INFORMATION TO USERS

This dissertation was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in "sectioning" the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from "photographs" if essential to the understanding of the dissertation. Silver prints of "photographs" may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

University Microfilms
300 North Zeeb Road
Ann Arbor, Michigan 48106
A Xerox Education Company
EBNER, George Chester, 1943-
AN ASSOCIATIVELY CONTROLLED FUNCTIONAL
MULTIPROCESSOR FOR REAL-TIME APPLICATIONS.

The Ohio State University, Ph.D., 1972
Engineering, electrical

University Microfilms, A XEROX Company, Ann Arbor, Michigan
AN ASSOCIATIVELY CONTROLLED
FUNCTIONAL MULTIPROCESSOR FOR
REAL-TIME APPLICATIONS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy in the Graduate
School of the Ohio State University

By
George Chester Ebner, B.S., S.M.

* * * * *

The Ohio State University
1972

Approved by

[Signature]
Adviser
Department of Electrical Engineering
To Audrey, Mark, and Jeffrey
ACKNOWLEDGMENTS

I wish to express my sincere appreciation to my wife, Audrey, and to our sons, Mark and Jeffrey, for their love, understanding, encouragement, and moral support during these years of graduate study.

I thank D. M. Rouse, J. E. Smathers, and C. W. Hoffner for many helpful discussions. I gratefully acknowledge my adviser, Professor K. J. Breeding, for his guidance throughout the dissertation.

I am deeply grateful for the generous support I have received from Bell Telephone Laboratories. In particular, I wish to thank my supervisor, J. E. Smathers, and my department head, J. H. Wuorinen, for their encouragement and assistance in obtaining acceptance for the Doctoral Support Program.

G. C. Ebner
VITA

March 8, 1943-- Born -- Columbus, Ohio

1964 -- --- -- -- B.S.E.E., Massachusetts Institute of Technology
      Cambridge, Massachusetts

1965 -- --- -- -- S.M., Massachusetts Institute of Technology,
      Cambridge, Massachusetts

1965 -- --- -- -- Member of Technical Staff, Bell Telephone
      Laboratories, Columbus, Ohio

PUBLICATIONS

"Static V-I Relationships in Transistors at High Injection Levels,"

FIELDS OF STUDY

Major Field: Electrical Engineering

   Studies in Switching Theory. Professors K. J. Breeding and
   R. B. McGhee

   Studies in Modern Control Theory. Professor H. Hemami

   Studies in Computer Science. Professor C. R. Foulk

   Studies in Mathematics. Professor A. M. Buoncristiani
TABLE OF CONTENTS

Dedication .................................................. ii
Acknowledgments .............................................. iii
Vita. ........................................................ iv
List of Tables ................................................ x
List of Illustrations ........................................ xi
List of Acronyms ............................................ xvii

Chapter
I. INTRODUCTION .......................................... 1
II. BACKGROUND ............................................ 8
   2.1 Multiprocessor Terminology ......................... 8
   2.2 Software Terminology ................................ 12
   2.3 The Evolution of Multiprocessing .................. 16
   2.4 Classic Multiprocessor Problems .................. 28
      2.4.1 Memory Access Conflicts ....................... 28
      2.4.2 Common Memory Interconnection .............. 35
      2.4.3 Memory Protection ........................ 37
      2.4.4 Executive Control ............................ 38

v
### III. THE PROPOSED SYSTEM

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Design Objectives</td>
<td>48</td>
</tr>
<tr>
<td>3.2 The Basic Structure</td>
<td>49</td>
</tr>
<tr>
<td>3.3 Memory Structure</td>
<td>53</td>
</tr>
<tr>
<td>3.3.1 Local Memory</td>
<td>54</td>
</tr>
<tr>
<td>3.3.2 Common Data Memory</td>
<td>55</td>
</tr>
<tr>
<td>3.3.3 Memory Interconnection</td>
<td>58</td>
</tr>
<tr>
<td>3.3.3.1 Time-Shared Bus for High-Speed Common Data Memory Access</td>
<td>61</td>
</tr>
<tr>
<td>3.3.3.2 Low-Speed Common Data Memory Interconnection</td>
<td>64</td>
</tr>
<tr>
<td>3.3.4 Common Data Memory Allocation</td>
<td>68</td>
</tr>
<tr>
<td>3.3.4.1 Method 1</td>
<td>69</td>
</tr>
<tr>
<td>3.3.4.2 Method 2</td>
<td>70</td>
</tr>
<tr>
<td>3.3.4.3 Method 3</td>
<td>70</td>
</tr>
<tr>
<td>3.4 Associative Control</td>
<td>70</td>
</tr>
<tr>
<td>3.4.1 Associative Memories</td>
<td>71</td>
</tr>
<tr>
<td>3.4.2 Simplified View of Intertask Communication</td>
<td>76</td>
</tr>
<tr>
<td>3.4.2.1 Control Options</td>
<td>88</td>
</tr>
<tr>
<td>3.5 Processors</td>
<td>90</td>
</tr>
<tr>
<td>3.6 Input/Output</td>
<td>92</td>
</tr>
<tr>
<td>3.6.1 Direct Memory Access to Secondary Storage or to Low-Speed Common Data Memory</td>
<td>93</td>
</tr>
<tr>
<td>3.6.2 Common Input/Output Channels</td>
<td>94</td>
</tr>
<tr>
<td>3.6.3 Dedicated Input/Output Processors</td>
<td>94</td>
</tr>
<tr>
<td>3.7 Summary</td>
<td>96</td>
</tr>
</tbody>
</table>
IV. INTERTASK COMMUNICATION ....................................... 97

4.1 Hardware Design Details Pertinent to Inter-
task Communication ..................................... 98

4.1.1 The Executive CAM ................................ 98
4.1.2 The QCAM ........................................ 106
4.1.3 Common Data Memory ................................ 116
4.1.4 Processors ........................................ 116

4.1.4.1 Register Memory and Control .......... 116
4.1.4.2 Return Counter ................ 118
4.1.4.3 Associative Memory Access Registers .. 121
4.1.4.4 Microprogram Control. ................. 121

4.2 Processor Microprograms for Task Control. .............. 121

4.2.1 ENTASK .......................................... 123
4.2.2 LW .............................................. 125

4.2.2.1 Task Termination .................. 127
4.2.2.2 Task Suspension .................. 128
4.2.2.3 Action Taken at Entry Point 1 ..... 129

4.2.3 QENTRY .......................................... 131
4.2.4 QRET ........................................... 133
4.2.5 WRET ........................................... 133

4.3 Control Macros ............................................ 136

4.3.1 CALLs ........................................... 136
4.3.2 RETURN ......................................... 138
4.3.3 END ............................................. 138
4.3.4 WAIT ............................................ 138
4.3.5 FORK and JOIN .................................. 138
4.4 Examples of How Control Macros are Used ........................................ 139
4.5 Repeated Tasks ........................................................................... 146
4.6 Summary ..................................................................................... 157

V. PERFORMANCE EVALUATION ............................................................. 162
5.1 Circuit Delay Assumptions ............................................................... 165
5.2 Simulations of Example 6 ................................................................. 167
   5.2.1 Theoretical Consideration ....................................................... 169
   5.2.2 Simulation of Single Instances of Execution ..................... 173
   5.2.3 Simulations of Multiple Instances of Execution ............. 177
5.3 Determination of Capacity ................................................................. 182
   5.3.1 Overhead Simulations ............................................................. 183
   5.3.2 Analytic Determination of Overhead .................................... 186
5.4 Time-Shared Bus Simulators ........................................................... 192
   5.4.1 TSB1 .................................................................................. 194
   5.4.2 TSB2 .................................................................................. 198
5.5 Summary ....................................................................................... 198

VI. APPLICATION OF THE ACFM TO REAL-TIME OBJECT IDENTIFICATION 204
6.1 Description of the Algorithm ........................................................... 205
   6.1.1 Overview ............................................................................ 205
   6.1.2 Silhouette Generation .......................................................... 207
   6.1.3 Advani's Simulation ............................................................... 210
6.2 ACFM Hardware Configuration ....................................................... 211
6.3 ACFM Software Description ........................................................... 213
   6.3.1 General Presentation ............................................................. 213
   6.3.2 Data Traffic Between Tasks .................................................. 221
6.4 Simulation Results ........................................... 223
6.5 Summary ....................................................... 227

VII. APPLICATION OF THE ACFM TO TELEPHONE SWITCHING CONTROL ... 229

7.1 Description of Telephone Switching Control ............. 230
  7.1.1 The Concept of Call States ............................... 233
  7.1.2 State Diagram of a Typical Call ......................... 234

7.2 Description of the ACFM Hardware Configuration
  for Telephone Switching Control ............................. 238
  7.2.1 Data Memory ............................................. 238
  7.2.2 The CPU ................................................. 241
  7.2.3 Memory Interconnection Subsystem ....................... 242

7.3 ACFM Software Structure ..................................... 243
  7.3.1 Scheduling Philosophy ................................... 245

7.4 Simulation Results ........................................... 246
7.5 Summary ....................................................... 262

VIII. CONCLUSIONS AND RECOMMENDATIONS FOR FURTHER WORK .......... 263

APPENDIXES

A. The Executive Control Simulator ............................. 267
B. Circuit Delay Assumptions .................................... 285

References ....................................................... 290
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-1</td>
<td>Simulations of Single Instances of Execution of Example 6</td>
<td>174</td>
</tr>
<tr>
<td>5-2</td>
<td>Simulation Results for Five Instances of Execution of Example 6 on Four Processors</td>
<td>179</td>
</tr>
<tr>
<td>5-3</td>
<td>Results of Typical Bus Simulation Using TSBI</td>
<td>197</td>
</tr>
<tr>
<td>6-1</td>
<td>Real-Time Object Identification Simulation Results</td>
<td>224</td>
</tr>
<tr>
<td>6-2</td>
<td>Results of Time-Shared Bus Simulations</td>
<td>226</td>
</tr>
<tr>
<td>7-1</td>
<td>Single Processor Simulation Results</td>
<td>255</td>
</tr>
<tr>
<td>7-2</td>
<td>Two-Processor Simulation Results</td>
<td>256</td>
</tr>
<tr>
<td>7-3</td>
<td>Three-Processor Simulation Results</td>
<td>258</td>
</tr>
<tr>
<td>7-4</td>
<td>Four-Processor Simulation Results</td>
<td>259</td>
</tr>
<tr>
<td>7-5</td>
<td>Simulation of Storage Interference at Common Data Memory Interface</td>
<td>260</td>
</tr>
</tbody>
</table>
## LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Conceptual Block Diagram of the ACFM</td>
<td>4</td>
</tr>
<tr>
<td>2-1</td>
<td>A Representative Multiprocessor Structure</td>
<td>10</td>
</tr>
<tr>
<td>2-2</td>
<td>The Burrough D825 Multiprocessor System</td>
<td>19</td>
</tr>
<tr>
<td>2-3</td>
<td>The IBM 9020 Multiprocessor System</td>
<td>20</td>
</tr>
<tr>
<td>2-4</td>
<td>The SAFEGUARD Multiprocessor System</td>
<td>22</td>
</tr>
<tr>
<td>2-5</td>
<td>The CDC 3600 System</td>
<td>23</td>
</tr>
<tr>
<td>2-6</td>
<td>The IBM Stretch (7031) Structure</td>
<td>24</td>
</tr>
<tr>
<td>2-7</td>
<td>The CDC 6700 Multiprocessor</td>
<td>26</td>
</tr>
<tr>
<td>2-8</td>
<td>Effective Processing Power vs N_p</td>
<td>30</td>
</tr>
<tr>
<td>2-9</td>
<td>Storage Utilization vs N_p</td>
<td>31</td>
</tr>
<tr>
<td>2-10</td>
<td>Control Hierarchy in a General Purpose Multiprocessor</td>
<td>40</td>
</tr>
<tr>
<td>2-11</td>
<td>A SAFEGUARD Universal Task List</td>
<td>43</td>
</tr>
<tr>
<td>2-12</td>
<td>Gunderson's Associatively Controlled Multiprocessor</td>
<td>45</td>
</tr>
<tr>
<td>3-1</td>
<td>The Basic ACFM Architecture</td>
<td>51</td>
</tr>
<tr>
<td>3-2</td>
<td>ACFM Local Memory Layout</td>
<td>56</td>
</tr>
<tr>
<td>3-3</td>
<td>ACFM Configuration for File-Oriented Application</td>
<td>59</td>
</tr>
<tr>
<td>3-4</td>
<td>ACFM Configuration for Telephone Switching Control</td>
<td>60</td>
</tr>
<tr>
<td>3-5</td>
<td>High-Speed Parallel Time-Shared Bus</td>
<td>62</td>
</tr>
</tbody>
</table>
3-6 State Diagram for Time-Shared Bus Controller
3-7 Serial Electronic Crossbar Switch
3-8 Serial Time-Shared Bus Arrangement
3-9 Associative Memory for ACFM Control
3-10 Two-Processor System with Executive CAM
3-11 Associative Operations Required to Pass Control From Task A to Task B
3-12 Two-Processor System with QCAM Included
3-13 Three-Processor System Illustrating Queuing of Requests in the QCAM
3-14 QCAM Word for Task Call with Return Linkage
3-15 QCAM Word for Return to Calling Task
3-16 Search Key for Return Search
3-17 Simplified State Diagram of the Executive Control Functions Performed by a Processor
3-18 Input/Output Options for an ACFM
4-1 Executive CAM Word Format
4-2 Enable Task Operation
4-3 Priority Task Search Operation
4-4 QCAM Word Format
4-5 Write Idle Word Operation
4-6 Task Read Operation
4-7 Clear Word Operation
4-8 Return Write Operation
4-9 Return Search Operation
4-10 Associative Memory for Register Group Control
<table>
<thead>
<tr>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-3</td>
<td>An Optimal Two-Processor Assignment for Example 6</td>
</tr>
<tr>
<td>5-4</td>
<td>An Optimal Three-Processor Assignment for Example 6</td>
</tr>
<tr>
<td>5-5</td>
<td>Gantt Charts for Two Different Two-Processor Assignments for Example 6</td>
</tr>
<tr>
<td>5-6</td>
<td>Coding of Example 6 with Precedence Relationships Coded Implicitly Within the Tasks</td>
</tr>
<tr>
<td>5-7</td>
<td>Gantt Chart of Four-Processor Simulation of Five Reentrant Instances-of-Execution of Example 6</td>
</tr>
<tr>
<td>5-8</td>
<td>Sequence of Processor Activity During Overhead Simulation</td>
</tr>
<tr>
<td>5-9</td>
<td>Processor Utilization vs Segment Length (Simulation Results)</td>
</tr>
<tr>
<td>5-10</td>
<td>Breakdown of Processor Real-Time</td>
</tr>
<tr>
<td>5-11</td>
<td>Pollaczek's Average Delay Formula</td>
</tr>
<tr>
<td>5-12</td>
<td>Processor Utilization vs Segment Length (Analytic Results)</td>
</tr>
<tr>
<td>5-13</td>
<td>Flowchart of TSB1 Simulator</td>
</tr>
<tr>
<td>5-14</td>
<td>Flowchart of TSB2 Simulator</td>
</tr>
<tr>
<td>5-15</td>
<td>Typical Results of Time-Shared Bus Simulation Using TSB2</td>
</tr>
<tr>
<td>5-16</td>
<td>Average Segment Length Required for a Given Level of Processor Utilization vs Np</td>
</tr>
<tr>
<td>6-1</td>
<td>Flowchart of Total Algorithm</td>
</tr>
<tr>
<td>6-2</td>
<td>Flowchart of the Silhouette Generation Sub-routine</td>
</tr>
<tr>
<td>6-3</td>
<td>ACFM Hardware Configuration for Real-Time Object Identification</td>
</tr>
<tr>
<td>6-4</td>
<td>Flowchart of REALMAIN</td>
</tr>
<tr>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>6-5</td>
<td>Flowchart of LOCMN.</td>
</tr>
<tr>
<td>6-6</td>
<td>Subroutine Level Diagram for REALMAIN</td>
</tr>
<tr>
<td>6-7</td>
<td>Flowchart of RGRES.</td>
</tr>
<tr>
<td>6-8</td>
<td>Flowchart of SILHETTE</td>
</tr>
<tr>
<td>6-9</td>
<td>Address Table Whose Address is Passed to the SILHETTE Routine</td>
</tr>
<tr>
<td>6-10</td>
<td>Address Table Whose Address is Passed to the SILCALC Routine</td>
</tr>
<tr>
<td>7-1</td>
<td>Conceptual View of a Toll Switching Office</td>
</tr>
<tr>
<td>7-2</td>
<td>State Diagram for a Typical Toll Telephone Call</td>
</tr>
<tr>
<td>7-3</td>
<td>Four-Processor ACFM Configuration for Telephone Switching Control</td>
</tr>
<tr>
<td>7-4</td>
<td>Conceptual View of the ACFM Software Structure for Telephone Switching Control</td>
</tr>
<tr>
<td>7-5</td>
<td>ORIG Call State Routine (State 1)</td>
</tr>
<tr>
<td>7-6</td>
<td>ENDWS Call State Routine (State 2)</td>
</tr>
<tr>
<td>7-7</td>
<td>DIGST1 Call State Routine (State 3)</td>
</tr>
<tr>
<td>7-8</td>
<td>DIGST2 Call State Routine (State 4)</td>
</tr>
<tr>
<td>7-9</td>
<td>BEGINWS Call State Routine (State 5)</td>
</tr>
<tr>
<td>7-10</td>
<td>OUTPULSE Call State Routine (State 6)</td>
</tr>
<tr>
<td>7-11</td>
<td>IDLEOUTP Call State Routine (State 7)</td>
</tr>
<tr>
<td>7-12</td>
<td>STABLE Call State Routine (State 8)</td>
</tr>
<tr>
<td>7-13</td>
<td>DISC Call State Routine (State 9)</td>
</tr>
<tr>
<td>7-14</td>
<td>Service Subroutines</td>
</tr>
<tr>
<td>7-15</td>
<td>Capacity in Calls Per Hour vs N_p</td>
</tr>
<tr>
<td>A-1</td>
<td>Segmentation of the ENTASK Microprogram into Three Procedures</td>
</tr>
<tr>
<td>A-2</td>
<td>Segmentation of the LW Microprogram into Eight Procedures</td>
</tr>
</tbody>
</table>
A-3 Segmentation of QENTRY Microprogram into Two Procedures ............................. 272
A-4 Segmentation of QRET Microprogram into Two Procedures ............................. 273
A-5 Single Procedure for WRET Microprogram ............................................ 274
A-6 Main Loop of Executive Control Simulator ............................................. 276
A-7 Simulator Coding for a Task Procedure .................................................. 278
A-8 Activity of Processor X During Simulation of TaskA .................................. 281
A-9 Complete Flowchart of Executive Control Simulator .................................. 282
B-1 Anticipated Times for Associative Control Operations .............................. 288
LIST OF ACRONYMS

ACFM  Associatively Controlled Functional Multiprocessor
BORAM  Block Oriented Random-Access Memory
CPU  Central Processing Unit
ENTASK  ENable TASK Microprogram
Executive CAM  Executive Control Associative Memory
LSI  Large Scale Integration
LW  Looking for Work Microprogram
Nm  Number of Memory Modules
Np  Number of Processors
QCAM  Queue Control Associative Memory
QENTRY  Queue ENTRY Microprogram
QRET  Queue RETurn Microprogram
RC  Return Counter
TSB1  Time-Shared Bus Simulator No. 1
TSB2  Time-Shared Bus Simulator No. 2
WRET  Write RETurn Microprogram
CHAPTER I

INTRODUCTION

Controlling a multiprocessor so that the processors act collectively to satisfy overall system requirements without conflicting with each other is a major problem. Successful multiprocessing systems have overcome this difficulty only at the expense of considerable hardware and software complexity. As a result, multiprocessing has largely been limited to applications for which no other solution is available. Examples include very large general purpose time-sharing systems and complex real-time control systems for anti-ballistic missile defense and for air traffic control.

The intent of this research is to develop a multiprocessor architecture which minimizes interference among processors, which simplifies the control problem, which can be expanded to cover a wide range of capacity requirements, and which can be customized according to the needs of a variety of real-time applications. The research is motivated by the belief that a multiprocessor structure tailored to the characteristics of real-time processing is needed. It is further
motivated by the belief that a multiprocessor structure and associated control philosophy is needed which is not intimately bound to the characteristics of any particular processor design. The increased interest in using networks of minicomputers, rather than a general purpose uniprocessor, for special purpose applications substantiates this need.\(^1\)

The proposed system is called the Associatively Controlled Functional Multiprocessor (henceforth referred to as the ACFM). The philosophy taken in the design of the ACFM is, first, to choose a structure which minimizes the amount of hardware shared by processors thereby making the system more distributed and less prone to interference among processors, and, second, to develop a control scheme which ties the system together so that it functions efficiently as a multiprocessing complex.

All program memory in the ACFM is dedicated on a per processor basis. Task programs are assigned to specific processors. It is in this sense that the ACFM is a functional multiprocessor; each processor can perform only a subset of all functions which are required by the application. Dedicated program memory removes the major source of storage interference but sacrifices some of the flexibility that is enjoyed by a system in which all program memory is shared. A storage backup system can compensate for this loss of flexibility by making it possible for the functional assignment of task programs to processors to be dynamically changed. The fixed software structure of real-time systems, combined with the statistical information that is available
on input arrival rates, allows a task assignment to be made that will tend to distribute the real-time load evenly among processors.

The heart of the control design is a shared associative control subsystem which, by virtue of the parallel search capability of associative memories, facilitates the performance of executive control operations. Each processor performs executive control functions on its own behalf using the associative control subsystem as a depository for requests to task programs that are assigned to other processors and as a scheduler and task dispenser to select the highest priority request waiting for it. The most critical control function in the system is called intertask communication. Intertask communication includes all operations involved in establishing linkages and passing control between task programs assigned to different processors.

The ACFM design is modular; conceptually it can be viewed as shown in Figure 1–1. The detailed designs of the common data memory subsystem, the input/output subsystem, the backup storage subsystem, and the interconnection subsystem which connects processors to shared hardware units depend upon the requirements of the particular application. These application dependent portions of the system are considered by enumerating the various design options which can easily be incorporated into the ACFM structure. The system can accommodate virtually any type of processor or an assortment of processor types. The hardware and software which is required by processors to perform executive operations can be incorporated into a microprogrammed interface unit which sits between each processor and the shared busses, or it can be an integrated portion of the processor design itself.
FIGURE 1-1
CONCEPTUAL BLOCK DIAGRAM OF THE ACFM
The major contribution of this dissertation is the development and evaluation of an algorithm for controlling a functional multiprocessor. The executive operations required for intertask communication are developed in detail, in some cases down to the microprogram level and in other cases down to the register transfer level. A repertoire of associative control operations is developed which provides the capability for the implementation of efficient scheduling algorithms; but, task scheduling itself is not a subject of this dissertation. A set of control macros is defined which provides the user with the basic operations needed to program the system. Programs will execute on the ACFM independently of how many processors are in the system and independently of how task programs are assigned to processors. The set of control macros allows parallelism within a single program structure to be specified. However, the problem of recognizing parallelism within an algorithm is not explicitly treated.

The performance of the system is evaluated by simulation. The applicability of the ACFM to two real-time applications is investigated. The first, three-dimensional object identification, achieves parallelism by executing internal branches within an overall algorithm in parallel. The second, telephone switching control, achieves parallelism by simultaneously processing many telephone calls which are at various stages of completion. The simulation results demonstrate that the ACFM structure and control philosophy is effective for both applications.

Chapter II is devoted to background material. Multiprocessor terminology is defined and a discussion of pertinent software
terminology is given. An historical development of the evolution of multiprocessing is presented along with a brief description of noteworthy multiprocessors which have actually been built. The major problems associated with multiprocessing are considered, and proposed solutions which have appeared in the literature are discussed.

Chapter III introduces the ACFM system. General guidelines for its development are established, and each of the major subsystems is considered. The memory structure is examined by considering the various types of memory, the methods for interconnecting processors to common memory, and the memory allocation strategies which are most appropriate for the ACFM. A description of associative memory operation is given after which a general description of the control philosophy is presented. A brief discussion of the requirements which the system places on processors is followed by a consideration of the input/output configurations possible within the ACFM structure.

Chapter IV is a detailed description of the ACFM control philosophy. The repertoire of associative control operations is developed and the executive operations performed by processors are described by a set of microprograms. The set of control macros is defined and several examples are given to illustrate their use.

Chapter V evaluates the performance of the control design by simulation. It verifies the logical integrity of the control design and estimates system overhead, that is, the percentage of time which processors spend doing executive operations. It also evaluates the performance of the time-shared bus which is chosen as the basic method for interconnecting processors to common data memory.
Chapters VI and VII examine, by simulation, the applicability of the ACFM structure to three-dimensional object identification and telephone switching control, respectively.

Chapter VIII presents conclusions and recommendations for further work.

Because of the length and complexity of the executive control simulator, Appendix A is devoted to its description. Appendix B discusses the assumptions which were made concerning what circuit propagation delays to use in the simulations.
CHAPTER II

BACKGROUND

The purpose of this chapter is to develop background material for the remainder of the dissertation. In Section 2.1 multiprocessing is defined and terminology for the important hardware components of a multiprocessor system is presented. Section 2.2 discusses the essential characteristics of real-time system software with emphasis on how it differs from the software of a general purpose system. It also defines pertinent software terminology for the ACFM system. Section 2.3 gives an historical development of multiprocessing. The classical problems of multiprocessor system design, along with solutions which have appeared in the literature, are considered in Section 2.4.

2.1 Multiprocessor Terminology

Computing systems may be classified according to the number of instruction and data streams which can be simultaneously processed as was suggested by Flynn.\(^{(2)}\) He defines a uniprocessor to be a computer which applies a single instruction stream to a single data stream, a parallel or array processor to be a computer which operates on multiple data streams under the control of a single instruction stream, and a multiprocessor to be a computer which applies multiple instruction streams to multiple data streams.
The following definition for a multiprocessor is used in this dissertation.

A **multiprocessor** is a computing system consisting of two or more independent processors which work collectively on a common data base to accomplish a particular set of system requirements.

The definition implies that the total system load must somehow be subdivided into parts which can be performed simultaneously by the several processors. Multiprocessors can, in fact, be classified according to how the system load is subdivided. There are two basic classifications.

In **functional multiprocessing**, each processor performs a dedicated set of functions which is a subset of the total set of functions required by the application. The processors in a functional organization may be identical hardware units which are programmed differently or they may be special units which are designed specifically for their intended functions.

In **direct multiprocessing**, any processor can perform any function in the total set of functions required by the application.

A continuum of machine organizations which mix functional and direct multiprocessing techniques is possible.

For purposes of defining hardware terminology used throughout this dissertation a very general multiprocessor structure is illustrated in Figure 2-1.
FIGURE 2-1
A REPRESENTATIVE MULTIPROCESSOR STRUCTURE
A CPU (Central Processing Unit) consists of the arithmetic, logic, and control portion of a processor. A CPU is capable of executing an instruction stream.

Program Memory contains instructions which can be fetched and executed by CPUs.

Data Memory contains transient information, system parameters, and other data items to be processed.

A processor consists of a CPU and possibly a local data memory and/or a private program memory. These memories contain information which is directly available only to the processor of which they are a part. The processors in the system are not necessarily identical.

Common data memory and common program memory contain information which is accessible to all processors in the system. A memory interconnection subsystem provides the required connections between processors and common memories.

The control subsystem consists of shared hardware which facilitates system control.
The **secondary storage** subsystem contains a copy of all programs and permanent data required for system functions. It can move information to or from any first-level memory in the system. (Common data memory, common program memory, local data memory, and private program memory comprise the first-level memory in the system.)

The **input-output subsystem** provides for communication with the external environment. It may be connected to all processors via a common bus or via a switching exchange, or it may interface directly with common memory or with secondary storage.

### 2.2 Software Terminology

The software structure of a real-time computing system can be broken down into three fundamental program classifications and two data classifications as is done in the book by Martin.⁴

- **Application Programs** carry out the actual processing of transactions or messages in the system.

- **Supervisory Programs** coordinate and schedule work for the application programs and perform service functions on their behalf. In a sense, application programs are subroutines of the supervisory programs.
Support Programs help keep the system running smoothly. Maintenance and diagnostic programs fall into this category.

Transient Data consists of temporary results, input/output data, and data which reflects the present system status. (The busy/idle state of input/output ports is an example of present system status information.)

Permanent Data specifies the particular system configuration including not only that of the internal processing system but also that of the external environment being controlled.

Real-time systems are engineered according to externally imposed constraints on response time.

Response time is the time it takes for a real-time system to detect an input stimulus in the controlled environment, perform indicated processing functions, and return appropriate output stimuli to the controlled environment. (3)

For example, an anti-ballistic missile defense system(11) has to detect inputs from the radar system, calculate the trajectory of the enemy missile, determine the required trajectory of the intercepting missile, and issue firing commands all within a few tens of milliseconds.
Real-time system design is quite different from general purpose time-sharing system design. Input stimuli (or user requests) cause a particular path to be taken through a fixed set of supervisory and application programs. The expected arrival rate for each type of input stimulus is known; response time requirements are clearly specified. In a general purpose time-sharing system, on the other hand, user requests consist of application programs and associated input data. The system has no way of knowing in advance what resources (memory space, I/O devices, system programs, etc.) are required to handle the user requests. There is little meaningful information available on the expected arrival rate of different types of user requests. Consequently, the system must have a very complex operating system which allows for all possibilities. The operating system attempts to schedule jobs so that turnaround time is minimized. There are, however, no critical requirements on response time.

