5-1-1985

Power System Simulation by Parallel Computation

D. L. Dickmander
Purdue University

D. P. Carroll
Purdue University

Power System Simulation by Parallel Computation

D. L. Dickmander
D. P. Carroll

TR-EE 85-7
May 1985

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

POWER SYSTEM SIMULATION BY PARALLEL COMPUTATION

D. L. Dickmander

D. P. Carroll

School of Electrical Engineering

Purdue University

West Lafayette, Indiana 47907

TR-EE 85-7

May 1985

ABSTRACT

The concept of parallel processing is applied to power system simulation. The Component Connection Model (CCM) and appropriate numerical methods, such as the Relaxation Algorithm, are established as a conceptual basis for the parallel simulation of small power networks and individual power system components. A commercially available multiprocessing system is introduced for the power system simulator, and the system is adapted to facilitate high-speed parallel simulations. Two separate strategies for controlling the parallel simulation, synchronous and asynchronous relaxation, are introduced, and their performances are evaluated for the parallel simulation of an induction motor drive system. The performances of the parallel methods are also compared to a similar simulation run on a single processor, and the results show that considerable simulation speed-up can be obtained when parallel processing is employed.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER 1
INTRODUCTION

1.1 Background

Digital computers have been used with great success in the power industry to study power system network problems such as load flow and transient stability. For large power networks, mainframe computers are typically used because of the extensive storage and computational speed requirements of these programs. The computational requirements are considerably reduced, however, for small-scale power systems. Analog simulator technology, characterized by the use of operational amplifiers in the solution of system differential equations, has been extensively applied in transient studies of small-scale networks. This method has also been used successfully in detailed studies of individual power system components such as induction and synchronous machines, transmission lines, and power conditioning equipment. The results of these analog-based studies are generally viewed as realistic, and the direct integration and inherently parallel operation of analog simulators promote high solution speeds. In addition, the direct relationship between the analog simulation circuit and the system being studied make analog programming very natural and instructive.

Certain drawbacks to the analog methods, however, have limited their applications in power systems research. One problem with the analog method is the lack of flexibility in changing the power component models. Hard-wired analog simulator circuit cards typically allow for changes in component parameters, but they do not allow for changes in the component models employed. When special-purpose component models are needed, the analog power system simulator must be augmented with a general-purpose analog computer. Other problems with the analog method are the high cost of hardware and the relatively low numerical resolution as compared to digital simulation.

The rapid acceleration in microprocessor speed and microcomputer technology has sparked interest in applying this technology to the simulation of small-scale power networks and individual power system components. The flexibility inherent in microprocessor systems makes this method an attractive alternative to analog simulators, and the relatively low hardware cost enables the application of more than one processing unit to the simulation task. The overall simulation task can be broken up into several subtasks, each executed by a separate processor. By distributing the computational effort of the simulation over several processors, the processing time requirements for the simulation are considerably lower than for an identical simulation run on a single, dedicated processor. Parallel processing offers a large degree of flexibility, and promises simulation speeds approaching those of present analog simulators.

Parallel processing is a very broad concept, encompassing a large variety of machine architectures and corresponding operating systems. Parallelism can be accomplished at both the word level (sometimes referred to as "pipelining") and at the program level ("multiprocessing"). Both of these structures have been applied to power system simulation tasks, and the results have shown that multiprocessing is the most promising approach in terms of simulation efficiency[1]. Multiprocessing has an additional advantage in that this technique lends itself very well to the modularity inherent in power networks, and allows for a conceptual correspondence between the simulator hardware and the system under study. For these reasons, this thesis will consider the multiprocessing approach only.

Several groups have studied parallel architectures and algorithms in attempts to optimize the solution of power system simulation problems. The major thrust of these efforts has been to increase the overall simulation speed by minimizing the performance degradation caused by two major factors: the bus contention problems which arise when large amounts of data are transferred between processors, and the problem of keeping all processors in the working area of the simulation busy. One group at the University of Erlangen-Nuremburg, West Germany[2], has done extensive work in evaluating different parallel architectures in the power system simulation context. In these studies, a ladder network representation of a transmission line was used as a test network, with a pi section consisting of two state variables (an inductor current and a capacitor voltage) modeled on each processor. Their work showed that the resultingly high amount of data transfer between processors was handled more efficiently by a size-invariant ("pyramid") topology than by a more conventional common-bus architecture. However, another study done at the Technical University Braunschweig, West Germany[3], showed that the data transfer time can be kept to an extremely low percentage of total processor time in the simulation of small-scale power networks. This study used the physically suggestive approach of modeling each bus of the power network on a separate processor. As a result, higher numbers of state variables were located on each processor, and a common-bus architecture easily handled the correspondingly fewer data transfers involved in the simulation runs. Other noteworthy work in this area includes a project at Carnegie-Mellon University[4,5], in which an efficient power network simulation algorithm was developed for a hybrid common-bus architecture. For the initial phase of multiprocessor simulator development presented in this thesis, a common-bus architecture was employed.

1.2 Scope of Thesis

The purpose of this research is to perform the initial phase in the development of a multiprocessor-based power system simulator which will augment the existing analog simulator at Purdue. This overall objective encompasses the following specific tasks:

- Establish a unified conceptual framework for the study of small power systems and their components in the multiprocessing environment.
- Develop a suitable system architecture and programming tools for the above framework.
- Test the simulator with a simple power system simulation application, and verify the results.
- Experiment with different methods of controlling the simulation run.

The organization of this thesis corresponds roughly to the outline above. In Chapter 2, the Component Connection Model and appropriate numerical methods are set forth as a conceptual basis for the simulator, and their application in the multiprocessing environment is explained. Chapter 3 presents an overview of the multiprocessing system used in this research, and describes the specific tools and techniques which were developed to apply power system simulation tasks to the multiprocessor. A test of the multiprocessor simulator is carried out in Chapter 4 for a small-scale power system application, and the results of this study are compared to a similar single-processor simulation. Two different methods of controlling the simulation, and their results, are also presented in Chapter 4.

CHAPTER 2
CONCEPTUAL BASIS OF SIMULATOR

Two basic issues come into play when considering the framework of a multiprocessing system: processing efficiency (speed) and conceptual simplicity. Although the two are not mutually exclusive, it has been shown that rigid adherence to a conceptually simple method may not lead to the most optimal solution speeds[4]. The framework chosen for this simulator is based on the Component Connection Model (CCM), and numerical methods such as the Relaxation Algorithm and the Newton-Raphson Algorithm. These tools provide a conceptually simple framework for the system, and are flexible enough to allow for changes which will increase the simulation efficiency.

2.1 The Component Connection Model

The Component Connection Model is a technique developed in recent years for modeling large-scale interconnected dynamical systems[6]. Briefly stated, the CCM is a method of decoupling a large system by separating the system's component dynamics from its interconnections. This separation allows the components to be viewed individually, and it introduces a modular structure to the system which is well-suited to the modularity inherent in power systems.

2.1.1 Components

The ability to characterize a power system component by the behavior of the voltage and current at its terminals encourages the general component model shown in Fig. 2.1.

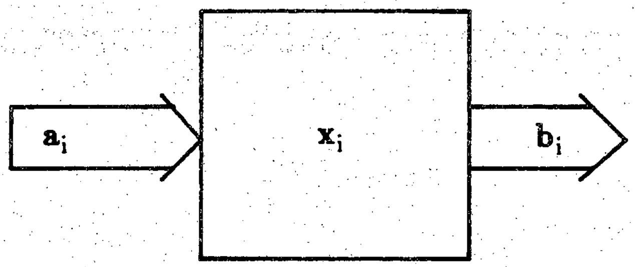


Figure 2.1. Power System Component Model

In this representation, $\mathbf{a}_i$ is a vector of component inputs (typically voltages or currents), $\mathbf{b}_i$ is the component output vector, and $\mathbf{x}_i$ is a vector of state variables internal to the component. The i subscript designates the i^{th} component of a system of N components. In the ensuing discussion, vector quantities are presented in bold face type; scalar quantities appear in normal type.

The input, output, and state variables for the power system component are related by the following general *nonlinear state model:*

$$\dot{\mathbf{x}}_i = \mathbf{f}(\mathbf{x}_i,\mathbf{a}_i) \tag{2.1a}$$

$$\mathbf{b}_i = \mathbf{g}(\mathbf{x}_i,\mathbf{a}_i) \tag{2.1b}$$

The nonlinear state model expresses each component state variable as a nonlinear function of the state vector $\mathbf{x}_i$ and the component input vector $\mathbf{a}_i$. The component output vector $\mathbf{b}_i$ is then determined by a separate function of the state vector and the component input vector. This state model is suitable for power system simulation because of the nonlinearities which occur naturally in many power system components.

2.1.2 Connections

Many real world systems, including power systems, may be represented by a large group of components of the form shown in Fig. 2.1. These components are connected together in some fashion, resulting in a *composite system,* which may or may not include an overall input vector **U** and output vector **Y**. It is possible to describe the dynamics of this composite system by the *composite system state model*

$$\dot{\mathbf{X}} = \mathbf{F}(\mathbf{X},\mathbf{U}) \tag{2.2a}$$

$$\mathbf{Y} = \mathbf{G}(\mathbf{X},\mathbf{U}) \tag{2.2b}$$

where $\mathbf{X}=\mathrm{col}(\mathbf{x}_1,\ldots\ldots,\mathbf{x}_N)$ for the N components in the system. The vector functions **F** and **G** now include both the dynamics of the system's components and the interconnections between components. This model, however, is not very useful for many simulation applications, because it destroys needed information about the component inputs and outputs and the connectivity structure of the system. A better approach is the Component Connection Model formulation, which describes the connectivity structure of a large system by the following set of linear algebraic equations:

$$\mathbf{A} = \mathrm{L}_{11}\mathbf{B} + \mathrm{L}_{12}\mathbf{U} \tag{2.3}$$

$$\mathbf{Y} = \mathrm{L}_{21}\mathbf{B} + \mathrm{L}_{12}\mathbf{U} \tag{2.4}$$

where $\mathbf{B}=\mathrm{col}(\mathbf{b}_1,\ldots\ldots,\mathbf{b}_N)$ is the *composite component output vector,*

$\mathbf{A}$=col($\mathbf{a}_1$,.....,$\mathbf{a}_N$) is the *composite component input vector,* $\mathbf{U}$ is the *composite system input vector,* and $\mathbf{Y}$ is the *composite system output vector.* The L_{ij} are sparse real matrices which map the system and component inputs and outputs.

Taken together, the components and connections described by the CCM can be viewed as a vector matrix block diagram as shown in Fig. 2.2.

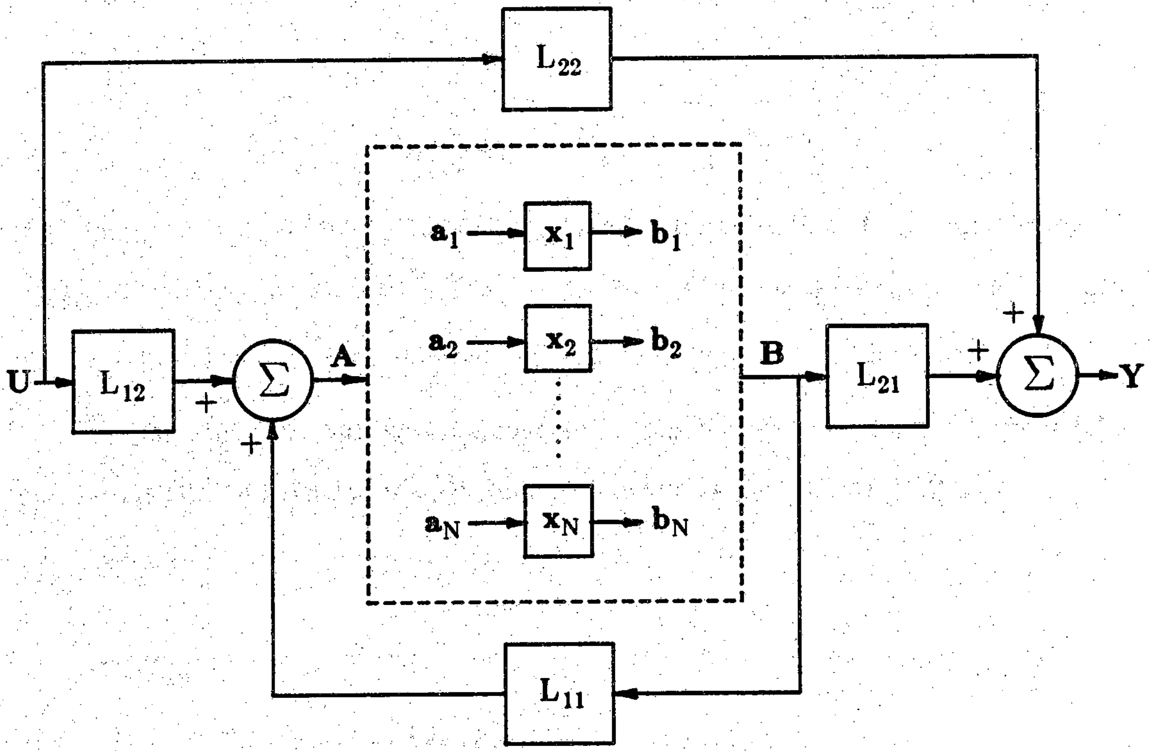


Figure 2.2. The Component Connection Model

This diagram is beneficial in understanding the relationships between the system and component inputs and outputs. It should be emphasized that the dynamics of the system are limited to the component models (in the center of Fig. 2.2), and do not appear in the interconnection equations.

The application of the CCM to power system simulation is straightforward, and can be adapted to the simulation task in mind. For example, if the short-term electromechanical phenomena (transient stability) of the system are of interest, the machines may be represented by their nonlinear state models, and the dynamics of the transmission system could be neglected. This would allow a static Z_{bus} representation of the transmission system, and the Z_{bus} matrix would then perform the dual functions of describing the system interconnections (similar to L_{11}) and modeling the system transmission lines. On the other hand, if the electromagnetic phenomena of the transmission system are of interest, the machines could be linearized around a steady-state operating point, and the transmission lines could be modeled as dynamic components.

2.1.3 Implementation of the CCM

In the CCM context, the multiprocessor architecture assumes the form shown in Fig. 2.3. The satellite processors define the working space of the simulator: all component dynamic models reside at this level. Prior to a simulation run, the necessary component simulation routines are transferred individually by the main processor into the memory space of each satellite processor. The master processor assumes responsibility for synchronization of the satellite processors and for overall control of the simulation. When included, the overall system input U and output Y are handled by the master processor. For the installation at Purdue, the A/D and D/A interface to the analog power system simulator will serve as the overall system input and output.

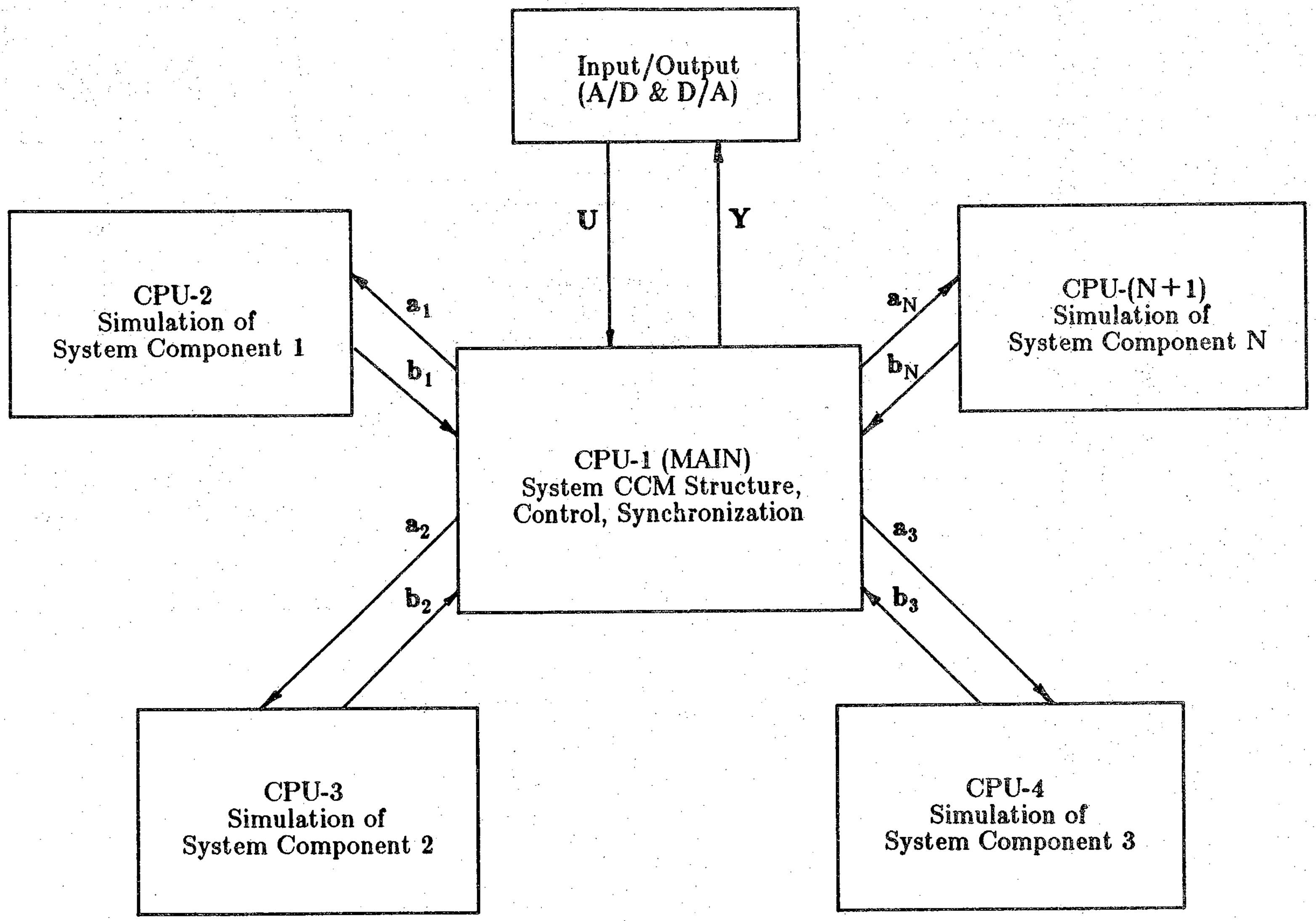


Figure 2.3. Conceptual Multiprocessor Architecture

This multiprocessing configuration has several advantages. First, it is readily adaptable to many parallel processing architectures, including the common-bus architecture used in this research. In addition, the configuration allows for a large degree of flexibility in changing component models for different simulation studies. Finally, the correspondence between the system under study and the simulator architecture should promote an interactive mode of operation with the user.

