pp. 729-743, September/October 1988.

[11] C. S. G. Lee and C. L. Chen, "Efficient Mapping Algorithms for Scheduling Robot Inverse Dynamics Computation on a Multiprocessor System," to appear in *IEEE Trans. on Syst. Man. Cybern.*

[12] M. W. Walker and D. E. Orin, "Efficient Dynamic Computer Simulation of Robot Mechanisms," *Trans. ASME J. Dynam. Syst. Meas. and Contr.*, Vol. 104, pp. 205-211, Sept. 1982.

[13] R. Featherstone, "The Calculation of Robot Dynamics Using Articulated-body Inertia," *Int. J. Robotics Res.*, Vol. 2, No. 1, pp. 13-30, Spring 1983.

[14] C. S. G. Lee and P. R. Chang, "Efficient Parallel Algorithms for Robot Forward Dynamics Computation," *IEEE Trans. on Syst. Man. Cybern.*, Vol. SMC-18, No. 2, pp. 238-251, Mar./Apr. 1988.

[15] M. Amin-Javaheri and D. E. Orin, "A Systolic Architecture for Computation of the Manipulator Inertia Matrix," *Proc. of 1987 IEEE Int'l Conf. on Robotics and Automation*, Raleigh, North Carolina, pp. 647-653, March 30-April 3, 1987.

[16] A. Fijany and A. K. Bejczy, "An Efficient Method for Computing the Manipulator Inertia Matrix," *2nd Int. Symp. on Robotics and Manufacturing Research*, Albuqurque, Nov., 1988.

[17] J. Denavit and R. B. Hartenberg, "A Kinematic Notation for Lower-pair Mechanisms based on Matrices,"

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

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

[20] A. Fijany and J. G. Pontnau, "Parallel Computation of the Jacobian for Robot Manipulators," *Proc. IASTED*, Santa Barbara, May 1987.

[21] D. E. Orin and W. W. Schrader, "Efficient Computation of the Jacobian for Robot Manipulators," *the Int. J. of Robotics Research*, Vol. 3, No. 4, pp. 66-75, Winter 1984.

[22] T. B. Yeung and C. S. G. Lee, "Efficient Parallel Algorithms and VLSI Architectures for Manipulator Jacobian Computation," *IEEE Trans. on Syst. Man. Cybern.*, Vol. SMC-19, No. 5, September/October 1989.

[23] P. R. Chang and C. S. G. Lee, "Residue Arithmetic VLSI Array Architecture for Manipulator Pseudo-Inverse Jacobian Computation," *IEEE Trans. on Robotics and Automation*, Vol. RA-5, No. 5, October 1989.

[24] R. Featherstone, "Position and Velocity Transformations Between Robot End-effector Coordinates and Joint Angles," *the Int. J. of Robotics Research*, Vol. 2, No. 2, Summer 1983, pp. 35-45.

[25] R. P. Paul, B. E. Shimano, and G. Mayer, "Kinematic Control Equations for Simple Manipulators," *IEEE Trans. on Syst. Man. Cybern.*, Vol. SMC-11, pp. 494-455, 1981.

[26] Y. T. Tsai and D. E. Orin, "A Strictly Convergent Real-time Solution for Inverse Kinematics of Robot Manipulators," *J. of Robotic Systems*, Vol. 4, No. 4, pp. 477-501, 1987.

[27] C. S. G. Lee and P. R. Chang, "A Maximum Pipelined CORDIC Architecture for Robot Inverse Kinematic Position Computation," *IEEE J. Robotics and Automation*, Vol. RA-3, No. 5, pp. 445-458, Oct. 1987.

Table 1. Characteristics of Basic Robotics Algorithms.