The software structure of a real-time control system should take advantage of the more completely specified environment in which it works. Simplicity is of major importance; the software must get the job done, but it should not be over designed. Ease of programming ought to be an important goal also, especially since software design and debugging costs are beginning to dominate overall system costs.

The application programs, supervisory programs, and support programs of a real-time multiprocessor system must be suitably broken down into tasks which can be allocated to processors for execution in parallel.
The task is the fundamental program building block in a multiprocessing system. A task performs a specific function and may, in general, call other tasks. Tasks are the basic entities assigned to processors for execution.

Tasks may be organized to form a task structure, also called a routine. In this case, the task structure is itself a task and is assigned to a processor. The tasks within the task structure may be viewed as subroutines.

Fundamentally, parallelism in a multiprocessor can be achieved in two ways. A single program can be organized into a task structure in which some tasks can be run in parallel, or many unrelated tasks can be simultaneously executed. The motive for the cooperative execution of elements of a single task structure is to finish the task structure more quickly. Fork and Join instructions, first conceived by Conway,\(^{(4)}\) are the vehicles for spawning and merging parallel branches, respectively. The motive for the parallel execution of unrelated or loosely related tasks is to increase the throughput of the system.

Two types of task coding are employed in the ACFM software structure, reentrant coding and serially reusable coding.\(^{(5)}\)

A reentrant task can have multiple instances of execution in progress at the same time. In other words, a reentrant task may execute for awhile on behalf of client A, be suspended, and then begin
execution on behalf of client B.\textsuperscript{1} This implies that transient information pertaining to the suspended instance of execution (client A) is kept separate from the transient information pertaining to the second instance of execution (client B).

A \textit{serially reusable} task may have at most one instance of execution in progress at any given time. A particular instance of execution must complete before a second can begin.

Both types of tasks may have queues of requests waiting for them. The difference between the two is that a waiting request to a reentrant task can begin execution as soon as the previous instance of execution is suspended; whereas a waiting request to a serially reusable task can begin execution only after the previous instance of execution has completed.

2.3 \textbf{The Evolution of Multiprocessing}

In early computer systems, the central processor was in direct control of virtually all system activity. Input-output equipment was controlled directly by the processor. If, for example a tape was to be read or written, all other activity was stopped while the computer

\textsuperscript{1}A suspended instance of execution of a task is one which has been discontinued but which will be reactivated at some future time.
controlled the transfer of information between the tape and core memory. Memory addresses were prepared, timing of tape gaps was calculated, and the remainder of the system sat idle.

The next step toward higher computing efficiency was the advent of data channels. These input-output channels were a great improvement. They made possible the simultaneous operation of peripheral equipment and the central processor. Each channel had direct access to memory, with its own address register and the ability to keep a count of the number of records transferred. An interrupt system was needed to insure priority of access to memory for peripheral information. Although data channels were much more efficient than direct software control of I/O, they still required the address preparation and arithmetic capabilities of the main computer.

The next advance toward multiprocessing was the use of a completely separate Input-Output processor with its own memory. The IBM 1401 which was used in many 7090-94 installations is a good example of the functional split of input-output and main computing activity between two different machines. In some cases, the 1401 provided all editing, data conversion, peripheral control and communication control functions for the system.

Data channels and separate I/O processors are still very much in vogue today. The IBM 360 and 370 series both use selector and multiplexer channels to control peripheral equipment. The CDC 6600 system uses a collection of ten peripheral processors to control input-output activity.
Multiprocessing came of age with the need for large systems, for ballistic missile defense and for air traffic control, whose requirements for enormous amounts of computing capacity exceeded the capability of the most powerful uniprocessor system. A characteristic of these systems was a complex interdependent logical structure requiring that system data be kept centrally. High availability was also an important design specification. The almost obvious answer to these needs was to build a system consisting of multiple processors and to develop control techniques which enabled the processors to act collectively to perform the intended system operations.

An early example (1962) of a multiprocessor for real-time applications was the Burrough D825(9) depicted in Figure 2-2. In this system up to four processor modules were interconnected via an electronic crossbar switch to up to 16 memory modules and to a group of input-output control modules which accessed up to 64 peripheral devices through an electronic crossbar switch input-output exchange.

More current multiprocessors for real-time applications include the IBM 9020 and the SAFEGUARD systems.

The IBM 9020 system(10) (1967) was designed in accordance with Federal Aviation Administration (FAA) requirements for a large-scale air traffic control system with high availability. A block diagram of the IBM 9020 system is shown in Figure 2-3. The system consists
FIGURE 2-2

THE BURROUGH D825 SYSTEM
FIGURE 2-3
THE IBM 9020 SYSTEM
of up to three computing elements (IBM 360s), 12 storage elements, three I/O control elements, and various types of peripheral equipment. Each computing element and each I/O control element has a private bus which connects to all storage elements in the system.

The SAFEGUARD system\(^{(11)}\) depicted in Figure 2-4 is the control complex of the ABM (Anti-Ballistic Missile) system currently under development in the United States. The basic modules in the SAFEGUARD system are processors, program stores, variable (data) stores, input-output control units, and a timing and status unit which controls the system configuration. The system is fully interconnected, that is, a separate bus exists for every intermodular interconnection. Interface switching units resolve conflicts on a fixed priority basis.

Multiprocessing has also come to fruition in general purpose computation with such systems as the CDC 3600, the IBM 7031, the UNIVAC 1108, MULTICS, IBM MP65, and the CDC 6700. The justification for large multiprogrammed multiprocessor systems is the belief that large complex problems requiring vast amounts of data are best solved by large complex computers.\(^{(1)}\)

One of the early commercial multiprocessors was the CDC 3600\(^{(12)}\) (1962) shown in Figure 2-5. Each processor and each input-output control module (housekeeping module) has a single memory bus which connects to all storage modules.

The IBM-Stretch (7031) system\(^{(13)}\) (1962) illustrated in Figure 2-6 used a time-shared bus to connect its two processors to all storage modules. The bus switch connects a processor to a storage module long
FIGURE 2-4
THE SAFEGUARD SYSTEM
FIGURE 2-5
THE CDC 3600 SYSTEM
Figure 2-6

The IBM Stretch (7031) Structure
enough for the information to be transferred; the bus is then free to handle other requests.

The CDC 6700\(^{(8)}\) (1967) illustrated in Figure 2-7 consists of two CDC 6600s. In each 6600 a functional split is made between input-output and central processing functions as in all of the multiprocessor systems discussed thus far. The input-output processor section consists of ten identical peripheral processors which share the input-output load. The central processor pushes the instruction execution rate for a single instruction stream to the limit imposed by the discrete component technology used. It consists of special hardware units which perform dedicated functions such as add, multiply, divide, etc., and elaborate control hardware to control the sequencing of instructions to these units.

The architectures of the UNIVAC 1108, MULTICS, and IBM MP65 systems do not differ markedly from those depicted in Figures 2-2 to 2-7.

Most multiprocessor systems in existence today were designed for applications for which a large powerful uniprocessor would not be adequate. However, a new trend is developing toward using networks of minicomputers rather than a powerful single processor for certain applications.\(^{(1)}\) The minicomputer network may be as effective as the large single computer — and much less expensive.

The important attributes of multiprocessing are high capacity, reliability, flexibility, and efficiency.

The capability of multiprocessors to handle real-time applications with complex interdependent logic structures requiring a common data
FIGURE 2-7
THE CDC 6700 MULTIPROCESSOR
base is well known. Almost equally obvious is the inherent reliability of a multiprocessor. Since the system load is shared by several processors, system availability is high. Redundant processors can give the system additional immunity to failures and can also provide extra capacity in periods of overload. A well-designed multiprocessor will degrade gracefully as processors fail; the system can continue to function, albeit at greatly reduced capacity, even when only one processor remains. Less obvious features of multiprocessing are its flexibility and efficiency.

The multiprocessor is potentially a very flexible architecture. As already indicated, a whole spectrum of possibilities exists for mixing functional and direct multiprocessing. A multiprocessor with \( N \) identical processors can be programmed to function as \( N \) independent computers or as one very large machine. A single processor can be singled out to use for program debugging while the other \( N-1 \) processors continue carrying the system load. If the design is modular, a basic system can be customized for specific applications by changing module types. A multiprocessor can grow to meet requirements for increased capacity or to permit new functions to be incorporated into the system.

A multiprocessor is more efficient in its use of resources than a single processor. Processors, busses, switching equipment, and input-output controllers may be time-shared. Memory modules and secondary storage may be space shared.

In the future, the concept of multiprocessing will become more popular because the technology needed to overcome the basic weaknesses
of the architecture is becoming available; viz., high-speed semiconductor memory, associative memory, read-only or read-mostly memory for microprogram control, and LSI (Large-Scale Integration) for modular logic design. The next section examines the classical problems characteristic of multiprocessing and summarizes previous work toward their solution.

2.4 Classic Multiprocessor Problems

The full attainment of the potential advantages of multiprocessing has been inhibited by several classic problems. Memory access conflicts, which result when processors compete for common memory modules, reduce multiprocessor capacity and efficiency. The complexity of the memory interconnection subsystem required to minimize memory access conflicts inhibits system flexibility and "growability." Data protection techniques are needed to prevent one processor from prematurely using data being modified by another. The executive control scheme must provide for the coordinated control of the entire system without introducing too much overhead. This section briefly examines the nature of these multiprocessor problems and surveys work toward their solution which has been reported in the literature.

2.4.1 Memory Access Conflicts

Whenever several processors share common memory modules, there is a finite probability that two or more processors may simultaneously attempt to access the same memory module. When this happens, one processor must be granted access to the memory module according to some priority assignment while other processors desiring access to the same
module queue up and wait. This queuing delay increases the effective memory access time and thereby degrades system performance.

The memory access conflict problem is, of course, most serious in a multiprocessor where all memory is shared. Its significance is best demonstrated by a plot of effective processing power versus the number of processors ($N_p$) for a fixed number of storage modules ($N_m$). Effective processing power is simply the total system processing capacity expressed in terms of the capacity of a single processor. Figure 2-8 illustrates such a plot for $N_m = 16$, 32, and 64. The dotted line represents the ideal case where the effective processing power is $N_p$ times the capacity of one processor. Referring to the figure, it is seen that the amount of interference depends upon the ratio of $N_m$ to $N_p$. For a given number of storage modules, $N_m$, adding additional processors brings diminishing returns in processing power owing to the greater probability of a memory access conflict. For a fixed number of processors, effective processing power can be increased by increasing the number of storage modules. This, however, decreases storage utilization as illustrated in Figure 2-9 and requires a more complex memory interconnection subsystem. (Storage utilization is the fraction of the maximum possible storage access rate actually used.) An optimum balance between $N_m$ and $N_p$ would provide the desired processing power while maximizing the cost effectiveness of the system.

2The data illustrated in Figures 2-8 and 2-9 were obtained from a simulation study by Rosenfeld (Reference 14) in which he actually modeled the performance of a direct multiprocessor in executing an electrical network problem.
FIGURE 2-8
EFFECTIVE PROCESSING POWER VS Np
STORAGE UTILIZATION VS \( N_p \)

FIGURE 2-9

STORAGE UTILIZATION VS \( N_p \)
There are two basic approaches to reducing the loss of effectiveness caused by storage interference. The first is to allocate common memory to programs and data and to assign tasks to processors in such a way that processors tend to operate in disjoint memory modules. The second is to reduce traffic to common memory by using local (per-processor) memory.

Because of the complex interdependence among tasks in multiprocessor systems, there is no known technique for allocating memory and assigning tasks which will minimize memory access conflict. Conway\(^4\) has suggested that main memory be divided into small enough modules so that not more than one program occupies any given module. This approach will indeed reduce memory interference simply because of the large number of storage modules. But, it also increases the complexity of the memory interconnection system. And, severe interference can still result when two or more processors try to execute the same common subroutine or attempt to read the same data table.

Mosier\(^5\) refers to an assigned mode of multiprocessor operation in which each processor is semipermanently assigned a set of tasks. This mode of operation tends to decouple the multiprocessor into independent computers working into disjoint memory modules. But it does nothing to reduce interference in accessing common subroutines or data tables.

Replication of programs and data tables eliminates the interference problem associated with common subroutines and data tables.\(^15\) There are, however, two disadvantages to replication. The first is
the redundant use of memory. The second is the administrative problem introduced by having to allocate multiple copies of programs to processors and by having to keep multiple copies of data tables updated.

Address interleaving is a technique which distributes program and data fetches over all modules. Consecutive memory addresses are located in different modules. (For example, in an N-way interleaving scheme, all addresses for which \( [\text{address}]_{\text{modulo } N} = 1 \) are physically located in module 1.) Whenever two processors attempt to execute the same program or read the same block of data, they conflict initially and then tend to get out of phase so that they are accessing different modules at any given time. The SAFEGUARD system\(^{11}\) and the CDC 6600\(^{8}\) use address interleaving. Its major weakness is that branching and looping within a program generally produces new points of conflict.

Local memory can dramatically reduce memory access conflicts provided that a high percentage of memory requests are directed to local memory rather than to common memory.

One approach which is made possible by recent advances in memory technology is the use of associative cache buffers between processors and main memory.\(^{16}\) Briefly, operation is as follows. Each time a processor reads word from main memory, it writes the word and its main memory address into the associative buffer anticipating that that same word will be required again soon. Prior to going to main memory, a processor searches the cache buffer to ascertain whether the desired word is in the buffer. If so, considerable time is saved since the associative buffer is much faster than main memory. Temporary results
during computations are also stored in the cache buffer; only at the end of a computation are final results written back into main memory. The effectiveness of the cache buffering scheme depends upon the fact that short loops often exist in programs and upon the fact that particular data items may be needed many times throughout a computation. In some cases, the effectiveness of the buffering scheme can be improved by reading a block of words or a very wide word into the cache whenever a fetch from main memory is necessary.

Another solution to the storage interference problem is to restrict common storage to an absolute minimum. It is suggested in reference 16 that the trend will be to reduce the amount of shared memory in future multiprocessor designs. In principle, common data is the only information which must be directly accessible to all processors. Programs and local data can be stored in private per-processor modules. In such a system, storage interference occurs only when processors read or write common data. Program fetches are fast because they incur neither queuing delay nor the delay of a memory interconnection subsystem. Local data memory holds private data and temporary results and it acts as a buffer for data from common data memory. The lesser need for common storage modules relaxes design requirements on the memory interconnection subsystem. This, in turn, makes growth-ability a more realistic objective. This approach, however, introduces other problems. The functional assignment of programs to processors can reduce system flexibility. If a fairly even load balance among processors is not achieved, system efficiency will be impaired. And, a
control philosophy is required which will permit programs in different processors to communicate with each other without introducing a serious overhead penalty.

2.4.2 Common Memory Interconnection

Closely allied to the storage interference problem is the problem of designing a memory interconnection subsystem to interface between processors and common memory modules. There is a definite tradeoff in memory segmentation. From the standpoint of reducing memory conflicts, many small common memory modules are desirable. The hardware designer, on the other hand, when faced with the technical problems of providing many interconnections between processors and memory modules, is inclined to favor a design with fewer, larger memory modules. There is the added consideration that an upper bound in segmentation exists due to losses in signal timing resulting from excessively long busses. This is particularly true of integrated circuit technology where interconnections tend to limit performance and dominate cost.\(^{(17)}\)

Several techniques for interconnecting processors to common memory modules have been tried in existing multiprocessors.

1. Interconnection via private address and data paths between each processor and each memory module.

Priority circuits associated with each memory module resolve conflicts and queue requests. Example:

SAFEGUARD\(^{(11)}\) (See Figure 2-4).
2. Interconnection via an electronic crossbar switch which allows \( N \) parallel address and data paths to simultaneously exist between the \( N \) processors and any \( N \) of the \( M \) memory modules. Example: Burroughs D-825\(^{(9)}\) (See Figure 2-2).

3. Interconnection via a single bus per-processor which connects to all common memory modules. Priority circuits resolve access conflicts at the memory module interface. Example: IBM 9020\(^{(10)}\) (See Figure 2-3).

4. Interconnection via a time-shared bussing arrangement. Example: IBM STRETCH\(^{(13)}\) (See Figure 2-6).

All of these techniques require a considerable amount of common hardware which often saddles a system with high getting-started cost. Unless the design of the interconnection subsystem is modular and expandable, system growth is confined within a narrow range.

The crossbar switch is the most flexible method of interconnection since any processor can connect to any memory module in a fraction of a microsecond. But, it is also the most expensive. A full Burroughs D-825 system may require around 300,000 switch points.\(^{(18)}\) The large switch matrix is, however, an ideal candidate for LSI. It may even be possible to develop a switch which can grow modularly with the system.

The multiple bus approaches employed in the SAFEGUARD and IBM 9020 systems are less flexible than the crossbar switch approach. And, because of the tremendous numbers of bus wires required, they are not
well-suited techniques for a system which is realized with integrated circuits.

The least expensive approach in terms of hardware is the time-shared bus. It reduces interconnection to a minimum. In a system where the traffic to common memory is not too high, it is probably the best solution to the interconnection problem.

2.4.3 Memory Protection

Memory protection is any constraint placed on a system that prevents processors from interfering with each other as they access common memory. The conflict resolution algorithm used to insure that only one processor gains access to a particular memory module at any given time is a type of memory protection. Of more concern in this section is the common data protection problem, also known as the interwrite problem. Data must be protected from being updated by a processor until all other processors using the data are ready for its updating. Failure to employ this type of protection will result in erroneous and inconsistent results from the multiprocessor system.

One solution for common data is given by Roder and Rosene. A software table of contents lists common data tables and usage counts of the number of programs that are currently using the data. A processor wishing to check or modify the table of contents may do so only after the table has been locked to other processors. To modify a data table, a processor creates a new copy of the table, enters its address in the table of contents, and marks it "NEW." It then decrements by one the usage count associated with the old copy of the data table.
programs finish using the old data table, they decrement the usage count. When the count reaches zero the old copy is destroyed by removing its table of contents entry. All new requests are directed to the new copy.

The rings-of-protection approach employed by the Multics system is a particularly illuminating way of looking at the common data protection problem. A number of concentric circles are set up, each of which represents certain data. Processors obtain access to data on a security-clearance basis with the processor assigned the larger ring having access to all data represented by the smaller contained rings.

Various hardware techniques are often employed in multiprocessors to assist in the common data protection problem and to protect programs and private data which are stored in common memory. These include complete hardware lockout, mask registers, limit registers, and the associative memory allocation technique developed by Conway. Memory paging is another, somewhat indirect, method of hardware protection.

2.4.4 Executive Control

Executive control performs the management function in a multiprocessor. It controls the system so that processors act collectively, without interfering with each other, to accomplish overall system requirements.

The question immediately arises whether one processor should be designated the control processor or whether all processors should share
equally in executive control. Actually, there are three basic ways
to incorporate control into a multiprocessing system.\(^23\)

1) **Specialized Control**

A special processor, different in design from all
other processors, controls the entire system.

2) **Floating Control**

One processor, identical to all other processors,
is designated the control processor for some
period of time. During that period, it controls
the entire system. The selection of a control
processor and the period of nomination may vary.

3) **Distributed Control**

Each processor executes its own control as re­
quired along with any general control function
which may be required at that time on behalf of the
entire system.

The specialized control approach sacrifices much of the flexi­
bility, efficiency, and reliability of multiprocessing. The concensus
of opinion throughout the literature is that floating and distributed
control are most in keeping with the objectives of multiprocessing.\(^23\)

Task scheduling, task allocation, memory allocation, peripheral
control, and maintenance and system configuration control all fall
under the domain of executive control.

The hierarchy of control in a general purpose multiprocessor is
depicted in Figure 2-10.\(^18\)
HIERARCHY OF CONTROL

LOADER

EXECUTIVE

MAINTENANCE AND SYSTEM RECONFIGURATION

PERIPHERAL CONTROL PROGRAM

CONTROLS I/O
MONITORS I/O
ASSIGNS CHANNELS AND ACCESS ARMS ON DISCS
RE-RUNS TAPE ERRORS

SCHEDULER & TASK ALLOCATOR

QUEUES ON FACILITIES
ASSIGNS FACILITIES

OPERATING PROGRAMS

MEMORY ALLOCATOR

ASSIGNS MEMORY
MAINTAINS TABLES OF PROGRAMS AND LOCATIONS
HANDLES MEMORY ERRORS

FIGURE 2-10
CONTROL HIERARCHY IN A GENERAL PURPOSE MULTIPROCESSOR
The scheduler examines program requirements, checks availability of peripherals and memory, and determines whether a program should start immediately or be queued, awaiting some facility. The task allocator provides for delivering programs to processors when they are ready for execution. The memory allocator handles the allocation of memory space, overlay of programs, and the calculation of memory assignment algorithms. The peripheral control program assigns and controls I/O devices and handles communication with the system console. Maintenance and configuration control includes error detection, fault recognition, diagnostics, and system recovery. If interrupt capability is needed in the system, interrupt control is also an executive function.

Minimization of executive control overhead is desirable in a general purpose multiprocessor, but convenience to the user is of greater importance. Consequently, in most general purpose multiprocessors (and time-shared uniprocessors), executive control is highly sophisticated providing all sorts of special features for many types of users. In a real-time control multiprocessor, such luxury cannot be tolerated; simplicity and low overhead are essential.

Fortunately, the working environment of a real-time control system is highly prespecified. The set of tasks is known. Scheduling can, for the most part, be predetermined. Much of the memory space can be permanently assigned in advance; the need for dynamic storage allocation is limited. Peripheral control is much simpler since the system communicates primarily with machines rather than with man. Maintenance
and system configuration control is, however, quite important since high availability is usually a design requirement. The most critical function of executive control in a real-time multiprocessor is the rapid deliverance of tasks to processors so that externally imposed constraints on response time can be satisfied.

The SAFEGUARD System employs universal task lists for executive control. A universal task list for a radar tracking program in a two processor system is illustrated in Figure 2-11.

The top of Figure 2-11 diagrams the precedence relationships between tasks. All predecessor tasks of a given task must be completed before the task can be executed. The middle part of Figure 2-11 depicts the task list as it appears in common memory prior to the initiation of radar tracking. The lower part of Figure 2-11 shows how the tasks are executed in real-time on the two processors. The scheduler program absolutely enables tasks 1 and 2. At \( t_1 \) processor 1 selects task 1 and resets its absolute enable bit. Processor 2 selects task 2 at \( t_2 \). At \( t_3 \) processor 1 completes task 1, conditionally enables task 4, finds conditions \( c_1 \) and \( c_2 \) satisfied, and absolutely enables task 4. It then scans the list and picks up task 4. Similarly, at \( t_4 \), processor 2 absolutely enables both task 3 and task 7 and selects task 3. At \( t_5 \), the first processor, upon completing task 4, conditionally enables task 6. Since it has two predecessors, task 6 is not absolutely enabled until \( t_7 \) when task 5 is completed. The process continues until all tasks have been executed.

When a universal task list is completed, the scheduler comes in and resets the entire list. It may also analyze the list and reorder
RETURNS

PRECEDENCE RELATIONSHIPS AMONG TASKS

<table>
<thead>
<tr>
<th>Successor Tasks</th>
<th>Program Start Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>TASK 4c1</td>
<td>TASK 1</td>
</tr>
<tr>
<td>TASK 3c1,7c</td>
<td>TASK 2</td>
</tr>
<tr>
<td>TASK 5c1</td>
<td>TASK 3</td>
</tr>
<tr>
<td>TASK 6c1</td>
<td>TASK 4</td>
</tr>
<tr>
<td>TASK 6c2</td>
<td>TASK 5</td>
</tr>
<tr>
<td>TASK 6</td>
<td>TASK 6</td>
</tr>
<tr>
<td>TASK 7</td>
<td>TASK 7</td>
</tr>
</tbody>
</table>

UNIVERSAL TASK LIST PRIOR TO INITIATION OF RADAR TRACKING

PROCESSOR ACTIVITY DURING EXECUTION OF RADAR TRACKING PROGRAM

FIGURE 2-11
A SAFEGUARD UNIVERSAL TASK LIST
tasks or change their precedence relationships. The linear ordering within each task list is based on analysis of task timing within the required external execution time frames. In the SAFEGUARD System, these lists are prepared in advance on an external IBM 360 machine.

Executive control programs require a data base, usually in the form of various tables. These tables contain information such as locations of the programs, precedence relations among tasks, task lengths, dynamic memory map, I/O control information, etc. Even in a real-time control system where many executive operations are predetermined, considerable overhead is consumed locating and modifying tabular information. In SAFEGUARD, for example, the task list has to be scanned sequentially each time a processor selects a new task. Associative memories offer an attractive solution to this problem. With their parallel search capability, they can simultaneously examine all words in a table and select one which has some desired property.

Gunderson, Heimerdinger, and Francis\(^{(25)}\) first recognized the advantages of associative control in a multiprocessor. They proposed a general purpose multiprocessor system which uses three associative control memories. A block diagram of their system is shown in Figure 2-12.

Referring to Figure 2-12, the segment word associative memory is the central control point of the system. It contains segment words, each of which contains the status, identity, and the description of a segment. (A segment, as defined by Gunderson et. al. is equivalent to a task as defined in Section 2.2). It is from this set of words
SEGMENT WORD ASSOCIATIVE MEMORY

I/O COMMAND ASSOCIATIVE MEMORY

PAGE CONTROL ASSOCIATIVE MEMORY

PROCESSOR SUBSYSTEM

MAIN MEMORY

INPUT/OUTPUT SUBSYSTEM

SECONDARY STORAGE SUBSYSTEM

FIGURE 2-12
GUNDERSON’S ASSOCIATIVELY CONTROLLED MULTIPROCESSOR
that each processor selects new work whenever it completes its present assignment. The parallel search capability of the segment word associative memory allows the highest priority segment to be selected for which the required pages are in main memory. A normal segment word designates three pages: the job page, a procedure page, and a data page which contain all the information necessary to execute the segment.

The input/output command associative memory contains I/O commands. I/O processors in the I/O subsystem select these commands and assign them to I/O channels. The ability to examine all I/O commands simultaneously allows I/O device usage to be optimized. The page word store controls the movement of pages, which are the basic blocks into which all data files and procedures are divided, between main memory and secondary storage.

Berg and Johnson(27) describe an associative control scheme for executive control in an advanced avonic computer system. Their system contains a specialized executive control processor, called the ECU, which uses associative memory to assist in performing executive functions. The ECU supervises all job assignments, data transfers, and overall system control. It also coordinates the dynamic configuration of variable sized processors from a number of byte-sized processing elements for the execution of modular programs, and controls the allocation of tasks to these processors.
CHAPTER III

THE PROPOSED SYSTEM

An overview of the important aspects of the ACFM design is presented in this chapter.

Section 3.1 enumerates the main objectives of the design. In Section 3.2 the system structure is introduced. The remaining sections of the chapter consider each of the major subsystems. Where there is considerable latitude in the design depending upon the nature of the application, several options are presented to illustrate the flexibility of the structure.

The memory structure is discussed in Section 3.3. The major types of local and common data memory which can be incorporated into the ACFM structure are considered; several time-shared bus designs for connecting common memory modules to processors are presented; and, three methods for dynamically allocating common data memory space are described.

The basic associative control concepts are introduced in Section 3.4. Associative memory operation is described and a simplified description of the ACFM control philosophy is progressively developed.
The flexibility of the ACFM in accommodating a variety of processor designs is discussed in Section 3.5. The input/output subsystem is considered in Section 3.6. Finally, Section 3.7 summarizes the chapter.

3.1 Design Objectives

The principle design objectives for the ACFM development are:

1. The design should take advantage of the following characteristics of real-time processing:
   a) The set of supervisory and application programs is fixed.
   b) The arrival rates of input stimuli are known.
   c) Response time requirements are specified.

2. The method for controlling the system should be simple. The overhead in delivering tasks to processors for execution should be minimal.

3. The hardware structure should eliminate system bottlenecks as much as possible. Shared subsystems should be reduced to the minimum required for control and for providing sufficient access to the common data base.

4. The structure should be modular so that the design can easily be customized to the needs of particular applications.

5. The structure should accommodate growth over a wide range of capacity requirements. The getting-started cost of small installations should be minimal.
6. The system should be easy to program. Software should be independent of the number of processors in the system.

7. The design should make optimum use of integrated circuit technology.
   a) The intermodular interconnection should be minimal or at least regular. (Busses are regular interconnections.)
   b) The modules should be regular in their logical structure in order to facilitate their implementation with LSI. This implies that various types of memory and functional chips should be used in preference to irregular logic.
   c) The number of chip and substrate types should be small.

8. The system should be both reliable and maintainable. It should degrade gracefully under failure and should absorb overloads without performance degradation.

3.2 The Basic Structure

The approach taken in the development of the ACFM architecture was, first, to take advantage of the more specified nature of real-time processing in order to reduce the amount of shared memory in the system and therewith the seriousness of the storage interference problem and, second, to develop control techniques which enable the system to perform efficiently as a multiprocessor.
A block diagram of the ACFM is shown in Figure 3-1. All program memory and a portion of data memory is dedicated on a per-processor basis. The shared elements in the system are an associative control subsystem, common data memory, a secondary storage subsystem, and an input/output subsystem. Time-shared busses connect these common subsystems to processors.