2.2 Numerical Methods

Digital simulation is the process of computing values for all state vectors and all component inputs and outputs at suitable time intervals. The coupling between the various components in a system requires that the component state equations and system connection equations be solved simultaneously. Some type of iterative technique is therefore necessary to converge to a global solution for each time step. In the modular multiprocessing context, the choice of numerical techniques is limited because of the nonlinear models employed, and the requirement that the component dynamic equations be kept separate from the system interconnection equations. Two numerical methods applicable within this framework are relaxation (predictor/corrector) algorithms, and the Newton-Raphson algorithm. Both of these techniques will be described briefly.

2.2.1 The Relaxation Algorithm

The thrust of the Relaxation Algorithm is to deal explicitly with the individual component state equations and neglect the composite component state model for the system. The procedure is to solve the individual component differential equations contained in (2.1a) by first converting them to equivalent integral equations of the form

$$x_i(t) = x_i(t_0) + \int_{t_0}^{t} f_i(\mathbf{x}(q),\mathbf{a}(q))dq \ . \tag{2.5}$$

Since we are now focusing on the dynamics of an individual component, the meaning of the i subscript has been changed. The i subscript now designates the i^{th} state variable in the n-dimensional state space of the component. The evaluation of this equation at a time instant t_k proceeds by approximating $f_i(\mathbf{x}(q),\mathbf{a}(q))$ by a polynomial evaluated at a discrete set of points t_j: $j=0,1,\ldots,k$, such that

$$f_i^j = f_i(\mathbf{x}(t_j),\mathbf{a}(t_j)) \ . \tag{2.6}$$

The numerical integration of equation (2.5) is carried out in two steps. First, an *explicit* integration scheme

$$x_i(t_k) = x_i(t_{k-1}) + \sum_{j=0}^{k-1} c_j f_i^j \tag{2.7}$$

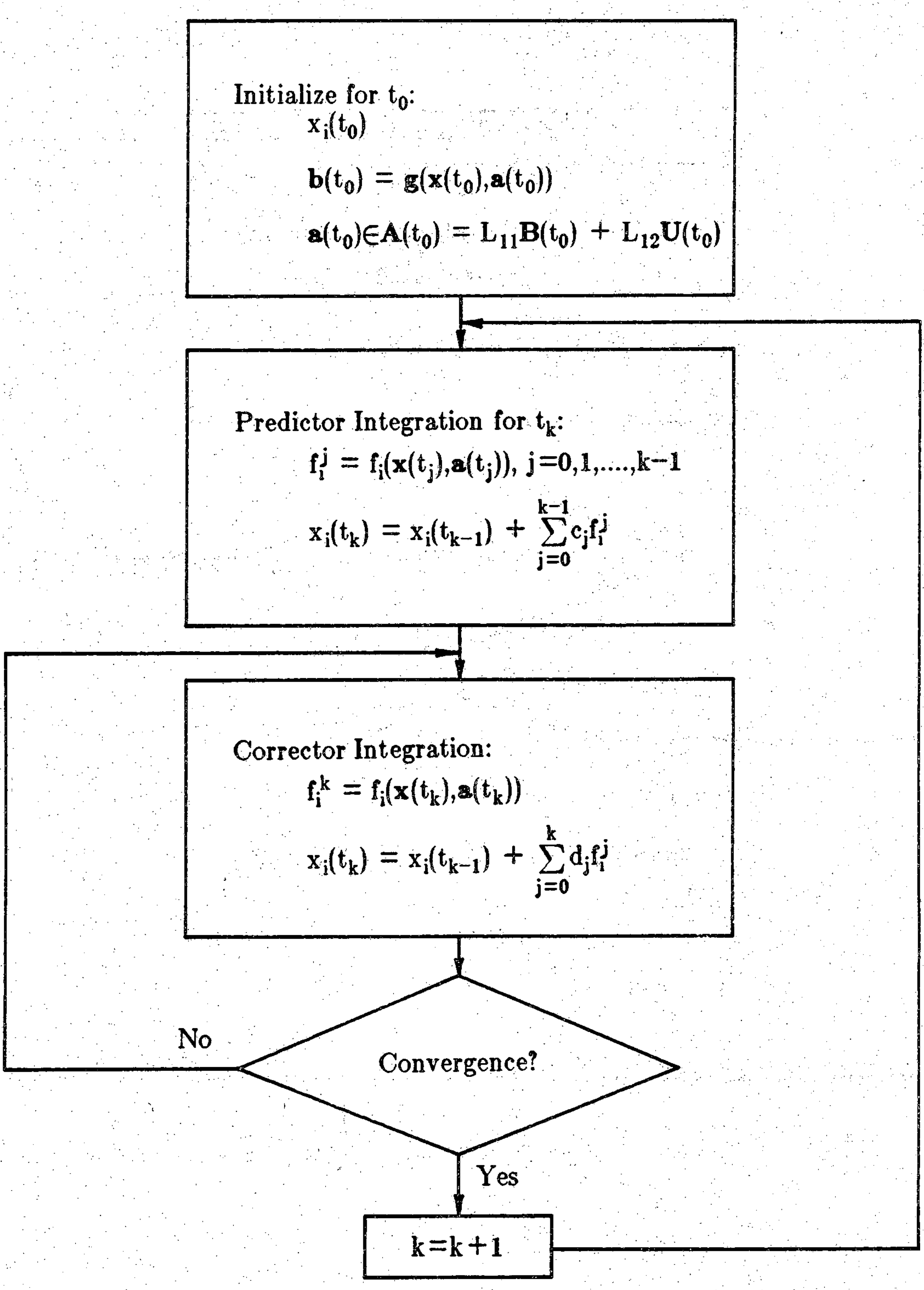
is used to *predict* values for the state variable $x_i(t_k)$. This value is then *corrected* with an *implicit* integration scheme:

$$x_i(t_k) = x_i(t_{k-1}) + \sum_{j=0}^{k} d_j f_i^j \tag{2.8}$$

The solution proceeds iteratively, by reevaluating equation (2.8) until suitable convergence occurs. The solution method is flowcharted in Fig. 2.4. A common implementation of the relaxation algorithm uses Euler integration

$$x_i(t_k) = x_i(t_{k-1}) + (t_k - t_{k-1}) f_i^{k-1} \tag{2.9}$$

for the predictor, and trapezoidal integration



Initialize for t_0:
$x_i(t_0)$
$\mathbf{b}(t_0) = \mathbf{g}(\mathbf{x}(t_0),\mathbf{a}(t_0))$
$\mathbf{a}(t_0) \in \mathbf{A}(t_0) = L_{11}\mathbf{B}(t_0) + L_{12}\mathbf{U}(t_0)$
Predictor Integration for t_k:
$f_i^j = f_i(\mathbf{x}(t_j),\mathbf{a}(t_j)),\ j=0,1,\ldots,k-1$
$x_i(t_k) = x_i(t_{k-1}) + \sum_{j=0}^{k-1} c_j f_i^j$
Corrector Integration:
$f_i^k = f_i(\mathbf{x}(t_k),\mathbf{a}(t_k))$
$x_i(t_k) = x_i(t_{k-1}) + \sum_{j=0}^{k} d_j f_i^j$
Convergence?
No
Yes
$k=k+1$

Figure 2.4. The Relaxation Algorithm

$$x_i(t_k) = x_i(t_{k-1}) + 0.5(t_k - t_{k-1})(f_i^k + f_i^{k-1}) \tag{2.10}$$

for the corrector. Other higher-order integration methods, such as Milne's method, Adams-Bashforth method, and Hammings method[7] are also applicable, and would reduce the number of corrector integrations necessary for convergence and allow for larger time steps in the simulation run. However, the number of floating-point operations per iteration is higher for these methods, so their effect on simulation speed is unknown.

2.2.2 The Newton-Raphson Algorithm

The Newton-Raphson method is an iterative process where one successively computes approximations $\mathbf{x}^k$ to the state vector solution $\mathbf{x}^*$ of a vector function $\mathbf{F}(\mathbf{x}^*)=\Theta$ for each time instant t_k (Θ is the zero vector). For a component's nonlinear state model (equation 2.1), the vector function $\mathbf{F}$ assumes the following form:

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} \dot{x}_1 - f_1(\mathbf{x},\mathbf{a}) \\ \dot{x}_2 - f_2(\mathbf{x},\mathbf{a}) \\ \cdot \\ \cdot \\ \dot{x}_n - f_n(\mathbf{x},\mathbf{a}) \end{bmatrix} = \Theta \tag{2.11}$$

An iterative technique is used to solve (2.11) for the state vector $\mathbf{x}$, and the component output equation (2.1b) is used to compute the output vector $\mathbf{b}$ for the component.

The technique used to solve (2.11) is the basis of the Newton-Raphson Algorithm. It can be shown that the stable limiting points $\mathbf{x}^*$ of the differential equation

$$\dot{x} = -J_F^{-1}(x)F(x) \tag{2.12}$$

are solutions to $\mathbf{F}(\mathbf{x})=\Theta$. In this equation, the partial derivatives of $\mathbf{F}(\mathbf{x})$ form the *Jacobian matrix*, denoted $J_F(\mathbf{x})$:

$$J_F(\mathbf{x}) = \frac{\delta \mathbf{F}}{\delta \mathbf{x}} = \begin{bmatrix} \frac{\delta F_1}{\delta x_1} & \frac{\delta F_1}{\delta x_2} & \cdots & \frac{\delta F_1}{\delta x_n} \\ \frac{\delta F_2}{\delta x_1} & \frac{\delta F_2}{\delta x_2} & \cdots & \frac{\delta F_2}{\delta x_n} \\ \vdots & \vdots & & \vdots \\ \frac{\delta F_n}{\delta x_1} & \frac{\delta F_n}{\delta x_2} & \cdots & \frac{\delta F_n}{\delta x_n} \end{bmatrix} \tag{2.13}$$

Assuming that the function $\mathbf{F}$ is differentiable in the neighborhood of $\mathbf{x}^*$, and assuming that $\mathbf{x}$ is within a sufficiently small neighborhood of $\mathbf{x}^*$, the Taylor Series expansion for $\mathbf{F}(\mathbf{x}^*)=\Theta$ reduces to a first order linear approximation[8]:

$$\Theta = \mathbf{F}(\mathbf{x}^*) \cong \mathbf{F}(\mathbf{x}) + \frac{\delta \mathbf{F}}{\delta \mathbf{x}}\Big|_{x} (\mathbf{x}^* - \mathbf{x}) \tag{2.14}$$

The Newton-Raphson iteration process is performed at each time instant t_k during the simulation run. The formula is obtained by identifying $\mathbf{x}^{j+1}$ (the newest estimate of $\mathbf{x}$) with $\mathbf{x}^*$, and $\mathbf{x}^j$ with $\mathbf{x}$; and by rearranging equation 2.14 (the j superscript designates the current Newton-Raphson iteration):

$$\mathbf{x}^{j+1} = \mathbf{x}^j - J_F^{-1}(\mathbf{x}^j)\mathbf{F}(\mathbf{x}^j) \tag{2.15}$$

It is possible to write (2.15) in a form which guarantees convergence to a solution. The result is the Modified Newton-Raphson Algorithm:

$$\mathbf{x}^{j+1} = \mathbf{x}^j - \lambda^j J_F^{-1}(\mathbf{x}^j)\mathbf{F}(\mathbf{x}^j) \tag{2.16}$$

In this formulation, λ^j is a positive scalar which controls the distance traveled in the Newton-Raphson search direction. The Modified Newton-Raphson method is flowcharted in Fig. 2.5. The $||\mathbf{F}||_2^2$ notation used in this flowchart denotes the square of the Euclidian norm of the vector function **F** and is computed by summing the squares of the components of **F**.

Before implementing the Newton-Raphson method on a computer, it is necessary to form an approximation for the state variable derivative $\dot{x}_i$ found in equation 2.11. The discrete approximation for $\dot{x}_i$ at a time instant t_k is of the form

$$\dot{x}_i^k \cong \sum_{j=0}^{m} d_j x_i^{k-j} \quad . \tag{2.17}$$

For example, Simpson's Rule may be used to obtain a second-order approximation to the derivative:

$$\dot{x}_i^k \cong \frac{3}{2h}x_i^k - \frac{2}{h}x_i^{k-1} + \frac{1}{2h}x_i^{k-2} \tag{2.18}$$

In this equation, h represents the size of the time step used in the simulation. When the Jacobian is formed, the partial derivatives are taken literally with respect to "x_i^k", and only the d_0 term of the derivative approximation appears in the Jacobian matrix.