Algorithms	CHARACTERISTICS					
	Type of Parallelism	Degree of Parallelism	Uniformity of Operations	Fundamental Operations	Data Dependency	Communication Requirement
Inverse Dynamics	Job level	Large grain	Yes	Matrix-Vector	HLR	(Regular) Permutation
Forward Dynamics	Job level	Large grain	Yes	Scalar ops. Reciprocal Matrix-Vector	HLR,HHLR SHLR, PNE System of Linear Eqs.	one-to-one Permutation Broadcast
Forward Kinematics	Job level	Medium or Fine grain	Yes	Matrix Mult. Trigonometric	HLR	(Regular) Permutation
Forward Jacobian	Job level	Medium or Fine grain	Yes	Matrix-Vector	HLR (Forward & Backward)	(Irregular) Permutation Broadcast
Inverse Jacobian (Direct)	Job level	Medium or Fine grain	Yes	Scalar ops. Reciprocal Vector ops.	Global	Permutation Broadcast
Inverse Jacobian (iterative)	Job level	Medium or Fine grain	Yes	Scalar ops. Reciprocal Vector ops.	Local	Permutation Broadcast
Inverse Kinematics (Direct)	Task level	Fine grain	No	Scalar ops. Reciprocal Square root Trigonometric	Global	one-to-one Broadcast
Inverse Kinematics (iterative)	Job level	Medium or Fine grain	Yes	Scalar ops. Reciprocal Matrix-Vector Trigonometric	Local	one-to-one Permutation Broadcast

Table 2. Robotics Benchmark Algorithm and Subtask Assignment.

G_i	Equations	T_i	Subtasks
G_1	$A[i]=A[i-1]\times B[i]$, $2\le i\le n$, $A[1]=B[1]$	$\bar{T}_1$	$\bar{T}_1[i]=\bar{T}_1[i-1]\times I_1[i]$, $2\le i\le n$, $\bar{T}_1[1]=I_1[1]$
G_2	$C[i]=A[i]\cdot D[i]$	T_2	$T_2[i]=\bar{T}_1[i]\cdot I_2[i]$
G_3	$E[i]=A[i]\cdot F[i]$	T_3	$T_3[i]=\bar{T}_1[i]\cdot I_3[i]$
G_4	$G[i]=G[i-1]+(E[i]H[i])J[i]$, $2\le i\le n$	T_4	$T_4[i]=T_3[i]I_4[i]$
	$G[1]=(E[1]H[1])J[1]$	T_5	$T_5[i]=T_4[i]I_5[i]$
		$\bar{T}_6$	$\bar{T}_6[i]=\bar{T}_6[i-1]+T_5[i]$, $2\le i\le n$, $\bar{T}_6[1]=T_5[1]$
G_5	$K[i]=L[i](C[i]+E[i])+G[i]+C[i]$	T_7	$T_7[i]=T_2[i]+T_3[i]$
	$M[i]=N[i]M[i-1]+K[i]$, $2\le i\le n$	T_8	$T_8[i]=T_7[i]I_6[i]$
	$M[1]=K[1]$	T_9	$T_9[i]=\bar{T}_6[i]+T_2[i]$
		T_{10}	$T_{10}[i]=T_8[i]+T_9[i]$
		$\bar{T}_{11}$	$\bar{T}_{11}[i]=I_7[i]\bar{T}_{11}[i-1]+T_{10}[i]$, $1\le i<n$, $\bar{T}_{11}[n]=T_{10}[n]$
G_6	$O[i]=(E[i-1]+M[i])+(G[i]+K[i+2])$	T_{12}	$T_{12}[i]=\bar{T}_6[i]+T_{10}[i+2]$
	$P[i]=A[i]\cdot G[i]$	T_{13}	$T_{13}[i]=T_3[i-1]+\bar{T}_{11}[i]$
	$Q[i]=M[i]\cdot C[i+1]+O[i+1]\cdot P[i]$	T_{14}	$T_{14}[i]=T_{12}[i]+T_{13}[i]$
		T_{15}	$T_{15}[i]=\bar{T}_1[i]\cdot\bar{T}_6[i]$
		T_{16}	$T_{16}[i]=T_{14}[i+1]\cdot T_{15}[i]$
		T_{17}	$T_{17}[i]=T_2[i+1]\cdot\bar{T}_{11}[i]$
		T_{18}	$T_{18}[i]=T_{16}[i]+T_{17}[i]$

where in this table, except separate indication, i is from 1 to n.
$B[i]$, $D[i]$, $F[i]$, $H[i]$, $J[i]$, $L[i]$, $N[i]$; $i=1,\ldots,n$; are assumed to be input variables.
$B[i]$ is a 3×3 matrix. $D[i]$, $F[i]$ are 3×1 vectors.
$H[i]$, $J[i]$, $L[i]$, $N[i]$ are all scalars.