The set of application programs, supervisory programs, and support programs for a given real-time application are broken down into tasks which are assigned semipermanently to particular processors. It is in this sense that the ACFM is a functional multiprocessor. Available statistics on the real-time usage of the various tasks provide the information necessary to achieve an even distribution of the real-time load among processors. The secondary storage subsystem, which contains a copy of all programs and permanent data in the system allows rarely used programs to be brought into the system on demand and permits programs to be redistributed among processors whenever processor failures or load imbalances require it.

The system can accommodate virtually any type of processor. The hardware and software which processors require to perform executive operations can be incorporated into a microprogrammed interface unit which sits between each processor and the shared busses, or it can be an integrated portion of the processor design itself. The single term processor will be used throughout the dissertation to refer to both implementations.
FIGURE 3-1
THE BASIC ACFM ARCHITECTURE
Control in the ACFM is distributed; each processor performs executive functions on its own behalf. The executive control functions required in the ACFM differ from those in a direct multi-processor organization. There is no dynamic task allocation function since tasks are functionally preassigned to processors. This functional assignment, however, introduces a control problem with inter-task communication. The path of a particular transaction through the various tasks in the system may result in considerable inter-leaving among processors of the total processing of that transaction. Consequently, the success of the ACFM structure depends upon the ability to smoothly pass control between tasks assigned to different processors.

An associative memory, called the Executive CAM (Executive Control Associative Memory) contains a word for each task in the system which specifies, among other things, the name of the task, the processor to which the task is assigned, the task's priority, and an enable bit which indicates whether the task has been called for execution. When a functional assignment of tasks to processors is made, the Executive CAM is initialized to reflect this assignment. Although task scheduling is largely prespecified by the functional assignment of tasks to processors, any particular processor schedules its own work by searching the Executive CAM for that task assigned to it which is ready for execution and which has highest priority. Priorities may be dynamically altered to effect more efficient scheduling whenever the application requires it. A second associative memory, called the
QCAM (Queue Control Associative Memory), queues requests to individual tasks in the system. Each entry in the QCAM identifies the task which is being queued for and carries with it the information needed to locate the data to be processed and to provide linkage back to the calling task when required. After a processor has determined which task to execute by searching the Executive CAM, it searches the QCAM to pick up a request to that task. To call a task, a processor first places a request in the QCAM and then sets the enable bit in the Executive CAM word associated with the called task. The control structure of the ACFM is effectively a task queuing network in which processors are servers.

Associative control techniques can also be used for dynamic storage allocation, paging, I/O control, and system reconfiguration control. These techniques are not, however, developed in this dissertation because they have been discussed elsewhere in the literature.

3.3 Memory Structure

The memory organization of the ACFM is based on the observation that the only centralized storage inherently necessary for multiprocessing is that which is required to store the common data base upon which the processors collectively operate. The advantages of reducing the amount of centralized storage are almost obvious: Performance is improved since a high percentage of memory accesses are directed to local memory where there is no opportunity for storage interference. Interconnection requirements between processors and memories are simplified. This, in turn, makes cost effective growth
a more realistic objective. Reliability is enhanced since a greater percentage of system hardware is distributed among independent units. Much of the memory protection problem is eliminated by the increased use of local storage; it is physically impossible for one processor to obliterate the program memory or local data memory of another.

The memory structure of the ACFM is intended to be flexible so that the design can be customized to meet the needs of each application. The number of memory modules and their characteristics, the type of memory interconnection subsystem, and the memory allocation strategy all depend upon the particular application. This section presents a general discussion of the ACFM memory structure.

3.3.1 Local Memory

The ACFM memory organization includes both local program memory and local data memory. As already described, all task programs are stored in local program memories. Local data memory has several functions. It stores private data belonging to the task programs in the corresponding local program memory, it serves as a scratch area for the storage of local transient information, and it buffers data brought in from common data memory.

In some applications, it may be desirable to provide both random-access and associative local data memory. The associative memory serves as a cache buffer between the processor and common data memory as described by Gunderson. It also implements mapping functions such as those which relate the name of a task, as it appears in the
Executive CAM, to the physical address of the task in local program memory and those which relate symbolic data addresses in the task programs to physical addresses in local data memory. In applications where the associative local data memory does not exist, such mapping functions are implemented by indirect addressing techniques. (27)

The address spaces of both local program memory and local data memory are subdivided into blocks in order to facilitate the assignment of task programs and their associated data to processors. Block sizes depend upon the particular application but probably fall in the 256 to 1024 word range. Each task program occupies some number of contiguous program memory blocks and requires a certain number of data memory blocks. Task programs always begin at the first address of a block.

The block structure of local memory provides the relocatability required for the redistribution of tasks among processors and also lays the groundwork for possible general purpose applications in which dynamic paging between secondary storage and local memory is necessary.

Figure 3-2 summarizes the types of local memory and their functions in the ACFM.

3.3.2 Common Data Memory

The design details of the common data memory subsystem: the number and type of modules, module access times, word size, module size, the method of interconnecting modules with processors, and the method of storage allocation depend heavily upon the data structure and the data flow requirements of the particular application. Basically, the
## Figure 3-2

**ACFM Local Memory Layout**

<table>
<thead>
<tr>
<th>Local Program Memory</th>
<th>Local Data Memory</th>
<th>Local Associative Memory (Optional)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 1 of Task A</td>
<td>Data Block 1 for Task A</td>
<td>Data Block 1 for Task C</td>
</tr>
<tr>
<td>Block 1 of Task B</td>
<td>Data Block 1 for Task B</td>
<td>Data Block 1 for Task C</td>
</tr>
<tr>
<td>Block 1 of Task C</td>
<td>Data Block 1 for Task C</td>
<td>Data Block 1 for Task C</td>
</tr>
</tbody>
</table>

- **Data Block**
  - Block 1 for Task A
  - Block 1 for Task B
  - Block 1 for Task C

- **Address**
  - Associative Cache Buffer for Common Data Memory Interface
  - Mapping of Task Names into Program Starting Addresses
  - Mapping of Symbolic Data Addresses into Data Memory Blocks
common data memory subsystem must have sufficient bandwidth (maximum rate of data flow) to keep the N processors supplied with data.

Several types of common data memory may be incorporated into the system:

1. High-speed random access memory is necessary for passing information between processors, for holding shared transient data, and for storing frequently accessed items from the common data base.

2. Low-speed random access data memory is needed whenever the application requires a large common data base in which any data item must be readily accessible but in which any particular data item is infrequently accessed.

3. Associative common data memory may be justifiable in applications where several levels of indirect addressing would otherwise be necessary in order to retrieve a data item. It is also useful in applications which require dynamic allocation of common data memory.

4. Secondary storage media such as disks, drums, or BORAMs (Block Oriented Random Access Memories) are needed to store large data files, a copy of permanent system data, and a copy of all programs.
Many common memory configurations are possible in the ACFM. The three-dimensional object identification application for example, requires large data files which represent the various objects that can be identified. An appropriate data memory structure is shown in Figure 3-3. The data files are stored in the BORAM which has an access time of several microseconds and a word transfer rate of 10 megahertz. Data is block transferred to high speed common data memory for processing. The associative cache buffers significantly reduce interference since the application involves long computations on localized blocks of data. (See Chapter VI.)

Other applications, such as telephone switching control, require random access to individual words in a very large common data base. A small percentage of the data are frequently accessed while the vast majority are referenced only infrequently. The common data memory structure, illustrated in Figure 3-4, consists of an appropriate balance between high-speed and low-speed random-access memory. (See Chapter VII.)

3.3.3 Memory Interconnection

Various techniques for interconnecting processors and common memory modules were discussed in Chapter II (Section 2.4.2). Of these, the time-shared bus is most consistent with the design objectives of the ACFM since it involves the least amount of interconnection. It can be engineered to sustain a high data flow rate. If, however, data flow requirements exceed the capacity of a single time-shared bus, additional busses can be added or a different method of interconnection, such as the electronic crossbar switch, can be employed.
FIGURE 3-3
ACFM CONFGURATION FOR FILE-ORIENTED APPLICATION
FIGURE 3-4

ACFM CONFIGURATION FOR TELEPHONE SWITCHING CONTROL
3.3.3.1 Time-Shared Bus for High-Speed Common Data Memory Access

An asynchronous parallel time-shared bus design for connecting processors with high-speed common data memory is depicted in Figure 3-5. Any processor may seize the input bus to transfer an address, an opcode, and optionally, data to be written to the desired memory module. Any memory module may seize the output bus to return a data word to a processor. The bus controller coordinates bus traffic.

A state diagram for the bus controller is shown in Figure 3-6. There are two independent control sequences: one for the input bus and another for the output bus.

When the input bus is idle, the input portion of the bus controller is in state 1 cyclically monitoring the bus for processor requests. When a processor request is recognized, state 2 is entered. The controller gates the input information onto the bus and tests the status of the desired memory module. If the module is available, a transition is made to state 3. The controller gates the input information from the bus into the module’s input register and records the number of the requesting processor so that the data word can be routed to the correct processor at the completion of a read operation. State 4 is then entered. The information is removed from the bus and a transition is made back to state 1. If the desired memory module is busy, the controller goes from state 2 directly to state 4 and then back to state 1. The unsatisfied request will be selected again after any waiting requests from other processors have been serviced.
FIGURE 3-5

HIGH-SPEED PARALLEL TIME-SHARED BUS
MONITOR INPUT BUS

PROCESSOR REQUEST RECOGNIZED

GATE INPUT INFORMATION ONTO BUS AND TEST DESIRED MEM MODULE

IDLE

REMOVE INPUT INFORMATION FROM BUS

GATE INFORMATION INTO MEMORY MODULE AND RECORD PROCESSOR NO.

INPUT BUS CONTROL

MEMORY REQUEST RECOGNIZED

GATE DATA WORD ONTO OUTPUT BUS

LOOK UP PROCESSOR NO. AND GATE DATA WORD TO THAT PROCESSOR

REMOVE INFORMATION FROM BUS

OUTPUT BUS CONTROL

FIGURE 3-6

STATE DIAGRAM FOR TIME-SHARED BUS CONTROLLER
The output portion of the bus controller cyclically monitors the output bus for memory requests (state 5). When one is recognized, the controller enters state 6. The data word which has been read is gated and the output bus and state 7 is entered. The controller looks up the appropriate processor number and gates the data word from the bus into the processor's data access register. State 8 is then entered. The controller removes the information from the output bus and goes back to state 5. On write operations the output portion of the bus controller is not needed.

3.3.3.2 Low-Speed Common Data Memory Interconnection

The function of low-speed random-access common data memory is to provide low cost storage for large amounts of data in applications where individual words from the data base, though accessed infrequently, must be available at microsecond speeds. Because the utilization of this type of memory is low, module size can be very large. The characteristically low access rate simplifies the interconnection problem. In many applications, low-speed common data memory modules can share the parallel time-shared bus described in the previous section with high-speed modules. In some applications, however, the access rate to low-speed common data memory may warrant a separate bus system. Two possible bussing arrangements, both serial, are presented here.

The first structure, depicted in Figure 3-7, is a serial electronic crossbar switch. When a processor desires access to a memory
FIGURE 3-7
SERIAL ELECTRONIC CROSSBAR SWITCH
module, the appropriate crosspoint is closed in the matrix, provided, of course, that the desired module is free. A single lead connects the processor and memory. The address, control information, and any data to be written is transmitted from a shift register in the processor to a similar shift register in the memory. The memory performs the read or write operation and, on a read, transmits the data word back to the processor. Assuming that each module contains 512K (where K = 1024) 64-bit words and that the module access time is 2 microseconds, a processor can obtain a word in under 3 microseconds. This is based on the assumption that the serial transmission rate is 100 megahertz (10 ns per bit).

The second interconnection scheme, depicted in Figure 3-8, is a variation of the time-shared bus idea. A single serial bus, which connects to all processors, is associated with each memory module. When a processor desires access to a memory module, it submits a request to the controller. If the module is idle, the processor is connected to the associated bus. It remains connected until the memory operation is complete. Serial transmission is effected in the same manner as described for the crossbar switch technique.

Both of the above methods provide low cost interconnection between processors and low-speed common data memory. The crossbar switch, though more expensive than the time-shared bus scheme, is superior for applications which require more than a few modules. The time-shared bus scheme is appropriate when the number of modules is small but is less attractive for large numbers of modules because a termination per module is necessary at each processor.
FIGURE 3-8
SERIAL TIME-SHARED BUS ARRANGEMENT
3.3.4 **Common Data Memory Allocation**

The need for dynamic storage allocation is not so great in a real-time system where the software structure is relatively static as it is in a general purpose system where there is considerable program and data traffic in and out of main storage. The tendency in real-time software design is to assign fixed addresses to programs and data tables in order to avoid, as much as possible, the added time associated with indirect addressing or with dynamic allocation and deallocation of storage. Even in general purpose systems, there is a tendency to assign fixed addresses to operating system programs and data areas.

In the ACFM structure, the assignment of local program and data memory does not change frequently. Once they are assigned, the addresses of task programs and their associated data blocks remain fixed until an unusual condition, such as a processor failure or a load imbalance, necessitates a reassignment. Fixed addresses are assigned to items in low-speed common data memory so that they can be retrieved directly. In most applications, a portion of high-speed common data memory will also have a fixed assignment. There is, however, a need to dynamically allocate and deallocate at least a portion of high-speed common data memory.

This need arises because processors use high-speed common data memory as a scratch area for passing data between tasks. Any particular task may have a queue of requests of arbitrary length waiting for it. With each request, there may be an area in common data memory which contains the data to be processed. This requirement
is most efficiently satisfied by the dynamic allocation of memory space. The need for dynamic memory allocation also arises in applications where the amount of common memory space required is a function of the type of input stimulus. It may not be economical to reserve enough space for the worst case.

The development of storage allocation strategies is not an objective of this research. Storage allocation problems are of no greater magnitude in the ACFM architecture than they are in other multiprocessor configurations. It is, however, appropriate to summarize several common data memory allocation philosophies which may be suitable for certain ACFM applications. (4, 19, 27, 28)

3.3.4.1 Method 1

The most general approach to dynamic storage allocation is to provide an associative common data memory module which controls the assignment of all or of a subset of high-speed random-access common data memory. Each word in the associative memory contains the address of a block of words in random-access memory and a status indicator. To allocate a block of words, a processor initiates a search over the status indicator field looking for an idle block. The associative memory marks the selected block busy and returns its address to the processor. Deallocation is accomplished by searching over the block address field and resetting the status indicator of the matching word. This approach can be expanded to effect more sophisticated strategies such as the allocation of blocks of arbitrary size.
3.3.4.2 **Method 2**

Each processor is assigned a subset of common data memory over which it has control. The processor maintains a dynamic map of its section of common data memory; no other processor can allocate or deallocate storage in that section, but any processor can read or write there. When a processor calls a task, it allocates enough space to hold the data being passed to the called task and the data to be returned to the calling task. When the calling task is returned to, the processor retrieves the returned information and deallocates the data area.

3.3.4.3 **Method 3**

When a transaction enters the system for processing, a storage area is allocated to hold all transient information which may be associated with the transaction during its lifetime in the system. Any task which is to perform services on behalf of a transaction is passed the address of its storage area so that it can obtain access to necessary information concerning the status of the transaction. When a transaction leaves the system, its storage area is released. Either Method 1 or Method 2 can be used to allocate and deallocate the storage areas. Or, linked list techniques can be employed to control the assignment of a pool of storage areas.

3.4 **Associative Control**

The capability of associative memories in performing search or processing operations simultaneously over all stored words can greatly simplify and speed up executive control operations in a multiprocessor.
The control operation which is most critical to the success of the ACFM is intertask communication, that is, the means whereby the "processing thread" is interleaved among processors having particular functional capabilities in order to accomplish overall processing requirements on behalf of some transaction. A primary objective of this research is to develop the intertask communication function in sufficient detail to demonstrate that the ACFM is an efficient architecture for real-time control applications.

This section presents a brief discussion of the operation of associative memories and then gives a simplified description of intertask communication.

3.4.1 Associative Memories

Associative or content-addressable memory differs from location-addressable random-access memory in the following ways:(29)

1) The address is not needed for storing or retrieving information.

2) During the retrieval process, information is simultaneously sensed from all (or a subset of all) locations to determine if it is associated with the desired information.

3) All associated information can be retrieved if desired, that is, the output process is multivalued.

In an associative memory, data is retrieved by specifying the information content of an arbitrary portion of a word. The specified
portion of the data is called a search key. The search key issimultaneously matched against all words in the memory. Bits which
are not part of the search key are called "don't cares"; they always
match. The matching word can be read out or modified and written
back. Multiple matches are handled by a resolver circuit which
selects one word, according to some algorithm, from among those which
match. The resolver circuit of the associative memory designed by
Koo(29), for example, selects the matching word with lowest physical
address.

Although location-mode addressing is not necessary for associative
operation, associative memories have an address system and can be
operated as random-access memories. An associative read or write is
actually a two-step operation: an address is obtained as a result of
the search, and then the memory is operated as a random-access memory
using that address.

Associative memories have bit-write capability, that is, any
portion of a word can be written without changing the contents of the
remainder of the word. Two types of content-addressed write operations
are possible. In the first type, an address is obtained as a result
of a search operation. The word at the specified address, or a
portion of it, is then written via a location-addressed write. In the
second type, the search operation identifies all matching words. The
bit drivers of the memory are then activated to simultaneously write
the same information in specified bit positions of all matching words.
Although the equality search operation is the most common and easiest to implement search algorithm in an associative memory, other algorithms are often useful such as maximum search, minimum search, less than or equal search, etc. In addition, associative memories can be designed to do processing functions such as to simultaneously increment or decrement certain fields in all matching words.

Recent advances in semiconductor technology have made high-speed associative memories a reality. Several designs have been proposed in the literature.\(^{29,20,21}\) By the mid 1970s the time for performing the basic operations of read, write, and equality search can be expected to fall in the 30 to 50 nanosecond range for memories of up to 50,000 bits.\(^{16}\)

The associative memory assumed for the control function in the ACFM is shown in Figure 3-9. It is capable of both content-addressable and location-addressable modes of operation. It can perform equality searches over any specified portion of a word. In addition, it is capable of maximum or minimum searches over one particular field called the priority field. Other features such as increment/decrement capability over certain fields are not specified here, but may be useful in certain applications.

The basic operations of the associative memory depicted in Figure 3-9 are content-addressed read, content-addressed write, location-addressed read, location-addressed write, and maximum/minimum priority read. The resolver circuit is of the type described by Koo\(^{29}\); it selects the matching word with lowest physical address.
TO AND FROM PROCESSORS

OP CODE
PROCESSOR INTERFACE REGISTER

CONTROL

ADDRESS REGISTER

ADDRESS DECODING CIRCUIT

RESOLVER

ADDRESS OF MATCHING WORD

SEARCH REGISTER

WRITE REGISTER

WRITE MASK REGISTER

ASSOCIATIVE MEMORY CELLS

SENSE AMPLIFIERS AND BIT DRIVERS

OUTPUT REGISTER

FIGURE 3-9
ASSOCIATIVE MEMORY FOR ACFM CONTROL
On a **content-addressed read**, the search mask register is loaded with a mask which specifies the bits which make up the search key. The search key itself is loaded into the search register and filtered through the search mask register to initiate an equality search operation. The resolver determines the address of the matching word which is then used in a location-addressing mode to load the desired word into the output register.

A **content-addressed write** begins in exactly the same way as a content-addressed read. Once the address of the matching word is determined, the write is performed in the same manner as in a location-addressed memory. The write register contains the information to be written. The write mask register specifies which bits are actually to be written and which bits are to be left unchanged.

An address is externally supplied for **location-addressed reads and writes**. The random-access portion of the memory is then utilized to carry out the desired operation. The contents of the write mask register specifies which bits are to be written on write operations.

A **maximum or minimum priority** read selects, from among those words which match the search key, the one which has a maximum or minimum value in the priority field. The cells which make up the priority field are specially designed to perform the maximum search function.

---

3 A possible implementation of the priority search is as follows: A content-addressed write simultaneously writes a management bit in all words which are to be involved in the maximum or minimum priority search. The priority search is then initiated to examine the priority of all words whose management bit is set. The desired word is gated to the output register. A content-addressed write then clears the management bits.
When a processor presents a request to the associative memory shown in Figure 3-9, it supplies an opcode which specifies not only what basic operation is to be performed but also which fields of the word are involved. This opcode is used to interrogate a read-only memory which supplies internal data operands, such as mask register contents, needed to perform the desired operation. This read-only memory makes it possible for processors to execute control operations by supplying a single word to the associative memory and receiving a single word in return.

3.4.2 Simplified View of Intertask Communication

The basic control problem in the ACFM is how to smoothly pass control between tasks which may be functionally assigned to different processors. To shed light on this problem, consider the two-processor system depicted in Figure 3-10. Tasks A and B reside in the private program memories of processors 1 and 2, respectively. Suppose task A calls task B. How is control passed?

The Executive CAM contains a word for each individual task in the system. This word consists of the task's unique name code, a unique name code for the processor to which the task is assigned, an enable bit, E, that indicates when the task has been called, and various other fields which are described in Chapter IV.

When, while executing task A, processor 1 encounters a Call Task B instruction, it initiates an associative write into the Executive CAM to set the enable bit for task B. The search key for this write consists of the coded name for task B in the task name field and don't
FIGURE 3-10

TWO-PROCESSOR SYSTEM WITH EXECUTIVE CAM
cares everywhere else. After enabling task B, processor 1 is free to look for other work. A processor looks for work by initiating an associative read of the Executive CAM using its own namecode and \( E = 1 \) as a search key. Presumably, processor 1 selects some other task to execute. Processor 2, the next time it looks for work, selects task B since task B is assigned to it and has been enabled by the call from task A. Processor 2 transfers control to task B after issuing an associative write to the Executive CAM to reset the enable bit for task B. The operation of passing control from task A to task B is summarized in Figure 3-11.

The previous example illustrated how processors pass control to tasks assigned to other processors and how processors select work. No provision was made, however, for passing data between tasks. How is this accomplished?

The Queue Control Associative Memory (QCAM) is incorporated into the system. The two-processor system of Figure 3-10 is shown in Figure 3-12 with the QCAM added. When processor 1 executes the Call Task B instruction, it first writes a queue word for task B into the QCAM and then enables task B in the Executive CAM. The queue word consists of the name code for task B, either a single data item to be passed or an address of where data being passed is located in common data memory, and other information to be described later. After processor 2 selects task B via the Executive CAM, it initiates an associative read of the QCAM using as a search key the name code for task B and don't cares elsewhere. Once it has read the QCAM word, processor 2 retrieves
EXECUTIVE CAM WORD FOR TASK B PRIOR TO ITS BEING CALLED BY TASK A

PROCESSOR 1 ISSUES ASSOCIATIVE WRITE TO ENABLE TASK B

EXECUTIVE CAM WORD FOR TASK B AFTER IT IS ENABLED

PROCESSOR 2 ISSUES AN ASSOCIATIVE READ TO SEARCH FOR WORK

PROCESSOR 2 ISSUES ASSOCIATIVE WRITE TO RESET ENABLE BIT

\[ \emptyset = \text{DON'T CARE (\emptyset \text{ BITS ALWAYS MATCH)}} \]

\[ U = \text{UNCHANGED (U BITS ARE LEFT UNCHANGED BY A WRITE)} \]

**FIGURE 3-11**

ASSOCIATIVE OPERATIONS REQUIRED TO PASS CONTROL FROM TASK A TO TASK B
FIGURE 3-12

TWO-PROCESSOR SYSTEM WITH QCAM INCLUDED
the passed data either from the QCAM word itself or from common data memory. It then begins executing task B. (Control bits in the QCAM word indicate whether data is being passed directly via the QCAM word or indirectly via common data memory.)

One might question the necessity of a second associative memory, the QCAM, for passing data items. Why not pass data items or data addresses via the Executive CAM? The answer to this question is that, in general, a particular task may have a queue of requests waiting for it. The single Executive CAM word assigned to a given task cannot keep track of an arbitrary number of requests to that task. For the system to run smoothly a processor calling a task must be able to post its request in the Executive CAM and go on to further work. The QCAM makes this possible.

Consider the three-processor system depicted in Figure 3-13. Suppose that task A has called task B and that task C also wants to call task B. If the Executive CAM word were used to pass data, processor 3 would have to wait until processor 2 had begun processing the call to task B from task A before posting its request, since the Executive CAM word for task B would not be available until then. With the QCAM in the system, processor 3 simply enters a queue word into the QCAM, enables task B in the Executive CAM and looks for more work. When it becomes free, processor 2 processes all requests in the queue for task B before looking for further work. (In the actual system, it is optional whether a processor exhausts the queue for a given task before looking for other work.)
EXECUTIVE CAM

<table>
<thead>
<tr>
<th>TASK A</th>
<th>PROCESSOR 1</th>
<th>OTHER INFORMATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>TASK B</td>
<td>PROCESSOR 2</td>
<td>OTHER INFORMATION</td>
</tr>
<tr>
<td>TASK C</td>
<td>PROCESSOR 3</td>
<td>OTHER INFORMATION</td>
</tr>
</tbody>
</table>

QCAM

<table>
<thead>
<tr>
<th>TASK B</th>
<th>DATA OR ADDRESS OF DATA</th>
<th>OTHER INFORMATION</th>
</tr>
</thead>
</table>

QUEUE OF TWO REQUESTS FOR TASK B

CPU 1
PROGRAM MEMORY 1
TASK A

CPU 2
PROGRAM MEMORY 2
TASK B

CPU 3
PROGRAM MEMORY 3
TASK C

FIGURE 3-13
THREE-PROCESSOR SYSTEM ILLUSTRATING QUEUING OF REQUESTS IN THE QCAM
The examples given thus far have shown how tasks assigned to one processor pass control, including data items, to tasks assigned to other processors. The use of the QCAM for handling queues of requests to a particular task has been described. No procedure has been developed, however, for initiating a subroutine call to a task which later returns control to the calling task. Consider again the two-processor system of Figure 3-12. Suppose task A in processor 1 calls task B in processor 2 and sets up return linkage. When task B is completed, control is returned to task A. This operation involves four steps:

1. Processor 1 passes control to task B including passed data and return linkage information.
2. Processor 2 selects task B, obtains the passed data, and executes task B.
3. Processor 2 sets up return linkage to task A including data results.
4. Processor 1 picks up the return from task B and reactivates task A.

Task A is said to be suspended from the time that the call to task B is initiated until the return from task B is processed by processor 1.

A more detailed description of passing control with return linkage is now given: When processor 1 encounters the instruction Call Task B, Ret Addr it builds a queue word having the form shown in Figure 3-14. The return linkage information includes the name code of the calling processor (processor 1), the physical address in private
<table>
<thead>
<tr>
<th>TASK B</th>
<th>DATA OR ADDRESS OF DATA BEING PASSED</th>
<th>CALLING PROCESSOR (PROCESSOR I)</th>
<th>RETURN ADDRESS</th>
<th>OTHER INFORMATION</th>
</tr>
</thead>
</table>

**FIGURE 3-14: QCAM WORD FOR TASK CALL WITH RETURN LINKAGE**

<table>
<thead>
<tr>
<th>SPECIAL RETURN CODE</th>
<th>DATA OR ADDRESS OF DATA BEING RETURNED</th>
<th>CALLING PROCESSOR (PROCESSOR I)</th>
<th>RETURN ADDRESS</th>
<th>OTHER INFORMATION</th>
</tr>
</thead>
</table>

**FIGURE 3-15: QCAM WORD FOR RETURN TO CALLING TASK**

<table>
<thead>
<tr>
<th>SPECIAL RETURN CODE</th>
<th>0-----0</th>
<th>PROCESSOR NAME</th>
<th>0-----0</th>
<th>0-----0</th>
</tr>
</thead>
</table>

**FIGURE 3-16: SEARCH KEY FOR RETURN SEARCH**
program memory to which control is to be returned, and information which allows the state of task A, at the time it was suspended, to be retrieved when control is returned. Processor 1 makes a note of the fact that it expects a return from a called task by incrementing a counter called the Return Counter (RC). It then enables task B via the Executive CAM. Processor 1 now looks for other work.

Processor 2 eventually selects task B and associatively reads the queue word from the QCAM. This queue word is retained in the QCAM in order to preserve the return linkage information for later use. Processor 2 obtains the passed data and executes task B. When it has finished task B, it modifies the reserved queue word as shown in Figure 3-15. The data to be returned, or its address in high-speed common data memory is placed in the second field of the queue word. The name of the calling task is replaced by the name code of a special task called RETURN. Processor 2 is now free to look for other work.

Whenever the return counter of a processor is not zero, that processor checks for a return from a called task before looking for other work. To check for a return, the processor does an associative read of the QCAM using the search key shown in Figure 3-16. If it finds a match, the processor uses the return information to reactivate the calling task. In the present example, processor 1 eventually picks up the return from the QCAM, fetches the returned data, and reactivates task A.