One problem with the Newton-Raphson technique is the effort involved in obtaining the Jacobian inverse $J_F^{-1}(x^j)$ needed to solve equation 2.16. For several reasons, it is advantageous to use the Crout Algorithm to obtain the upper- and lower-triangular factorization $J_F=L_FU_F$. The inverses of the L_F and U_F matrices are then easily obtained, and the Jacobian inverse is constructed from $J_F^{-1}=U_F^{-1}L_F^{-1}$. For a linear component model, the Jacobian becomes a matrix of constant terms, and its inverse can be calculated by hand

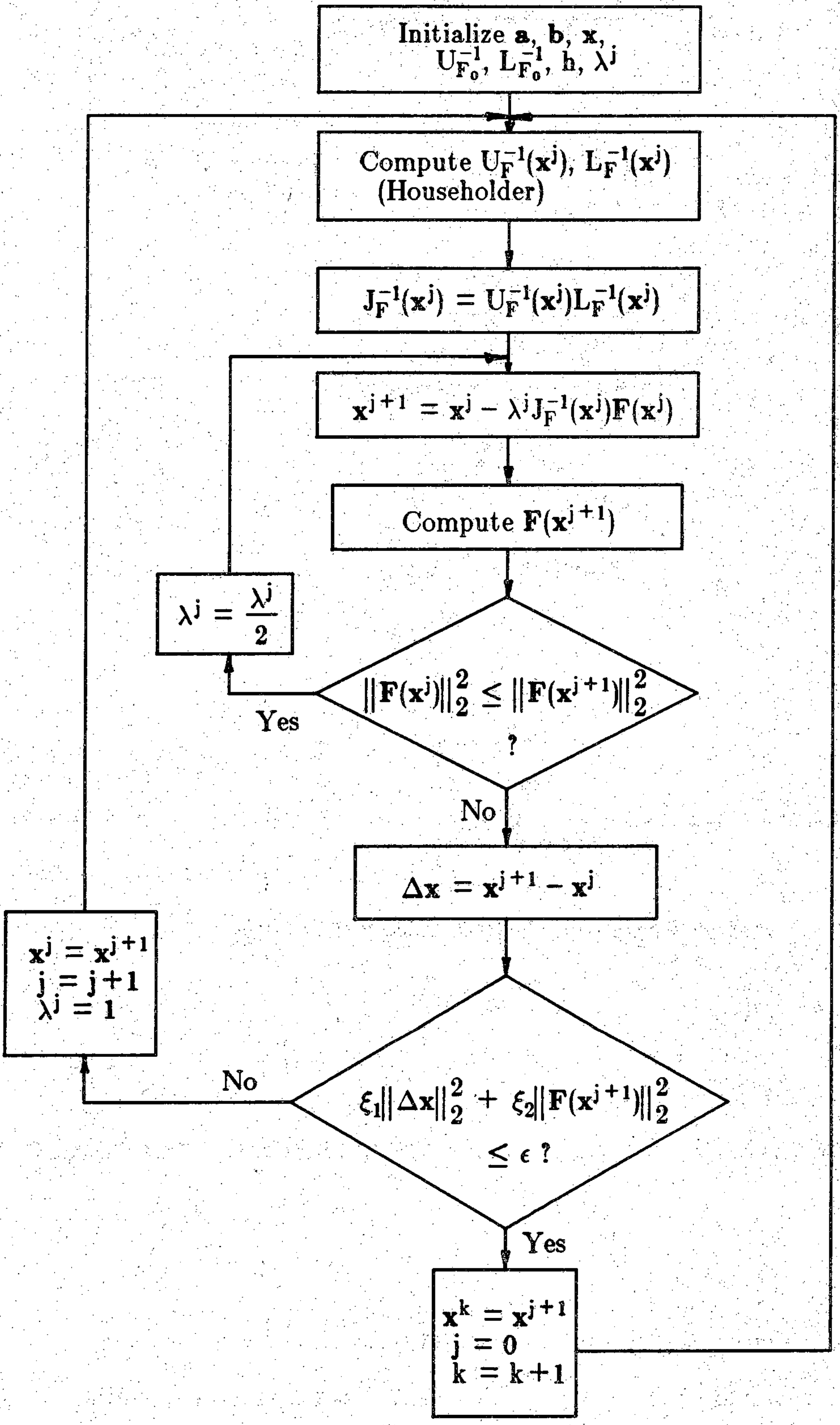

Initialize $\mathbf{a}$, $\mathbf{b}$, $\mathbf{x}$, $U_{F_0}^{-1}$, $L_{F_0}^{-1}$, h, λ^j
Compute $U_F^{-1}(\mathbf{x}^j)$, $L_F^{-1}(\mathbf{x}^j)$ (Householder)
$J_F^{-1}(\mathbf{x}^j) = U_F^{-1}(\mathbf{x}^j)L_F^{-1}(\mathbf{x}^j)$
$\mathbf{x}^{j+1} = \mathbf{x}^j - \lambda^j J_F^{-1}(\mathbf{x}^j)\mathbf{F}(\mathbf{x}^j)$
Compute $\mathbf{F}(\mathbf{x}^{j+1})$
$\|\mathbf{F}(\mathbf{x}^j)\|_2^2 \leq \|\mathbf{F}(\mathbf{x}^{j+1})\|_2^2$?
Yes
$\lambda^j = \frac{\lambda^j}{2}$
No
$\Delta\mathbf{x} = \mathbf{x}^{j+1} - \mathbf{x}^j$
$\xi_1\|\Delta\mathbf{x}\|_2^2 + \xi_2\|\mathbf{F}(\mathbf{x}^{j+1})\|_2^2 \leq \epsilon$?
No
$\mathbf{x}^j = \mathbf{x}^{j+1}$, $j = j+1$, $\lambda^j = 1$
Yes
$\mathbf{x}^k = \mathbf{x}^{j+1}$, $j = 0$, $k = k+1$

Figure 2.5. The Newton-Raphson Algorithm

in advance and built into the simulation program. A nonlinear component model, however, may cause certain terms of the Jacobian to vary, forcing the recalculation of J_F^{-1} as the simulation progresses. If these perturbations are of low rank (relative to the dimension of J_F), Householder's formula may be employed in the calculation of the Jacobian inverse. To use this method effectively, it is necessary to arrange the Jacobian in such a way that the nonlinear terms appear toward its lower right-hand corner. This arrangement prevents the perturbations from "spreading" as the $L_F U_F$ factorization is carried out. L_F and U_F are found in advance, and the simulation routine computes L_F^{-1} and U_F^{-1} using Householder's formula.

Both the relaxation algorithm and the Newton-Raphson Algorithm were investigated in the preparation of a simulation routine for the (nonlinear) symmetrical induction machine model described in Chapter 4. For this model, the nonlinear coupling between state variables resulted in perturbations to eight of the 25 terms for the five-by-five Jacobian. The extent of the perturbations, as compared to the dimension of the Jacobian, prevented an efficient Newton-Raphson implementation. In this case, the relaxation algorithm appeared to be more efficient in terms of the number of floating-point calculations necessary for convergence. It is felt that the Newton-Raphson algorithm may be more effectively applied in simulations of linear power system component models, such as the common T-equivalent representation for a transformer or a transmission line. For these reasons, the Relaxation Algorithm was chosen for subsequent studies in this thesis.

2.3 Inter-Processor Data Transfer

One problem which arises for multiprocessor-based simulation is the necessity of data transfer between processors. To achieve overall system convergence for each time step, each satellite processor must share the results of its iterations by making its component output vector **b** available to the other processors as defined by the system connection matrix L_{11}. Knowledge of the system connectivity structure may reside in matrix form at the master processor level, or in column form in each of the satellite processors. The former configuration leads to a centralized data exchange, in which the master processor distributes the results of each satellite processor iteration. A first approach to centralized data transfer would be for the master processor to perform a global data transfer operation after all satellite processors have finished their respective iterations. This method, however, would eventually suffer from performance degradation because of the large amount of data transfer involved in a large-scale system simulation.

A better method takes advantage of the staggered iteration cycle times among the satellite processors. This method is illustrated in Fig. 2.6, which shows only the activities of the satellite processors; the master processor is not shown for simplicity. Using this staggered method, the data transfers are performed by the master processor as each satellite processor completes its iteration. In this way, the data transfers for the "fast" components are overlapped in time onto the iteration cycles for the "slower" components.

It is also possible to "decentralize" the data transfer effort. Consider the CCM input equation (2.3) written in column form, ignoring for the moment the system input **U**:

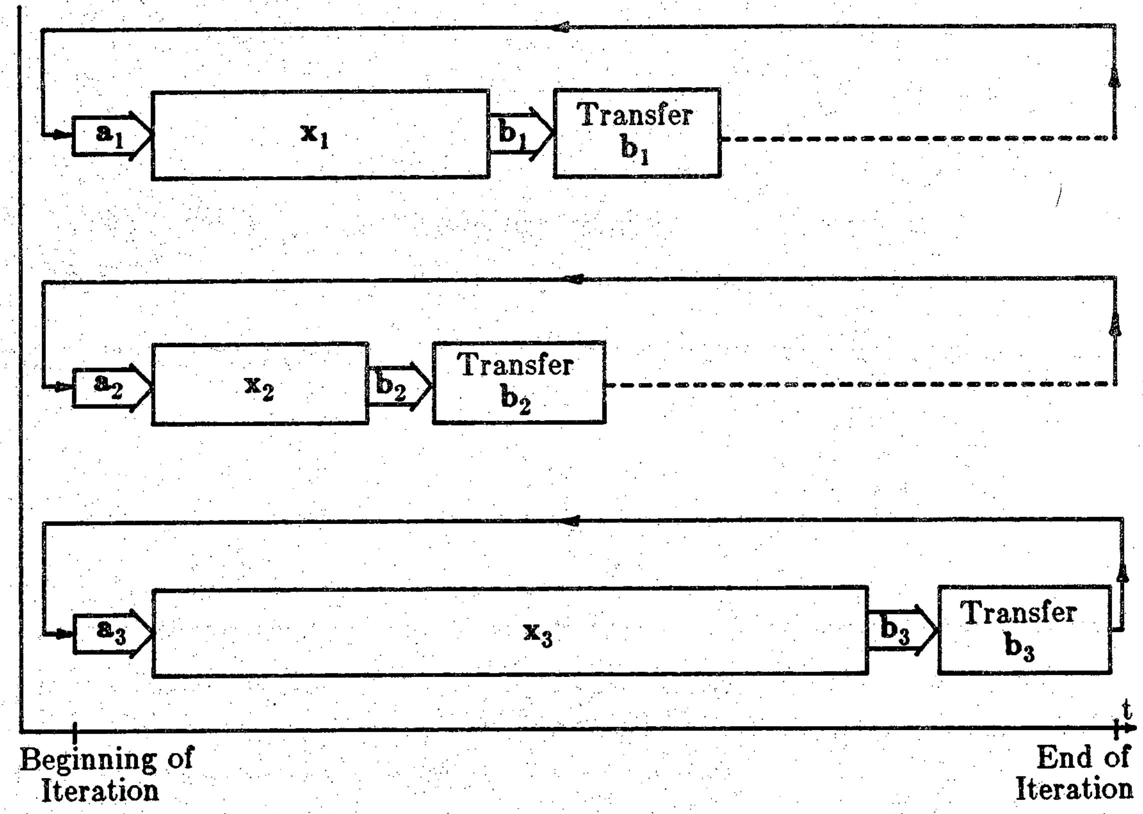


Figure 2.6. Iteration Cycle for a Three-Processor Simulation

$$\begin{bmatrix} a_1 \\ a_2 \\ \cdot \\ \cdot \\ \cdot \\ a_N \end{bmatrix} = \begin{bmatrix} \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ L_{11}^1 & L_{11}^2 & \ldots & L_{11}^N \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \cdot \\ \cdot \\ \cdot \\ b_N \end{bmatrix} \tag{2.11}$$

L_{11}^i corresponds to the ith column of the L_{11} matrix, and can be thought of as an "output distribution vector" for the ith component. For example, the L_{11}^1 vector identifies which of the system's N components require the output of component #1 in their calculations. By providing each processor with its corresponding output distribution vector, the data exchange effort can be delegated to the satellite processors: each processor assumes responsibility for distributing its results among the other processors as it concludes each iteration. This approach would free the master processor from data transfer responsibility, allowing it to concentrate on input/output, control, and synchronization.

2.4 Simulation Run-Time Control

As mentioned previously, the master processor assumes responsibility for controlling and synchronizing the activities of the satellite processors (refer to Fig. 2.3). Each satellite processor, in turn, solves its component's dynamic equations at a time instant t_k via some iterative task. Each iteration in this task includes the following steps: an *input* stage, where the component input vector **a** is calculated from the relevant output (**b**) vectors of the other components; a *calculation* stage, where the values for the state vector (**x**) are computed; and, finally, an *output* stage, where the resulting component output vector (**b**) is made available to the other satellite processors in the system. The iteration cycles and data exchanges between processors can be thought of as a "relaxation" toward the global solution for the time step involved. There are two basic methods whereby the master processor can control this global convergence: ***synchronous relaxation*** and ***asynchronous relaxation.***

The synchronous technique is the most straightforward method of controlling the simulation. This method compensates for the variations in iteration cycle times among the satellite processors by synchronizing all processors at the input stage of each iteration. After any unfinished data transfers are completed, the master processor signals all satellite processors to execute the iteration. As mentioned previously, the satellite processors complete their iterations at different times because of variations between the component models employed. When the synchronous technique is implemented, no satellite processor is allowed to proceed to the next iteration until all processors have finished the current iteration, and until all data transfer is completed. After several synchronized iterations, suitable system convergence occurs and the simulation proceeds to the next time step.

Asynchronous relaxation is a technique which was originally developed for the parallel iterative solution of the linear algebraic equation $\mathbf{Ax}=\mathbf{b}$, and the developers of the technique called it *Chaotic Relaxation.* Most of the literature on this technique[9,10] deals with the Jacobi iterative method for solving $\mathbf{Ax}=\mathbf{b}$, but the technique is also applicable to the iterative solution of the nonlinear differential equations used in this thesis. The basis of asynchronous, or chaotic, relaxation is to allow the iterative tasks on the satellite processors to "free run", performing data transfers as soon as updated output values become available. In this way, each satellite processor computes its input vector ($\mathbf{a}$) from the most recent output vectors available from the other system processors, and releases its output vector ($\mathbf{b}$) as soon as it is available, proceeding immediately to the next iteration. All processors continue iterating independently, and are instructed to move to the next time step by the master processor as soon as system convergence occurs.

The synchronous technique has much to recommend it in terms of ease of implementation and debugging, but the efficiency of this technique may suffer somewhat in cases where the iteration cycle times vary widely among the various components in the simulation. In these cases, the asynchronous method may be advantageously employed. A comparison of these techniques is presented in Chapter 4 for an example power system simulation.

CHAPTER 3
DESCRIPTION OF THE MULTIPROCESSOR SIMULATOR

This chapter describes in detail the initial phase of the development of the multiprocessor-based power system simulator. One of the goals of the chapter is to give the reader a basic understanding of how the multiprocessor is used for power system simulation. An overview of the multiprocessing system is followed by a description of the basic tools used to assign tasks to the satellite processors. The procedures involved in developing and executing code on the simulator are then described.

3.1 System Overview

The system chosen for this project is the Intel System 86/380$^{\text{TM}}$, which uses a common-bus architecture based on Multibus$^{\text{TM}}$ hardware (Fig. 3.1). The bus arbitration necessary to coordinate inter-processor and peripheral communications is handled by the Multibus hardware. The simulator peripherals consist of an ADM-36 terminal, a Printronix line printer, and a chassis which houses the Intel storage devices. A 35 megabyte Winchester hard disk system is used as the primary storage device; it carries the operating system file structure and user files. An 8" floppy disk drive is also available for backing up applications software and for updating or revising the RMX–86$^{\text{TM}}$ operating system.

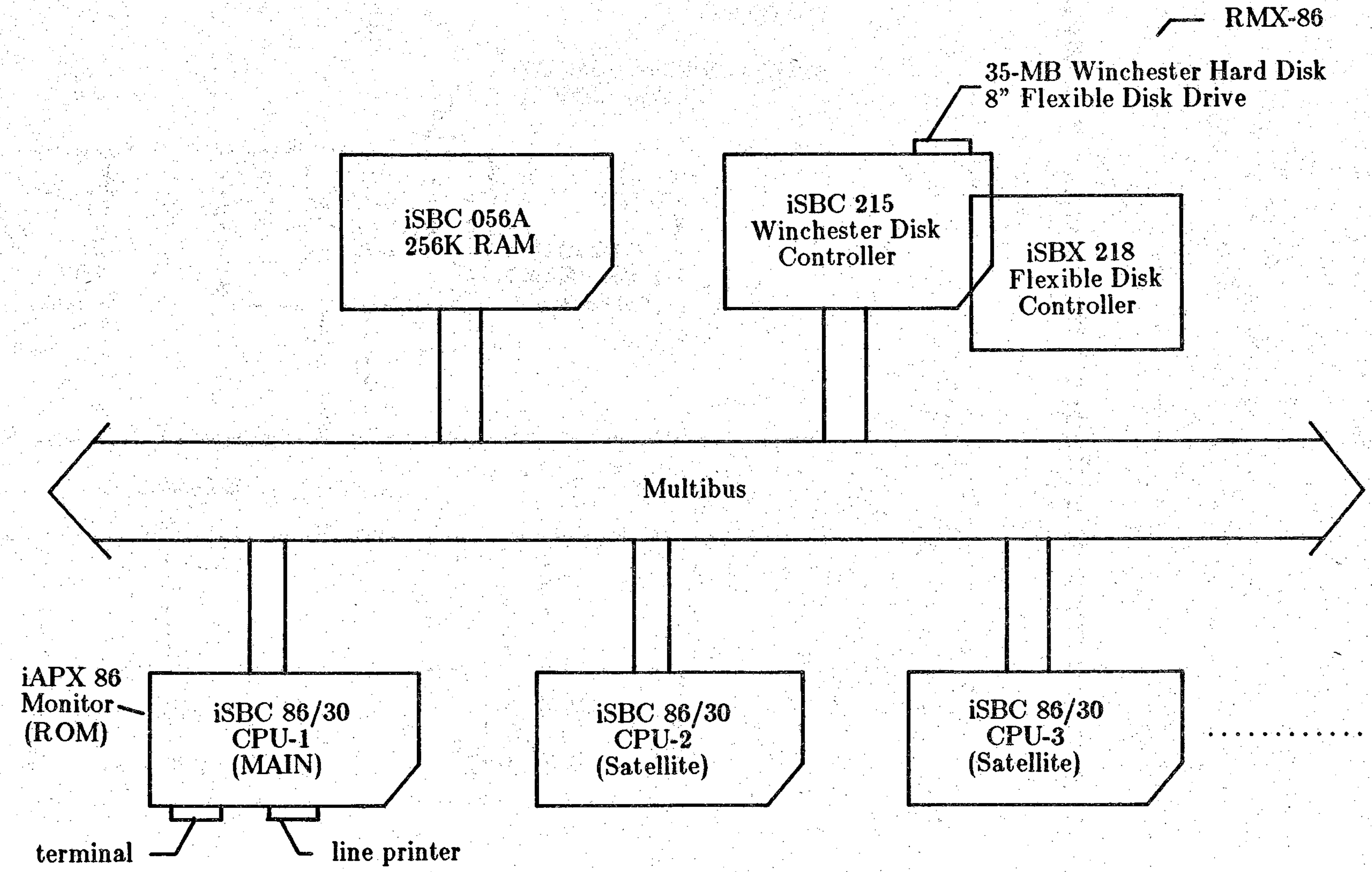


Figure 3.1. Multiprocessor Architecture

A separate Intel chassis houses the multibus hardware, the Winchester disk controller, the central processing units, and the system memory. At the present time, the system processing and memory capabilities consist of three iSBC^{TM} 86/30 Single Board Computer cards with 128K RAM each, and a separate iSBC 056A 256K RAM board for a total system memory of 640K, of which 448K are multibus accessible. Extra slots available on the multibus will allow the number of processor boards to be increased to a maximum of eleven, and a future interface to the analog power system simulator will be provided by an 88/40 A/D and D/A card.