Table 3. Simplified and Reordered Task Table of Benchmark Algorithm.

	SIMPLIFIED TASK TABLE			REORDERED TASK TABLE		
ROW	TB[ROW]	OP[ROW]	DTR	TB[ROW]	OP[ROW]	DTR
1	$\bar{T}_1$	$\bar{T}_1^{-1} \times I_1$		$\bar{T}_1$	$\bar{T}_1^{-1} \times I_1$	
2	T_2	$\bar{T}_1 \cdot I_2$		T_2	$\bar{T}_1 \cdot I_2$	
3	T_3	$\bar{T}_1 \cdot I_3$		T_3	$\bar{T}_1 \cdot I_3$	
4	T_4	$T_3 I_4$		T_7	T_2+T_3	
5	T_5	$T_4 I_5$		T_4	$T_3 I_4$	
6	$\bar{T}_6$	$\bar{T}_6^{-1}+T_5$		T_5	$T_4 I_5$	
7	T_7	T_2+T_3	×	$\bar{T}_6$	$\bar{T}_6^{-1}+T_5$	
8	T_8	$T_7 I_6$		T_{15}	$\bar{T}_1 \cdot \bar{T}_6$	
9	T_9	$\bar{T}_6+T_2$	×	T_9	$\bar{T}_6+T_2$	
10	T_{10}	T_8+T_9		T_8	$T_7 I_6$	
11	$\bar{T}_{11}$	$I_7 \bar{T}_{11}^{-1}+T_{10}$		T_{10}	T_8+T_9	
12	T_{12}	$\bar{T}_6+T_{10}^{+2}$	×	T_{12}	$\bar{T}_6+T_{10}^{+2}$	
13	T_{13}	$T_3^{-1}+\bar{T}_{11}$	×	$\bar{T}_{11}$	$I_7 \bar{T}_{11}^{-1}+T_{10}$	
14	T_{14}	$T_{12}+T_{13}$		T_{17}	$T_2^{+1} \cdot \bar{T}_{11}$	
15	T_{15}	$\bar{T}_1 \cdot \bar{T}_6$	×	T_{13}	$T_3^{-1}+\bar{T}_{11}$	
16	T_{16}	$T_{14}^{+1} \cdot T_{15}$		T_{14}	$T_{12}+T_{13}$	
17	T_{17}	$T_2^{+1} \cdot \bar{T}_{11}$	×	T_{16}	$T_{14}^{+1} \cdot T_{15}$	
18	T_{18}	$T_{16}+T_{17}$		T_{18}	$T_{16}+T_{17}$	

where T_1 and T_6 are HLR equations and T_{11} is an HHLR equation.

Table 4. Control Table for Benchmark Algorithm.

1	2	3	4	5	6	7	8	9			10
T_i	Operand 1	Source 1	Network Type	Operand 2	Source 2	Network Type	Operation	Output Destination			Comment
								GDR	IFD	RIN2	
$\bar{T}_1$	$\bar{T}_1^{-1}$	IFD	*	I_1	IDB	-	MM	×		-	* HLR Eqn.
T_2	$\bar{T}_1$	OWR	-	I_2	IDB	-	MV	×		-	
T_3	$\bar{T}_1$	IWR	-	I_3	IDB	-	MV	×		-	
T_7	T_3	OWR	-	T_2	GDR	RIN1-1	VA	×		-	
T_4	T_3	IWR	-	I_4	IDB	-	SV			-	
T_5	T_4	OWR	-	I_5	IDB	-	SV			-	
$\bar{T}_6$	$\bar{T}_6^{-1}$	IFD	*	T_5	OWR	-	VA	×		-	* HLR Eqn.
T_{15}	$\bar{T}_6$	OWR	-	$\bar{T}_1$	GDR	RIN1-1	MV	×		-	
T_9	$\bar{T}_6$	IWR	-	T_2	GDR	RIN1-1	VA	×		-	
T_8	T_7	GDR	RIN1-1	I_6	IDB	-	SV			-	
T_{10}	T_8	OWR	-	T_9	GDR	RIN1-1	VA	×	×	2	
T_{12}	T_{10}^2	IFD	RIN2-2	$\bar{T}_6$	GDR	RIN1-1	VA	×		-	
$\bar{T}_{11}$	I_7	IDB	-	*	*	*	*			-	* HHLR Eqn.
T_{17}	$\bar{T}_{11}$	OWR	-	T_2^{+1}	GDR	RIN1-3	VI	×		-	
T_{13}	$\bar{T}_{11}$	IWR	-	T_3^{-1}	GDR	RIN1-4	VA			-	
T_{14}	T_{13}	OWR	-	T_{12}	GDR	RIN1-1	VA		×	3	
T_{16}	T_{14}^{+1}	IFD	RIN2-3	T_{15}	GDR	RIN1-1	VI			-	
T_{18}	T_{16}	OWR	-	T_{17}	GDR	RIN1-1	SA	×		-	Result