A key point in the ACFM control philosophy is that a processor is immediately free to select and execute another task as soon as the
task being executed is either completed or suspended. When task A calls task B in the previous example, task A is suspended. Processor 1 can select and execute some other task in the interim period while processor 2 is executing task B. Thus, although the processing of a particular transaction, say transaction i, may be interleaved among several processors, each of the processors involved can be doing useful work on behalf of other transactions when it is not executing tasks on behalf of transaction i.

The state diagram in Figure 3-17 describes the executive control functions performed by each processor. It summarizes the control philosophy as it has been developed thus far. A processor spends most of its time in state 1, the task execution state. It leaves state 1 whenever the task being executed either places a call to some other task or completes execution.

When the task being executed places a call to some other task, state 2 is entered. While it is in state 2, the processor formats a QCAM word, writes it in the QCAM, and then enables the called task via the Executive CAM. If the call does not establish return linkage, the processor enters state 4 and terminates the calling task. If, however, return linkage is set up, the processor advances to state 3 and suspends the calling task. After the actions of either state 3 or state 4 are complete, the processor advances to state 7 to begin looking for new work.

A task is terminated when it places a call without return linkage to another task or when it completes execution either by returning to
FIGURE 3-17
SIMPLIFIED STATE DIAGRAM OF THE EXECUTIVE CONTROL
FUNCTIONS PERFORMED BY A PROCESSOR
a calling task or by encountering an END macro. When an END macro is encountered, the processor moves from state 1 to state 4, terminates the task and enters state 7. A return to a calling task is effected by entering state 5. In state 5, the processor writes the special RETURN code and the data to be returned, or its address, in the QCAM word which was reserved when the task was originally called. It then enters state 4, terminates the called task, and advances to state 7.

In state 7, the processor tests the Return Counter (RC). If RC > 0, the processor searches the QCAM for returns to suspended tasks. If a return is found, the processor moves to state 8, re-awakens the task to which control is being returned, and advances to state 6. If RC = 0, or if RC > 0 and no returns were found, the processor enters state 9 and searches the Executive CAM for new work. After a task is selected, the processor enters state 10, selects a request to the new task from the QCAM, and moves to state 6. In state 6, the processor passes control to the newly selected (or reawakened) task and advances to state 1 where it begins to execute the task.

3.4.2.1 Control Options

In addition to the basic operations required for communication between tasks, the ACFM control repertoire includes FORK and JOIN macros which permit the programmer to specify parallelism within a program. The implementation of this parallelism is such that a program with internal parallel branches will execute regardless of how many processors are in the system, as long as there is at least one.
Several task control options are implemented: the task-type option, the queue option, and the lock option. They are specified by control bits in the Executive CAM word.

**Task-Type Option**

Tasks in the ACFM may be either reentrant or serially reusable. The type is indicated by the R bit: $R = 1$ designates a reentrant task and $R = 0$ designates a serially reusable task.

One additional task type, called a Direct Task is provided. A direct task does not receive data from any task which calls it, and it can have only one request waiting for it at any given time. Consequently, it requires no QCAM entry. The direct task provides an expedient way of notifying the system of the occurrence of rare events such as external real-time clock signals. The D bit specifies a direct task. If $D = 1$ the task is a direct task; the R bit is ignored. If $D = 0$, the R bit is examined to determine whether the task is reentrant or serially reusable.

**Queue Option**

The control bit, Q, specifies whether all requests in the queue for the task are to be processed before looking for other work ($Q = 1$) or whether only one request is to be processed before searching for possible
higher priority work. Even when the queue option is indicated for a given task, the processor still gives priority to returns from called tasks to suspended calling tasks. After a request from the queue for the given task is completed, the processor processes any returns which may exist to suspended tasks before selecting the next queue entry for that task.

**Lock Option**

A serially reusable task which can be suspended by a call to some other task may require a lock to prevent it from being reselected while the original instance of execution is still suspended. The need for a lock is indicated by the L bit. If L = 1, an inhibit bit in the Executive CAM word is set at the time the task is selected. This prevents it from being selected again. When the original instance of execution completes or, if the queue option is specified, when the queue is emptied, the inhibit bit is reset.

**3.5 Processors**

All processors must be equipped to interface with common data memory and the associative control subsystem, and they must be capable of executing the basic control operations required for intertask
communication. (As described in Section 3.2, these functions may be implemented in a microprogrammed interface unit to which processors are connected.) Beyond this, however, there are no restrictions on processor design.

The ACFM structure provides a basic control framework within which a group of processors can act collectively. Since all interfaces with shared subsystems are asynchronous, there are no critical timing requirements which processors must satisfy; the system can handle processors of any speed. The structure can accommodate identical processors which are programmed differently or a collection of different types of processors with different instruction sets, each designed to perform a specific set of functions. Processors do not even have to be programmable units; they can be functional circuits which perform a fixed set of operations. The same control philosophy applies to high-capacity systems made up of large powerful processors and to networks of minicomputers designed to handle a special application more economically than a powerful uniprocessor.

The ACFM structure can grow in a variety of ways to meet increasing capacity requirements. An initial installation may consist of one or more processors which are programmed to perform all system functions. When more real-time capacity is needed, more processors can be added. An added processor may be programmed to perform all tasks; it may be programmed to perform only certain tasks for which more processing capacity is needed; or it may be a special purpose machine designed to relieve other processors of certain functions such as input-output
control. In some systems, it may be more economical to replace existing processors with faster versions than to add more processors of the same type. If the cost of hardware continues to decline as anticipated, the concept of throwaway computers may become realistic. In this case, when a particular processor fails or becomes obsolete, it can simply be thrown away and replaced. In any case, the distributed nature of the ACFM architecture minimizes the difficulty of growing the system.

The two applications examined in this paper require entirely different types of processors. Real-time object identification requires processors which perform floating point arithmetic efficiently. Telephone switching control requires little arithmetic capability beyond fixed point addition but requires enhanced logical decision capability.

Real-time object identification is dominated by one very time-consuming task, silhouette construction (see Chapter VI), which is executed simultaneously on all processors at various times during the course of identifying an object. Growth is accomplished by adding more processors and assigning the silhouette routine to them.

Telephone switching control involves performing tasks on behalf of many telephone calls which simultaneously exist in the system at various stages of completion. Growth is accomplished by splitting off particular functions and assigning them to additional processors.

3.6 Input/Output

Many input/output configurations are possible in the ACFM structure; the choice of a particular configuration is, of course,
highly application dependent. Three diverse approaches to input/output are described in this section to emphasize the inherent flexibility of the ACFM organization in this area.

3.6.1 Direct Memory Access to Secondary Storage or to Low-Speed Common Data Memory

In some applications, it is desirable to shield the multiprocessor from the environment which it controls by buffering input/output data in common memory.

In telephone switching control, for example, inputs to the system are sensed by repetitively scanning input ports for activity. Critical timing requirements must be satisfied to guarantee that no information (e.g., digits) is lost. It is often more economical to allocate this simple function to autonomous units which collect data and deposit it in memory buffers rather than to consume valuable processor time looking for changes of state. In the ACFM, such autonomous units communicate directly with low-speed common data memory. Processors are programmed to interrogate input buffers often enough to prevent them from overflowing under normal load conditions. The output of signals to the controlled environment is accomplished by loading information into buffers which are then emptied by autonomous output units.

A similar input strategy is appropriate for real-time object identification. The optical device which photographs the object to be identified loads a coordinate description of that object directly into the same BORAM (see Figure 3-3) that contains descriptions of all objects that the system is capable of recognizing.
3.6.2 Common Input/Output Channels

Input/output channels are connected to all processors via a time-shared bus or an electronic crossbar switch. Or, the interconnection subsystem which connects low-speed common data memory modules to processors is shared by input-output channels. This approach may be mandatory in systems where high availability and complete fail-soft operation is necessary. Any processor can take over input-output control.

A shared input-output control associative memory can be used, if desired, to optimize the use of input-output devices in this configuration.\(^{(25)}\)

3.6.3 Dedicated Input/Output Processors

One or more processors, designated I/O processors, are specially designed to control input/output devices. The intertask communication control philosophy automatically provides other processors with the capability to initiate input-output requests; they initiate input-output requests by calling tasks assigned to I/O processors. Data is passed via common data memory. I/O processors notify other processors of the initiation or completion of I/O requests by returning to calling tasks. When they receive new inputs from the controlled environment, I/O processors buffer the data in common data memory and call tasks, assigned to other processors, which begin to process the new origins.

The three basic approaches to input-output in the ACFM are summarized in Figure 3-18.
AUTONOMOUS I/O UNITS

ASSOCIATIVE CONTROL

COMMON DATA MEMORY

BORAM

UNIT 1 UNIT 2 UNIT 3 UNIT 4

PROCESSORS

A) DIRECT MEMORY ACCESS TO SECONDARY STORAGE (OR TO LOW-SPEED COMMON DATA MEMORY)

ASSOCIATIVE CONTROL

COMMON DATA MEMORY

DEVICE CHANNELS

B) SHARED I/O CHANNELS

ASSOCIATIVE CONTROL

COMMON DATA MEMORY

DEVICES

I/O CHANNELS

C) I/O DEDICATED TO PARTICULAR PROCESSORS

FIGURE 3-18
INPUT/OUTPUT OPTIONS
3.7 Summary

The ACFM system is designed to meet the objectives listed in Section 3.1. The functional assignment of tasks to processors takes advantage of the three main characteristics of real-time processing. Private program memory and local data memory minimize the need for common memory. This greatly reduces the potential for storage interference making it possible to use a time-shared bus to connect processors to common data memory modules. Growability over a wide range of capacity requirements is more easily achieved in the distributed structure that results. Likewise, graceful degradation is a more realistic objective.

The ACFM structure is modular. This chapter has demonstrated that a wide variety of design options can be accommodated. The memory structure and the input/output subsystem can be tailored to the specific requirements of each application. Virtually any type of processor can be incorporated into the system. The combination of modularity and reduced interconnection with common subsystems is conducive to an LSI implementation. This implies higher reliability.

The control philosophy ties the system together so that it can function effectively as a multiprocessing complex. The Executive CAM facilitates task scheduling. The QCAM absorbs requests to tasks so that control can be passed smoothly between tasks without any direct communication between the processors involved. The remainder of the dissertation develops this control philosophy in detail and evaluates its effectiveness.
CHAPTER IV

INTERTASK COMMUNICATION

The purpose of this chapter is to develop control techniques for intertask communication in sufficient detail to demonstrate the practicality of the ACFM architecture.

The hardware details pertinent to intertask communication are presented in Section 4.1 including basic formats for Executive CAM and QCAM words, descriptions of the operations which the Executive CAM and QCAM must perform, and a description of the hardware which each processor requires in order to operate in the system. Section 4.2 presents a detailed discussion of the microprograms which processors execute while performing executive functions.

In Section 4.3, the various control macros necessary for intertask communication are described in terms of the executive microprograms which they invoke. The repertoire of control macros includes not only the basic macros which provide for communication between tasks but also macros which permit task structures with parallel branches to be specified and executed. The set of control macros is
designed so that intertask communication is entirely independent, as far as the programmer is concerned, of how tasks are actually assigned to processors and of how many processors are in the system.

Several programming examples are given in Section 4.4 to illustrate how control macros are used to specify software structures. Section 4.5 explains how the system handles repeated tasks, that is, tasks which are assigned to more than one processor.

Finally, Section 4.6 summarizes the control philosophy of the ACFM and presents the rationale behind the various design decisions which were made during its development.

4.1 Hardware Design Details Pertinent to Intertask Communication

The hardware blocks which are involved in intertask communication are the Executive CAM, the QCAM, high-speed common data memory, and the processors. The reader is referred to Figure 3-1 for a block diagram which shows how these blocks connect into the ACFM structure.

4.1.1 The Executive CAM

The Executive CAM is a high-speed associative memory of the type discussed in Section 3.4.1. Each task in the system is assigned an Executive CAM word, called a task word, having the format shown in Figure 4-1. The various fields which make up an Executive CAM word are described as follows:

**TASKNAME** is a unique identifier assigned to individual tasks in the system.

**E** is an enable bit which, when set, indicates that the task has been called for execution.
<table>
<thead>
<tr>
<th>TASKNAME</th>
<th>E</th>
<th>N</th>
<th>H</th>
<th>PRIORITY</th>
<th>PROCESSOR NAME</th>
<th>R</th>
<th>D</th>
<th>Q</th>
<th>L</th>
<th>C</th>
<th>B</th>
<th>I</th>
</tr>
</thead>
</table>

FIGURE 4-1
EXECUTIVE CAM WORD FORMAT
**INH** is a inhibit bit used to prevent a serially reusable task from being selected whenever it has another instance of execution in progress.

**PRIORITY** is a field which indicates the priority of the task relative to other tasks in the system. The Executive CAM can select that task which has maximum (minimum) priority from among those tasks which are enabled and waiting for a given processor. Consequently, a processor is able to select the most urgent task awaiting its service.

**PROCESSOR NAME** is a namecode for the processor to which the task is assigned. (This code is designed to allow a given task to be assigned to more than one processor as explained in Section 4.5.)

**R, D, Q, and L** are control bits which specify the type of task.

- **R** indicates whether the task is reentrant \((R = 1)\) or serially reusable \((R = 0)\).

- **D** designates a direct task \((D = 1)\). A direct task requires no QCAM entry.
Q indicates whether the queue of requests is to be completely exhausted before any other task is selected (Q = 1) or whether only one request is to be processed before checking to see if other tasks of higher priority are waiting (Q = 0).

L indicates whether a serially reusable task is to be locked when it is selected for execution (L = 1). (It is locked by setting the INH bit in the Executive CAM word for the task.)

C is a special control bit. After a word is read by an associative read, C = 0. After a word is written by an associative write, C = 1.

B/I indicates whether the Executive CAM word has a task assigned to it (B/I = 1) or whether it is unused (B/I = 0).

The reasons for the various fields and control bits which make up an Executive CAM word will become clear as the control philosophy is developed.

As described in Section 3.4.1, the Executive CAM (and the QCAM) is equipped with a read-only memory controller which recognizes operation codes sent from a processor and supplies internal data
operands, such as mask register contents needed to perform the desired operation.

The basic Executive CAM operations used for intertask communication include: **Set Enable** and **Reset Enable** which control the enable bit, \( E \); **Priority Task Search** and **Task Search** which select a task, from among those assigned to the requesting processor, which is enabled and thus ready for execution; and, **Set Inhibit** and **Reset Inhibit** which control the inhibit bit, \( \text{INH} \). All operations return a match/nomatch signal to the requesting processor which indicates whether or not a matching word was found. A more detailed description of these basic operations is now given:

**Set Enable**

The processor supplies the correct operation code and a search key consisting of the taskname for the task to be enabled and \( B/I = 1 \). The Executive CAM does an equality search to locate the task word and then sets its enable bit to one. The \( C \) bit is also set to one. The contents of internal Executive CAM registers during a Set Enable operation are depicted in Figure 4-2.

**Reset Enable**

The processor supplies the correct opcode and a search key made up of the taskname, \( B/I = 1 \), and \( C = 0 \). The Executive CAM locates the matching word and sets its enable bit to zero. Note that the enable bit is actually reset by the Reset Enable operation only if \( C = 0 \). The reason for the \( C \) bit is explained in Section 4.2.2.
FIGURE 4-2
ENABLE TASK OPERATION
**Priority Task Search**

The processor supplies the opcode and a search key consisting of its own namecode, \( E = 1, \text{INH} = 0 \), and \( B/I = 1 \). The Executive CAM selects, from among all words which match the search key, the word with maximum (or minimum, as designated by the opcode) priority and returns its contents and a match signal to the processor. The \( C \) bit of the selected task word is set to zero. If there are no matching words, a nonmatch signal is returned to the processor. The contents of internal Executive CAM registers during a Priority Task Search operation are shown in Figure 4-3.

**Task Search**

This operation is identical to the Priority Task Search operation except that the priority field is ignored. The matching word with lowest physical address is returned to the processor.

**Set Inhibit and Reset Inhibit**

The processor supplies the correct opcode and a search key consisting of the taskname and \( B/I = 1 \). The Executive CAM selects the matching word and either sets or resets its inhibit bit.

The Executive CAM operations described above are only a subset of the total repertoire, but they are the basic ones used in inter-task communication. In addition to these, there are operations...
- CONTENTS SUPPLIED BY EXECUTIVE CAM EXCEPT THAT THE PROCESSOR NAME IS COPIED FROM THE SEARCH REGISTER INTO THE SEARCH MASK REGISTER FOR REASONS GIVEN IN SECTION 4.5. THE MAX OR MIN IN THE PRIORITY FIELD INDICATES THAT A MAX OR MIN SEARCH IS DONE OVER THIS FIELD ON THOSE WORDS WHICH MATCH THE REST OF THE SEARCH KEY.

+ CONTENTS SUPPLIED BY PROCESSOR.

FIGURE 4-3
PRIORITY TASK SEARCH OPERATION
which change the priority of a task, write a new task word, erase an
existing task word, change the task control bits, etc.

A typical size for the Executive CAM word is 32 bits assigned as
follows: taskname, 10 bits; priority, 6 bits; processor name, 8 bits;
and E, INH, R, D, Q, L, C, and B/I 1 bit each. The number of Executive
CAM words depends upon the application, but in most cases a 512-word
memory should be sufficient.

4.1.2 The QCAM

The QCAM is a high-speed associative memory much like the
Executive CAM except that it does not require maximum (minimum) search
capability.

The basic format of a QCAM word is shown in Figure 4-4. The
various fields which make up a QCAM word are described below:

**TASKNAME** - identifies the task being called. On returns,
this field contains the special return code.

**FIELD A** - is blank or it contains either a data item being
passed or the address in common data memory of
data being passed.

**FIELD B** - is blank or it contains the name code of the
calling processor.

**FIELD C** - is blank or it contains a 4-bit register group
identifier. (See Section 4.1.4.)
<table>
<thead>
<tr>
<th>TASKNAME</th>
<th>FIELD A</th>
<th>FIELD B</th>
<th>FIELD C</th>
<th>FIELD D</th>
<th>FIELD E</th>
<th>CONT</th>
</tr>
</thead>
</table>

**FIGURE 4-4**

**OCAM WORD FORMAT**
FIELD D - is blank or it contains the physical address in private program memory to which control is to be returned.

FIELD E - contains the physical address of the QCAM word.

CONTROL - specifies how the various fields above are to be interpreted.

K - is a single bit used to reserve (Keep) the QCAM word while it is holding return linkage information.

B/I - indicates whether the word is occupied (B/I = 1) or whether it is idle (B/I = 0).

The meaning of the various fields listed above will become clear as the control philosophy is developed.

Like the Executive CAM, the QCAM has a read-only memory controller which interprets coded operations sent by a processor and supplies internal operands.

The QCAM operations used in intertask communication are: Write Idle Word which selects an unused word and loads it with the information pertaining to a task request; Task Read which selects a request from the queue of requests waiting for a particular task; Clear Word which erases a word; Return Write which sets up a return to a calling task; and, Return Search which is used by processors to search for
returns to suspended calling tasks. All QCAM operations return a match/nomatch signal to the processor. In addition, on the task read operation a signal is returned to the processor which specifies whether zero, one, or more than one word matched the search key. The reason for these signals are discussed in Section 4.2. A more detailed description of each of the basic QCAM operations is given below:

**Write Idle Word**

The processor supplies the information to be written in a QCAM word and the correct operation code. The QCAM locates an idle word (B/I = 0) and stores the information there. The contents of internal QCAM registers during a Write Idle Word operation are shown in Figure 4-5.

**Task Read**

The processor supplies the opcode and a search key consisting of the taskname, K = 0, and B/I = 1. The QCAM does an equality search and returns to the processor that matching word with lowest physical address. After the read, the reserve bit, K, is set to one. Figure 4-6 depicts the contents of internal QCAM registers during a Task Read operation.

**Clear Word**

The processor supplies the address (field E) of the word to be erased and the proper opcode. The QCAM sets the word to all zeros except for field E which it leaves
**FIGURE 4-5**

WRITE IDLE WORD OPERATION

* CONTENTS SUPPLIED BY QCAM
+ CONTENTS SUPPLIED BY PROCESSOR
** FIELD E ALWAYS CONTAINS THE PHYSICAL ADDRESS OF THE QCAM WORD, AND IS NOT MODIFIED BY WRITE OPERATIONS
**TASKNAME** FIELD A FIELD B FIELD C FIELD D FIELD E CONTENTS

**SEARCH MASK REGISTER**

<table>
<thead>
<tr>
<th>TASKNAME</th>
<th>FIELD A</th>
<th>FIELD B</th>
<th>FIELD C</th>
<th>FIELD D</th>
<th>FIELD E</th>
<th>CONTENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1--------</td>
<td>0-------</td>
<td>0-------</td>
<td>0-------</td>
<td>0-------</td>
<td>0-------</td>
<td>0-------</td>
</tr>
</tbody>
</table>

**SEARCH REGISTER**

<table>
<thead>
<tr>
<th>TASKNAME</th>
<th>FIELD A</th>
<th>FIELD B</th>
<th>FIELD C</th>
<th>FIELD D</th>
<th>FIELD E</th>
<th>CONTENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0--------</td>
<td>0-------</td>
<td>0-------</td>
<td>0-------</td>
<td>0-------</td>
<td>0-------</td>
<td>0-------</td>
</tr>
</tbody>
</table>

**OPCODE REGISTER**

<table>
<thead>
<tr>
<th>TASKNAME</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0--------</td>
<td></td>
</tr>
</tbody>
</table>

* CONTENTS SUPPLIED BY QCAM

+ CONTENTS SUPPLIED BY PROCESSOR

**FIGURE 4-6**

**TASK READ OPERATION**
intact. The contents of internal QCAM registers during a Clear Word operation are shown in Figure 4-7.

**Return Write**

The processor supplies the RETURN code, field A, the control field, $K = 0$, $B/I = 1$, the QCAM address (field E), and the correct opcode. The QCAM selects the proper word and writes the information in the appropriate fields leaving other fields unchanged. Figure 4-8 illustrates the contents of internal QCAM registers during a Return Write operation.

**Return Search**

The processor provides the opcode and a search key consisting of the RETURN code in the taskname field, the processor namecode in field B, $K = 0$, and $B/I = 1$. The QCAM does an equality search and returns to the processor that matching word with lowest physical address. Figure 4-9 depicts internal QCAM register contents during a Return Search operation.

The width of a QCAM word will vary with the application but it is approximately 90 bits assigned as follows: taskname, 10 bits; field A, 32 bits; field B, 8 bits; field C, 4 bits; field D, 15 bits; field E, 11 bits; control, 8 bits; and K and $B/I$ 1 bit each. The number of QCAM words depends upon the application. In particular, the number of words depends upon the number of tasks and the expected
<table>
<thead>
<tr>
<th>Register Type</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search Mask Register*</td>
<td></td>
</tr>
<tr>
<td>Search Register+</td>
<td></td>
</tr>
<tr>
<td>Write Mask Register*</td>
<td></td>
</tr>
<tr>
<td>Write Data Register*</td>
<td></td>
</tr>
<tr>
<td>Opcode Register+</td>
<td></td>
</tr>
</tbody>
</table>

* Contents supplied by QCAM

+ Contents supplied by processor

**FIGURE 4-7**

Clear Word Operation
SEARCH MASK REGISTER *

SEARCH REGISTER *

WRITE MASK REGISTER*

WRITE DATA REGISTER+

OPCODE REGISTER +

* CONTENTS SUPPLIED BY QCAM

+ CONTENTS SUPPLIED BY PROCESSOR

FIGURE 4-8

RETURN WRITE OPERATION
<table>
<thead>
<tr>
<th>TASKNAME</th>
<th>FIELD A</th>
<th>FIELD B</th>
<th>FIELD C</th>
<th>FIELD D</th>
<th>FIELD E</th>
<th>CONT K</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1------</td>
<td>0-------</td>
<td>1------</td>
<td>0-0</td>
<td>0-------</td>
<td>0------</td>
</tr>
</tbody>
</table>

**SEARCH MASK REGISTER**

**SEARCH REGISTER**

**OPCODE REGISTER**

* CONTENTS SUPPLIED BY QCAM

+ CONTENTS SUPPLIED BY PROCESSOR

**FIGURE 4-9**

RETURN SEARCH OPERATION
length of the queue of requests for each task. In most cases, 2048 words should be sufficient.

4.1.3 Common Data Memory

The high-speed common data memory may be used to pass data between tasks. If a single data word is being passed from one task to another, it is passed directly in the QCAM word. Otherwise, an address of an area in high-speed common data memory is passed in the QCAM word and the actual data is passed via high-speed common data memory. A detailed discussion of common data memory was given in Section 3.3.2.

4.1.4 Processors

Several processor design details are essential to intertask communication. The required hardware includes: a high-speed register memory to facilitate task suspension; a counter to keep track of the number of two-way calls which have been placed but not yet completed; access registers to interface with the Executive CAM and the QCAM; and a microprogram memory for executive control routines.

4.1.4.1 Register Memory and Control

A processor must be able to quickly suspend operation of one task in preparation for executing another. Likewise, a processor must be able to quickly reactivate a suspended task when control is returned to it. One way to implement this requirement is with a high-speed register memory.

A 256-word high-speed register memory consisting of 16 groups of 16 general purpose registers is assumed to exist in each processor.
The eight-bit address to this memory is divided into two concatenated four-bit fields. The high order 4-bits, called the register group identifier (RGI), select one of 16 register groups. The low order 4-bits select a particular register within a group.

A register group may be in any of four possible states:

1. **IDLE** - If a register group is not currently assigned to a task, it is in the idle state.

2. **ACTIVE** - The register group being used by the task currently in execution is in the active state. Only one register group can be active at any given time.

3. **SUSPENDED** - The register group corresponding to a suspended task is in the suspended state. The information it contains is retained until the suspended instance of execution is reactivated.

4. **WAITING** - While the processor is sequentially processing a queue of requests to a particular task, say task A, it checks for possible returns from called tasks each time it finishes processing a queue entry. If a return is picked up, the register group for task A is temporarily placed in the waiting state.
An associative memory consisting of 16 2-bit words is used to control the allocation and deallocation of register groups. The 4-bit addresses of these words correspond to register group identifiers. Each word contains the state code for the associated register group. The layout of this associative memory is shown in Figure 4-10. To set a register group to a particular state, the memory is operated in a random-access mode; to locate a register group that is in a particular state, the state code is applied as a search key to the 2-bit associative registers.

The first register, RO, of each register group is called the **Task Register**. When a task is selected for execution, the taskname, the control bits from the Executive CAM word (R, D, Q, and L), the physical address of the QCAM word (field E), and the control field of the QCAM word are copied into the task register. The content of the task register is summarized in Figure 4-11. The task register controls the execution of the task as described in Section 4-2.

The second register of a register group, R1, is used to hold the passed data or its address.

### 4.1.4.2 Return Counter

Each processor has a special hardware counter called the **Return Counter** (RC). Each time a call with return linkage is set up, the RC is incremented. When a return to a calling task is processed, the RC is decremented. When the processor searches for work, it first checks the return counter. If the contents of the RC are not zero, it searches the QCAM for a return before looking for new work.
FIGURE 4-10
ASSOCIATIVE MEMORY FOR REGISTER GROUP CONTROL
FIGURE 4-11

TASK REGISTER CONTENTS
4.1.4.3 **Associative Memory Access Registers**

Each processor contains two registers, the Executive CAM Access Register and the QCAM Access Register, which communicate with the Executive CAM and the QCAM, respectively. (These are also referred to as the Executive CAM Register and the QCAM Register.) On an associative control operation, the processor loads the appropriate register and then initiates the operation. Data coming from the associative control memories is placed into these registers. Figure 4-12 shows the layout of these registers.

4.1.4.4 **Microprogram Control**

Because the various intertask control operations are rather involved, a microprogrammed implementation is recommended. This is also consistent with the design goal of making optimal use of large-scale integration. The several control microprograms are described in Section 4.2.

4.2 **Processor Microprograms For Task Control**

Five task control microprograms, which implement the executive control functions performed by processors, are described in this section. In the next section, the various control instructions (or control macros) are explained in terms of the paths which they cause to be taken through these microprograms. The microprograms are:

1. **ENTASK** - (ENABLE TASK) is invoked each time the processor enables a task for execution.
### Figure 4-12

**Layout of Associative Memory Access Registers**

<table>
<thead>
<tr>
<th>TASKNAME</th>
<th>FIELD A</th>
<th>FIELD B</th>
<th>FIELD C</th>
<th>FIELD D</th>
<th>FIELD E</th>
<th>CONT</th>
<th>B</th>
<th>OPCODE</th>
<th>MISC</th>
<th>CONT</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXECUTIVE CAM REGISTER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TASKNAME</th>
<th>FIELD A</th>
<th>FIELD B</th>
<th>FIELD C</th>
<th>FIELD D</th>
<th>FIELD E</th>
<th>CONT</th>
<th>B</th>
<th>OPCODE</th>
<th>MISC</th>
<th>CONT</th>
</tr>
</thead>
<tbody>
<tr>
<td>QCAM REGISTER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2. **LW** - (LOOKING FOR WORK) is the microprogram which the processor executes each time it either finishes or suspends a task and is ready to search for more work to do.

3. **QENTRY** - (QUEUE ENTRY) is called whenever the processor selects an entry from the queue of requests waiting for a particular task.

4. **QRET** - (QUEUE RETURN) is the routine which picks up a return from a called task and returns control to the calling task.