The heart of the system is the 86/30 Single Board Computer card, which is used for both the main and satellite processing units. This board is based on the 8086 16-bit Central Processing Unit, coupled with an 8087 Numeric Data Co-processor which handles all floating-point calculations. The 8086 processor's instruction set is very flexible, and includes special-purpose instructions for iteration control and data string transfers.

Intel's iRMX–86^{TM} Operating System was chosen to provide an environment for program development and execution. This operating system is especially suited for the power system simulator because of its "real-time" interrupt processing features, which may be useful when the analog simulator interface is installed. The RMX-86 operating system provides a system file structure and a group of Human Interface commands, which supply the tools necessary for file manipulations and peripheral usage. An extensive library of System Calls allow access to RMX-86 features from within applications programs. Several other software packages accommodated by RMX-86 include line and screen editors, high-level language compilers (Fortran-86, PL/M-86, C-86), the 8086 assembler ASM-86, and program development tools such as

LINK-86 and LOC-86, which produce executable object code.

Another Intel-supplied software package used extensively during simulation code debugging is the iAPX-86 monitor, which is located in ROM on the main processor board. This monitor allows direct access to the system memory, and it includes a bootstrap loader which is used to load the RMX-86 operating system and the satellite processor simulation routines from the Winchester.

3.2 Simulator Development

Before the Intel system could be used for power system simulation, methods for loading and executing simulation programs on the satellite processors had to be developed. Intel provides a software package (MMX^{TM}) for inter-processor communications, but this package was not intended for high-speed data transfers, and it was felt that the extensive overhead in the MMX package would not allow the desired simulation speeds. In addition, the MMX package required that an RMX-86 nucleus operating system be located on each satellite processor board, limiting the amount of memory available for applications programs. For the purposes of power system simulation, the following multiprocessor features were needed:

- A well defined satellite processor memory structure: common memory areas for input and output data and control flags; local area for simulation code.

- Algorithms for loading code on the satellite processors.

The main processor, with its exclusive access to RMX-86 features, assumes all program development tasks, providing the satellite processors with executable machine code. The algorithms and techniques developed to meet the above requirements are described in subsequent sections.

3.2.1 Satellite Processor Memory

After loading, each satellite processor's (128K) memory space assumes the layout developed in Fig. 3.2. The top 32K of the memory block is located on the system multibus, and is accessible to all other processors in the system. This area holds a command/status region (described later) and a separate data region for the input and output variables used by the component simulation routine. The lower 96K of the memory space, accessible only to the local satellite processor, holds an interrupt pointer table and the simulation object code.

It is helpful at this point to diverge slightly and describe in some detail the simulation code executed by the satellite processor. This code is separated into two parts: an initialization "shell" main module written in assembly language, and a (compiled) high-level language power system component simulation routine written by the user. This format was necessary because of the need for explicitly defined multibus memory addresses for input and output variables and run-time control flags. On entry, the assembly language shell program initializes the 8086 segment registers, calls routines which initialize the floating-point processor and the high-level language run-time environment, and then invokes the power system component simulation routine.

The component simulation is written as a *subroutine* which is called from the assembly language shell. For example,

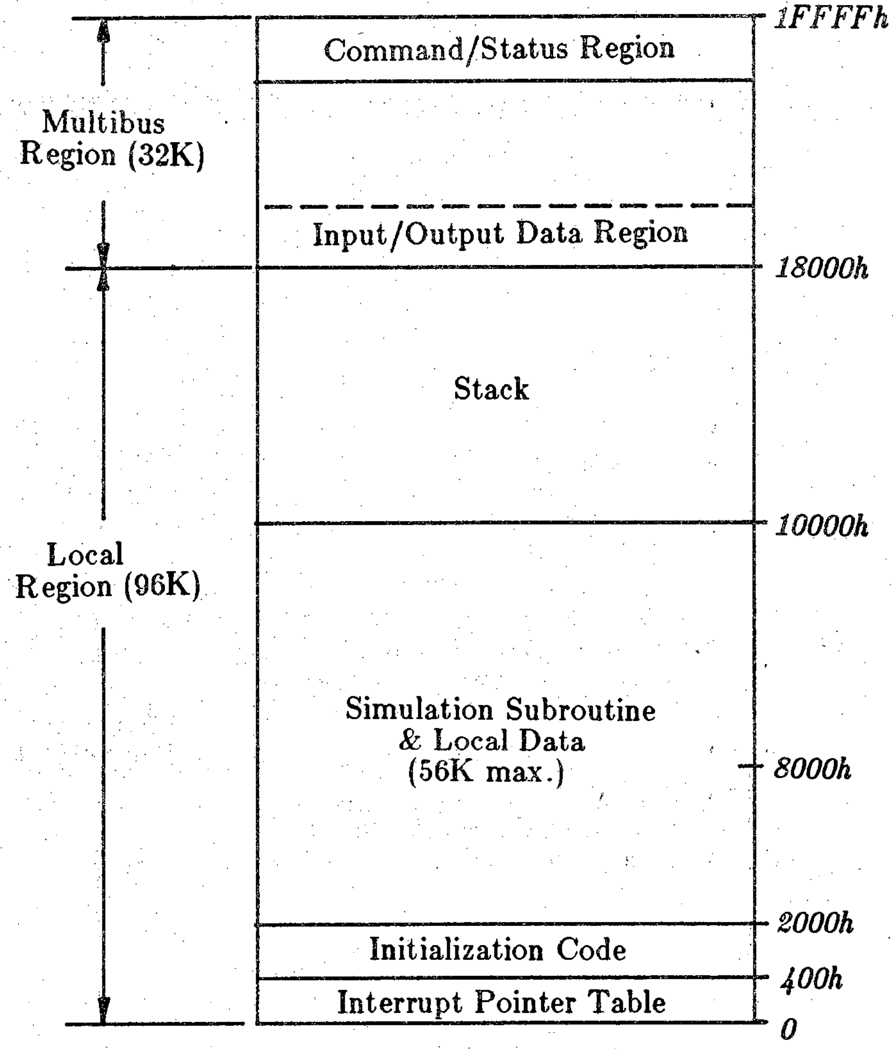


Figure 3.2. Satellite Processor Memory Space

subroutine cpu2(output variable list, input variable list, run-time control flags)

would be the first line of a Fortran component simulation routine intended for processor #2. To satisfy the high-level language conventions for subroutine parameter passing, the assembly language shell program pushes the input/output variable addresses and control flag addresses onto the 8086 run-time stack immediately before calling the simulation routine.

3.2.2 Loading the Satellite Processors

The task of moving simulation code into a satellite processor's memory space is largely transparent to the user, and is accomplished by two routines: a main processor routine called "prime", working in conjunction with a monitor program which resides in ROM on each satellite processor board. Figure 3.3 shows a flowchart of the satellite processor monitor. The monitor processes commands from the command/status memory region (Fig. 3.4) mentioned earlier. This region consists of a command word, a status word, and other memory locations which hold a destination address, a source address, and the length of a memory block in bytes. After being vectored to a "home" location, the processor polls the command word until a legal command appears. When commanded to "feed", the processor moves a block of code (the simulation routine) from a designated multibus area into its local memory space. The processor executes this code when the "cafe" command is received.

The main processor "prime" routine coordinates the loading process. This routine, which is linked to the simulation subroutine and bootstrap loaded by the iAPX-86 monitor, establishes a segment of main processor memory as a "holding area" identical to the lower 64K of the satellite processor's local

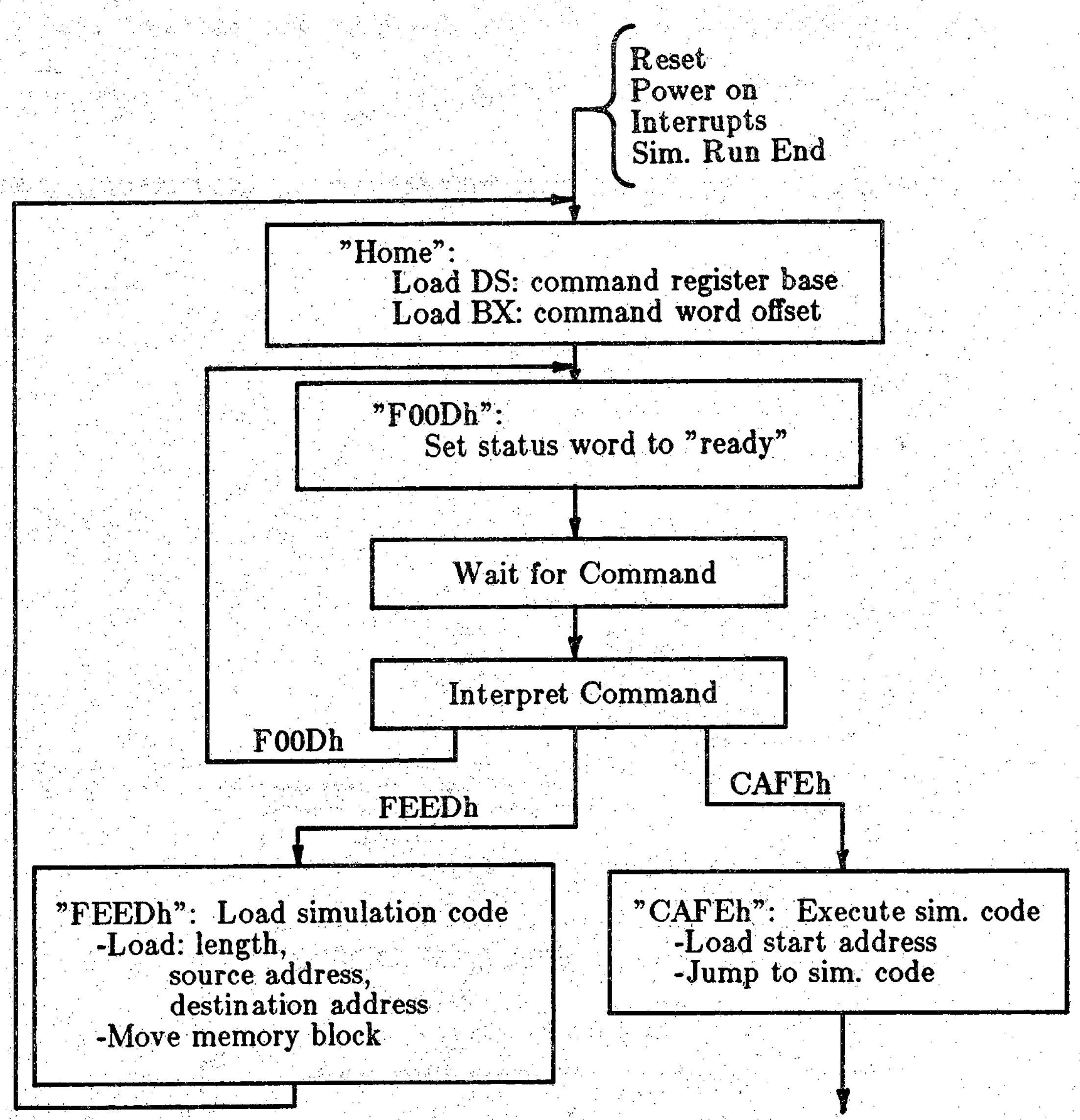


Figure 3.3. Satellite Processor Monitor

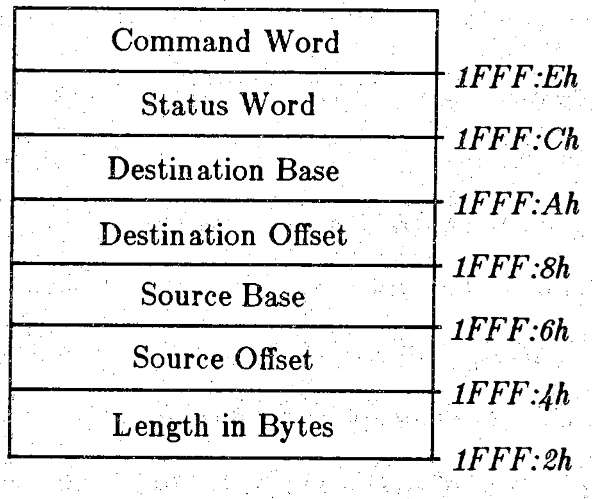


Figure 3.4. Satellite Processor Command/Status Region

memory (Fig. 3.2). The interrupt pointer table, assembly language shell routine, and simulation subroutine are placed at appropriate addresses within this area, and the satellite processor is then commanded to move the entire 64K block into its memory space. The starting address of the code is written into the command/status region of the satellite processor, and the prime routine then returns control to the iAPX-86 monitor. Each of the system's satellite processors is loaded using a separate version of the "prime" routine.

Figure 3.5 shows a memory map of the entire multiprocessing system. This map uses italics to designate the addresses of memory segments accessible only to the local processor; all other addresses designate multibus-accessible memory regions. The physical (hardware) separation of the various memory regions is highlighted in this figure. The memory areas located on the satellite processor boards are shown as "pages" stacked in order, with the multibus region of each satellite processor's memory space fully visible, and the local region partially hidden behind the previous "page" of memory. The map also

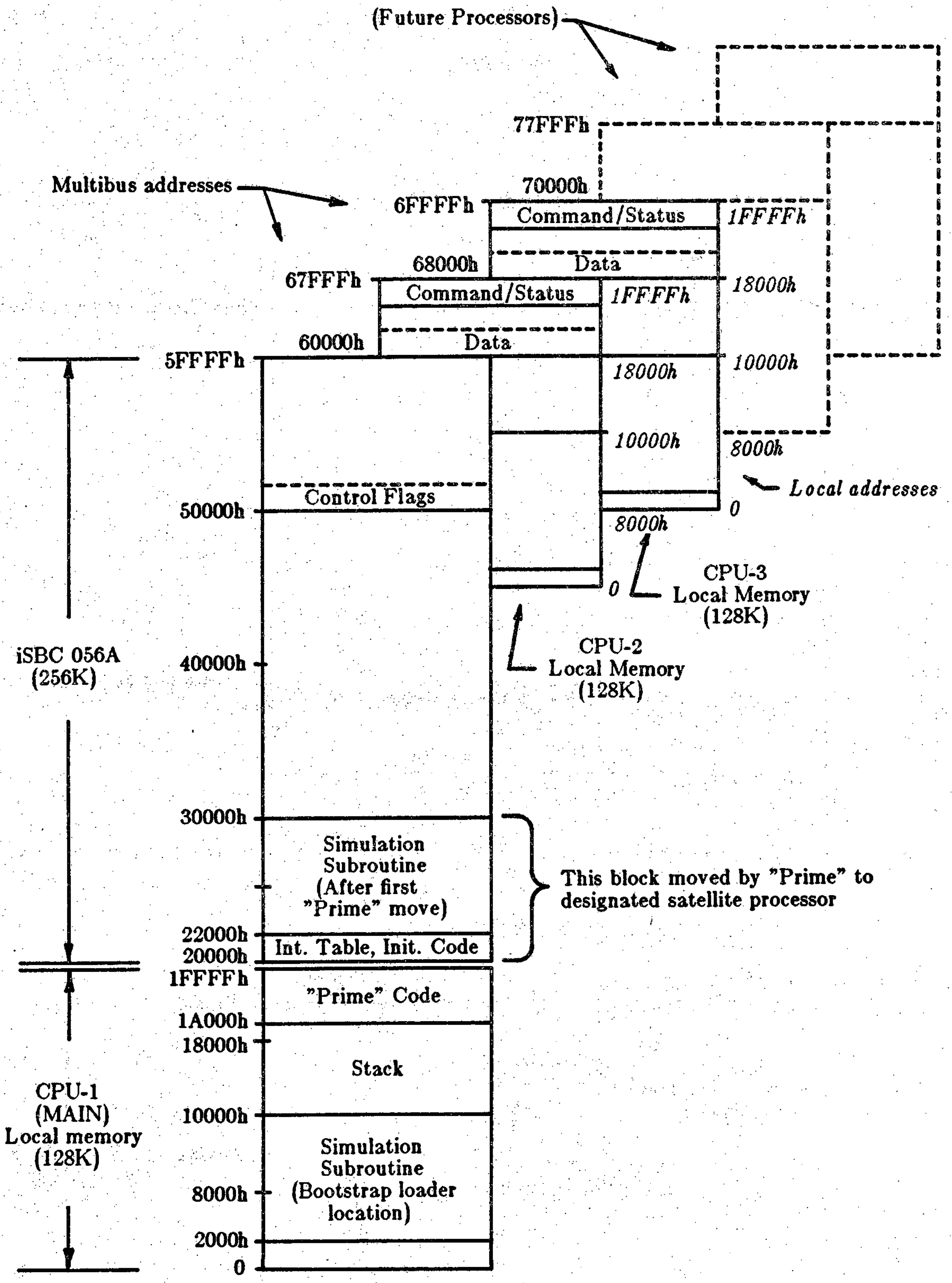


Figure 3.5. Multiprocessor Memory Map

shows the locations of various code segments after the "prime"/simulation routine package is bootstrap loaded.