Connection type 1: straight connection;
Connection type 2: uniform module shift ($d = 2$);
Connection type 3: uniform module shift ($d = 1$)
Connection type 4: uniform module shift ($d = -1$)

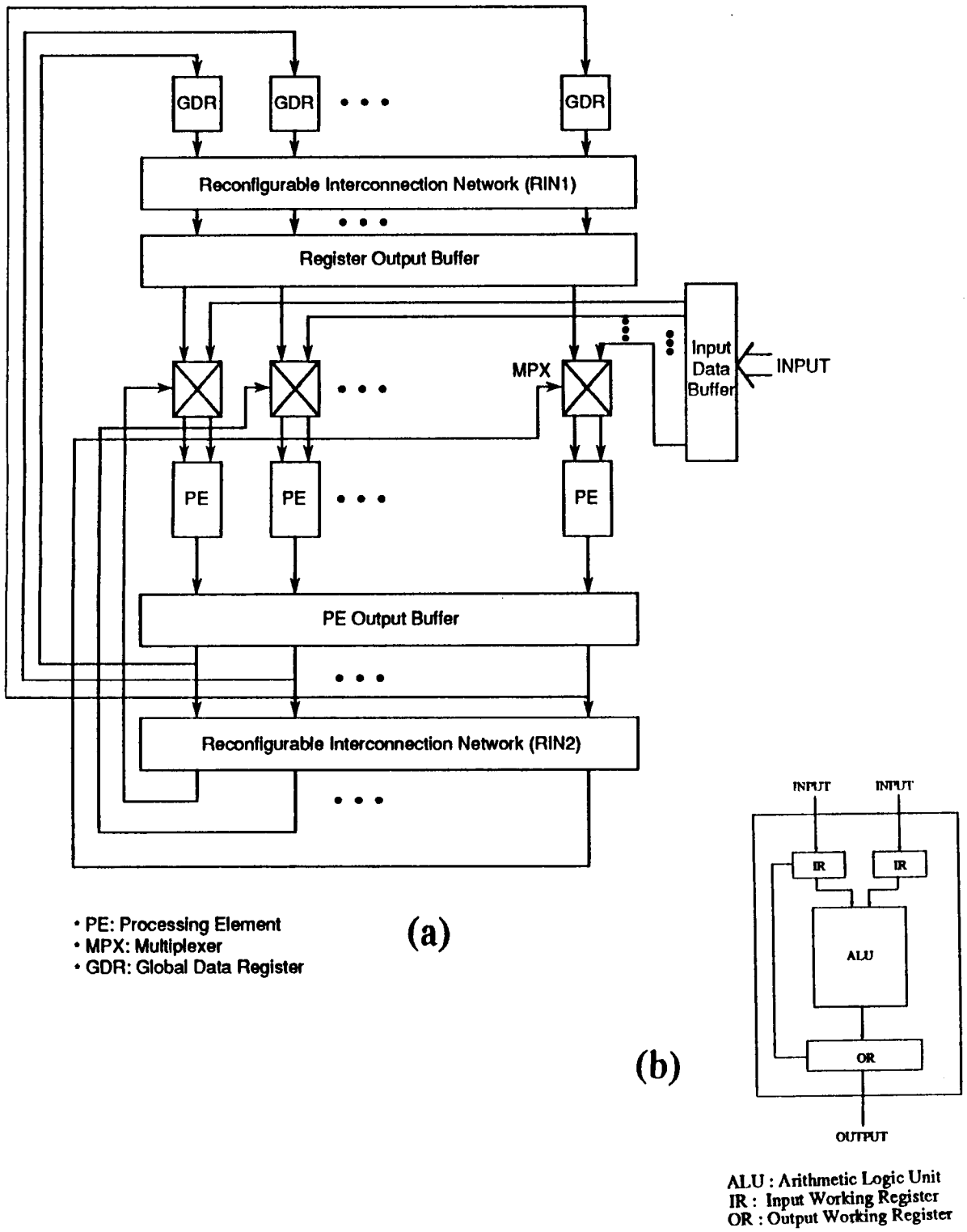
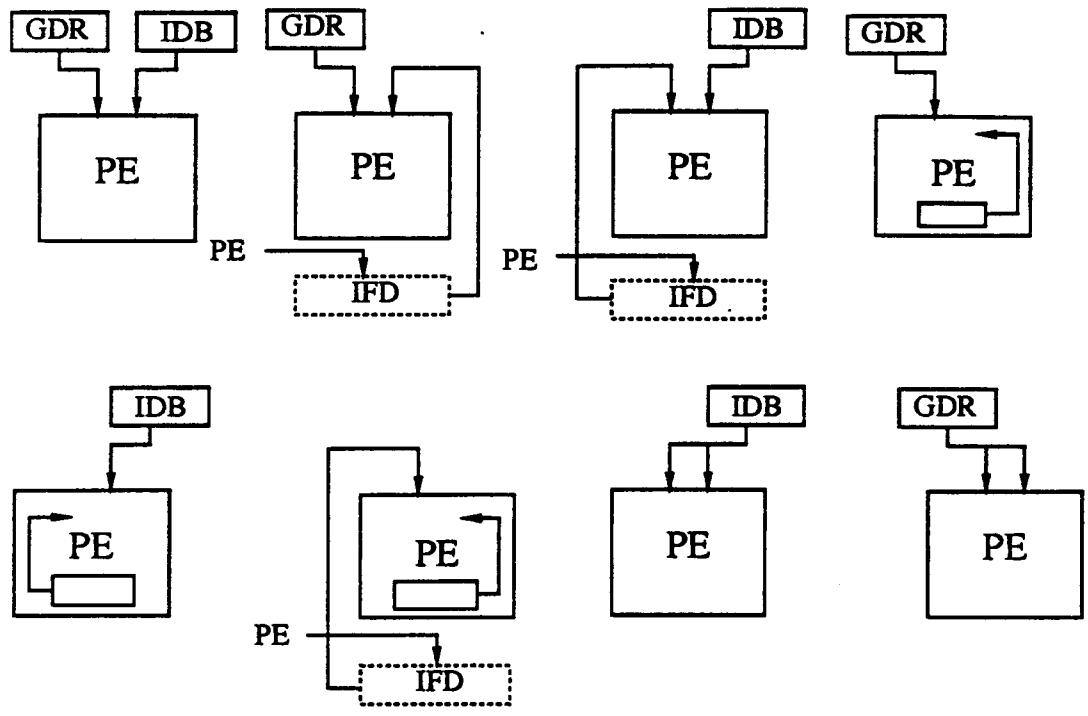
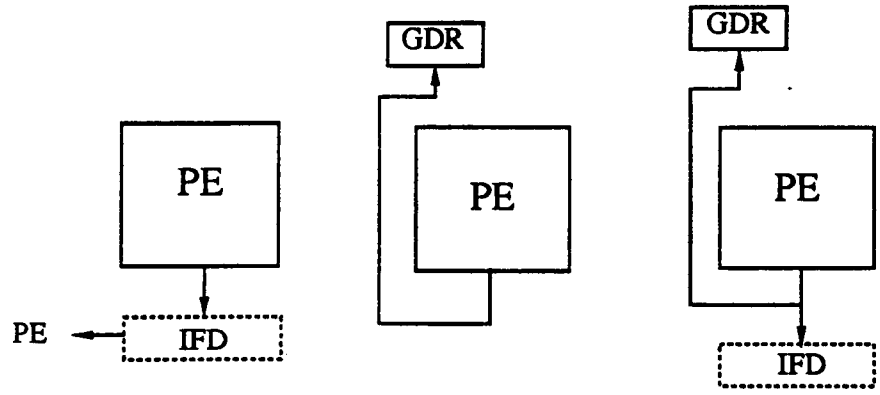