5. **WRET** - (WRITE RETURN) is executed at the completion of a called task in order to set up the return linkage to the calling task.

### 4.2.1 ENTASK

A flowchart of the ENTASK microprogram is given in Figure 4-13. Whenever a processor encounters some type of a CALL macro in the task it is executing, that is, when it encounters an instruction which calls some other task, it enters ENTASK. The function of ENTASK is to build the appropriate QCAM word in the processor's QCAM access register and then to write the word into the QCAM and enable the task via the Executive CAM. If return linkage is required, the return counter is incremented. On a call to a direct task (See Section 4.3), the part of ENTASK is bypassed which constructs the QCAM word and writes it.
WRITE TASKNAME OF CALLED TASK INTO EXECUTIVE CAM & QCAM REGISTERS

YES

D = 1

NO

DATA PASSED

YES

DIRECT OR INDIRECT

NO

WRITE DATA ADDRESS INTO FIELD A OF QCAM REGISTER

RETURN LINKAGE REQUIRED?

YES

WRITE RETURN LINKAGE INFORMATION INTO QCAM REGISTER

WRITE CONTROL CODE IN QCAM REG

QCAM ORDER WRITE IDLE WORD

EXECUTIVE CAM ORDER ENABLE TASK

RETURN LINKAGE REQUIRED?

YES

NO

SP3 OF LW

NO

DISPATCHING CALL?

YES

FETCH NEXT COMMAND IN ACTIVE TASK

SP3 OF LW

NO

DISPATCHING CALL?

YES

RC = RC + 1

NO

FIGURE 4-13
FLOWCHART OF ENTASK MICROPROGRAM
A **Dispatching Call** is one which enables the called task but does not cause either the suspension or termination of the calling task. Tasks use dispatching calls to dispatch work to the system. In ENTASK, a dispatching call causes control to be passed to the next instruction in the calling task after the called task has been enabled.

If a call is not a dispatching call, it causes the calling task to be terminated if the call does not provide for return linkage, or suspended if the call does set up return linkage. In the former case, control is passed to entry point 2 of LW. In the latter case, control is passed to entry point 3 of LW.

### 4.2.2 LW

LW is the most complex of the five task control microprograms. Figure 4-14 is a flowchart of LW. It has four entry points:

1. Entry point 1 (EP1) is entered when the processor is first started up and it is entered periodically during idle periods when little or no work is available for the processor. It is also entered after preliminary processing is completed at entry points 2 or 3.

2. Entry point 2 (EP2) is entered after a task is terminated either by a CALL without return linkage or by an END macro which ends a task.

3. Entry point 3 (EP3) is entered after a task is suspended by a CALL with return linkage, by a WAIT macro, or by a JOIN macro.
FIGURE 4-14
FLOWCHART OF THE LW MICROPROGRAM
4. Entry point 4 (EP4) is entered when a task has been suspended and the return linkage just set up is the only pending return.

The LW microprogram decides what type of work to do next. The first priority is to process any returns which may be waiting. If no returns are waiting, the LW routine decides whether to select another entry from the queue of requests waiting for the task just completed or to select a new task by interrogating the Executive CAM. The LW routine will now be described in detail.

4.2.2.1 Task Termination

When a task is terminated by an END macro or a call without return linkage, entry point 2 of LW is entered. If the D bit in the task register of the active register group is one, indicating that the task being completed is a direct task, the active register group is idled and entry point 1 of LW is entered. (The active register group is the one which was being used by the task being terminated.)

If the D bit is zero, the QC bit is examined. Recall that the QC bit was originally initialized from the Q bit of the Executive CAM word associated with the task just completed. An initial value of zero indicates that only one waiting request for the task is to be processed before going to the Executive CAM for more work. An initial value of one means that the entire queue of waiting requests is to be processed before any new tasks are selected. In the latter case, the value of QC remains one until the last queue entry is selected at which time it is set to zero. Referring to the flowchart (Figure 4-14),
if QC = 0 at the time the task is terminated, the register group that was assigned to the task just completed is idled and entry point 1 of LW is entered. If QC = 1, the register group for the task just completed is set to the waiting state and entry point 1 of LW is entered. The register group is placed in the waiting state to allow any returns to be processed before selecting the next queue entry for the task that was just completed.

Before the active register group is idled to terminate a task, a check is made to see if the L bit of the active task register is one. If so, an Executive CAM order is issued to reset the INH bit in the task word for that task.

4.2.2.2 Task Suspension

When a task is to be suspended, entry point 3 of LW is entered. A new queue entry for the suspended task can be processed immediately only if the task is reentrant (R = 1) and if QC = 1. If both of these conditions are not satisfied, the active register group (which is assigned to the task being suspended) is set to the suspended state. If the value of RC (The Return Counter) is one, entry point 4 of LW is entered. This bypasses the search for returns since the only return that exists is the one that is associated with the call just initiated, and it has not had time to be processed by the processor to which the called task is assigned. If RC = 1, entry point 1 of LW is entered.

If the suspended task is reentrant and if QC = 1, a new queue entry for that task can be processed. The task register of the active register group is copied into the task register of an idle register group (if one exists) which is then set to the waiting state. The QC
bit in the task register of the active register group is then set to zero. This is done to insure that only the first suspension of a given instance of execution creates a new instance of execution. The active register group is placed in the suspended state. Either EP1 or EP4 of LW is entered as described in the previous paragraph.

4.2.2.3 Action Taken at Entry Point 1

When entry point 1 of LW is entered, the return counter is tested. If it is not zero, a Return Search QCAM order is executed to look for returns to suspended calling tasks. If one is found the QRET routine is entered. If none are found or if RC = 0, a test is made to see if there are any waiting register groups. If there are, one is selected and marked active and point B (See Figure 4-14) is entered. If there are no register groups in the waiting state, a new task is selected via either a Task Search order or a Priority Task Search order to the Executive CAM.

If no work is found in the Executive CAM, an idle routine is activated. This routine may perform maintenance exercises or it may simply wait for some time interval. In both cases, control is eventually returned to EP1 of LW.

Once a task is selected from the Executive CAM, an idle register group is seized and marked active. The TASKNAME, and the control bits (R, D, Q, L) are copied from the Executive CAM Register into the task register of this register group. A test is then made to see if the selected task is a Direct Task (D = 1). If it is, a Reset Enable Executive CAM order is issued to reset the task's enable bit. Control is then passed to the task. If the task is not a Direct Task, a test
is made to determine whether the task requires a lock \((L = 1)\). If so, an Inhibit Task Executive CAM order is issued. The position in the flowchart (Figure 4-14) is now point B.

At point B, a task read order is sent to the QCAM. Three responses to this order are possible: a) If no match is found, a Reset Enable Executive CAM order is issued, the active register group is idled, and control is returned to point A of LW. b) A single match indicates that the last queue entry was selected. The QC bit in the task register is set to zero and a Reset Enable Executive CAM order is issued. Control is passed to the QENTRY routine. c) If more than one match exists, QENTRY is entered directly.

A Reset Enable Executive CAM order actually resets the enable bit only if the C bit of the selected Executive CAM word is zero. Recall that the C bit is set to one by a Set Enable operation and set to zero by a Task Search operation. The C bit acts as an interlock to prevent a task call from being missed. Without the C bit, this could happen if the following sequence occurred:

Task A selected by processor J
First request for task A processed

_____

_____

Last request for task A selected
Processor K posts a request to task A
Processor K enables task A
Processor J resets enable for task A.
If another processor (Processor K) calls task A and obtains access to the associative control memories between the time that processor J selects the last queue entry for task A and the time that processor J resets the enable bit for task A, the call will be missed. The C bit, however, prevents this because it is set to one when processor K issues the Set Enable order for task A. The Reset Enable order issued by processor J does not reset the enable bit because $C = 0$. Hence, the request is picked up later when processor J again selects task A. If a new request to task A arrives earlier while the queue is still being processed, the enable bit will remain set after the queue is exhausted. Later, when processor J reselects task A it will find no work and will simply return to point A of LW and select a new task. If the Q bit of the Executive CAM word for task A is zero, the enable bit will remain set until all queued requests have been processed even though they are processed one at a time with other higher priority tasks interleaved with them.

4.2.3 QENTRY

After a task has been selected and a QCAM word corresponding to a request for that task has been read, the QENTRY routine is entered. (See the flowchart in Figure 4-15.) The physical address of the QCAM word (field E) and the control field (which indicates the type of call) are copied into the task register of the active register group. Field A, which either contains data being passed or the address of data being passed (or is empty), is copied into R1 of the active register group.

The control field of the QCAM register is now examined to determine whether the call returns control to the calling task. If it does not,
QENTRY

COPY FIELD A OF QCAM REG INTO RI OF ACTIVE REG GROUP

COPY FIELD E AND CONTROL FIELD FROM QCAM REG INTO RO OF ACTIVE REG GROUP

IS THIS A TASK WHICH RETURNS TO A CALLING TASK?

YES

RECALL THAT THE QCAM TASK READ ORDER SET K=1, AUTOMATICALLY RESERVING THE WORD FOR RETURN LINKAGE

NO

QCAM ORDER CLEAR WORD

TRANSFER CONTROL TO THE TASK Whose Name IS IN RO OF THE ACTIVE REG GROUP

FIGURE 4-15
FLOWCHART OF THE QENTRY MICROPROGRAM
the QCAM word is idled by a Clear Word order. If it does, the QCAM word is left intact until control is returned by a return macro. The K(Keep) bit was set to one when the Task Read order was executed. This reserves the word and prevents it from being selected by other task reads. Control is now transferred to the called task. The information in Rl is used to locate the data to be processed.

4.2.4 QRET

The QRET routine is entered whenever the processor finds a return to a calling task via a Return Search command to the QCAM. (See the flowchart in Figure 4-16). Field C of the QCAM word contains the register group identifier of the suspended task which is to be reawakened. The corresponding register group is set to the active state. Field A of the QCAM word, which contains returned data or its address is copied into Rl of the now reactivated register group. The return address from Field D is placed in the processor's program address register. The return counter, RC, is decremented. A Clear Word order is issued to idle the QCAM word used for the return and control is passed to the address in the program address register.

4.2.5 WRET

The WRET routine is executed whenever a called task encounters a RETURN macro. (See the flowchart in Figure 4-17.) The return code, the data to be returned or its address, and the control field are placed in the QCAM register. The control field specifies whether the data being returned is passed directly or indirectly. The physical address of the QCAM word containing the return linkage information is moved from the task register to field E of the QCAM register. A Return Write
A QCAM return word is waiting in the QCAM register.

Figure 4-16

Flowchart of the QRET microprogram
WRET

MOVE RETURN CODE INTO TASKNAME FIELD OF QCAM REGISTER

WRITE DATA OR ADDRESS OF DATA TO BE RETURNED INTO FIELD A OF QCAM REGISTER

WRITE CONTROL INFORMATION IN CONTROL FIELD OF QCAM REGISTER

SET K=0, B/I=1 IN QCAM REGISTER, MOVE QCAM ADDRESS FROM TASK REGISTER OF ACTIVE REGISTER GROUP INTO QCAM REGISTER

QCAM ORDER RETURN WRITE

EP2 OF LW

FIGURE 4-17
FLOWCHART OF THE WRET MICROPROGRAM
QCAM order is then issued. (After this Return Write has been com-
pleted, the return is ready to be picked up by the processor which
placed the call to the called task.) Control is then passed to EP2 of
LW since the called task has completed execution and is ready to be
terminated.

4.3 Control Macros

Control macros provide the user with the tools he needs to pro-
gram the ACFM.

CALL macros pass control from one task to another and may provide
for return linkage. (Two-way calls provide for return linkage; one-
way calls do not.) Data may be passed either directly or indirectly
between tasks.

The RETURN macro is used by a called task to return control to the
calling task. After the return is set up, the called task is termi-
nated and the processor searches for new work.

The FORK macro initiates the execution of parallel branches with-
in a program. Its counterpart, the JOIN macro, provides for the
merging of parallel branches.

These control macros will now be described in terms of the micro-
programs presented in Section 4-2. In Section 4-4, a number of
examples are presented to clarify how macros are used.

4.3.1 CALLS

All types of CALLS, except the direct call (which requires no
QCAM word), have the single mnemonic, CALL. The format of a CALL macro
is:

CALL TASKNAME, α, β, γ
The fields have the following interpretation:

**TASKNAME** is the name of the task being called.

*a* indicates the addressing mode.

A blank means no data is passed.

The letter D indicates that a single data item is passed to the called task via the QCAM word.

The letter I indicates that the address of a parameter area in common data memory is passed via the QCAM.

*δ* indicates whether the CALL is a dispatching CALL or not.

The letter D indicates a dispatching CALL; otherwise the field is blank.

*γ* is the physical address in private program memory to which control is returned following the completion of a two-way CALL. If the *γ* field is blank, the CALL is one-way.

Direct CALLs have the mnemonic DCALL and the format:

```
DCALL    TASKNAME, δ
```

The fields have the following meaning:

**TASKNAME** is the name of the task being called.
The letter D indicates a dispatching CALL; otherwise the \( \delta \) field is blank.

All CALLs invoke the \textbf{ENTASK} microprogram. After the call is set up, control is passed to an appropriate entry point in the LW microprogram or to the next instruction in the calling task in the case of dispatching calls. (See Section 4.2.1.)

4.3.2 \textbf{RETURN}

The \textsc{return} macro returns control from a called task to a calling task by invoking the \textsc{wret} microprogram which then passes control to entry point 2 of LW to terminate the called task.

4.3.3 \textbf{END}

The \textsc{end} macro simply enters entry point 2 of the LW microprogram causing the active task to be terminated.

4.3.4 \textbf{WAIT}

The \textsc{wait} macro enters entry point 3 of the LW routine causing the active task to be suspended.

4.3.5 \textbf{FORK} and \textbf{JOIN}

The \textsc{fork} macro has the format:

\textsc{fork \( \textit{N,COUNTER} \)}

The \textit{N} field specifies the number of parallel branches to be created and the \texttt{COUNTER} field is a data memory location which holds the count \textit{N}. When the macro is executed, location \texttt{COUNTER} is simply initialized to the value \textit{N}.

The \textsc{join} macro has the format:

\textsc{join \texttt{COUNTER}}
When the JOIN macro is encountered, the value in location COUNTER is decremented by one. If the new count is not zero, entry point 3 of LW is entered causing the task to suspend. If the new count is zero, the next instruction is executed.

The FORK and JOIN macros do not invoke any of the five executive microprograms described in Section 4.2.

4.4 Examples of How Control Macros are Used

This section presents six programming examples which illustrate how the control macros described in the previous section are used to interconnect tasks into larger software structures.

Example 1 Task Dispenser

The coding of a task dispenser routine is shown in Figure 4-18. The routine, MAIN, sequentially executes the three dispatching CALL macros and is then terminated by an END macro. (Refer to Section 4.2.1 for a description of dispatching calls.) The dispatching CALLs post requests to TASKA, TASKB, and TASKC enabling them for execution by the processors to which they are assigned.

Example 2 Use of the DCALL Macro

The DCALL macro is used to call direct tasks. Direct tasks are performed in response to the occurrence of unusual events such as external real-time clock signals. A call to a direct task does not require a QCAM word since no data is passed to it and since only one request to it may exist at any given time.

Consider a direct task, called RTASK, which is activated by a real-time clock signal which interrupts processor 1 every 10
FIGURE 4-18
EXAMPLE 1: TASK DISPENSER ROUTINE

EXAMPLE 2: TWO-WAY CALL
milliseconds. When processor 1 is interrupted by the clock, it simply executes the macro:

```
DCALL RTASK
```

This enables RTASK so that it can be selected for execution by the processor to which it is assigned.

In some applications, it may be desirable to wire the enable bit in the Executive CAM word of a direct task directly to the real-time clock. Then, when the clock turns on the enable, the rest of the system is notified of the event since the direct task can then be selected for execution.

As described in Section 4.5, a direct task may be assigned to all processors and given highest priority so that, as soon as any processor becomes available, it is executed. This provides a convenient way to interrupt the system without first having to decide which processor to interrupt.

**Example 2 Two-Way Call**

Consider the code depicted in Figure 4-19. Task A is assigned to processor 1 and task B is assigned to processor 2. The CALL macro in task A specifies that task B is to be called, that data is passed to it indirectly via common data memory, and that control is to be returned to address AD1. After processor 1 executes the CALL macro in task A, it suspends task A and searches for other work. Processor 2 eventually selects task B, executes it, and executes the RETURN macro which sets up a return in the QCAM to processor 1. Since the return counter of processor 1 is not zero, processor 1 looks for returns each
time it searches for new work. Eventually it finds the return to
task A and reactivates it beginning at address AD1. Figure 4-20
diagrams the activity of processors 1 and 2 during the two-way call.

Example 4 Simple Program With Parallel Branches

The structure of a program which has parallel branches is re­
presented in a MAIN task which calls the various tasks to be executed
in the proper sequence. The MAIN task and the tasks it calls may be
arbitrarily assigned to any number of processors. Consider Figure
4-21: Figure 4-21 a) illustrates the structure of a program with
parallel branches and Figure 4-21 b) depicts the coding for corre­
sponding MAIN task.

Task A is called after which the MAIN task is suspended. When
task A completes, it returns control to the MAIN task at address AD1.
The FORK macro initializes the counter, CT1, to four (for 4 parallel
branches). The calls to tasks B, C, D, and E are dispatching calls.
Consequently, the MAIN task initiates all four calls before being
suspended by the WAIT macro. Each time one of the tasks B, C, D, or
E completes, it returns control to the MAIN task at address AD2. The
JOIN macro decrements CT1 and suspends the MAIN task as long as CT1 = 0.
When CT1 reaches zero, the call to task F is executed. When task F
completes, it returns control to the MAIN task which is then terminated
by the END macro.

Example 5 Parallel Structure Requiring Two FORKS

Figure 4-22 illustrates the structure and MAIN task for a program
requiring two FORK macros. At each node in the structure where there
CALL TO TASKB SUSPENDS TASKA

RETURN TO TASKA PICKED UP
TASKA REAWAKENED

EXECUTING / OTHER WORK / EXECUTING

TASKA * _________________

EXECUTING
OTHER WORK
EXECUTING TASKA

PROCESSOR ACTIVITY DURING A TWO-WAY CALL

FIGURE 4-20
a) STRUCTURE OF MAIN TASK  

b) CODING OF MAIN TASK

* IN CODING PARALLEL STRUCTURES, THE INSTRUCTION PAIR

\[
\text{CALL TASK, } a, D, X \\
\text{WAIT}
\]

CAN BE REPLACED BY

\[
\text{CALL TASK, } a,, X
\]

* SINCE A NON-DISPATCHING TWO-WAY CALL AUTOMATICALLY SUSPENDS THE CALLING TASK. THE WAIT MACRO IS USED IN EXAMPLES 4, 5, AND 6, HOWEVER, TO EMPHASIZE THE FACT THAT THE MAIN TASK IS SUSPENDED AFTER PARALLEL BRANCHES HAVE BEEN SPAWNED FOLLOWING A FORK.

FIGURE 4-21

EXAMPLE 4: A SIMPLE PARALLEL TASK STRUCTURE
EXAMPLE 5: A PARALLEL TASK STRUCTURE REQUIRING TWO FORK MACROS

**FIGURE 4-22**

**a) STRUCTURE**

**b) MAIN TASK**

```
    START
      |    A
      |   AD1
      |    |
      |   B  C
      |    |
      D  E  |
      AD2  AD3
      |
      F
     AD5
    END

    MAIN
    AD1 CALL A,D,,AD1
         FORK 2,CT1
         CALL B,I,D,AD2
         CALL C,I,D,AD3
         WAIT
    AD2 FORK 2,CT2
         CALL D,I,D,AD4
         CALL E,D,D,AD4
         WAIT
    AD4 JOIN CT2
    AD3 JOIN CT1
    AD5 CALL F,I,,AD5
    END
```
is branching, a corresponding FORK macro is coded in the MAIN task. The counter CT1 will not reach zero until task C is complete and count CT2 has reached zero. CT2 will not reach zero until both tasks D and E are finished.

Example 6 More Complex Parallel Structure

A still more complex parallel structure and the coding for the associated MAIN task is depicted in Figure 4-23. The FORK macro at address AD1 initiates the parallel execution of tasks B and C. When task B completes, the FORK macro at address AD2 spawns the parallel execution of tasks D and E. Similarly, when task C completes, the FORK macro at address AD3 spawns the parallel execution of tasks G and H. The JOIN macro at address AD4 insures that tasks D and E both complete before task F is enabled. Likewise, the JOIN macro at address AD5 insures that tasks G and H both complete before task I is enabled. Finally, the JOIN macro at address AD6 insures that tasks F and I both complete before task J is enabled.

An alternate way of coding the structure of Example 6 is to break it down into three MAIN tasks as shown in Figure 4-24. This amounts to having two parallel structures within one larger parallel structure. The coding for the three main tasks, MAIN1, MAIN2, and MAIN 3 is given in Figure 4-25.

4.5 Repeated Tasks

In certain applications, it is necessary to have copies of a particular task in the program memories of more than one processor. As shown in Figure 4-26, tasks D and E are assigned to all three
FIGURE 4-23
EXAMPLE 6: A MORE COMPLEX PARALLEL STRUCTURE
FIGURE 4-24

ALTERNATE STRUCTURE FOR EXAMPLE 6
ALTERNATE CODING OF EXAMPLE 6
### Table: Task Assignment to Processors

<table>
<thead>
<tr>
<th>Task</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task A</td>
<td>Proc 1</td>
</tr>
<tr>
<td>Task B</td>
<td>Proc 2</td>
</tr>
<tr>
<td>Task C</td>
<td>Proc 3</td>
</tr>
<tr>
<td>Task D</td>
<td>All Proc</td>
</tr>
<tr>
<td>Task E</td>
<td>All Proc</td>
</tr>
</tbody>
</table>

### Diagram: Three-Processor System with Repeated Tasks

**EXECUTIVE CAM**

- **CPU 1**
  - Program Memory 1
  - Tasks: Task A, Task D, Task E

**CPU 2**
- Program Memory 2
- Tasks: Task B, Task D, Task E

**CPU 3**
- Program Memory 3
- Tasks: Task C, Task D, Task E

**QCAM**

**Figure 4-26**

Three-Processor System with Repeated Tasks
processors. Consequently, any of the three processors can execute these tasks.

There are several reasons why repeated tasks might be required:

1. Critical application programs may have to be resident in more than one processor to insure maximum reliability.

2. To achieve fast responses to critical external real-time demands may require either an interrupt system (which is difficult to implement in a multiprocessor) or a sufficient replication of critical real-time programs to guarantee that some processor will become free in a short enough time to execute the real-time task. (Of course, these critical tasks are given the highest priority.)

3. Whenever the system application requires that the same program executed in parallel by several processors, replication is necessary.

Referring again to Figure 4-26, note that tasks D and E have only a single Executive CAM word assigned to them. The processor namecode field, however, indicates that tasks D and E are assigned to all processors. Ideally, any processor should be able to select either a private task (one that is assigned only to it) or a repeated task with a single Task Search (or Priority Task Search) Executive CAM order. One way to accomplish this is to code processor names in an M/N code.

Consider a 4/8 processor namecode consisting of four ones and four zeros. Suppose the following namecode assignments are given to the
processors of Figure 4-26:

Processor 1 - 11110000
Processor 2 - 00111100
Processor 3 - 00001111

Let the processor namecode fields for tasks D and E consist of all ones. As shown in Figure 4-3 of Section 4.1.1, a Task Search (or Priority Task Search) operation places the name code of the processor searching for work in both the search register and the search mask register. The contents of these two internal Executive CAM registers during a Task Search (or Priority Task Search) operation initiated by processor 1 of Figure 4-26 are shown in Figure 4-27. This search will match all tasks which have four ones in the first four positions. This includes task A which is a private task of processor 1 and tasks D and E which are repeated in all three processors. Similarly, processors 2 and 3 are able to select either their private task or the repeated tasks with a single search.

The M/N namecode also permits repeated tasks to be assigned to a subset of processors. For example, if the processor name field of the Executive CAM word for task D of Figure 4-26 were 11111100, task D could be selected by either processor 1 or processor 2, but not by processor 3.

There are other ways of coding processor names, besides an M/N code, which will allow repeated tasks to exist in the system. A straight binary code would not be adequate, however, because the
**FIGURE 4-27**

INTERNAL EXECUTIVE CAM REGISTERS DURING A TASK SEARCH OPERATION INITIATED BY PROCESSOR 1
logical OR of two or more processor names would produce another legitimate namecode. Repeated task assignments would therefore lead to erroneous operation.

The control of repeated tasks is clarified by the following example.

Example 7  **Parallel Executions of a Single Task**

The structure and the main routine for a program which invokes four simultaneous executions of a repeated task is shown in Figure 4-28. The loop in the MAIN routine dispatches the four calls to task B.

The program of Example 7 might be assigned to a four processor system as illustrated in Figure 4-29. Processor 1 selects the MAIN task and then calls task A. The MAIN task is suspended while processor 2 executes task A. When processor 2 finishes task A, it executes a RETURN macro. Processor 1 picks up the return and reactivates the MAIN routine. The FORK macro is executed and four calls to task B are dispatched. This causes four queue words to be created in the QCAM. The WAIT macro causes the MAIN routine to be suspended again. All four processors pick up a queue entry for task B and begin processing. The physical QCAM address field of the task register in the active register group of each processor specifies which QCAM word is associated with that processor. As the processors complete task B, they place returns in the QCAM. When processor 1 finishes task B, it places its return in the QCAM and then begins to pick up the returns to the MAIN routine. It picks up the returns one at a time, stores the data and executes the
EXAMPLE 7: A PARALLEL TASK STRUCTURE WITH REPEATED TASKS
## FIGURE 4-29

A FOUR-PROCESSOR SYSTEM FOR EXAMPLE 7

<table>
<thead>
<tr>
<th>TASKNAME</th>
<th>E</th>
<th>INH</th>
<th>PRIORITY</th>
<th>PROCESSOR</th>
<th>NAME</th>
<th>R</th>
<th>D</th>
<th>Q</th>
<th>L</th>
<th>C</th>
<th>B/I</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN</td>
<td></td>
<td></td>
<td></td>
<td>1 1 1 1 0 0 0 0 0 0 0 0 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TASK A</td>
<td></td>
<td></td>
<td></td>
<td>0 1 1 1 0 0 0 0 0 0 1 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TASK B</td>
<td></td>
<td></td>
<td></td>
<td>1 1 1 1 1 1 1 1 0 0 1 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TASK C</td>
<td></td>
<td></td>
<td></td>
<td>0 0 0 0 1 1 1 1 0 0 1 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
JOIN macro. When all four returns have been picked up, processor 1 calls task C and suspends the MAIN routine. When processor 4 completes task C, processor 1 picks up the return and completes the MAIN routine.

4.6 Summary

The purpose of this section is to clarify why certain design decisions were made in the development of the ACFM control philosophy and to summarize the various control options which are available.

The primary objectives of the control design were:

1. To smoothly and rapidly pass control between tasks which are assigned to processors on a functional basis.

2. To facilitate scheduling so that each processor automatically selects the most urgent task assigned to it.

3. To maximize processor utilization.

4. To accommodate various task types and to handle a wide variety of software structures independently of how many processors are in the system.

The design decision to provide two associative control memories has much to do with the realization of all four of the above objectives.

The Executive CAM contains information pertaining to individual tasks: the name of a task, the processor or processors to which it is assigned, its priority, its type, and how its execution is to be controlled. The Executive CAM is designed to act as a scoreboard where
calls to tasks are posted and as a scheduler which permits each processor to select the highest priority task assigned to it which has been enabled.

The QCAM contains information pertaining to instances of execution of tasks. It acts as a queue memory which holds requests to tasks until they are selected for execution. The QCAM makes it possible for a processor to post a call to a task and immediately search for other work since it alleviates any need for direct communication between processors in coordinating communication between tasks. The QCAM does all of the bookkeeping required to keep track of linkages between tasks and of the data which are to be processed.

The technique for handling return linkage from called task to calling task was developed to give top priority to the reactivation and completion of suspended tasks. This is necessary to minimize the probability that processor register space is tied up with information pertaining to suspended tasks.

The return linkage information which is held in the QCAM during the execution of a two-way call consists of: the name of the calling processor, the identity of the register group assigned to the calling task, and the address in the calling task to which control is to be returned. The name of the calling processor is, of course, necessary to route the return to the correct processor; the call could, in general, come from any task assigned to any processor or from a repeated task assigned to more than one processor. The register group identifier is required because, if the task is reentrant, multiple instances of execution of that task may be in progress on the same
processor; the correct instance of execution must be identified and reactivated when a return from a called task is processed. The return address is passed in the QCAM with the request because a particular instance of execution of a task may have an arbitrary number of two-way calls in progress. To provide for the storage of an arbitrary number of return addresses in the processor and to associate each of them with the proper return would be uneconomical. A task which spawns a multiplicity of parallel branches is a good example of a task which can have many calls in progress.

**Summary of Control Options**

The ACFM is designed to accommodate a wide variety of software structures. The basic concept of using two associative control memories, the Executive CAM and the QCAM, is tailored to the needs of applications, such as telephone switching control, which achieve parallelism by simultaneously sequencing many transactions through a fixed set of independent tasks. The FORK and JOIN macros make the system equally well suited to applications which achieve parallelism by spawning parallel branches within one large program. Repeated tasks make it possible to provide enhanced real-time capacity for performing certain tasks. They also make it possible to efficiently handle parallel structures which require that the same task be executed simultaneously on several processors.