3.3 Simulating Power Systems on the Multiprocessor

The software developments described above are general tools which allow high-level language programs to be executed on the satellite processors. This section will now present some general guidelines which must be followed for simulation routine development and execution on the multiprocessor.

3.3.1 Simulation Program Development

As mentioned previously, the high-level language routines written for the satellite processors are coded as subroutines which are called from assembly language main modules. Any of the system's high level languages (Fortran-86, PL/M-86, C-86) may be used for the simulation subroutine, as long as the language convention for passing subroutine parameters is followed. Fortran-86 was chosen for the example simulation described in Chapter 4.

Figure 3.6 shows the procedure used to produce executable code for the satellite processors. The LINK-86 utility is used to link together the compiled component simulation routine, the assembled "prime" routine, and the necessary run-time and floating-point libraries. LOC-86 is then used to transform the "load-time locatable" code generated by LINK-86 into (bootstrap loadable) absolute object code.

Figure 3.7 diagrams the process of developing a control routine to be executed by the main processor during the simulation run. The control routine handles such tasks as synchronizing the activities of the satellite processors and providing formatted output of simulation results. Assembly language is

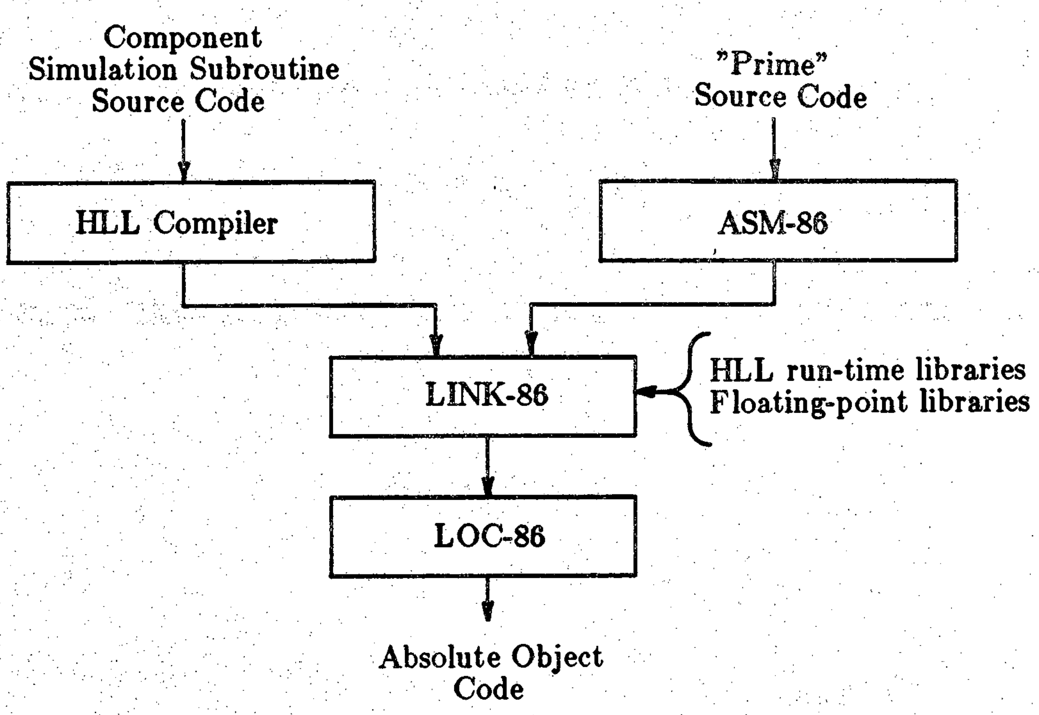


Figure 3.6. Satellite Processor Code Development Process

probably the wisest choice for this program because the iAPX-86 monitor/debugger operates at the machine code level. Any formatted input/output routines are most easily coded in a high level language such as Fortran-86. The assembled control code and compiled code for formatted output are linked to the necessary run-time libraries to produce code which may be loaded on the main processor by the RMX-86 Application Loader.

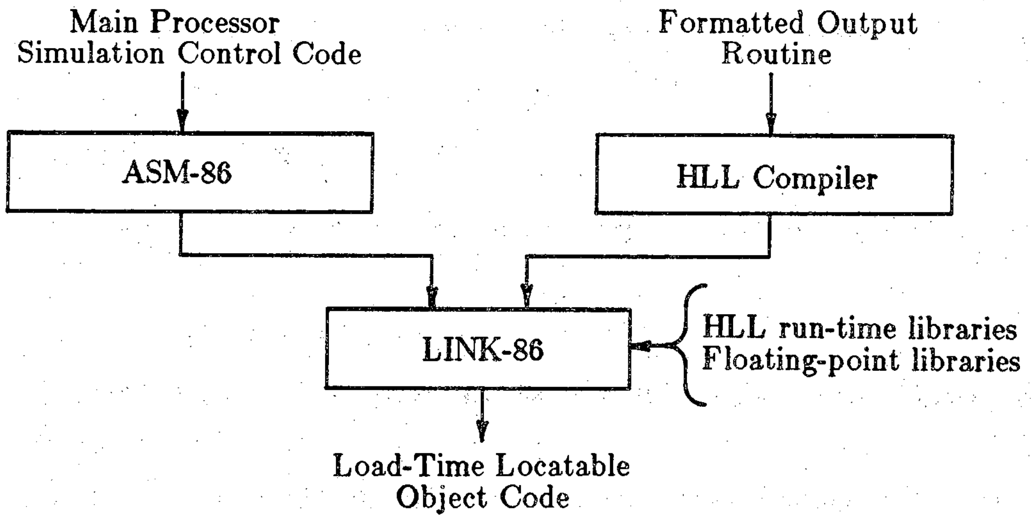


Figure 3.7. Main Processor Code Development Process

3.3.2 Simulation Run Procedure

The following procedure is used to run a simulation on the multiprocessor:

[1] Invoke the iAPX-86 Monitor (exit RMX-86).

[2] Bootstrap load the individual satellite processor loader ("prime") routines.

[3] Reload the RMX-86 operating system.

[4] Load and execute the simulation control routine on the main processor.

The tools and procedures described in this chapter will allow the multiprocessor to simulate a wide variety of power system configurations. An understanding of 8086 assembly language and the basic features of RMX-86 will allow the user to adapt high-level language component simulation routines to the multiprocessing environment. An example of the application of these tools and techniques to a power system simulation is described in Chapter 4.

CHAPTER 4
SIMULATION OF AN INDUCTION MOTOR DRIVE SYSTEM

A three-processor power system simulation was developed to evaluate the multiprocessing techniques described in Chapter 3. The primary goals of this test were twofold: to develop a good understanding of how to adapt power system simulation to the multiprocessing environment; and to get a rough idea of the capabilities of the multiprocessor in terms of computational speed and efficiency. An induction motor drive system was chosen for the initial power system studies on the multiprocessor.

4.1 Background

In many applications, it is necessary to be able to control the amplitude and frequency of the stator voltages applied to an induction machine. By controlling these parameters, it is possible to produce usable machine torque at a variety of rotor speeds. It is often necessary to apply voltages of relatively low amplitude and low frequency to start a large induction machine. This technique avoids the problem of large stator currents which can occur when a machine is accelerated from stall by sudden application of rated stator voltage and frequency. The induction motor drive system studied uses a controlled rectifier/inverter design to achieve the desired stator voltage and frequency control.

Rectifier-inverter systems are being used extensively in many present-day power system applications, and their associated control systems are a source of interesting problems which are being studied at the present time. Previous research has shown that pulse-width modulation (PWM) inverters can be developed to produce a smooth, nonpulsating machine torque at various rotor speeds, and extensive work has been done to reduce the harmonic losses associated with these drives. The complexity of the resulting control systems has prompted the use of microprocessors[11,12] in the design of PWM controls. This approach enables the introduction of more sophisticated PWM techniques, by allowing a variety of voltage waveform patterns (located in ROM) for various operating speeds and load torques. Microprocessors are also being applied extensively in the design of HVDC convertor control systems.

It seems inevitable that conventional analog/hybrid simulation techniques will not be able to keep pace with the increasing sophistication of these control systems. Purely digital techniques will soon be necessary to achieve accurate simulations of both machine drives and HVDC convertor control systems.

Factors such as these led to the choice of the induction motor drive system for the initial multiprocessor work. A multiprocessor simulation for this system was developed using the three available system processors: one satellite processor modeled the drive system component; the other satellite processor simulated the induction machine component; and the main processor assumed overall control of the simulation. The CCM formulation (equation 2.3) for this simulation study is trivial:

$$\begin{bmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix} \qquad (4.1)$$

This equation merely states that the output vector for component #1 (the drive system) serves as the input vector for component #2 (the induction machine), and vice versa. The models used for these components, and the input and output vectors employed, will be described next.

4.2 Drive System Model

Figure 4.1 shows a simplified diagram of the system studied. The system consists of a three phase 60Hz power source, a phase-controlled rectifier and associated filter, a self-commutated inverter, and a three-phase 6-pole symmetrical induction machine. For the purposes of the multiprocessor simulation, the three phase source is considered an infinite bus, and the rectifier is therefore simplified to a variable DC source behind an equivalent impedance r_c. The controlled rectifier output voltage V_r is modeled by the equation

$$V_r = V_o e_c - r_c I_r \tag{4.2}$$

where V_o is a base value of the dc source, and I_r is the filter input current. It can be shown[11] that if the slip frequency ω_s is held constant, then a constant machine torque is developed by holding the stator voltage/frequency ratio constant. The e_c factor in equation 4.2 defines this proportionality:

$$e_c = K_v \cdot \omega \tag{4.3}$$

where K_v is a constant. The dynamics of the filter are obtained by applying Kirchoff's voltage and current laws to the filter circuit, resulting in the following linear differential equations:

$$\dot{I}_r = -\frac{R_f}{L_f} I_r + \frac{1}{L_f}(V_r - V_i) \tag{4.4}$$

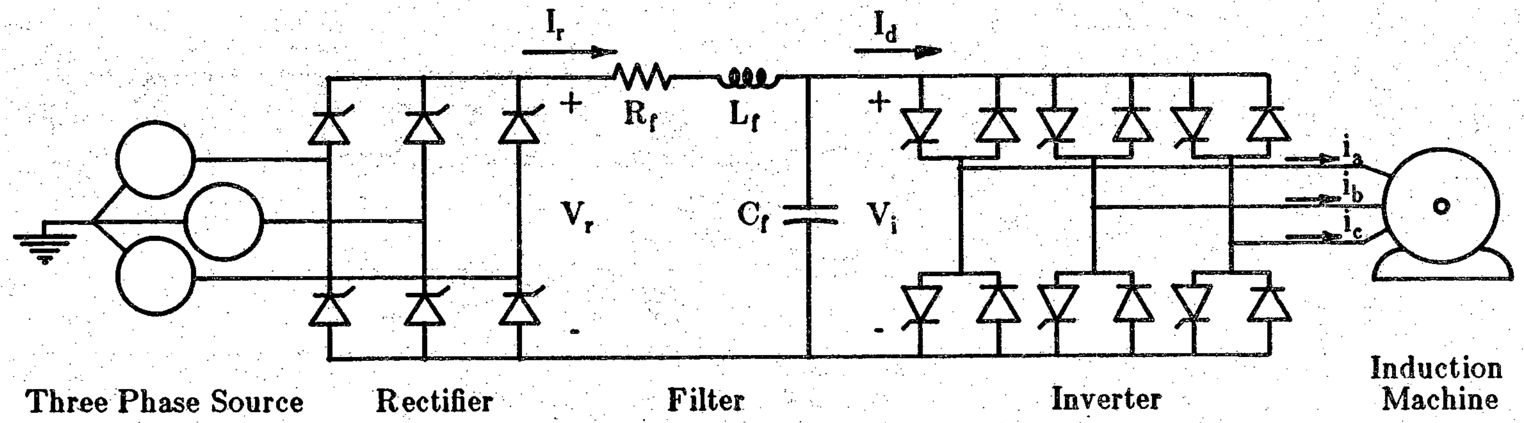


Figure 4.1. Induction Motor Drive System

$$\dot{V}_i = \frac{1}{C_f}(I_r - I_d) \tag{4.5}$$

The inverter thyristor firing rate is determined by the feedback control system shown in Fig. 4.2.

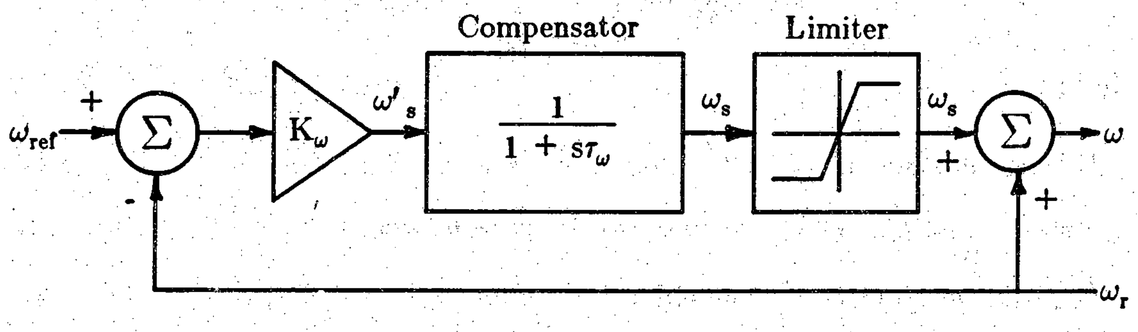


Figure 4.2. Drive Control System

The control system uses the machine's rotor speed, ω_r, to determine an uncompensated slip frequency, ω'_s, given by the equation

$$\omega'_s = K_\omega(\omega_{ref} - \omega_r) \ . \tag{4.6}$$

The ω_{ref} term in this equation is a set point to which the machine is to be accelerated, and K_ω is a gain constant in the control system. In an attempt to tune the control system to the mechanical dynamics of the machine rotor, the transfer function

$$\omega_s = \frac{1}{1+s\tau_\omega}\omega'_s \tag{4.7}$$

was used to apply a frequency dependent gain to the uncompensated slip frequency ω'_s. This transfer function may be represented in the time domain by the differential equation

$$\dot{\omega}_s = \frac{1}{\tau_\omega}(\omega'_s - \omega_s) \tag{4.8}$$

where the compensator characteristic is set as needed by adjusting the time constant τ_ω. The slip frequency is limited to a certain range of values ($\pm$ 25 rad/sec for the system studied), and the resulting frequency of the applied stator voltages, ω, is determined by the equation

$$\omega = \omega_s + \omega_r \ . \tag{4.9}$$

Because of the variable-frequency design of the drive system, it was not practical to choose a fixed time step for the numerical integration of state variables. In the simulated inverter system, the voltage output waveform is derived from a stored pole pattern, with the thyristor status defined at regular angular intervals. For this reason, the simulation is made to operate with a fixed angle step $d\theta$. The corresponding time step, dt, is then determined by the relationship

$$dt = \frac{d\theta}{\omega} \ . \tag{4.10}$$

The drive system simulation routine performs this calculation, and uses the results in its predictor/corrector integrations. The time step dt is also included in the drive system output vector for the integration of the machine state variables. The input vector ($\mathbf{a}_1$) for the drive system simulation includes the machine stator currents and the machine rotor speed ω_r, and the output vector ($\mathbf{b}_1$) consists of the stator voltages and the time step dt. For reasons which are explained more fully in the machine model development, the drive system "abc" variables are transformed to "qd" stationary reference frame variables according to the relationships

$$i_{abcs} = (\overline{K}_s)^{-1} i_{qds} \tag{4.11}$$

$$V_{qds} = \overline{K}_s V_{abcs} \tag{4.12}$$

where the stationary reference frame transformation matrices[13] are

$$\overline{K}_s = \frac{2}{3}\begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & -\frac{\sqrt{3}}{2} & \frac{\sqrt{3}}{2} \end{bmatrix} \tag{4.13}$$

and

$$(\overline{K}_s)^{-1} = \begin{bmatrix} 1 & 0 \\ -\frac{1}{2} & -\frac{\sqrt{3}}{2} \\ -\frac{1}{2} & \frac{\sqrt{3}}{2} \end{bmatrix} \tag{4.14}$$

This transformation of variables could be performed by either the drive system simulation routine or the induction machine simulation routine. In this study, it was convenient to locate the variable transformation in the drive system simulation so that the number of variables transferred between processors could be reduced.