Figure 1. (a) Structure of Dual Network SIMD Machine.
(b) The Structure of Processing Element.



(a) Input Operands Flow Diagrams



(b) Output Result Flow Diagrams

Figure 2. Data Path Flow of Processing Element.

Algorithm FOHRA (*First-Order Homogeneous Recurrence Algorithm*).

F1. [Initialization] Given the terms a_i, $0 \le i \le n$, let $X^{(k)}(i)$ be the ith sequence at the kth splitting and $s = \lceil \log_2(n+1) \rceil$. Set the sequence at the initial step, $X^{(0)}(i) \leftarrow a_i$, $0 \le i \le n$.

F2. [Compute x_i parallelly]
for $k \leftarrow 1$ **to** s, **do**

$$X^{(k)}(i) = \begin{cases} X^{(k-1)}(i-2^{k-1}) * X^{(k-1)}(i) & , \text{ if } 2^{k-1} \le i \le n \\ X^{(k-1)}(i) & , \text{ if } 0 \le i < 2^{k-1} \end{cases}$$

end {for}

Set $x_i \leftarrow X^{(s)}(i)$, $1 \le i \le n$.

END FOHRA.

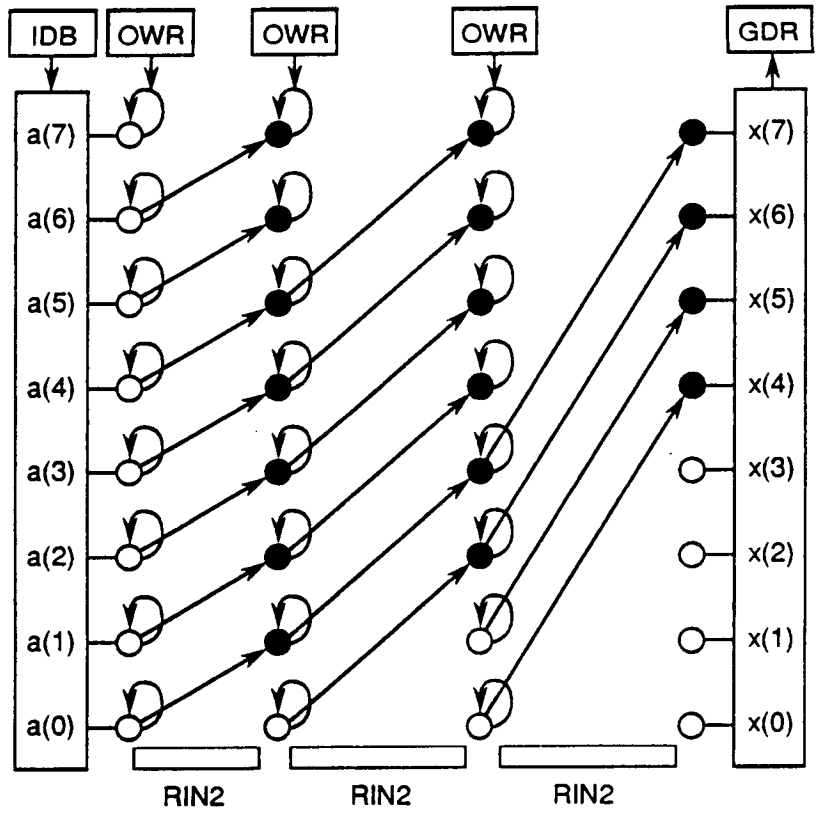


Figure 3. Mapping Diagram of First-Order Homogeneous Linear Recurrence Equations on DN-SIMD Machine.