The three basic task types provide considerable flexibility in task coding. The direct task is useful for handling unusual events which initiate particular system responses. Reentrant and serially reusable tasks implement the basic set of supervisory and application
programs. If a task can be coded either way, reentrant coding is preferred as long as the storage requirement for each instance of execution is not excessive. The lock option for serially reusable tasks prevents a task from being selected while it has a previous instance of execution in progress.

The queue option allows the user to specify whether all requests in the queue for a given task are to be processed immediately or whether other higher priority work is to be interleaved if it exists. Executive CAM orders which set or reset the Q bit can be used to program the system to respond to overload conditions by emptying long queues which have built up.

The option of passing a single data word directly in the QCAM word is included to speed up calls to tasks which require only one input data word from the calling task or which return only a single data word to the calling task. The indirect addressing option allows multiple arguments to be passed to a task. Any desired level of indirect addressing is possible. The parameter area is common data memory which is specified by the address passed via the QCAM word may itself contain addresses which point to data or to other addresses.

The state diagram presented in Figure 4-30 summarizes the sequences which processors execute during intertask communication. The executive microprogram that performs the operations required for each state is indicated in parentheses. The state diagram includes all transitions caused by the various control macros and all transitions brought about by the task control bits (R, D, L, and Q).
FIGURE 4-30
DETAILED STATE DIAGRAM OF THE EXECUTIVE CONTROL FUNCTIONS PERFORMED BY A PROCESSOR
CHAPTER V

PERFORMANCE EVALUATION

The structure of the ACFM and its control philosophy were presented in the previous two chapters. The purpose of this and the following two chapters is to evaluate the performance of the ACFM. Specifically, this chapter verifies the logical integrity of the control design, estimates the overhead required for intertask communication, and determines the data handling capacity of the high-speed time shared bus described in Section 3.3.3.1. Chapters VI and VII examine the real-time object identification and telephone switching control applications respectively.

A simulator was developed to model the performance of the system in executing various software structures. This simulator, called the Executive Control simulator, is described in Appendix A. An important feature of the simulation is that the level at which it is performed varies throughout the execution of the program. The actual computations performed by application programs are simply represented by lumped times which are supplied as inputs to the simulator. The simulator models the Executive CAM, the QCAM, the processor hardware
involved in task control, and the five executive microprograms described in Section 4.2. The simulator is constructed so that the number of processors, the structure of application programs, and the assignment of tasks to processors can easily be varied.

The most important performance indicators which the simulator measures are processor utilization, executive control overhead, and run time. **Processor utilization** is the percentage of real-time which each processor spends doing the productive work of executing task programs. **Executive control overhead** (simply called overhead) is the percentage of real-time which each processor consumes executing the control operations required for intertask communication. Overhead can be divided into two components: nonassociative overhead which is the percentage of real-time spent performing executive operations other than Executive CAM and QCAM operations, and associative control overhead which is the percentage of real-time spent performing Executive CAM and QCAM operations. This latter component includes the delay caused by interference at the associative control interface. **Runtime** is simply the simulated execution time of whatever program is being simulated.

Section 5.1 discusses the circuit propagation delay assumptions which were used in the simulations.

In Section 5.2, example 6 of Section 4.4 is simulated in various ways to verify the logical consistency of the control design. Simulations of example 6 demonstrate that programs with parallel branches
will execute independently of how many processors are in the system and that the various task control options function as intended.

The overhead required for intertask communication is determined in Section 5.3 as a function of the number of processors and of segment length. **Segment length** is the amount of time a processor spends executing a task between when the task is begun (or is re-awakened following a period of suspension) and when it is either terminated or suspended. (Segment length corresponds to the time spent in the task execution state of Figure 4-30.) The real-time that remains after overhead is subtracted is available for task execution; it represents the maximum possible processor utilization, or, equivalently, the maximum system capacity. Achieving this level of utilization requires, of course, that enough work be available for all processors. Simulation runs were made of 1, 4, 8, and 16 processors with constant segment lengths of 5, 10, 25, 50, and 100 microseconds. The results show that above 95 percent utilization is possible for relatively short segment lengths. An approximate analytic determination of processor utilization is presented which agrees quite well with simulation results.

The data carrying capacity of the high-speed time shared bus is estimated in Section 5.4 by two different simulators. The first, written in PL/I, determines the maximum data handling capacity of the bus for given numbers of memory modules and processors. It is used to determine how fast processors can get data when each processor submits a new request immediately following the satisfaction of its previous request. The second simulator, written in GPSS (General
Purpose System Simulator), determines the average time required for a common data memory read when the average data request rate is given.

Section 5.5 summarizes the results of the performance evaluation.

5.1 Circuit Delay Assumptions

The circuit delays assumed for processor logic and for the associative memories are consistent with anticipated improvements in integrated circuit technology over the next few years.

The times for performing operations within the five executive microprograms are based upon the assumptions that the basic microinstruction cycle is 50 nanoseconds and that register-to-register transfers take 25 nanoseconds.\(^{32}\) The times for executing individual operations within the microprograms are given in Appendix B.

The times assumed for Executive CAM and QCAM operations are 300 nanoseconds for write operations and 350 nanoseconds for read operations. This includes the propagation delay of the bus which connects processors to the associative control memories. The extra time for read operations takes into account the added delay required to return information to a processor. These times are somewhat conservative in view of the prediction that the times for performing read, write, and equality search operations will be in the 30 to 50 nanosecond range for associative memories of up to 50,000 bits by the mid 1970s.\(^{16}\) Times, based on this prediction, for performing the basic ACFM associative control operations are presented in Appendix B.
Based upon the above assumptions, the times for performing typical executive sequences are as follows:

Select a task and pass control to it - 1.6 microseconds.

Execute a one-way call, terminate the calling task,
select a new task and pass control to it - 2.475 microseconds.

Execute a two-way call, suspend the calling task,
select a new task, and pass control to it - 2.725 microseconds.

Execute a two-way dispatching call and return control to the calling task - 0.95 microseconds.

Execute a FORK macro - 0.10 microseconds.

Execute a JOIN macro, find that the count is nonzero,
suspend the active task, select a new task, and pass control to it - 1.925 microseconds.

Execute a JOIN macro, find the count zero, and pass control to the next instruction - 0.15 microseconds.

Execute a RETURN macro, terminate the active task,
select a new task, and pass control to it - 2.1 microseconds.
Pick up a return, reawaken the calling task, and pass control to it - 1.0 microseconds.

Execute a WAIT macro, select a new task, and pass control to it - 1.775 microseconds.

Execute an END macro, select a new task, and pass control to it - 1.70 microseconds.

Execute a DCALL macro, terminate the calling task, and pass control to it - 2.125 microseconds.

5.2 Simulations of Example 6

Example 6 of Section 4.4, coded with one main structure routine as shown in Figure 4-23, was simulated in various ways to verify the validity of the control design. The program structure of example 6 is illustrated in Figure 5-1 along with a fixed set of execution times which were arbitrarily chosen for tasks A through J.

This section is subdivided into three parts: Section 5.2.1 presents enough background material on parallel task structures to determine the best possible runtimes of example 6 on various numbers of processors. Section 5.2.2 presents simulation results of single instances of execution of example 6 and compares the runtimes to those which are theoretically possible. Section 5.2.3 discusses simulations of multiple instances of execution of example 6 which demonstrate the various task control options.
FIGURE 5-1

EXAMPLE 6 WITH FIXED TASK EXECUTION TIMES
5.2.1 Theoretical Consideration

A parallel structure such as the one for example 6 describes a partial ordering on the set of tasks. If two tasks, $T_1$ and $T_2$ are related by $T_1 < T_2$, then $T_1$ is a predecessor of $T_2$ and $T_2$ is a successor of $T_1$. The relation $T_1 < T_2$ is a precedence relation. It states that $T_1$ must be completed before $T_2$ can begin.\(^{33}\) If no precedence relation exists between two tasks, $T_1$ and $T_2$, they can be executed in parallel provided that all predecessors of both $T_1$ and $T_2$ have been completed. An ordered sequence of tasks $T_1, T_2, T_3\ldots$ is a sequence obeying $T_1 < T_2 < T_3 < \ldots$. The longest ordered sequence of tasks through a parallel structure, where total execution time is the measure of length, is called a critical path.\(^{34}\)

The best possible runtime for a parallel structure is equal to the critical path length. For example 6, ABEFJ, ABDFJ, and ACHIJ are critical paths. The length of each of these paths is 450 microseconds. Thus, the minimum possible runtime for example 6, regardless of how many processors are employed, is 450 microseconds.

In order to execute a parallel structure in the minimum possible time, all predecessors of tasks in all critical paths must be completed in time to insure that no task in any critical path is delayed. To illustrate this point, and to present the simulation results of Section 5.2.2, Gantt charts are used. A Gantt chart is simply a time map of the activity of each processor in the system.\(^{35}\)

Figure 5-2 is a Gantt chart of the execution of example 6 on four processors. Four processors happen to be the minimum number required
FIGURE 5-2: AN OPTIMAL FOUR-PROCESSOR ASSIGNMENT FOR EXAMPLE 6
to complete example 6 in minimum time. Processor 1 is assigned all five tasks in the critical path ABEFJ. Since ACHIJ is also a critical path, tasks C, H, and I must be executed between $t = 100$ microseconds when task A is completed and $t = 400$ microseconds when task J must begin. Tasks C, H, and I are assigned to processor 2. Since ABDFJ is also a critical path, task D must be executed in parallel with task E; task D is assigned to processor 3. To insure that task I can begin on time, both of its predecessors, tasks G and H, must complete by $t = 350$ microseconds. Task G, which is bounded by $C < G < I$, must be executed in the interval between $t = 200$ microseconds and $t = 350$ microseconds. Since there is no available time for task G in this interval on any of the other processors, task G is assigned to processor 4.\(^4\) There are many other four-processor task assignments which will execute in minimum time.

Without delving deeper into the problems of scheduling tasks within parallel structures, optimal schedules for two and three processor runs of example 6 are shown in Figures 5-3 and 5-4.\(^5\) The best possible runtimes are 550 microseconds for a two-processor system and 500 microseconds for a three processor system. The best possible runtime on a single processor system is simply the sum of the task execution times since all tasks must be executed sequentially. For example 6 this number is 900 microseconds.

\(^4\)If task G were split into two parts it could be executed on processor 3 with task D interleaved.

\(^5\)Reference 33 presents a discussion of the scheduling of tasks on a multiprocessor system.
FIGURE 5-3
AN OPTIMAL TWO-PROCESSOR ASSIGNMENT FOR EXAMPLE 6

FIGURE 5-4
AN OPTIMAL THREE-PROCESSOR ASSIGNMENT FOR EXAMPLE 6
5.2.2 Simulations of Single Instances of Execution

A single instance of execution of example 6 was simulated on 1, 2, 3, and 4 processor systems. The results are tabulated in Table 5-1.

The simulation results clearly show that a program with parallel branches will execute independently of how many processors are in the system. With the exception of the first two-processor simulation, the overhead involved in intertask communication (runtime minus best possible runtime) is constant at about 50 microseconds regardless of how many processors are involved. Except for the single processor case, the processor utilization is a relatively meaningless number since the precedence relationships among tasks result in periods where there is not enough work for all processors.

Two different task assignments were used in the two-processor case to illustrate a task assignment problem which can result when one processor coordinates the execution of a parallel structure. Both task assignments for the two-processor case (see Table 5-1) are optimal in the sense that they can theoretically be executed in 550 microseconds. Assignment 1, however, takes 55 microseconds longer than assignment 2 when executed on the ACFM with a single main routine assigned to processor 1. The reason for this is illustrated by the Gantt Charts in Figure 5-5. Refer to the actual ACFM run of assignment 1. When processor 2 completes task C, it is unable to obtain new work since processor 1 is not available to pick up the return to the main routine from task C and enable tasks G and H. This delay later
<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Task Assignment</th>
<th>Runtime</th>
<th>Best Possible Runtime</th>
<th>Processor Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>All Assigned to Processor 1</td>
<td>946 µsec</td>
<td>900 µsec</td>
<td>95.1%</td>
</tr>
<tr>
<td>2</td>
<td>(Assignment 1)</td>
<td>Processor 1: MAIN, A, B, E, F, G, J</td>
<td>654 µsec</td>
<td>550 µsec</td>
</tr>
<tr>
<td></td>
<td>Processor 2: C, D, H, I</td>
<td>Processor 2: 53.5%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(Assignment 2)</td>
<td>Processor 1: MAIN, A, C, D, F, H, J</td>
<td>599 µsec</td>
<td>550 µsec</td>
</tr>
<tr>
<td></td>
<td>Processor 2: B, E, G, I</td>
<td>Processor 2: 58.4%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Processor 1: MAIN, A, C, E, G, J</td>
<td>Processor 1: 73.6%</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Processor 2: B, D, F</td>
<td>548 µsec</td>
<td>500 µsec</td>
<td>Processor 2: 54.7%</td>
</tr>
<tr>
<td></td>
<td>Processor 3: H, I</td>
<td>Processor 3: 36.4%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Processor 1: MAIN, A, C, D, J</td>
<td>499 µsec</td>
<td>450 µsec</td>
<td>Processor 1: 60.6%</td>
</tr>
<tr>
<td></td>
<td>Processor 2: B, E, F</td>
<td>Processor 2: 60.0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Processor 3: H, I</td>
<td>Processor 3: 40.4%</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Processor 4: G</td>
<td>Processor 4: 20.2%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TABLE 5-1**

SIMULATION OF SINGLE INSTANCES OF EXECUTION OF EXAMPLE 6
ASSIGNMENT 1 (THEORETICAL)

PROCESSOR 1
A B E G F J

PROCESSOR 2
C H D I

ASSIGNMENT 1 (ACTUAL ACFM RUN)

PROCESSOR 1
A B E G F J

PROCESSOR 2
C H D I

ASSIGNMENT 2 (THEORETICAL)

PROCESSOR 1
A C H D F J

PROCESSOR 2
B G E I

ASSIGNMENT 2 (ACTUAL ACFM RUN)

PROCESSOR 1
A C H D F J

PROCESSOR 2
B G E I

FIGURE 5-5 GANTT CHARTS FOR TWO DIFFERENT TWO-PROCESSOR ASSIGNMENTS FOR EXAMPLE 6
introduces delay in the schedule for processor 1 when it completes task G and finds that task F cannot be executed because its predecessor tasks D and E are not yet complete. Assignment 2 executes without delay because the breaks between tasks on processor 1 occur at appropriate times to keep processor 2 supplied with work.

When task runtimes are known, the problem described above can usually be avoided by an appropriate task assignment. When runtimes are unknown, there are two ways to program parallel structures for the ACFM so that they complete in the shortest possible time. Both, of course, assume that tasks within different parallel branches are assigned to different processors to the extent that this is possible with the available number of processors.

1. Assign the main routine which specifies the precedence relationships among tasks to a separate processor. (Single tasks, such as tasks A and J in example 6 which must be executed sequentially, can also be assigned to this processor.) As long as this processor is assigned no other work, it is always available to pick up returns from tasks assigned to other processors and to enable the appropriate successor tasks.

2. Code the FORK and JOIN instructions as part of the tasks which make up the parallel structure, rather than in a main routine. Store the FORK counts in common data memory so they are available to all
processors. Figure 5-6 illustrates this type of coding. Each task enables its own successor tasks. At points where parallel branches merge, the JOIN macros insure that all branches are completed before any successor tasks are enabled.

Either of the above techniques can achieve the shortest possible runtime of a program with parallel branches. It should be pointed out, however, that the runtime of a program with parallel branches is uncertain if other work is interleaved with it because, unless the runtimes of the interleaved tasks are short compared to those of the parallel structure or unless a task preemption mechanism is implemented, there is no guarantee that a processor will be available when it is needed to execute a task within the parallel structure.

5.2.3 Simulations of Multiple Instances of Execution

The simulations described in this section demonstrate the ability of the control design to handle both serially reusable and reentrant tasks, and verify the capability for handling repeated tasks. Three simulations were run, each of which simulated five instances of execution of example 6 on four processors. The results are summarized in Table 5-2.

In the first simulation in Table 5-2, the main task was serially reusable. The task assignment was identical to the one used for the four-processor simulation described in Section 5.2.2. Both the queue and the lock option were specified. The lock option insured that only one instance of execution of the main task was executed at a time.
FIGURE 5-6
CODING OF EXAMPLE 6 WITH PRECEDENCE RELATIONSHIPS
CODED IMPLICITLY WITHIN THE TASKS
<table>
<thead>
<tr>
<th>TYPE OF SIMULATION</th>
<th>TASK ASSIGNMENT</th>
<th>RUNTIME</th>
<th>AVERAGE TIME PER INSTANCE OF EXECUTION</th>
<th>PROCESSOR UTILIZATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serially Reusable</td>
<td>Processor 1: MAIN,A,C,D,J</td>
<td></td>
<td></td>
<td>Processor 1: 60.5%</td>
</tr>
<tr>
<td>Main Routine With</td>
<td>Processor 2: B,E,F</td>
<td></td>
<td></td>
<td>Processor 2: 60.5%</td>
</tr>
<tr>
<td>Q=1 and L=1</td>
<td>Processor 3: H,I</td>
<td>2485 sec</td>
<td>497 sec</td>
<td>Processor 3: 40.3%</td>
</tr>
<tr>
<td></td>
<td>Processor 4: G</td>
<td></td>
<td></td>
<td>Processor 4: 20.2%</td>
</tr>
<tr>
<td>All Tasks Reentrant</td>
<td>Same as Above</td>
<td>1824 sec</td>
<td>365 sec</td>
<td>Processor 1: 82.4%</td>
</tr>
<tr>
<td>Q = 0</td>
<td></td>
<td></td>
<td></td>
<td>Processor 2: 82.4%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Processor 3: 54.8%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Processor 4: 27.5%</td>
</tr>
<tr>
<td>All Tasks Reentrant</td>
<td>Processor 1: MAIN,B,D,F,J</td>
<td></td>
<td></td>
<td>Processor 1: 79.6%</td>
</tr>
<tr>
<td>Repeated Assignment</td>
<td>Processor 2: A,C,H,I,J</td>
<td>1548 sec</td>
<td>309.6 sec</td>
<td>Processor 2: 59.7%</td>
</tr>
<tr>
<td></td>
<td>Processor 3: A,C,H,I,J</td>
<td></td>
<td></td>
<td>Processor 3: 76.2%</td>
</tr>
<tr>
<td></td>
<td>Processor 4: A,B,E,G</td>
<td></td>
<td></td>
<td>Processor 4: 82.7%</td>
</tr>
</tbody>
</table>

**TABLE 5-2**

SIMULATION RESULTS FOR FIVE INSTANCES OF EXECUTION OF EXAMPLE 6 ON FOUR PROCESSORS
The total runtime of 2485 microseconds is roughly equal to five times the runtime given in Table 5-1 for the single execution of example 6. The slight difference of two microseconds per instance of execution (497 versus 499 microseconds) is due to the fact that the queue option alleviated the need to interrogate the Executive CAM each time a new instance of execution began.

In the second simulation described in Table 5-2, all tasks were reentrant. The pipelining effect that resulted when multiple instances of execution were simultaneously in progress increased processor utilization and shortened the average runtime per instance of execution to 365 microseconds. The Gantt chart in Figure 5-7 clearly shows the progress of each instance of execution as it was being executed. Note that, because parts of other instances of execution were interleaved with it, the first instance of execution required about 725 microseconds. The other instances of execution, however, completed in rapid succession once the pipelining effect was established.

The third simulation described in Table 5-2 demonstrated the ability of the ACFM to handle repeated tasks. The main task was reentrant. Certain of the tasks A through J were assigned to more than one processor. No attempt was made, however, to choose an optimal assignment. The total runtime for five instances of execution was 1548 microseconds, or an average of 309.6 microseconds per instance of execution. Processor utilization was higher and more balanced than in the other simulations.
FIGURE 5-7
GANTT CHART OF FOUR-PROCESSOR SIMULATION OF FIVE RE-ENTRANT
INSTANCES-OF-EXECUTION OF EXAMPLE 6
5.3 Determination of Capacity

Perhaps the most meaningful performance measure for the ACFM control design is executive control overhead, that is, the percentage of time which processors consume executing the control operations required for intertask communication. Overhead is a useful performance measure because it indicates how much real-time is available in each processor for the productive work of executing task programs. This is effectively a measure of system capacity.

Overhead can be divided into two parts: Associative control overhead and nonassociative control overhead.

Associative control overhead can further be subdivided into the percentage of real time actually spent executing Executive CAM and QCAM operations and the percentage of real time spent waiting for the associative control subsystem to become available. This latter number is a measure of how much interference is present at the associative control interface. It increases both when the segment length is shortened and when the number of processors is increased since, in either case, the request rate to the associative control subsystem increases thereby increasing the probability of interference. The other components of overhead, namely nonassociative control overhead and associative control overhead excluding interference effects, increase with decreasing segment length simply because the executive operations are performed more often.
5.3.1 Overhead Simulations

Two task programs, called task A and task B, were assigned to all processors in the overhead simulations. Whenever task A was selected, it executed for a length of time equal to the segment length and then placed a two-way call to task B. When task B was selected, it executed for a length of time equal to the segment length and then returned control to task A. Since this particular choice of task programs included most of the basic associative control operations and required all five executive microprograms, it provided a reasonable approximation of the executive control activity which might be expected in a typical application. All processors were supplied with a full work load by initializing the QCAM with 100 requests to task A. Each simulation was run until 80 of these requests had been serviced.

The activity of each processor during the simulation consisted of the repetitive execution of the sequence depicted in Figure 5-8. The time spent doing executive operations in this sequence is 5.1 microseconds: 1.9 microseconds for nonassociative control operations, and 3.2 microseconds for associative control operations exclusive of any delay due to interference. A total of ten associative control operations are required in the sequence.

Simulation results for simulations of 1, 4, 8, and 16 processors with constant segment lengths of 5, 10, 25, 50, and 100 microseconds are plotted in Figure 5-9. The curves show that overhead represents only a few percent of processor real time except when segment lengths are very short. At a segment length of 100 microseconds, for example,
1 - SELECT A REQUEST FOR TASK A.

2 - EXECUTE TASK A FOR A CONSTANT TIME EQUAL TO THE SEGMENT LENGTH.

3 - PLACE A TWO-WAY CALL TO TASK B AND SUSPEND TASK A.

4 - SEARCH FOR WORK AND SELECT TASK B.

5 - EXECUTE TASK B FOR A CONSTANT TIME EQUAL TO THE SEGMENT LENGTH.

6 - EXECUTE A RETURN MACRO TO RETURN CONTROL TO TASK A AND TERMINATE TASK B.

7 - SEARCH FOR RETURNS AND PICK UP THE RETURN TO TASK A.

8 - REACTIVATE TASK A AND EXECUTE AN END MACRO TO TERMINATE IT.

FIGURE 5-8
SEQUENCE OF PROCESSOR ACTIVITY DURING OVERHEAD SIMULATION
Figure 5-9

Processor Utilization vs Segment Length - Simulation Results

Utilization (Percent)

Segment Length (Microseconds)
processor utilization is around 97% for any number of processors up to 16. This means that the capacity of the ACFM system can theoretically approach .97 Np for Np ≤ 16 provided that the segment length is at least 100 microseconds. The utilization curve for Np = 1 falls off at lower segment lengths because executive operations are required more frequently and therefore consume a greater percentage of real time. There is, of course, no interference at the associative control interface when Np = 1. The more rapid decrease in utilization with decreasing segment length for Np = 4, 8, and 16 is entirely caused by interference at the associative control interface. If there were no interference, the utilization curve for Np = 1 would apply for any value of Np.

Bar graphs, which account for 100% of processor real time, are presented in Figure 5-10, for segment lengths of 5, 10, and 25 microseconds. These graphs clearly show how real time is consumed. They dramatize the interference effects which result at short segment lengths for Np = 8 and Np = 16. Graphs for segment lengths of 50 and 100 microseconds are not shown because, in both cases, utilization is high even for Np = 16.

5.3.2 Analytic Determination of Overhead

This section presents an approximate analytic approach to determining overhead or, equivalently, maximum possible processor utilization. By definition, overhead is given by:

\[
\text{Overhead} = 100 \times \frac{(\text{Executive Control Time})}{(\text{Task Execution Time}) + (\text{Executive Control Time})}
\]  

(5.32-1)
Figure 5-10
Breakdown of Processor Real-Time
Maximum possible utilization is simply:

\[ U = 100 - \text{Overhead} \]  

(5.32-2)

For the sequence of operations in Figure 5-5, maximum possible utilization is given by

\[ U = 100 \times \left[ \frac{2S_L}{2S_L + 5.1 + 10D} \right] \]  

(5.32-3)

where \( S_L \) is the segment length and \( D \) is the delay which is incurred on an associative control operation due to interference. Both \( S_L \) and \( D \) are expressed in microseconds. The factor of 2 in equation 5.32-3 appears because two segments are executed each time the sequence is performed. The factor of 10 accounts for the 10 associative operations which are performed in the sequence. The constant term, 5.1, is the time, in microseconds, required for executive control operations exclusive of any delay due to interference.

For the single processor case, the calculation of \( U \) versus \( S_L \) is trivial since \( D = 0 \). The difficulty lies in determining \( D \) for arbitrary values of \( N_p \) (Number of processors).

The delay, \( D \), due to interference can be obtained by viewing the associative control subsystem as a single server with constant holding time and by assuming that requests arrive according to a Poisson process. A formula is given by Pollaczek(36) for estimating the average delay on all requests:

\[ \frac{D}{h} = \sum_{w=1}^{\infty} e^{-\rho w} \left[ \sum_{u=w}^{\infty} \frac{(\rho w)^u}{u!} - \frac{1}{\rho} \sum_{u=w+1}^{\infty} \frac{(\rho w)^u}{u!} \right] \]  

(5.32-4)
where $\rho$ is the load offered to the server and $h$ is the server holding time. Equation 5.32-4 is plotted in Figure 5-11.

In the present case, a processor submits 10 requests to the associative control subsystem every $2S_L + 5.1 + 10D$ microseconds. The request rate for $N_p$ processors is therefore:

$$\lambda = \frac{N_p}{0.2S_L + 0.51 + D} \text{ Requests per microsecond} \quad (5.32-5)$$

and the offered load to the associative control subsystem is:

$$\rho = h\lambda = 0.32\lambda \quad (5.32-6)$$

since an average of 320 nanoseconds is required to perform an associative control operation.

The value of $U$ for given values of $S_L$ and $N_p$ can be determined by applying the following iteration technique:

1. Using $\rho = \frac{0.32 N_p}{0.2S_L + 0.51}$

   determine $D/h$ from the curve of Figure 5-11.

2. Use the value of $D$ obtained in Step 1 to make a new estimate for $\rho$:

   $$\rho = \frac{0.32 N_p}{0.2S_L + 0.51 + D}$$

3. Repeat Step 2 until successive values of $D$ are approximately equal.
FIGURE 5-11 POLLACZEK'S AVERAGE DELAY FORMULA
4. Use the resulting value of \( D \) in equation 5.32-3 to determine \( U \).

For \( \rho_0 < 0.4 \), the initial value of \( D \) is sufficient to determine \( U \). The iteration technique is always successful for \( \rho_0 < 1 \). For \( \rho_0 \gg 1 \) the following estimate of \( D \) is used:

\[
D = 0.16 (N_p - 1) \tag{5.32-7}
\]

Equation 5.32-7 was determined by assuming that, when the offered load exceeds one, every processor always has a request waiting for the associative control subsystem. Since the bus controller cyclically looks for processor requests, and since any given processor may submit a request at any point in the bus cycle, a processor may wait 0, 1, 2 --- up to \( N_p - 1 \) holding times to obtain access to the associative control subsystem. The average wait time is given by:

\[
D = h/N_p \left[ \sum_{k=1}^{N_p-1} k \right] = h/N_p \left[ N_p(N_p-1)/2 \right] = \frac{h}{2}(N_p-1) = 0.16(N_p-1) \tag{5.32-8}
\]

In the range \( 1 \leq \rho_0 < 2 \) the following approach can be used:

1. Choose \( D \) large enough to make \( \rho_0 < 1 \).
2. Apply the iteration technique.
3. If \( D \) converges, use the resulting value in equation 5.32-3.
4. If it does not converge, use equation 5.32-8 to determine \( D \) and substitute it into equation 5.32-3.
The analytic utilization curves for $N_p = 1, 4, 8, 16, 32,$ and 64 are given in Figure 5-12 along with the data points obtained by simulation. The analytic approach and the simulation results agree quite well with each other. The curves for 32 and 64 processors indicate that high utilization is possible for segment lengths in excess of 100 microseconds.

5.4 Time-Shared Bus Simulators

Two simulators are described in this section which estimate the data carrying capacity of the time-shared bus discussed in Section 3.3.3.1. The first simulator, called TSB1, is written in PL/I. It determines the maximum data flow rate which the bus can sustain when each processor submits a new request as soon as its previous request is satisfied. The second simulator, called TSB2, is written in GPSS. It determines the data handling capacity of the bus for any given request rate. Only one simulation of each type is presented in this section. The simulators were used primarily to estimate storage interference in the applications discussed in Chapters VI and VII.