4.3 Induction Machine Model

As is the case with many power system components, it is convenient to use reference frame theory as a basis for developing an induction machine model suitable for computer simulation. When expressed in machine (abc) variables, the stator and rotor voltage equations for a symmetrical induction machine contain nonlinear terms which arise from time-varying mutual inductances between the stator and rotor windings. These nonlinearities may be eliminated by transforming the stator and rotor machine variables to a frame of reference which rotates at an arbitrary angular velocity. This change of variables is accomplished by a pair of trigonometric transformation matrices[13]. The resulting equations express all voltages, currents, and flux linkages per second in terms of an orthogonal "qd" set, and a "0" quantity which accounts for any imbalances in the machine variables. When saturation effects are ignored, and flux linkages per second and rotor speed are used as the state variables, the following arbitrary reference frame dynamic model arises[13]:

$$\dot{\Psi}_{qs} = \omega_b[V_{qs} - \frac{\omega}{\omega_b}\Psi_{ds} + \frac{r_s}{X_{ls}}(\Psi_{mq} - \Psi_{qs})] \tag{4.15}$$

$$\dot{\Psi}_{ds} = \omega_b[V_{ds} + \frac{\omega}{\omega_b}\Psi_{qs} + \frac{r_s}{X_{ls}}(\Psi_{md} - \Psi_{ds})] \tag{4.16}$$

$$\dot{\Psi}_{0s} = \omega_b[V_{0s} - \frac{r_s}{X_{ls}}\Psi_{0s}] \tag{4.17}$$

$$\dot{\Psi}'_{qr} = \omega_b[V'_{qr} - (\frac{\omega-\omega_r}{\omega_b})\Psi'_{dr} + \frac{r'_r}{X'_{lr}}(\Psi_{mq} - \Psi'_{qr})] \tag{4.18}$$

$$\dot{\Psi}'_{dr} = \omega_b[V'_{dr} + (\frac{\omega-\omega_r}{\omega_b})\Psi'_{qr} + \frac{r'_r}{X'_{lr}}(\Psi_{md} - \Psi'_{dr})] \tag{4.19}$$

$$\dot{\Psi}'_{0r} = \omega_b[V'_{0r} - \frac{r'_r}{X'_{lr}}\Psi'_{0r}] \tag{4.20}$$

$$\dot{\omega}_r = \frac{\omega_b}{2H}(T_e - T_L) \tag{4.21}$$

where

$$\Psi_{mq} = X_{aq}\left(\frac{\Psi_{qs}}{X_{ls}} + \frac{\Psi'_{qr}}{X'_{lr}}\right) \tag{4.22}$$

$$\Psi_{md} = X_{ad}\left(\frac{\Psi_{ds}}{X_{ls}} + \frac{\Psi'_{dr}}{X'_{lr}}\right) \tag{4.23}$$

and

$$X_{aq} = X_{ad} = \left(\frac{1}{X_m} + \frac{1}{X_{ls}} + \frac{1}{X'_{lr}}\right)^{-1} \tag{4.24}$$

and

ω_b	= base angular velocity
ω_r	= rotor angular velocity
ω	= reference frame angular velocity
r_s	= stator resistance
X_{ls}	= stator leakage reactance
r'_r	= rotor resistance (referred to stator)
X'_{lr}	= rotor leakage reactance (referred to stator)
X_m	= magnetizing reactance
H	= rotor inertia constant (seconds)
T_L	= load torque (per unit)
T_e	= electromagnetic torque (per unit)

The s and r subscripts in these equations are used to distinguish stator and rotor quantities; the prime ($'$) superscript indicates rotor quantities referred to the stator windings via the machine winding ratio. The stator currents i_{qd0s} and rotor currents i'_{qd0r} are expressed as linear combinations of state variables:

$$i_{qs} = \frac{1}{X_{ls}}(\Psi_{qs} - \Psi_{mq}) \tag{4.25}$$

$$i_{ds} = \frac{1}{X_{ls}}(\Psi_{ds} - \Psi_{md}) \tag{4.26}$$

$$i_{0s} = \frac{1}{X_{ls}}(\Psi_{0s}) \tag{4.27}$$

$$i'_{qr} = \frac{1}{X'_{lr}}(\Psi'_{qr} - \Psi_{mq}) \tag{4.28}$$

$$i'_{dr} = \frac{1}{X'_{lr}}(\Psi'_{dr} - \Psi_{md}) \tag{4.29}$$

$$i'_{0r} = \frac{1}{X'_{lr}}(\Psi'_{0r}) \tag{4.30}$$

The electromagnetic torque is then expressed (in per unit) as

$$T_e = \Psi_{ds} i_{qs} - \Psi_{qs} i_{ds} \ . \tag{4.31}$$

Equations 4.15 - 4.31 represent a fully modeled induction machine, viewed from a reference frame rotating at an arbitrary angular velocity ω, for both unbalanced and balanced conditions. For the purposes of this simulation exercise, some assumptions may be introduced at this point to reduce the size and complexity of the model. First, the rotor circuit is assumed to be completely internal to the machine. This assumption eliminates the external rotor voltages V'_{qd0r}; and the rotor currents (equations 4.28 - 4.30) are no longer of interest and are left out of the simulation. Second, balanced three phase conditions are assumed for the simulation, eliminating all "zero" quantities (equations 4.17, 4.20, and 4.27). Finally, the discontinuous nature of the voltages produced by the drive system prompted the use of the stationary reference frame (ω=0)[13,14]. With these assumptions, the electrical dynamics equations (4.15 - 4.20) reduce to the following set of equations:

$$\dot{\Psi}_{qs} = \omega_b[V_{qs} + \frac{r_s}{X_{ls}}(\Psi_{mq} - \Psi_{qs})] \tag{4.32}$$

$$\dot{\Psi}_{ds} = \omega_b[V_{ds} + \frac{r_s}{X_{ls}}(\Psi_{md} - \Psi_{ds})] \tag{4.33}$$

$$\dot{\Psi}'_{qr} = \omega_r \Psi'_{dr} + \frac{\omega_b r'_r}{X'_{lr}}(\Psi_{mq} - \Psi'_{qr}) \tag{4.34}$$

$$\dot{\Psi}'_{dr} = -\omega_r \Psi'_{qr} + \frac{\omega_b r'_r}{X'_{lr}}(\Psi_{md} - \Psi'_{dr}) \tag{4.35}$$

The mechanical dynamics equation (4.21) remains unchanged for this representation, as do the torque equation (4.31) and the stator current equations (4.25 and 4.26). It is interesting to note that the nonlinearities in this model appear as products of state variables in the rotor electrical dynamics equations (4.34 and 4.35), and in the mechanical dynamics equation (4.21). To see the nonlinearities in the mechanical dynamics equation, it is necessary to decompose the electromagnetic torque equation (4.31) into its state variable representation by making appropriate substitutions for i_{qs} and i_{ds}. The input vector ($\mathbf{a}_2$) for the induction machine simulation includes the stator voltages V_{qds} and the time step dt; the output vector ($\mathbf{b}_2$) consists of the stator currents i_{qds} and the rotor speed, ω_r.

4.4 Implementation on the Multiprocessor

The induction motor drive system simulation was an extension of the conceptual ideas of Chapter 2 and the multiprocessing techniques discussed in Chapter 3. Two separate simulation packages were prepared, one using synchronous relaxation and the other using the asynchronous (or chaotic) relaxation technique described in Chapter 2. As discussed in Chapter 3, the component simulation algorithms were written as high-level language (Fortran-86) subroutines, and the main processor control algorithms were written in the 8086 assembly language ASM-86. Physically, the control algorithms for the separate simulation packages were executed by the master processor, cpu #1 (refer to Fig. 3.5). The drive system was simulated on cpu #2, and the induction machine algorithm was located on cpu #3.

Flowcharts of the routines used to simulate the drive system and the induction machine are presented in Figs. 4.3 and 4.4, respectively. In both of these programs, a relaxation method, using the Euler/trapezoid technique presented in Chapter 2, is used to solve for the component state variables. For the drive system algorithm, these variables are the DC filter quantities V_i and I_r, and the slip frequency ω_s; the induction machine state variables are the flux linkages per second Ψ_{qs}, Ψ_{ds}, Ψ'_{qr}, and Ψ'_{dr}, and the rotor speed, ω_r. Both the drive system program and the induction machine program are direct applications of the relaxation algorithm presented in Chapter 2 (Fig. 2.4), although considerable amounts of extra code were incorporated in the drive system simulation routine for controlling the thyristor gating pulses to produce the desired six-step three phase voltage output waveforms.

One source of performance degradation in multiprocessing systems is the competition among processors for access to the system's common memory areas. To limit the amount of common memory area access, the input/output data read and write functions occur at only two points in each simulation algorithm; one for the predictor integration and one for the corrector integration. At other times, the input and output vectors are stored in secondary local memory locations for calculation purposes following read

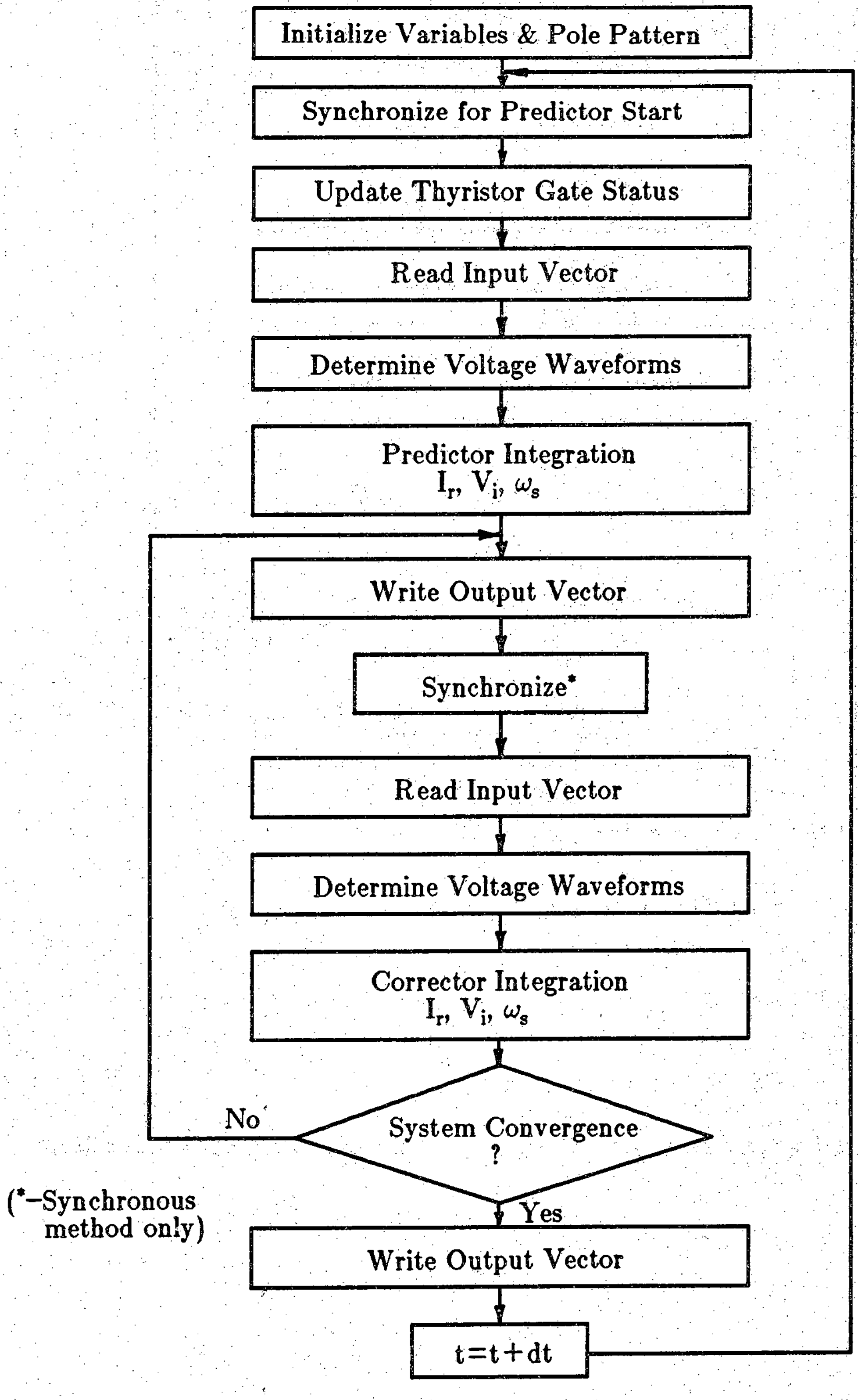

Initialize Variables & Pole Pattern
Synchronize for Predictor Start
Update Thyristor Gate Status
Read Input Vector
Determine Voltage Waveforms
Predictor Integration I_r, V_i, ω_s
Write Output Vector
Synchronize*
Read Input Vector
Determine Voltage Waveforms
Corrector Integration I_r, V_i, ω_s
System Convergence ?
No
Yes
Write Output Vector
t=t+dt
(*–Synchronous method only)

Figure 4.3. Drive System Simulation Flowchart

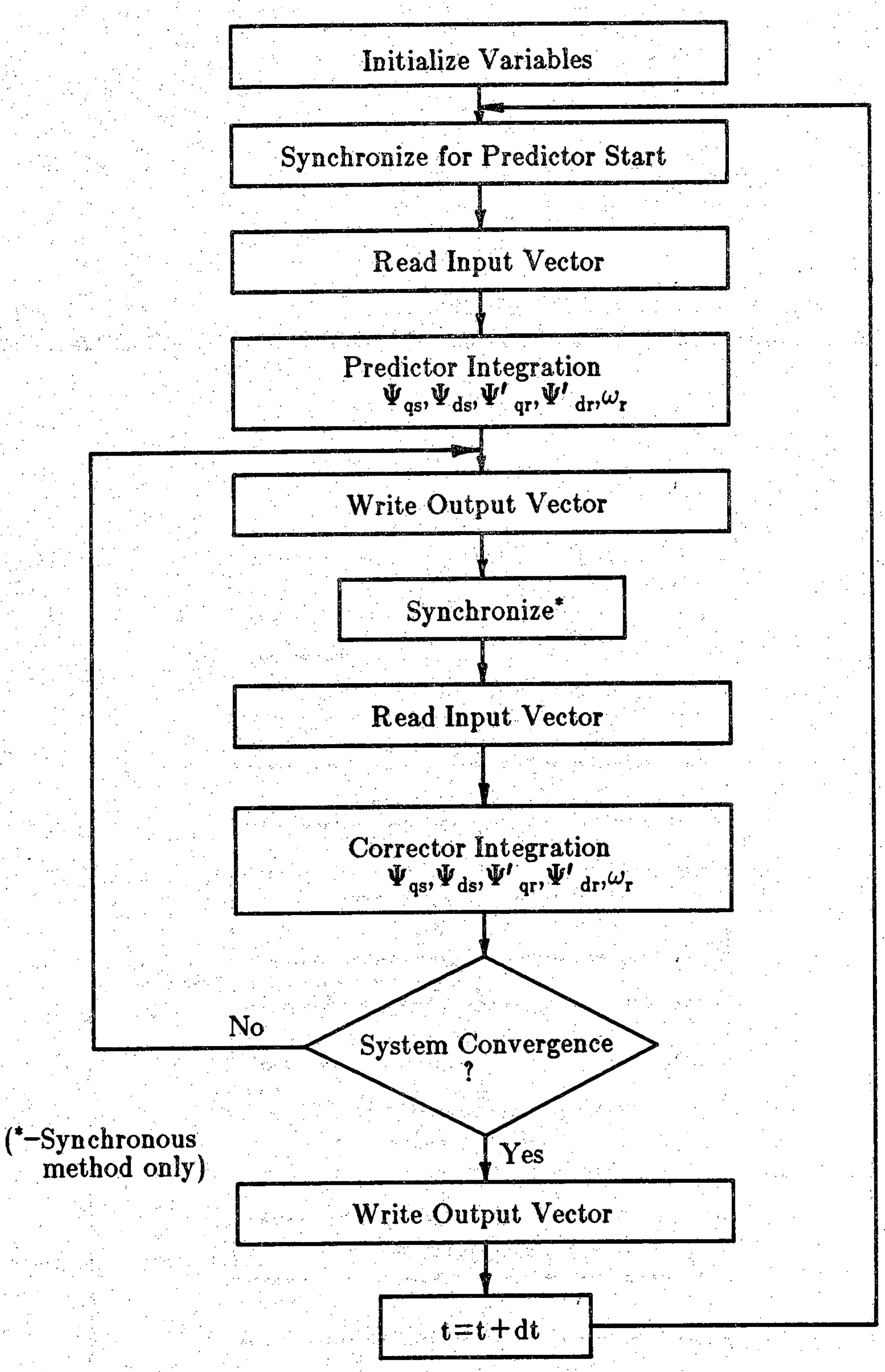


Figure 4.4. Induction Machine Simulation Flowchart

operations (for input data) and prior to write operations (for output data). As shown in Fig. 3.4, the input and output vectors for the drive system start at multibus memory address 6000:0, and the corresponding area for the induction machine simulation starts at 6800:0. Each real variable within these memory areas occupies four bytes of memory.