A bus transmission time of 100 nanoseconds is assumed. This includes the time required by the bus controller to coordinate bus traffic. When a processor submits a request, it takes 100 nanoseconds to recognize the request and to gate the address and, optionally, data to be written onto the bus provided that the bus is idle. The time required to gate the address from the bus into desired memory module is considered to be part of the memory cycle time. When a memory has a data word ready to send back to a processor, it takes 100 nanoseconds
FIGURE 5-12 PROCESSOR UTILIZATION vs SEGMENT LENGTH (ANALYTIC RESULTS)
for the bus controller to recognize the memory request and gate the data over the bus to the processor. Recall that an independent bus path is provided for each direction of transmission.

5.4.1 TSB1

The simulator, TSB1, is capable of determining maximum data flow rate as a function of the number of processors, the number of memory modules, and the memory cycle time.

A flowchart of TSB1 is given in Figure 5-13. The simulator measures time by keeping a running account of the current time for each processor (PROCTIME(I)), the current time for memory modules (MEMTIME(I)), the current time for the input bus (BUS1T), and the current time for the output bus (BUS2T).

When TSB1 enters the main loop (Point A in the flowchart) an iteration count is tested. If the count is not zero, that memory module or processor with minimum current time, which is requesting access to the bus, is selected. If the selected unit is a memory module, the module time and output bus time are updated to indicate that the data has been gated to the appropriate processor. The time for that processor is updated and a count, which specifies how many processor requests are to be satisfied, is decremented. The main loop is then entered.

If the selected unit is a processor, a memory module is chosen if that processor is not already waiting for a particular module to become available. This can either be done randomly or according to some other strategy such as address interleaving. The processor
TSBI

INITIALIZATION

POINT A

COUNT = 0?

NO

SCAN FOR REQ FROM MEMORIES AND PROCESSORS

SELECT REQUEST WITH MINIMUM TIME

J = INDEX OF SELECTED UNIT

PROC MEMORY?

MEMORY REQUEST

PROC OR MEMORY?

PROCESSOR REQUEST

SELECT A MEMORY MODULE IF NONE IS ASSIGNED

Y = INDEX OF MODULE

K = MAX(PROCTIME(J), BUS 1T)

PROC(J) = K + BT

BUS 1T = K + BT

TST MEM MODULE

IDLE

BUSY

MARK MODULE Y BUSY

Z = MPROC(Y)

PROCTIME(Z) = BUS 2T

MARK MODULE J IDLE

COUNT = COUNT - 1

K = MAX(MEMTIME(Y), BUS 2T)

MEMTIME(Y) = K

BUS 2T = K

Z = MPROC(Y)

PROCTIME(Z) = K

COUNT = COUNT - 1

MEMTIME(Y) = MEMTIME(Y) + MC

MPROC(Y) = J

FIGURE 5-13
FLOWCHART OF TSBI SIMULATOR
time and input bus time are updated to simulate the gating of an address onto the input bus. The selected memory module is then tested. If the module is idle, it is marked busy and the time for that module is updated by the memory cycle time. A field, called MPROC, is set to indicate which processor is requesting the module and the main loop is entered. If the memory module is busy, a calculation is made to see if it will become available in time to process the processor request. If not, the main loop is entered. If so, the memory module time and the output bus time are updated to indicate the gating of an output word to the appropriate processor. The MPROC field of that memory module is then used to select the processor which is to receive the data word. The time for that processor is updated. The count is decremented. Finally, the memory time is updated by the memory cycle time, the MPROC field is changed to identify the new processor whose request is being served, and the main loop is entered.

Table 5-3 presents the results of a simulation with four processors and eight memory modules. A memory module was randomly selected each time a processor submitted a request. The memory cycle time was 200 nanoseconds. The results show that each processor was able to obtain a data word every 550 nanoseconds. The data flow rate on the bus was 7.3 million words per second. The difference between this data flow rate and the theoretical bus capacity of 10 million words per second (one word every 100 nanoseconds) is due to storage interference.
<table>
<thead>
<tr>
<th>Number of Processors:</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Memory Modules:</td>
<td>8</td>
</tr>
<tr>
<td>Average Data Read Time:</td>
<td>550 Nanoseconds</td>
</tr>
<tr>
<td>Bus Time Per Data Word Read:</td>
<td>137 Nanoseconds</td>
</tr>
<tr>
<td>Data Flow Rate:</td>
<td>$7.3 \times 10^6$ Word/Sec</td>
</tr>
</tbody>
</table>

**TABLE 5-3**

RESULTS OF TYPICAL BUS SIMULATION USING TSB1
5.4.2 TSB2

The simulator TSB2 uses the GPSS\(^{(6)}\) simulation language to statistically determine the memory read time when the bus request rate is given. The request process was assumed to be Poisson. A flowchart of the simulator is given in Figure 5-14. The simulator monitors the time it takes a transaction to move from point A to point B. It then produces statistics on this time.

The results of a simulation of a common memory subsystem consisting of eight modules with cycle times of 200 nanoseconds is summarized in Figure 5-15. The request rate was 2.66 million requests per second. The mean read time was 426.9 nanoseconds and the standard deviation was 56 nanoseconds. Seventy percent of the data reads required only 400 nanoseconds.

5.5 Summary

The results presented in this chapter demonstrate the performance potential of the ACFM structure and its associated control philosophy. The simulations of Example 6 described in Section 5.2.2 emphasize the versatility of the ACFM in handling parallel structures. The control design can sequence the execution of tasks in a parallel structure according to the set of precedence relations with little overhead penalty. Parallel structures execute independently of both the number of processors and the manner in which tasks are assigned to

\(^{(6)}\)IBM's General Purpose System Simulator
TSB2

GENERATE REQUEST ACCORDING TO POSITION DISTRIBUTION

RANDOMLY SELECT MODULE

POINT A

QUEUE FOR INPUT BUS

SEIZE INPUT BUS AND LEAVE QUEUE

ADVANCE BY BUS TIME (100) NS

YES MODULE BUSY

NO

ADVANCE BY MEMORY CYCLE TIME

QUEUE FOR OUTPUT BUS

SEIZE OUTPUT BUS AND LEAVE QUEUE

ADVANCE BY BUS TIME (100) NS

POINT B

- TERMINATE TRANSACTION AND TABULATE RESULTS
- END SIMULATION WHEN COUNT EXCEEDED

FIGURE 5.14
FLOWCHART OF TSB2 SIMULATOR
FIGURE 5-15 TYPICAL RESULTS OF TIME-SHARED BUS SIMULATION USING TSB2
processors. Two methods were described for coding parallel structures for the ACFM which result in minimum runtime even when task execution times are unknown.

The simulations of Example 6 presented in Section 5.2.3 verify the ability of the ACFM control design to handle both serially reusable and reentrant tasks. They also demonstrate the capability for handling repeated tasks and illustrate the use of the queue and lock options.

The results of the overhead study described in Section 5.3 show that the ACFM has enormous capacity potential. Figure 5-16 summarizes these results. Each curve corresponds to a given level of utilization; it gives the average segment length required to achieve that level of utilization for a given number of processors. The nonlinear portion of each curve is in the region where overhead, exclusive of interference effects, dominates the behavior of the system. The linear portion of each curve is in the region where interference at the associative control interface is the principal contributor to overhead. This set of curves shows that high utilization is possible for relatively short segment lengths even when the number of processors is large. The degree of utilization actually achieved in an application, however, depends upon several other factors:

1. The amount of work available for each processor.
2. The effectiveness of the scheduling algorithm.
3. The existence of a common data memory subsystem which can sustain the required data flow rate with little storage interference.
Figure 5-16

Average segment length required for a given level of processor utilization vs Np.
The following two chapters show that high utilization can be achieved in real-time object identification and telephone switching control.

Section 5.4 described two simulators which estimate the data handling capacity of the time-shared bus. Simulation results demonstrate that a quite-high data flow rate is attainable.
CHAPTER VI

APPLICATION OF THE ACFM TO REAL-TIME OBJECT IDENTIFICATION

The applicability of the ACFM architecture to a particular solution of the problem of three-dimensional object identification is investigated in this chapter. In broader terms, the results of this chapter demonstrate that the ACFM is well-suited to applications in which parallelism is achieved by breaking a single application program into tasks which can be executed in parallel.

Section 6.1 presents a brief description of the problem and the algorithm which solves it. For details, the reader is referred to the dissertation by Advani. (37)

Section 6.2 discusses the appropriate ACFM configuration for the application and Section 6.3 describes how the algorithm is programmed for the ACFM. The results of simulations of the execution of this algorithm on one, four, and twenty-eight processors are presented in Section 6.4. Finally, in Section 6.5 it is concluded that the ACFM can greatly reduce the computation time required for object identification.
6.1 Description of the Application

6.1.1 Overview

The basic problem of three-dimensional object identification can be stated as follows: Given a set of data points on the boundary of a two-dimensional picture of an unknown three-dimensional object, identify the object to which the data points belong and determine its three translations and three rotations (the three Euler angles) with respect to some coordinate system. (The set of three translations and three rotations will be called a parameter vector throughout this chapter.) A recognition algorithm for this problem has been developed for the digital computer by Advani.\(^37\)

A description of each object that can be identified is stored in the memory of the digital computer. The description consists of a set of three-dimensional coordinate points and a connection matrix which specifies the connectivity of these points. When the computer is supplied with a set of data points on the boundary of a picture of some object to be identified, it performs an approximate calculation to select the first object to be tested. (This object is referred to as the test object.) An initial value is then chosen for the parameter vector. Using this guess, a silhouette of the test object is generated by a silhouette generation subroutine. A silhouette is a set of points on the boundary of a two-dimensional projection of the test object. The recognition algorithm then uses a gradient descent approach to optimize the parameter vector in such a fashion that the misalignment
between a silhouette of the test object and the given set of data points is minimized. The misalignment is represented by an appropriate error function. If the minimized error function is smaller than some value, the object being identified is considered to be the test object and the adjusted parameters are taken as its positional coordinates. Otherwise, a new test object is selected and the entire procedure is repeated.

The routine which finds a local minimum of the error function is called LOCMN. A detailed flowchart of LOCMN is given in a paper by R. B. McGhee.\(^{(38)}\) Basically, LOCMN repetitively calls a regression analysis routine, REGRES, which computes the next parameter estimate. The REGRES routine calls the silhouette generation routine seven times using slightly different values for the six positional parameters in order to calculate the partial derivatives of the error function with respect to each of these six parameters. The derivatives are used to find a new parameter vector which will reduce the magnitude of the error function. The new error function is then calculated. The process continues until the error function becomes smaller than some value or until a loop count criterion is exceeded. Actually, two other routines, GRASER and GRPREX may be called by LOCMN. GRASER determines an optimum gradient by Newton-Raphson techniques. GRPREX calculates the gradient whenever a range constraint on the parameters is violated. Both of these routines are ignored in the present discussion since they are seldom required.)
A flowchart of the complete algorithm for finding the global minimum of the error function is depicted in Figure 6-1. Because there exists the possibility that the initial guess for the three rotational coordinates will lead to a local minimum of the error function rather than the global minimum, a routine called RANSER (RAndom SEaRch) is used to randomly choose many sets of initial values for the rotational coordinates with the expectation that one of these sets will lead to the global minimum.

6.1.2 Silhouette Generation

The generation of the silhouette of a test object is the most time consuming task in the recognition algorithm. And, since a new silhouette must be generated each time the optimization procedure adjusts any of the components of the parameter vector, silhouette generation accounts for most of the total computational time.

The silhouette generation algorithm is described in detail in Chapter III of Advani's dissertation. Briefly, the input to the silhouette routine consists of the parameter vector, the set of three-dimensional coordinates of the test object, and the corresponding connection matrix. As shown in Figure 6-2, the first step is to adjust the three-dimensional coordinates to the position specified by the six positional parameters and then to project them onto a two-dimensional plane. The coordinate axes of this plane are labeled U and V. That point with maximum U coordinate is selected as the first boundary point. The connection matrix is then used to find all lines
CHOOSE INITIAL PARAMETER VECTOR $\hat{C}$

EVALUATE ERROR FUNCTION $\phi$

SOLUTION STABLE?

NO → ERROR EXIT

YES → CALL LOCMN

$\hat{C}_0 = \text{PARAMETER VECTOR}$

$\phi_0 = \text{ERROR FUNCTION}$

$K = 1$

$K < \text{TRIAL}$?

NO → $K \leftarrow K + 1$

YES → CALL RAMSER

$\hat{C}_1 = \text{BEST VALUE}$

FOR PARAMETER VECTOR

CALL LOCMN

$\hat{C}_1 = \text{PARAMETER VECTOR}$

$\phi_1 = \text{ERROR FUNCTION}$

$\phi_1 < \phi_0$?

YES → $\hat{C}_0 = \hat{C}_1$

NO → $\hat{C}_0 = \hat{C}_1$

PRINT OUT FINAL PARAMETER VECTOR $\hat{C}_0$ AND MINIMUM ERROR FUNCTION $\phi_0$

END

FIGURE 6-1: FLOW CHART OF TOTAL ALGORITHM
SILHOUETTE

CALCULATE THE COORDINATES OF THE PROJECTED POINTS FOR THE GIVEN PARAMETERS

I = I

BP(I) = PROJECTED POINT WITH MAXIMUM U COORDINATE

FIND ALL LINES WHICH CONNECT TO BP(I) AND SELECT THE ONE MAKING MINIMUM ANGLE WITH THE PREVIOUS SIDE OF THE BOUNDARY. (FOR THE FIRST BOUNDARY POINT, ANGLES ARE WITH RESPECT TO THE U-AXIS.)

REFER TO THE SELECTED LINE AS LINE L

TEST ALL OTHER LINES IN THE PROJECTION FOR REAL INTERSECTIONS WITH LINE L

ANY REAL INTERSECTIONS?

NO

YES

PICK THE ONE CLOSEST TO BP(I)

I = I + 1

BP(I) = COORDINATES OF CLOSEST INTERSECTION

MODIFY CONNECTION MATRIX TO ACCOUNT FOR THE NEW POINT

I = I + 1

BP(I) = POINT AT THE END OF LINE L

FIGURE 6-2
FLOWCHART OF THE SILHOUETTE GENERATION SUBROUTINE
which connect to this boundary point. That line which makes the minimum angle with the U-axis (or with the previous side of the boundary for later boundary points) is selected. The real intersections between this line and all other lines in the projection are determined. If any exist, the coordinates of the closest intersection to the most recently determined boundary point are the coordinates of the next boundary point. If there are no real intersections, the next boundary point is the point at the end of the line making minimum angle with the previous side of the boundary. The algorithm is continued until the initial boundary point is encountered again. The coordinates of the boundary points (BFs) are returned to the calling program.

6.1.3 Advani's Simulation

Advani applied his algorithm to the identification of a prototype of an airplane. The input data consisted of 79 points around the boundary of a picture of the airplane. Because of the considerable computational time required for the execution of the total algorithm, Advani set the number of trials \(N_{\text{trial}}\) in Figure 6-1) to one so that only one execution of the LOCMM routine was required. Even with this simplification, it took 80 minutes of runtime on a PDP-10 computer to identify the airplane. Most of this execution time was spent generating silhouettes. (Approximately 110 silhouettes were required, each of which took about 40 seconds to generate.) It is this airplane identification problem that is simulated on the ACFM.
6.2 ACFM Hardware Configuration

The ACFM configuration chosen for the object identification problem is depicted in Figure 6-3.

The coordinate descriptions of all objects which can be identified are stored in a Block Oriented Random Access Memory (BORAM). As indicated in Section 3.3.2, the BORAM has an access time of a few microseconds and a word transfer rate of 10 megahertz. Data are block transferred to the common data memory for processing. The input/output subsystem communicates directly with the BORAM.

Local data memory consists of associative cache buffers. These are particularly appropriate for the object identification problem since long computations on localized blocks of data are required.

Because periods where there is a high data flow rate to and from high-speed common data memory are very short compared to the time that processors spend doing computations on localized blocks of data stored in the associative buffers, the design of high-speed common data memory is not critical.

The CPUs are high-speed floating point units. Each processor is assumed to be 10 times as fast as the PDP-10 processor. The factor of 10 is based upon an investigation of the speeds for performing floating point arithmetic that are realistically possible with current integrated circuit technology.\(^{39}\) The reason for this assumption is that a fair evaluation of the performance of the ACFM in this application dictates that processor speeds be compatible with the speeds assumed for executive control operations.
HIGH-SPEED COMMON DATA MEMORY

BORAM

EXECUTIVE CAM

INPUT/OUTPUT SUBSYSTEM

FLOATING POINT CPU

PROGRAM MEMORY

ASSOCIATIVE CACHE

FLOATING POINT CPU

PROGRAM MEMORY

ASSOCIATIVE CACHE

FLOATING POINT CPU

PROGRAM MEMORY

ASSOCIATIVE CACHE

FIGURE 6-3
ACFM HARDWARE CONFIGURATION FOR REAL-TIME OBJECT IDENTIFICATION
6.3.1 ACFM Software Description

6.3.1 General Presentation

Because the computations performed by application programs were represented by lumped times, it was not possible to simulate the data dependent decisions which determine the actual flow through the routines that implement the recognition algorithm. Consequently, the application was programmed for the simulator to represent the trace of a typical run.

The main routine, called REALMAIN, is equivalent to the program depicted in Figure 6-2 with N_trial = 1. A flowchart of REALMAIN is shown in Figure 6-4. Lumped times represent computations performed within REALMAIN exclusive of those performed by the various subroutines it calls. The SUMSQ subroutine calls a subroutine called PRSS which in turn calls the silhouette generation subroutine, SILHETTE. The end result of calling the SUMSQ subroutine is to determine the error function corresponding to a given parameter vector. The LOCMN subroutine is called next.

A flowchart of the LOCMN subroutine is depicted in Figure 6-5. In a typical run, ten iterations are required to converge on a local minimum of the error function. Each iteration requires a call to the RGRES subroutine followed by a call to the SUMSQ subroutine. RGRES calls PRSS seven times. PRSS in turn calls SILHETTE. Consequently, eight silhouettes are generated during each iteration (seven by RGRES and one by SUMSQ), for a total of 80 by the LOCMN subroutine.
CALL SUMSQ ROUTINE TO DETERMINE ERROR FUNCTION

CALL LOCMN ROUTINE TO FIND OPTIMUM PARAMETERS AND MINIMUM ERROR FUNCTION

LUMPED TIME OF 1 MILLISECOND FOR ALL OTHER PROCESSING IN REALMAIN

END

FIGURE 6-4: FLOWCHART OF REALMAIN
LOCNM

COUNT = 11

COUNT = COUNT - 1

COUNT = 0?

YES

RETURN

NO

CALL RGRES ROUTINE

LUMPED TIME OF 0.1 MILLISECOND FOR CALCULATIONS

CALL SUMSQ ROUTINE

LUMPED TIME OF 0.1 MILLISECOND FOR CALCULATIONS

7 SILHOUETTES

1 SILHOUETTE

FIGURE 6-5: FLOWCHART OF LOCNM
The total of 81 silhouettes generated during an execution of REALMAIN is less than the 110 that an actual run on the PDP-10 requires. The difference is due to the fact that the optimum gradient routine GRASER is not included in the simulation.

A subroutine level diagram is presented in Figure 6-6 in order to clarify the subroutine linkages within REALMAIN.

Parallelism within the recognition algorithm is represented in two ways: The seven silhouettes required by the RGRES subroutine are generated in parallel. This is possible because each silhouette is generated using the original parameter vector with one parameter perturbed. (The seven silhouettes include one with the unperturbed parameter vector and six with each parameter perturbed.) The results are then collected and used to determine a set of partial derivatives as described in Section 6.1.1. A flowchart of RGRES is depicted in Figure 6-7.

The silhouette generation routine is segmented into four parallel parts. Instead of selecting only the point with maximum U coordinate as the first boundary point, four starting points are selected corresponding to max U, max V, min U, and min V. The SILHETTE subroutine is shown in Figure 6-8. The initial lumped time represents the time required to transform the set of three-dimensional object coordinates for the test object to the position specified by the input parameter vector and the time required to project these transformed points onto the U-V plane. While this is being done, the four starting points are selected. A FORK macro then spawns four calls
FIGURE 6-6  SUBROUTINE LEVEL DIAGRAM FOR REALMAIN
FIGURE 6-7:
FLOWCHART OF RGRES

LUMPED TIME OF 1 MILLISECOND FOR CALCULATIONS PERFORMED BY RGRES
FIGURE 6-8:
FLOWCHART OF SILHETTE
to a subroutine called SILCALC which determines all the boundary points lying between the starting point it is given and the next starting point. For example, one execution of SILCALC calculates boundary points lying between the boundary point with maximum U coordinate and the one with maximum V coordinate; another execution of SILCALC calculates boundary points lying between the boundary point with maximum V coordinate and the one with minimum U coordinate; etc. When all four executions of SILCALC are completed, the SILHETTE subroutine forms the complete list of boundary points and returns to the calling task.

As indicated in Section 6.1.3, it takes an average of 40 seconds to generate a silhouette of the airplane on the PDP-10. The assumption that an ACFM processor is 10 times as fast as the PDP-10 reduces this time to four seconds. It is assumed that each execution of SILCALC calculates one fourth of the silhouette in one second. This is a simplifying assumption in view of the fact that the four extreme points selected as starting points will not necessarily divide the boundary into four approximately equal sections. Because the silhouette generation routine accounts for almost all of the total execution time of the algorithm, the lumped execution times for other parts of the algorithm are unimportant in the simulation. These were somewhat arbitrarily chosen to be either one millisecond or one/tenth millisecond as indicated in the flowcharts of Figures 6-4, 6-5, and 6-7.
6.3.2 Data Traffic Between Tasks

This section describes the manner in which data would be handled in an ACFM implementation of the object identification problem.

The input data, the three-dimensional coordinates and the connection matrix for the test object, and other parameters of importance are transferred from the BORAM to high-speed common data memory at the beginning of the run. The addresses of these blocks of data in common data memory remain fixed during the execution of the algorithm.

When a task is called, it is passed an address via the QCAM. This address points to a table of addresses in common data memory. The addresses in this table, in turn, point to the required blocks of data. For example, the address table for a call to the SILHETTE subroutine is shown in Figure 6-9. The SILHETTE subroutine calculates the projected U-V coordinates using as input the P, Q, R coordinates of the test object and the parameter vector. It then spawns four calls to the SILCALC subroutine. The address table passed to the SILCALC subroutine by the SILHETTE subroutine is shown in Figure 6-10.

At the time that the REGRES subroutine spawns seven parallel executions of the SILHETTE subroutine and each execution of SILHETTE, in turn, spawns four executions of the SILCALC subroutine, there is potentially considerable congestion at the common data memory interface. For example, in a 28 processor system in which the SILCALC subroutine is assigned to every processor, there is a period when all processors are trying to fetch their copy of the data. Simulation results presented in Section 6.4 show, however, that by interleaving
Address of Three-Dimensional Test Object Coordinate Table (P, Q, R coordinates)
Address of Six Positional Parameters
Address of Connection Matrix
Address of Table of Other Data Items
Address of Area Where Boundary Points are to be Returned

FIGURE 6-9
ADDRESS TABLE WHOSE ADDRESS IS PASSED TO THE SILHETTE ROUTINE

Address of U-V Coordinates
Address of Connection Matrix
Address of Four Starting Points
Address of Starting Point for this Instance of Execution
Address of Other Pertinent Information
Address Where Boundary Points are to be Returned

FIGURE 6-10
ADDRESS TABLE WHOSE ADDRESS IS PASSED TO THE SILCALC ROUTINE
data addresses among all modules, it is possible to greatly reduce this congestion. As a processor reads the data, a copy is recorded in its associative cache buffer eliminating the need for going to common memory the next time the data is needed.

Data is passed between other routines in the algorithm in a manner similar to that described for the silhouette routine.

6.4 Simulation Results

The results of simulation runs of the airplane identification problem on one, four, and twenty-eight processors are tabulated in Table 6-1.

The total sequential runtime for the entire algorithm is 324.185 seconds. The runtime for the single processor simulation also came out to be 324.185 seconds since the executive control overhead was negligible compared to the long computations performed within the silhouette generation subroutine. Processor utilization was, for all practical purposes, 100 percent.

In the four processor simulation, all tasks were assigned to one processor except SILCALC, which was assigned to all four. The total runtime was 81.221 seconds or approximately one-fourth the runtime of the single processor case. The utilization of each of the four processors was in excess of 99.7 percent.

The twenty-eight processor simulation fully exploited the parallelism programmed into the algorithm. The REALMAIN, LOCMN and SUMSQ tasks were assigned to processor one. The SILHETTE subroutine
<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Task Assignments</th>
<th>Runtime</th>
<th>Processor Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>All tasks assigned to Processor 1</td>
<td>324.185 seconds</td>
<td>100%</td>
</tr>
<tr>
<td>4</td>
<td>Processor 1: All tasks</td>
<td>81.221 seconds</td>
<td>Processor 1: 99.9%</td>
</tr>
<tr>
<td></td>
<td>Processors 2-4: SILCALC</td>
<td></td>
<td>Processors 2-4: 99.7%</td>
</tr>
<tr>
<td>28</td>
<td>Processor 1: All tasks</td>
<td>21.142 seconds</td>
<td>Processors 1-4: 99.5%</td>
</tr>
<tr>
<td></td>
<td>Processors 2-7: SILHETTE and SILCALC</td>
<td></td>
<td>Processors 5-28: 47.3%</td>
</tr>
<tr>
<td></td>
<td>Processors 8-28: SILCALC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
was assigned to processors one to seven and the SILCALC subroutine was assigned to all processors. During those periods when the RGRES subroutine was executing, all twenty-eight processors were busy executing the SILCALC subroutine. During periods when the SUMSQ subroutine was executing, only the first four processors were being utilized since only one silhouette was being generated. The total runtime was 21.142 seconds or a factor of 15.34 shorter than the single processor run. The utilization of processors one to four was 99.5 percent. The utilization of each of the other processors was 47.3 percent because they had available work only when the RGRES subroutine was executing.

The interface with high-speed common data memory was simulated using the simulator, TSB1, (see Section 5.4.1) to determine how much congestion occurs when all processors are simultaneously attempting to obtain input data for the SILCALC subroutine. The results are summarized in Table 6-2.

For the four-processor case, four memory modules with cycle times of 200 nanoseconds were assumed. Addresses were assumed to be interleaved among the four modules so that each processor reads a word from module 1, then reads a word from module 2, etc. The results show that interference was completely negligible. The average read time per word was 400.8 nanoseconds, only slightly greater than the interference-free read time of 400 nanoseconds. Since only about 400 input words are required by SILCALC, the time to read the input data is negligible compared to the execution time of SILCALC.
<table>
<thead>
<tr>
<th>Number of Memory Modules</th>
<th>Module Cycle Time</th>
<th>Worst-Case Read Time Per Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>Four Processor Simulation</td>
<td>4</td>
<td>200 nanoseconds</td>
</tr>
<tr>
<td>Twenty-eight Processor Simulation</td>
<td>7</td>
<td>600 nanoseconds</td>
</tr>
</tbody>
</table>

**TABLE 6-2**

**RESULTS OF TIME-SHARED BUS SIMULATIONS**
For the twenty-eight processor case, seven storage modules with access times of 600 nanoseconds were assumed. Again, addresses were interleaved among storage modules. The simulation results show that the average read time is 2.8 microseconds per word when all 28 processors are trying to obtain data. This is the best possible read time under such conditions since, due to the cyclic nature of the time-shared bus, a given processor can obtain access to the bus only once every 2.8 microseconds. Faster memory modules do not change this situation. The 600 nanosecond cycle time is optimum since the request rate per module is one request per 700 nanoseconds. With this request rate, a given module is gating a word onto the output bus at the same time an address is being sent over the input bus. The read time of 2.8 microseconds is a worst-case number. In reality the request rate to common memory would be much lower owing to the fact that processors are performing calculations on data words as they are received. However, even the worst-case read time of 1.12 milliseconds for 400 words is negligible compared to the execution time of the SILCALC routine.

6.5 Summary

The results of this chapter demonstrate that the ACFM control philosophy is well-suited to applications in which parallelism is achieved by spawning internal parallel branches within a larger program structure. The application was a particularly effective test of the control design since it involved repeated tasks, parallel structures, and five levels of subroutine depth.
The simulations show that the object identification time could be shortened by a factor of 153.4 if the algorithm were run on a twenty-eight processor system consisting of processors which are 10 times faster than the PDP-10. A factor of 15.34 is directly attributable to multiprocessing. It is likely that this factor would be increased substantially if the algorithm were examined more closely for possible parallelism. It is clear that the runtime of the complete recognition algorithm as depicted in Figure 6-1 could be dramatically shortened by multiprocessing since the many silhouettes which must be generated by the RANSER subroutine can be generated in parallel.