Two separate main processor control algorithms were developed; one for the synchronous relaxation technique (Fig. 4.5), and one for the asynchronous (chaotic) method (Fig. 4.6). As discussed in Chapter 2, the synchronous method is characterized by a tight control on the satellite processor iterations, while the asynchronous method allows the satellite processors to run free, with the master processor performing data transfers whenever new data is available. In both cases, the satellite processors are synchronized at the beginning of each time step, prior to predictor integration. The synchronous method uses an extra synchronization point in the corrector loop of each satellite processor algorithm (refer to Figs 4.3 and 4.4); this point is omitted for the asynchronous method.

Except for a formatted (screen or disk) output subroutine coded in Fortran-86, the main processor routines were coded in 8086 assembly language. These routines control the satellite processors (cpu #2 and cpu #3) via multibus-located run-time control flags. The flags appear to the Fortran component simulation algorithms (executed on the satellite processors) as integer data types; they appear to the control algorithm as words (two bytes) of memory located in an area starting at multibus address 5000:0 (refer to Fig. 3.5). For example, the synchronous relaxation control algorithm operates on the following set of control flags:

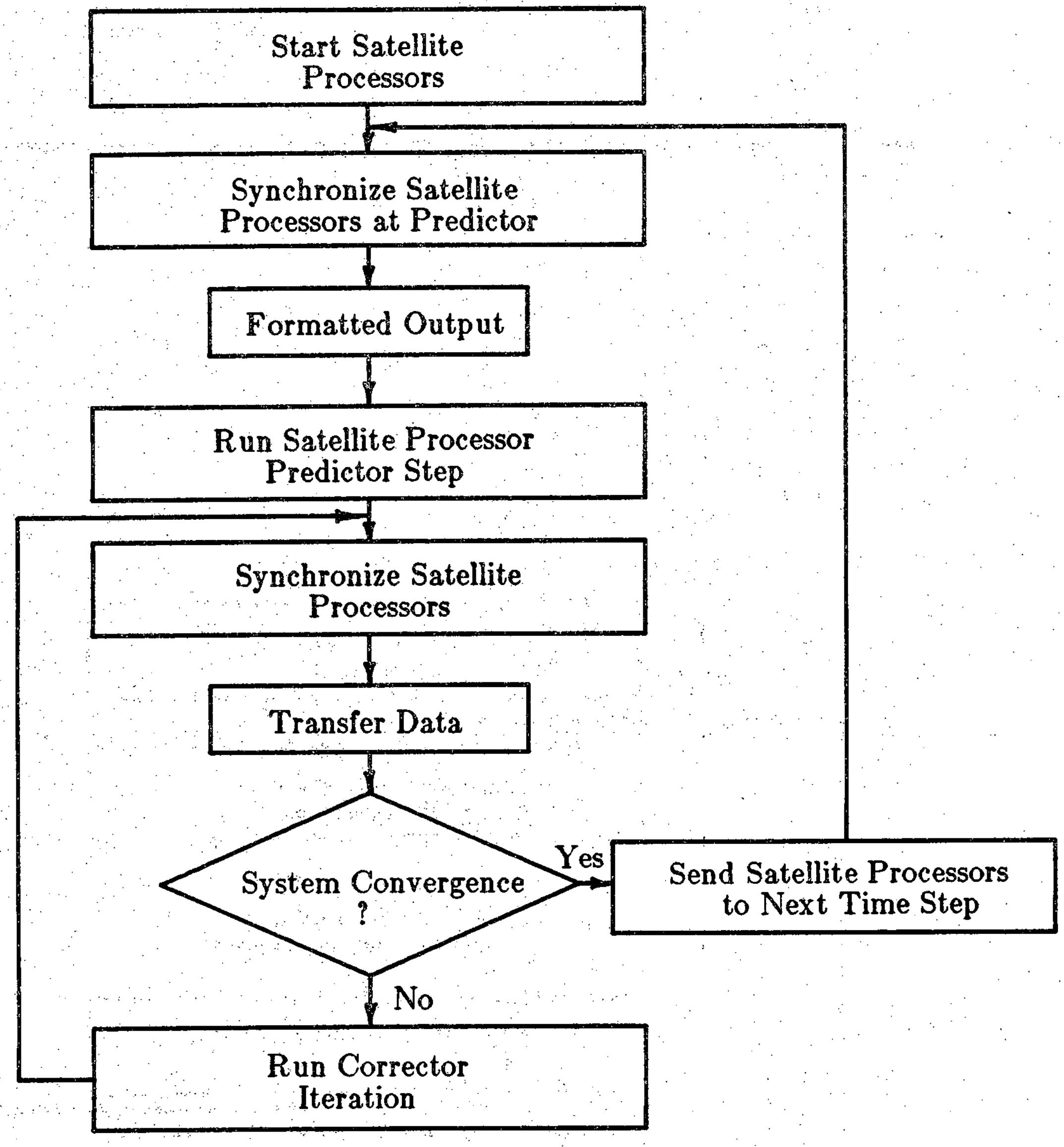

Start Satellite Processors
Synchronize Satellite Processors at Predictor
Formatted Output
Run Satellite Processor Predictor Step
Synchronize Satellite Processors
Transfer Data
System Convergence ?
Yes
Send Satellite Processors to Next Time Step
No
Run Corrector Iteration

Figure 4.5. Synchronous Relaxation Control Algorithm

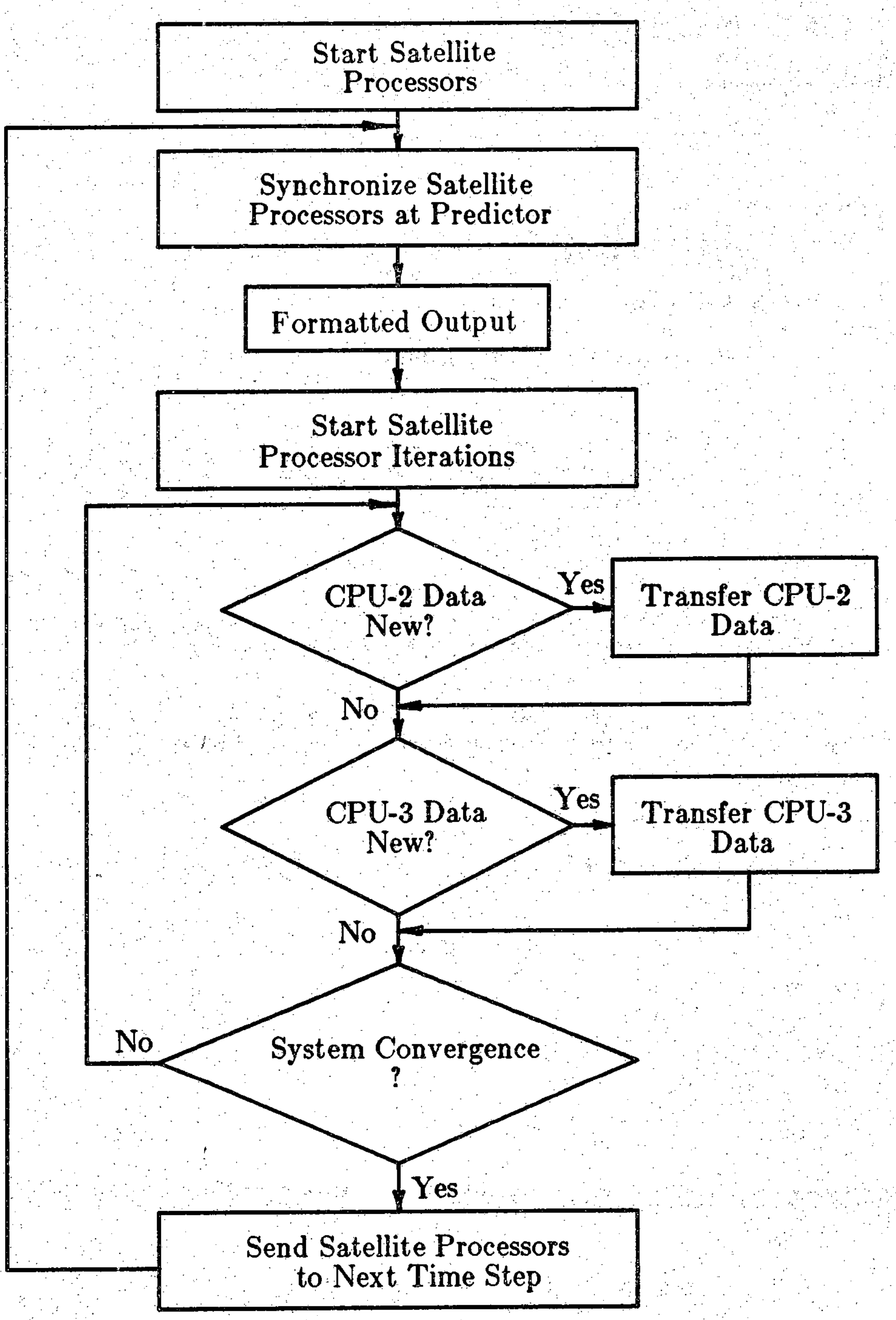

Start Satellite Processors
Synchronize Satellite Processors at Predictor
Formatted Output
Start Satellite Processor Iterations
CPU-2 Data New?
Yes
Transfer CPU-2 Data
No
CPU-3 Data New?
Yes
Transfer CPU-3 Data
No
System Convergence ?
No
Yes
Send Satellite Processors to Next Time Step

Figure 4.6. Asynchronous Relaxation Control Algorithm

- *rdy2 & rdy3:* set by satellite processors after they reach a synchronization point.
- *go2 & go3:* instructs satellite processors to resume execution.
- *conv2 & conv3:* set by each satellite processor after its simulation algorithm converges.
- *pred:* instructs satellite processors to proceed to next time step predictor.
- *done:* identifies end of simulation run.

The "go" and "rdy" flags are used for synchronization of the system processors. The main processor determines overall system convergence by periodically checking the "conv" flags controlled by the individual satellite processors. When all component simulations have converged, the "pred" flag is set by the master processor, sending the satellite processors to the next time step.

The asynchronous method required two additional control flag sets:

- *mbox2 & mbox3:* set by satellite processors to announce updated output data.
- *bus2 & bus3:* multibus data arbitration - when set, denies other processors access to common data area.

The main processor is notified of satellite processor data updates via the "mbox" flags. Special multibus arbitration coding was included in the asynchronous relaxation algorithm to prevent the scenario of one processor writing into a real data location (four bytes of memory) while another is reading from that location. This arbitration was accomplished by the "bus" flags above.

As discussed in Chapter 3, the need for absolute addressing of input/output data and control flags was satisfied by writing each component simulation routine as a Fortran-86 subroutine which is called from an assembly language main module. This main module, part of the "prime" package described in Chapter 3, initializes the 8086 segment registers and calls subroutines which initialize the Fortran-86 environment and the 8087 numeric data processor. The pre-defined addresses of the input/output variables and control flags are pushed onto the 8086 stack prior to invoking the simulation subroutine. The input/output variables and control flags then appear as parameters in the Fortran-86 subroutine statement. For example,

```
subroutine cpu2(vqs,vds,dt,iqs,ids,wr,go2,rdy2,conv2,pred,
          done)
```

is the first line of the drive system component simulation routine. In the simulation routine, the input/output variables are treated as "real" data types, and the control flags are treated as "integer" types.

The procedure used to develop executable code for the satellite processors followed the outline presented in Fig. 3.6. After a suitable Fortran-86 simulation routine was developed, the "prime" package was modified for the desired input/output variables and control flags. Absolute object code for the satellite processors was then produced by the LINK-86 and LOC-86 utilities. A similar procedure was followed for the assembly language control code executed by the main processor (refer to Fig. 3.7).

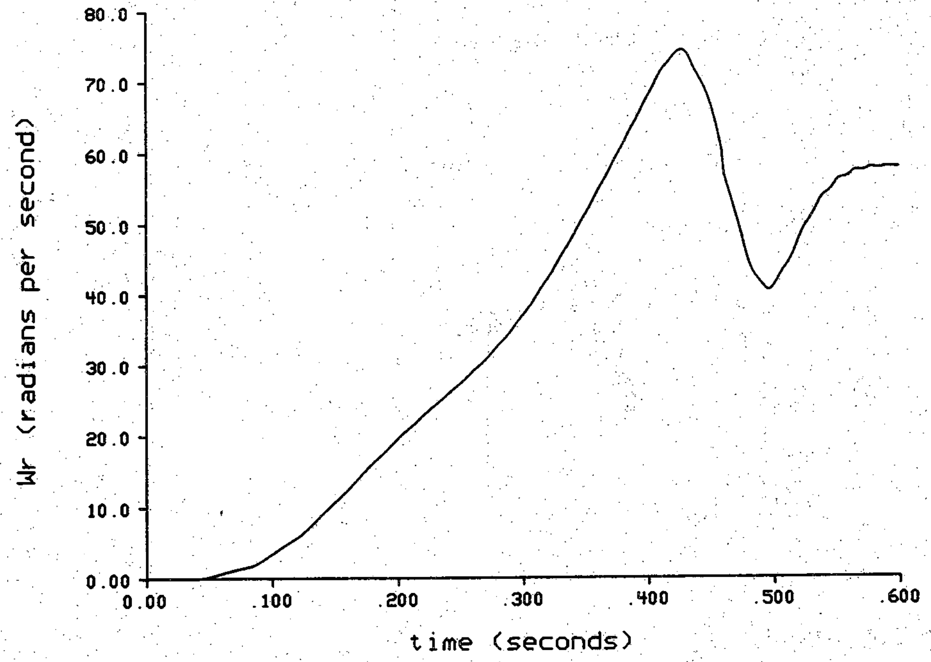
4.5 Results

To verify the performance of the multiprocessor, its simulation run results were compared to the results of a similar simulation executed by a VAX-11/780. The simulation performed was a free acceleration of the induction machine to a reference speed (ω_{ref}) of 63 radians per second (10Hz), approximately 0.6 seconds of real time. The solution trajectories generated by the VAX for rotor speed and per unit torque are presented in Fig. 4.7, and the phase voltage and current waveforms are shown in Fig. 4.8. The data generated by the multiprocessor were identical to the VAX results for both the synchronous and asynchronous techniques.

The voltage and current waveforms of Fig. 4.8 show the expected increases in frequency and amplitude as the machine accelerates, and they also illustrate the voltage amplitude and frequency compensation performed by the drive control system as the rotor overshoots the reference speed. The effect of this compensation also appears in the torque/rotor speed trajectory (Fig. 4.7), where the rotor deceleration is accompanied by negative values of electromagnetic torque.

In both the multiprocessor simulation and the VAX simulation, the solutions for the quantities of interest were calculated using an angular step of one degree. As mentioned previously, this simulation technique was used because of the design of the stored-pattern inverter used in the drive system. With this simulation technique, increases in the frequency of the applied stator voltages are reflected as decreases in the time increment dt, while the angular displacement remains constant. The simulation run data showed that 1870 1° angular steps were required for a 0.6 second acceleration of the machine; this figure of 1870 steps was used as a basis for measurements of computational speed and efficiency. The simulations were not carried beyond 1870 steps because of the potential distortion of results due to accumulation of numerical roundoff error.

The most important criterion for the evaluation of the multiprocessor's performance is, of course, the gain in simulation speed for the multiprocessor



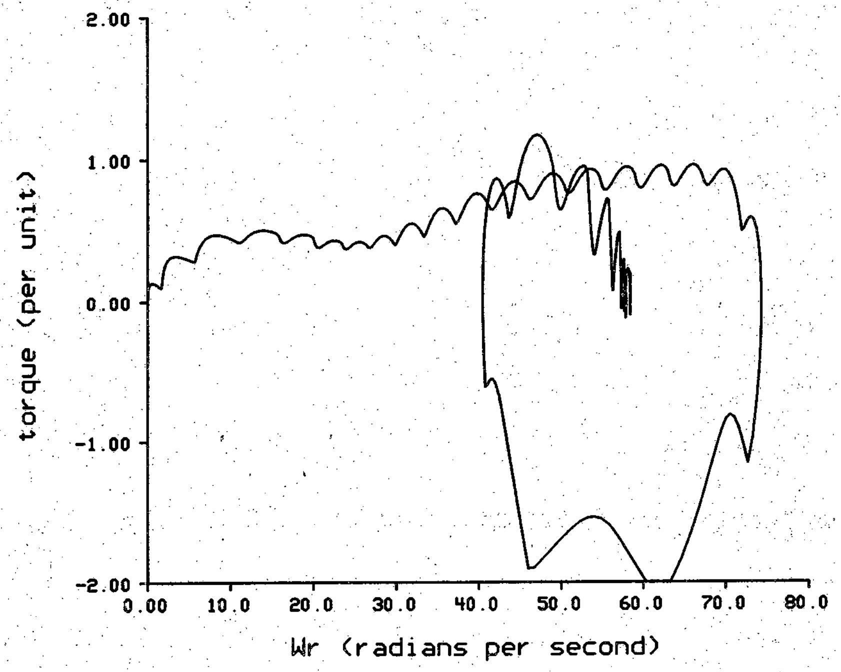
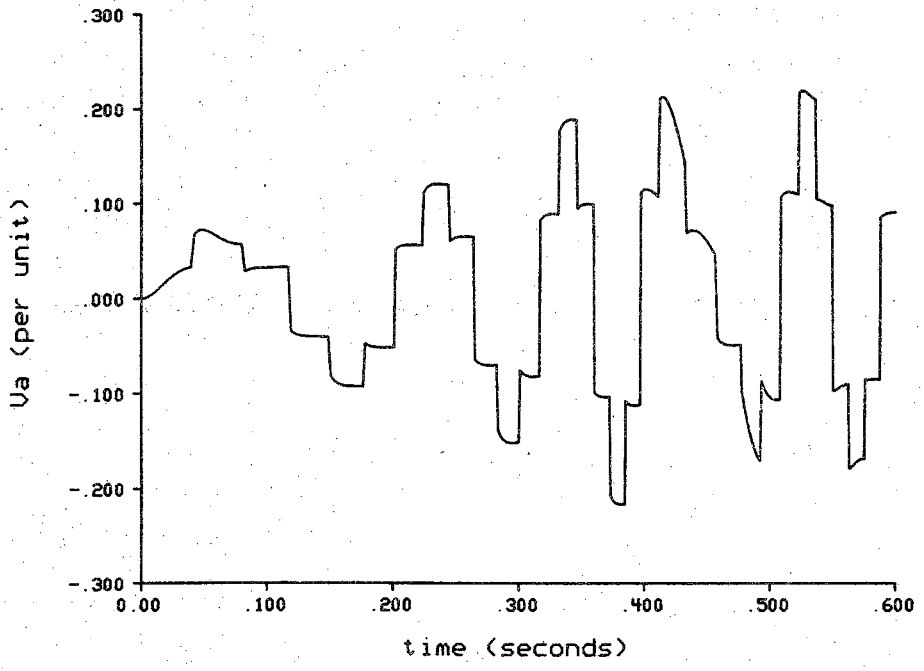


Figure 4.7. Rotor Speed and Torque Trajectories



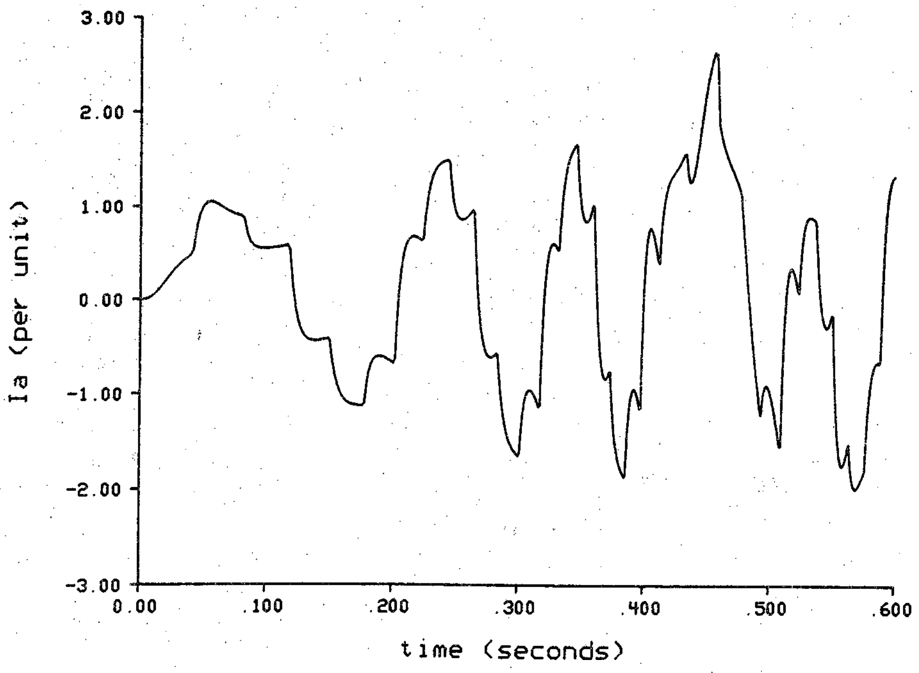


Figure 4.8. Voltage and Current Waveforms

configuration. For this evaluation, three separate 1870-step free acceleration simulations were prepared. The first simulation was run on a single iSBC 86/30 (cpu #1) board, with the separate algorithms for the induction machine and the drive system consolidated into one program. The second and third simulations were the two parallel methods described previously, one for the synchronous relaxation technique and the other for asynchronous relaxation. Because of the extensive processor time requirements for formatted data output, all formatted (terminal or disk) data output was omitted from these timing runs. This omission should not be overly restrictive, since any desired data could be stored in RAM during the simulation run, and written to terminal or disk after the end of the run.

The results of these runs are presented in Table 4.1. This table compares the performance of the three simulation techniques in terms of five measurements: the overall time in seconds used by the hardware for the 1870-step simulation; the average step convergence time in milliseconds; the time required for a single iteration in milliseconds; the average number of iterations necessary for convergence; and, finally, the greatest number of iterations for a single step recorded during the run. The data show that, for the two parallel methods tested, the synchronous relaxation technique gave the best overall performance.

To compare the performance of the two parallel techniques, it is helpful to introduce two quantities which are used as the basis for evaluating multiprocessing systems in terms of computational speed. ***Speed-up*** is defined as the ratio

Table 4.1. Data for 1870-Step (0.6 sec) Runs

Measurement	Single Processor	Parallel, Synchronous	Parallel, Asynchronous
Computation Time (sec.)	37.5	21.5	23.0
Avg. Convergence Time (msec.)	20.0	11.5	12.3
Iteration Time (msec.)	8.8	5.0	M/C: 5.0 Drive: 5.5
Avg. No. of Iterations	2.3	2.3	M/C: 2.4 Drive: 2.2
Greatest No. of Iterations	11	7	M/C: 7 Drive: 6

$$S = \frac{T_1}{T_P} , \qquad (4.36)$$

where T_1 is the computation time when one processor is used, and T_P is the time when P working-area processors are used. The theoretical maximum value for Speed-up is S=P, but real multiprocessing systems cannot achieve this value because of performance degradations due to the time required for data transfers, the time required for coordination of the processors, and, more importantly, unequal distribution of the simulation effort among the processors.

Another value commonly used is *Efficiency*, η:

$$\eta = \frac{S}{N} \tag{4.37}$$

where N is the number of processors in the working area of the simulation. The Efficiency η gives an indication of how intensively the N processors are being used; its optimum value is $\eta=1$. Table 4.2 presents the Speed-up and Efficiency values obtained for the induction motor drive system simulation.

Table 4.2. Speed-up and Efficiency

Method	S	η
Synchronous	1.74	0.87
Asynchronous	1.63	0.82

It is common practice to omit the control (master) processor from Speed-up and Efficiency calculations, because its control and synchronization activities are considered to be outside the realm of the simulation working area. With this in mind, the optimum Speed-up value for the induction motor drive system simulation is S=2.0. Referring to Table 4.2, the synchronous relaxation method achieved S=1.74, η=0.87.

The performance values turned in by the asynchronous relaxation method were slightly lower. This was somewhat surprising at first, but the reason for the poorer performance can be easily explained. When the asynchronous method was first investigated, it was felt that this method would produce superior results in cases of unequal distribution of the simulation effort among the satellite processors. However, the iteration time calculations for the asynchronous method (Table 4.1) suggest that the simulation effort is almost equally distributed between the drive system processor and the induction machine processor. This condition is not well suited for asynchronous relaxation.

To illustrate the reason for this, consider a typical angle step solution requiring two iterations on each satellite processor. As shown in Table 4.1, each machine processor iteration requires 5.0 milliseconds, while each drive system iteration requires 5.5 milliseconds. At 10 milliseconds (two machine processor iterations) into the calculation, the machine processor announces convergence, while the drive system processor is still 1 millisecond away from announcing its convergence. Since the control processor sees only one "set" convergence flag, it allows the machine processor to initiate a third, unnecessary, iteration. The solution for this angular step therefore uses 15 milliseconds of processor time, when 11 would have been sufficient. The effects of this type of inefficiency add up over the simulation run, yielding the higher values of computation and average convergence times recorded in Table 4.1 for the asynchronous method.

One of the plans for the multiprocessing system used in this research is to provide a D/A and A/D interface to the existing analog power system simulator. The analog simulator can be operated at a variety of speeds ranging from 20 to 200 times slower than real time (60 Hz. base). At its slowest setting, 200 times slower than real time, 1° of a sinusoidal waveform corresponds to 9.26 milliseconds. The best 1° convergence time obtained for the induction motor drive system averaged 11.5 milliseconds, a little too slow to keep pace with the analog simulator. If 2° or even 3° resolutions were permissible, it is felt that the average multiprocessor convergence times would probably be able to keep pace with the analog system. In addition, the 8086 processor and 8087 numeric data co-processor used in this research are currently being run at a clock frequency of 5MHz. The system clock frequency may be upgraded to 8MHz when an 8MHz-compatible 8087 co-processor becomes available. This hardware upgrade would favorably affect the convergence times recorded in Table 4.1. However, It should be cautioned that digital simulations involving discontinuous waveforms, such as those produced by the induction motor drive system, are numerically ill-conditioned at waveform transition points. As a result, the convergence times at these points are much longer than the average convergence times, as shown by the "Greatest Number of Iterations" row in Table 4.1.

CHAPTER 5
CONCLUSIONS

The initial phase of the multiprocessor power system simulation research has shown that parallel processing can be effectively applied to power system simulation, and that considerable speed-up can be obtained by distributing the simulation effort over several processors. By examining the iteration cycle times for an example simulation, it was determined that operating the satellite processor iterations in a synchronous fashion produces the best results in cases where the simulation effort is distributed nearly equally among the satellite processors.

In general, however, an even distribution of the simulation task may prove to be difficult because of the wide range of complexity among various power system component models. It would be possible to divide the simulation of an especially complex component over two or more processors to achieve a more even distribution of the simulation effort, but this division of a component model may introduce other problems. Many power system component models are characterized by tightly coupled groups of equations, such as the electrical dynamics equations for the induction machine described in this thesis. Dividing such a tight group of equations among two or more processors would dramatically increase the amount of data transferred between processors by introducing state- and auxiliary-variable transfers, and this approach would also deviate from the conceptual simplicity of allowing only component input/output data transfers.

One alternative to dividing a complex component over two or more processors may be the asynchronous relaxation method. This method maintains the conceptually simple component input/output data transfers, although it has been shown that the results of this technique are inferior to an evenly distributed simulation operated synchronously. There are certainly tradeoffs here between conceptual simplicity and optimum simulation speed; further work on the multiprocessor should address this question.

This initial phase of the multiprocessor simulator development has raised other questions which are worthy of further research. One area which has not yet been addressed is the variety of numerical methods which are applicable to power system component simulations. The example power system simulation presented in Chapter 4 used only the Euler/trapezoid relaxation technique; a wide variety of other integration methods[7] are available which may allow larger angle or time step sizes and more attractive convergence times. However, the additional floating-point operations introduced by these methods would extend the iteration cycle times. Tradeoffs are involved here, also. Another technique which may be effective in linear or nearly-linear component models is the Newton-Raphson algorithm. The extent of the nonlinearities in the induction machine model described in Chapter 4 prevented an efficient implementation of this method, but the Newton-Raphson technique may prove to be effective for other component model simulations.

Other areas of potential research could focus on the hardware and software associated with the multiprocessor itself. At the time of this writing, the A/D and D/A interface to the analog simulator is being installed. Further software and hardware work is needed to allow the multiprocessor to operate effectively in tandem with the analog power system simulator. Another extensive software project would be the development of a user-interactive operating system algorithm for multiprocessor simulations. At the present time, knowledge of 8086 assembly language and the multiprocessor hardware is necessary for effective use of the multiprocessor. A high-level operating system algorithm, performing the mechanics of setting up the parallel simulation, would pull the user away from the hardware level and would allow the user to concentrate on the power system simulation and its results.

As the development of the multiprocessor simulator continues, other possible long-range hardware upgrades should be kept in mind. In the future, if the system should grow to a large number of processors, the existing common-bus architecture may begin to hinder simulation speeds because of data transfer traffic. If this problem should occur, some sort of size-invariant multiprocessor topology[1] could be implemented to eliminate the bus contention problems. In addition, future developments in microprocessor technology should be viewed with their potential simulation applications in mind. Motorola recently introduced the MC68020, a CMOS design with a full 32-bit architecture[15]. This processor operates at 16MHz, and a compatible numeric data co-processor is being designed at this time. The advanced architecture and high speed of this system would produce excellent results if applied to power system simulation.

The results of this initial study of the multiprocessor simulator are encouraging, and further work should produce a cost-effective and useful addition to Purdue's analog power system simulation facilities.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Electric Power Research Institute, "Technology Assessment Study of Near Term Computer Capabilities and Their Impact on Power Flow and Stability Simulation Programs," Final Report El - 946, 1978.

[2] A. Viegas de Vasconcelos and G. Hosemann, "Transient Studies on a Multiprocessor," *IEEE Power Engineering Society Winter Meeting,* 1984.

[3] K. Schmidt and W. Leonhard, "Simulation of Electric Power Systems by Parallel Computation," *Proceedings of the Twelfth Annual Pittsburgh Conference on Modeling and Simulation,* 1981.

[4] I. Durham, R.C. Dugan, and S.N. Talukdar, "Power System Simulation on a Multiprocessor," *IEEE Power Engineering Society Summer Meeting,* 1979.

[5] I. Durham, R.C. Dugan, and S.N. Talukdar, "An Algorithm for Power System Simulation by Parallel Processing," *IEEE Power Engineering Society Summer Meeting,* 1979.

[6] R.A. DeCarlo and R. Saeks, *Interconnected Dynamical Systems,* Marcel Dekker, Inc., New York, N.Y., 1981.

[7] T.E. Shoup, *A Practical Guide to Computer Methods for Engineers,* Prentice-Hall, New Jersey, 1979.

[8] Notes accompanying EE 677, *Interconnected Dynamical Systems* (unpublished).

[9] D. Chazan and W. Miranker, "Chaotic Relaxation," *Linear Algebra and Its Applications,* Vol. 2, 1969, pp. 199-222.

[10] W.E. McBride, "Simulating the Purely Asynchronous Method on the Finite Element Machine," *Proceedings of the Twelfth Annual Pittsburgh Conference on Modeling and Simulation,* 1981.

[11] J.M. Murphy, L.S. Howard, and R.G. Hoft, "Microprocessor Control of a PWM Inverter Induction Motor Drive," *IEEE Power Electronics Specialists Conference,* 1979.

[12] A. Bellini and G. Figalli, "Analysis and Comparison of Different PWM Techniques for Induction Motor Drives," *Instituto di Automatica dell Universita di Roma,* 1977.

[13] P.C. Krause, *Analysis of Electrical Machinery,* (to be published).

[14] P.C. Krause, "Simulation of Symmetrical Induction Machinery," *IEEE Transactions on Power Apparatus and Systems,* Vol. Pas-84, No. 11, 1965.

[15] Institute of Electrical and Electronics Engineers, *The Institute,* Vol. 8, No. 9, September 1984.

APPENDICES

Appendix A
Drive System Parameters

Slip Frequency Limit: ±25 rad/sec.

$K_v = 0.00265$

$K_\omega = 20.0$

$V_o = 1.654$

$\tau_\omega = 0.07$

$r_c = 0.01528\ \Omega$

$R_f = 0.025\ \Omega$

$L_f = 0.001326$ H

$C_f = 0.18812$ F

Appendix B
Induction Machine Parameters

Rating:	10 hp
Voltage:	220 V (line-to-line), 60 Hz
Poles:	6
Inertia:	0.5 sec.

Parameters in Per Unit:

$r_s = 0.0453$

$X_{ls} = 0.0775$

$r'_r = 0.0222$

$X'_{lr} = 0.0322$

$X_m = 2.042$