The problem that was simulated in this chapter brought up an interesting point. Parallelism was achieved by simultaneously executing many copies of the same task. This type problem would not run effectively on a direct multiprocessor because severe storage interference would occur when many processors tried to execute the same code. Address interleaving or associative buffers would improve performance, but the most effective solution would be to bring a copy of the entire task program into a local program memory associated with each processor. The problem appears to be appropriate for an array processor. A closer look reveals, however, that the detailed instruction trace of each execution of the task is quite data dependent and quite different from the traces of other executions of the task. This situation would be difficult to program on an array processor. Again, the conclusion is that the problem is best solved on an ACFM-like structure.
CHAPTER VII

APPLICATION OF THE ACFM TO TELEPHONE SWITCHING CONTROL

This chapter examines the applicability of the ACFM to telephone switching control. Multiple processors are effectively utilized in this application by simultaneously sequencing many transactions, which are at various stages of completion, through a fixed set of independent task programs. The application is fundamentally different from the real-time object identification application which achieved parallelism by breaking one large program into segments which could be executed in parallel.

Section 7.1 presents a brief description of telephone switching control. A toll switching office, which communicates only with other switching offices, is chosen for the discussion because the operations required to switch a telephone call through a toll office are simpler to describe than those required in a local switching office which communicates with customer telephones.

An appropriate ACFM hardware configuration for telephone switching control is discussed in Section 7.2. Section 7.3 describes how the application would be programmed for the ACFM. Section 7.4 presents the results of a simulation study to determine the capacity of the ACFM system in terms of the number of calls which can be processed per hour.
Results indicate that capacity grows almost linearly from 1.37 million to 4.96 million calls per hour as the number of processors is increased from one to four. A summary is given in Section 7.5.

7.1 Description of Telephone Switching Control

The primary function of a toll telephone switching office is to connect incoming calls arriving on incoming trunks through a switching network to outgoing trunks which carry the calls on toward their destinations. A trunk is simply a transmission circuit which carries a telephone conversation. During the interval when it is being switched (call setup), a call requires the use of various resources that are time-shared among all calls. Telephone switching control involves both the management of these resources and the performance of processing functions on behalf of calls such as, for example, the analysis of incoming digits to determine where the call is to be routed.

In modern telephone switching offices, a digital computer performs all common control operations. The No. 1 ESS System is a noteworthy example of such an office. A conceptual block diagram of an electronic toll office is depicted in Figure 7-1. The switching network can connect any incoming trunk to any outgoing trunk, or it can connect any trunk to any of several types of service circuits. Service circuits include, for example, digit receivers and digit transmitters. Input information for the control computer is provided by input units which monitor various points in the system where information must be obtained. These points include trunks and digit receivers. Output
FIGURE 7-1,
CONCEPTUAL VIEW OF A TOLL SWITCHING OFFICE
units connect to various points in the system where actions must be controlled by the control computer. These actions include the dispatching of network commands which set up or break down connections through the network, the initiation of supervisory signals which go over trunks to other offices, and the outpulsing of digits to destination offices.

The control computer time shares a fixed set of application programs among calls which are at various stages of call setup or are in the process of being disconnected. These application programs require a rather large data base. Several basic data types are pertinent to this discussion:

**Translation Data** is permanent data which describes the specific layout of facilities in the particular office, for example: the association of input/output port numbers with memory locations that contain status information concerning these ports; and, the association of groups of dialed digits with trunk routes.

**Status Data** indicates the current status of various points in the office including, for example: trunk registers which reflect the state of trunks and service circuits and which contain data pertinent to the particular state; and, a network map which specifies the state of links in the network.
Transient Per-Call Data consists of blocks of data called call registers which hold information pertaining to individual calls during the process of call setup.

7.1.1 The Concept of Call States

Up to this point in the discussion, the call-processing job has been considered to be broken down into a set of independent tasks, but nothing has been said regarding what is the basis for this segmentation. The placing of a telephone call results in a series of stimuli being applied to the processing complex in a time sequence. In response to each of these stimuli, some sequence of actions is performed on behalf of the associated call. The stimuli consist of supervisory signals passed between telephone offices, digits, and signals originating in the peripheral hardware of the office as, for example, an outpulsing-complete signal from a digit transmitter. In addition, stimuli in the form of software timers indicate when actions are to be performed on behalf of certain calls. Since the times between occurrences of stimuli for a particular call are long compared to the internal processing speed and are often statistical in nature, they tend to naturally segment a call into states.

The segmentation of a call into states separated by real-time breaks is the accepted way of describing the sequence of transactions for an individual call. This representation is strongly analogous to the state diagram of a sequential circuit. The next state is a function of the present state and the input stimulus.
7.1.2 State Diagram of a Typical Call

A simplified state diagram of a typical call is depicted in Figure 7-2. Corresponding to each state there is a task program which will be called a call state routine. Each call state routine may call various service subroutines. Three service subroutines are required by the call being described:

The translation subroutine performs all translations including: those which relate input/output port numbers to trunk registers; route translations; and trunk hunt translations which select idle service circuits and which select an available trunk going to the desired destination.

The network subroutine interrogates the network map to find a path between two specified end points and then issues a network command to the output unit to establish the connection. It also handles the tearing down of connections.

The digit analysis subroutine analyzes incoming digits.

The flow of a call through the state diagram of Figure 7-2 will now be described:

State 1: The first stimulus on the call is the seizure of an incoming trunk by the previous switching office. In response to this seizure, the origination routine is executed. The origination routine allocates a call register for the call and initializes it. It then selects a digit receiver
SEIZURE
ORIGINATION
150 MILLISECOND
TIMEOUT
WAITING FOR
DIGITS
FIRST DIGIT
GROUP
DIGIT
ANALYSIS,
ROUTE
TRANSLATION,
AND TRUNK
HUNT
REMAINING
DIGITS
RECEIVE
DIGITS AND
IDLE DIGIT
COLLECTION
BEGINNING OF
START SIGNAL
FOR NEXT OFFICE
WAIT FOR
END OF
START SIGNAL
FIGURE 7-2
STATE DIAGRAM
FOR A TYPICAL
TOLL TELEPHONE CALL
and issues a network command to connect the digit receiver to the incoming trunk. This operation requires both the translation and network subroutines. An output order is issued to begin sending a signal, which will be called a start signal, back to the previous office telling it to commence sending digits. At this point, an entry is made in a timing list to time 150 milliseconds, the duration of this start signal. The call remains in State 1 until a 150-millisecond timeout occurs.

**State 2:** When the 150-millisecond timeout occurs, State 2 is entered. An output order is then issued to remove the start signal. The call remains in State 2 until the first digit group arrives.

**State 3:** When the first digit group arrives, State 3 is entered. The digit analysis subroutine is called to determine how the digits are to be handled. The translation subroutine is called twice: once to perform a route translation; and, once to perform a trunk hunt to select an idle outgoing trunk. The call remains in State 3 until the remaining digits arrive.

**State 4:** State 4 is entered when the remaining digits arrive. The digit analysis subroutine is called to analyze the digits. The digit collection process is then idled by calling the network subroutine to remove
the connection between the incoming trunk and the digit receiver and by calling the translation subroutine to mark the digit receiver idle. The translation subroutine is again called to select an idle digit transmitter. Finally, the network subroutine is called to connect this transmitter to the outgoing trunk. The call remains in State 4 until the beginning of a start signal is received from the next office.

State 5: When the beginning of a start signal is received, State 5 is entered. The call remains in State 5 until the end of the start signal occurs.

State 6: When the end of the start signal is received from the next office the outpulsing routine is called to format digits and send them to the output unit which sends them to digit transmitter for outpulsing. The call remains in State 6 until outpulsing is complete.

State 7: State 7 is a transient state. When an outpulsing complete signal is received from the digit transmitter, the outpulsing process is idled by calling the translation and network subroutines to idle the digit transmitter and to remove the network connection between the transmitter and the outgoing trunk. The network subroutine is then called again to connect the incoming trunk to the outgoing trunk. Finally, the call register
is deallocated since the process of call setup is complete. State 8 is then entered.

State 8: The call remains in State 8 while it is waiting for an answer signal and through the entire period of conversation until a disconnect occurs. On the average, four stimuli occur while the call is in State 8. Each time a stable state stimulus occurs, the stable state routine is called to respond to the stimulus in an appropriate manner. Eventually, one of the stable state stimuli is recognized as a disconnect and the transient State 9 is entered.

State 9: The disconnect routine is called to idle all facilities used by the call and to remove all network connections. The translation and network subroutines are required. The call is now complete.

7.2 Description of the ACFM Hardware Configuration for Telephone Switching Control

A complete block diagram of a four-processor ACFM system for telephone switching control is shown in Figure 7-3. Based upon the simulation results presented in Section 7.4, the capacity of the system will meet the needs of the largest anticipated toll office.

7.2.1 Data Memory

Four types of common data memory are provided:

1) A BORAM (Block Oriented Random Access Memory) contains a copy of all programs and permanent data tables. It
FIGURE 7-3
FOUR-PROCESSOR ACFM CONFIGURATION FOR TELEPHONE SWITCHING CONTROL
acts as a high-speed backup storage subsystem
making system reconfiguration possible at microsecond speeds.

2) Low-speed random access common data memory provides
bulk storage for trunk registers and translation
data. It satisfies the requirement that individual
data items, although accessed infrequently, must
be available within a few microseconds when needed.
A module size of 500K (where K = 1024) 32-bit words
and a cycle time of 2 microseconds is assumed. (32)
(As many as six modules might be needed in a large
installation.)

3) High-speed random access common data memory is used
for call registers and other frequently accessed
data. A module size of 32K and a cycle time of 200
nanoseconds is assumed. (32) (The number of modules
is in the range of 8 to 20.)

4) An associative common data memory module controls
the dynamic allocation and deallocation of call
registers.

Local data memory serves as a scratch area for processors and also
holds all data which is private to task programs assigned to the cor-
responding local program memory. The network map required by the net-
work subroutine is an example of data which is private to one task
program. A cycle time of 100 nanoseconds is assumed for local data
memory. (32)
7.2.2 The CPU

Telephone switching control involves making many data-dependent logical decisions on data fields of varying sizes. Consequently, the CPU instruction repertoire includes instructions which isolate an arbitrarily positioned field of arbitrary size within a data word; instructions which pack data fields into words for writing into data memory; instructions which conditionally transfer to some other point in the program based upon the results of a Boolean decision involving some data item; and, Boolean instructions which logically combine data fields. The arithmetic capability of the CPU is minimal; addition, which is used primarily for address calculation, is the only necessary operation.

The basic cycle time of a CPU is assumed to be 200 nanoseconds. Execution times for instructions classified according to the level of memory they access are as follows:

- Instructions which obtain operands from and store results in general purpose registers (Register Transfer Instructions): 200 nanoseconds.

- Instructions which read from or write into local data memory (Local Data Memory Instructions): 300 nanoseconds.

- Instructions which either read from or write into high-speed common data memory (High-Speed Common Data Memory Instructions): 600 nanoseconds.
Instructions which read from or write into low-speed common data memory (Low-Speed Common Data Memory Instructions)\(^7\): 2.4 microseconds.

A study of an actual trace of a call in a No. 1 ESS office revealed that approximately 44 percent of all instructions are of the register transfer type and that 56 percent read from or write into data memory.\(^{(42)}\) The following instruction mix is somewhat arbitrarily assumed for the ACFM system:

- **Register Transfer Instructions:** 44 percent
- **Local Data Memory Instructions:** 28 percent
- **High-Speed Common Data Memory Instructions:** 25 percent
- **Low-Speed Common Data Memory Instructions:** 3 percent

With this instruction mix, the average instruction execution time is around 400 nanoseconds.

### 7.2.3 Memory Interconnection Subsystem

All four types of common data memory reside on the same time-shared bus. This is possible because, with the assumed instruction mix, the request rate to the time-shared bus in a four-processor system is one request per 350 nanoseconds. For fewer than four processors, the request rate is even lower. Simulation results presented in Section 7.4 indicate that the bus can handle this request rate with minimal queuing delay.

---

\(^7\)The times for instructions which access common data memory include 200 nanoseconds for two-way time-shared bus transmission and 200 nanoseconds for processor actions following the reception of the data word.
7.3 ACFM Software Structure

A conceptual view of the ACFM software structure for telephone switching control is depicted in Figure 7-4. The input unit and the output unit are assumed to be autonomous units which communicate with the processing complex through buffers in low-speed common data memory. (The input/output function could equally well be performed by a dedicated processor in the ACFM structure.) For each of the nine states of the call diagrammed in Figure 7-2, there is a corresponding call state routine. The call state routines invoke the three service subroutines, described in Section 7.1, as needed. Call state routines are activated in two ways: 1) By an input routine which unloads the real-time stimuli in the input buffer and then calls the call state routines according to the type of stimuli and the present call state; and 2) by a software timer routine which generates stimuli when software timers timeout and then calls the call state routines as described in 1) above.

Calls which are in States 1 through 6 have a call register assigned to them. The call register address is stored in the trunk register(s) associated with the call. When an input stimulus is read from the input buffer a translation is performed, using the input port number as input, to locate the corresponding trunk register. The input type and the input data are then written into the call register pointed to by this trunk register. After the appropriate call state routine is determined, the call register address is passed to it. All information needed by the call state routine is available in the call register.
FIGURE 7-4

CONCEPTUAL VIEW OF THE ACFM SOFTWARE
STRUCTURE FOR TELEPHONE SWITCHING CONTROL
Originating calls and calls which are in State 8 have no associated call register. In this case a translation is performed to determine the trunk register address. The trunk register state is then used to determine whether to call the origination routine or the stable state routine. In either case, the input type, input data, and trunk register are passed to the called routine.

7.3.1 Scheduling Philosophy

The telephone switching control application has not really been described in enough detail to say much about how work should be scheduled. Only a minimal number of application programs have been considered; and, supervisory and support programs have not been discussed at all. Nevertheless, a few comments about scheduling bear directly on the simulation discussed in the next section.

A scheduling philosophy similar to No. 1 ESS is assumed. Both the input routine and the software timer routine are activated by periodic real-time clock signals. Each time these routines are activated, up to a certain number of calls to each call state routine are permitted. The thresholds are chosen according to how much processing can be done within the basic real-time clock period; they should never be reached except in periods of very heavy telephone traffic. The calls to the call state routines produce queues of requests in the QCAM. All requests are then processed. If the work is completed before the end of the basic clock period, administrative and maintenance programs are executed. (Actually, in a real system, the scheduling philosophy must assure that vital administrative and maintenance functions are not deferred indefinitely.)
7.4 Simulation Results

The nine call state routines, the three service subroutines, and an administrative and maintenance routine were coded for the executive control simulator. The execution times for the segments within these routines (except for the administrative and maintenance routines) were estimated from information obtained from private conversations with a telephone system programming expert. The information obtained was then transformed into equivalent ACFM CPU cycle times. Figures 7-5 through 7-14 give flowcharts of the thirteen tasks with execution times indicated.

Based upon these numbers, a call of the type described in Section 7.1.2 requires about 2220 microseconds of CPU time. The percentage of this total time required by each of the twelve call processing tasks was the basis for assigning tasks to processors in such a way that the real-time load was evenly distributed. The administrative and maintenance task was assigned to all processors. Its purpose was simply to absorb any idle processor time which would otherwise have resulted.

The effect of processing many telephone calls simultaneously was achieved by initializing the QCAM with a queue of requests to each call state routine. Seven requests were supplied to each call state routine except the stable state routine (State 8). It was given twenty-eight requests since a call receives, on the average, four stable state stimuli. (The total of 84 requests in the QCAM corresponded to portions of 84 different calls.) The simulations were run until all requests
FIGURE 7-5
ORIG CALL STATE ROUTINE (STATE 1)
ENDWS

LUMPED TIME

END

FIGURE 7-6
ENDWS CALL STATE ROUTINE (STATE 2)

DIGSTI

DIGIT ANALYSIS SUBROUTINE

TRANSLATION SUBROUTINE

LUMPED TIME

ONE WAY CALL TO TRANSLATION SUBROUTINE

END

FIGURE 7-7
DIGSTI CALL STATE ROUTINE (STATE 3)
DIGST2 CALL STATE ROUTINE (STATE 4)

DIGIT ANALYSIS SUBROUTINE 64 $\mu$SEC

NETWORK SUBROUTINE 67.5 $\mu$SEC

TRANSLATION SUBROUTINE 45 $\mu$SEC

TRANSLATION SUBROUTINE 32 $\mu$SEC

NETWORK SUBROUTINE 90 $\mu$SEC

LUMPED CYCLES 48 $\mu$SEC

END

FIGURE 7-8

BEGINWS CALL STATE ROUTINE (STATE 5)

BEGINWS

LUMPED TIME 40 $\mu$SEC

END

FIGURE 7-9
FIGURE 7-10
OUTPULSE CALL STATE ROUTINE (STATE 6)

FIGURE 7-11
IDLEOUTP CALL STATE ROUTINE (STATE 7)
FIGURE 7-12
STABLE CALL STATE ROUTINE (STATE 8)
FIGURE 7-13 DISC CALL STATE ROUTINE (STATE 9)
FIGURE 7-14 SERVICE SUBROUTINES

TIME SUPPLIED BY CALLING ROUTINE

TIME SUPPLIED BY CALLING ROUTINE

TIME SUPPLIED BY CALLING ROUTINE

TRANSLATION SUBROUTINE

NETWORK SUBROUTINE

DIGIT ANALYSIS SUBROUTINE
had been processed. The total runtime was then used to estimate how many calls could be handled per hour.

Simulations of 1, 2, 3, and 4 processor systems were run. The simulations for the 2, 3, and 4 processor cases were supplemented with GPSS simulations of the time-shared bus (see Section 5.4.2) to confirm that storage interference is minimal at the common data memory interface. The results are summarized in Tables 7-1 to 7-5.

The results of the single processor simulation are tabulated in Table 7-1. All tasks were serially reusable. The lock option was necessary for those call state routines which place two-way calls to the service subroutines. The total amount of processing accomplished is indicated by the task counts. More than the total work for seven calls was completed in 18.38 milliseconds. The extra work amounted to eight disconnects which were produced by stable state stimuli processed in the stable state routine, STABLE. Processor utilization was 96.35 percent. A conservative capacity estimate, based upon the fact that equivalent of seven calls were completed, is 1.37 million calls per hour.

The two-processor simulation is summarized in Table 7-2. The task assignment theoretically placed 51.5 percent or the real-time load on processor 1 and 48.5 percent on processor 2. Utilization was in excess of 95.5 percent for both processors. Again, more than the equivalent amount of the processing for seven calls was completed. Because there were periods of time when no other work was available, processor 1 executed the administrative and maintenance routine (AANDM) 37 times
Task Options: All tasks were serially reusable. The lock option was specified where needed.

Task Counts:

<table>
<thead>
<tr>
<th>Task</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORIG</td>
<td>7</td>
</tr>
<tr>
<td>ENDWS</td>
<td>7</td>
</tr>
<tr>
<td>DIGST1</td>
<td>7</td>
</tr>
<tr>
<td>DIGST2</td>
<td>7</td>
</tr>
<tr>
<td>BEGINWS</td>
<td>7</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>7</td>
</tr>
<tr>
<td>IDLEOUTP</td>
<td>7</td>
</tr>
<tr>
<td>STABLE</td>
<td>28</td>
</tr>
<tr>
<td>DISC</td>
<td>15</td>
</tr>
<tr>
<td>NETWORK</td>
<td>50</td>
</tr>
<tr>
<td>TRANSLATION</td>
<td>71</td>
</tr>
<tr>
<td>DIGIT ANALYSIS</td>
<td>14</td>
</tr>
<tr>
<td>ADMINISTRATIVE AND MAINTENANCE</td>
<td>0</td>
</tr>
</tbody>
</table>

Processor Utilization: 96.35%

Runtime: 18.38 Milliseconds

Average Time Per Call: 2.625 Milliseconds

Capacity: 1.37 Million Calls Per Hour

TABLE 7-1
SINGLE PROCESSOR SIMULATION RESULTS
Processor 1

Task Assignment and Task Control Options

- IDLEOUTP Q,L
- DIGST1 R
- TRANSLATION Q
- DIGITANL Q
- OUTPULSE
- BEGINWS Q
- AANDM D

Task Counts

- IDLEOUTP 7
- DIGST1 7
- TRANSLATION 71
- DIGITANL 14
- OUTPULSE 7
- BEGINWS 7
- AANDM 37

Real-Time Load: 51.5%
Processor Utilization: 96.8%
Runtime: 9.7 Milliseconds
Time Per Call: 1.385 Milliseconds
Capacity: 2.6 Million Calls Per Hour

Processor 2

- ORIG Q,L
- DIGST2 Q,L
- NETWORK Q
- DISC R
- STABLE
- ENDWS Q
- AANDM D

- ORIG 7
- DIGST2 7
- NETWORK 50
- DISC 15
- STABLE 28
- ENDWS 7
- AANDM 0

Real-Time Load: 48.5%
Processor Utilization: 95.56%

TABLE 7-2

TWO-PROCESSOR SIMULATION RESULTS
(925 microseconds total). The capacity estimate based upon seven calls is 2.6 million calls per hour.

Task control options in the 2, 3, and 4 processor simulations were chosen to optimize performance. Ideally, all tasks should be reentrant since, by having multiple instances of execution of each call state routine in progress, work is distributed to the service subroutines sooner. The service subroutines operate more efficiently when the queue option is specified and the queues are long. All tasks cannot be reentrant, however, since processor register space is limited.

Simulation results for the three and four processor systems are summarized in Tables 7-3 and 7-4, respectively. The capacities were 4.29 and 5.32 million calls per hour, respectively.

Table 7-5 summarizes the results of GPSS simulations of the time-shared bus using the simulator, TSB2. It was assumed that each request to common data memory is equally likely to be directed to any module. The number of storage modules for each simulation run was chosen to reflect the increase in storage capacity which is implied by a traffic increase. Since the minimum time for a common data memory read is 400 nanoseconds, the simulation results indicate that storage interference is not significant.

A graph of capacity versus the number of processors is shown in Figure 7-15. The capacity estimates were reduced according to the amount of storage interference which was predicted by the time-shared bus simulations. The curve shows that capacity increases almost linearly with the number of processors.
### Table 7-3

THREE-PROCESSOR SIMULATION RESULTS

<table>
<thead>
<tr>
<th>Task Assignment</th>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Processor 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IDLEOUTP R</td>
<td>DIGST1 R</td>
<td>ORIG R</td>
</tr>
<tr>
<td>And Task Control Options</td>
<td>TRANSLATION Q</td>
<td>NETWORK Q</td>
<td>DIGST2 R</td>
</tr>
<tr>
<td></td>
<td>STABLE</td>
<td>BEGINWS</td>
<td>DIGITALN L Q</td>
</tr>
<tr>
<td></td>
<td>AANDM D</td>
<td>DISC R</td>
<td>ENDWS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AANDM D</td>
<td>OUTPUTS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AANDM D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task Counts</th>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Processor 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IDLEOUTP 7</td>
<td>DIGST1 7</td>
<td>ORIG 7</td>
</tr>
<tr>
<td></td>
<td>TRANSLATION 65</td>
<td>NETWORK 44</td>
<td>DIGST2 7</td>
</tr>
<tr>
<td></td>
<td>STABLE 28</td>
<td>BEGINWS 7</td>
<td>DIGITALN 14</td>
</tr>
<tr>
<td></td>
<td>AANDM 25</td>
<td>DISC 9</td>
<td>ENDWS 7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AANDM 0</td>
<td>OUTPUTS 7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AANDM 16</td>
</tr>
</tbody>
</table>

- **Real-Time Load:** 33.54% 33.18% 33.25%
- **Processor Utilization:** 96.13% 96.61% 96.14%
- **Runtime:** 5.88 Milliseconds
- **Time Per Call:** 0.84 Milliseconds
- **Capacity:** 4.29 Million Calls Per Hour
<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Processor 3</th>
<th>Processor 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task Assignment</td>
<td>TRANSLATION Q</td>
<td>NETWORK Q</td>
<td>ORIG R</td>
</tr>
<tr>
<td>And Task Control</td>
<td>IDLEOUTP</td>
<td>DIGST2 R</td>
<td>DISC R</td>
</tr>
<tr>
<td>Options</td>
<td>AANDM D</td>
<td>BEGINWS</td>
<td>STABLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENDWS</td>
<td></td>
</tr>
<tr>
<td>AANDM D</td>
<td>AANDM D</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task Counts</th>
<th>TRANSLATION 66</th>
<th>NETWORK 46</th>
<th>ORIG 7</th>
<th>DIGITANL 14</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLEOUTP</td>
<td>7</td>
<td>DIGST2 7</td>
<td>DISC 9</td>
<td>OUTPULSE 7</td>
</tr>
<tr>
<td>AANDM</td>
<td>11</td>
<td>BEGINWS 7</td>
<td>STABLE 20</td>
<td>DIGST1 7</td>
</tr>
<tr>
<td>AANDM</td>
<td>15</td>
<td>ENDWS 7</td>
<td></td>
<td>AANDM 36</td>
</tr>
<tr>
<td>AANDM</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Real-Time Load: 26.489% 25.25% 25.02% 23.22%
- Processor Utilization: 96.19% 95.1% 95.65% 98.7%
- Runtime: 4.74 Milliseconds
- Time Per Call: .677 Milliseconds
- Capacity: 5.32 Million Calls Per Hour

**TABLE 7-4**
Four-Processor Simulation Results
<table>
<thead>
<tr>
<th></th>
<th>2 Processors</th>
<th>3 Processors</th>
<th>4 Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Modules:</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>Requests Per Microsecond to Time-Shared Bus:</td>
<td>1.43</td>
<td>2.14</td>
<td>2.86</td>
</tr>
<tr>
<td>Average Memory Read Time in Nanoseconds:</td>
<td>412.3</td>
<td>417.7</td>
<td>427</td>
</tr>
<tr>
<td>Percent Interference:</td>
<td>3.07</td>
<td>4.42</td>
<td>6.75</td>
</tr>
</tbody>
</table>

TABLE 7-5

SIMULATION OF STORAGE INTERFERENCE AT COMMON DATA MEMORY INTERFACE
Figure 7-15
CAPACITY IN CALLS PER HOUR VS Np

IDEAL LINEAR RELATIONSHIP

\[ y = \frac{4.07}{4.96} x + 1.37 \]
7.5 **Summary**

A simplified description of a call through a toll switching office was used as a model to estimate, by simulation, the call handling capacity of an ACFM system. The results indicate that about 5 million calls could be handled by a four-processor ACFM system. The actual numbers are, however, not so important as the fact that the simulations demonstrate the usefulness of a functional multiprocessor approach to telephone switching control. The results show that an almost linear growth curve is possible. This is an important result in view of the fact that growability has always been an elusive objective in telephone system design.

In a broader sense, the results of this chapter have demonstrated the capability of the ACFM for applications which achieve parallelism by sequencing many transactions through a fixed set of task programs. High utilization was achieved in spite of the fact that the average segment length was only about 55 microseconds.
CHAPTER VIII

CONCLUSIONS AND RECOMMENDATIONS
FOR FURTHER WORK

The ACFM and its control philosophy provide a generic multi-
processor structure which can be customized to the needs of a wide
variety of real-time applications. The major advantages of the system
are listed below.

1. Major system bottlenecks are eliminated by the
distributed structure. It minimizes the amount
of shared hardware and also simplifies the problem of interconnecting processors and shared
modules.

2. The design is modular. Except for the hardware
and software required for control, the design is
entirely independent of the types of modules that
are incorporated into the system. The system can
accommodate virtually any processor design, in-
cluding special purpose functional units.
3. The modular architecture is ideal for an LSI implementation. Most of the control hardware consists of several types of memory; viz., associative memory, high-speed semiconductor memory, and read-only control memory. Intermodular interconnections are regular.

4. The control philosophy allows control to be passed smoothly between tasks assigned to different processors with little overhead penalty. The concept of the QCAM is largely responsible for this. The QCAM makes it possible for a processor to post a call to a task and immediately search for other work since it alleviates any need for direct communication between processors in coordinating communication between tasks.

5. A number of task control options are provided which allow a considerable variety of software structures to be handled.

6. Programming does not require knowledge of either the number of processors or the manner in which tasks are assigned to processors. Programming the system is comparable to programming a time-shared uniprocessor.

7. The system can be expanded to cover a considerable capacity range.
This dissertation demonstrates the practicality of a functional approach to multiprocessing. The major contribution of the dissertation is the development and evaluation of an algorithm for controlling a functional multiprocessor. The two applications which were examined demonstrate the flexibility of the ACFM system and its control algorithm. The particular solution to a real-time object identification problem demonstrated the capability of the ACFM in handling parallel task structures. The telephone switching control application demonstrated the effectiveness of the system in sequencing many transactions through a set of independent task programs. In both applications, the time-shared bus met the data flow requirements between common data memory and processors without introducing any significant amount of storage interference.

Several areas are suggested for future research:

1) Investigate the applicability of the ACFM to general purpose processing. It should be possible to develop paging algorithms which would permit user tasks to be brought into the system and dynamically assigned to processors.

2) Develop algorithms for dynamically varying the assignment of tasks to processors in real-time applications so that adjustments to varying loading conditions can be easily effected.
3) Study the maintainability of the structure and develop procedures for fault detection, fault recognition and system reconfiguration.

4) Develop an appropriate queuing model for the system and study the use of various queue disciplines in the QCAM.

5) Actually build an ACFM system using several types of commercial minicomputers. Use this prototype as a vehicle to determine the difficulty of interfacing with a variety of minicomputer designs, and as a vehicle for studying the validity of the conclusions drawn from the present research.
APPENDIX A

EXECUTIVE CONTROL SIMULATOR

As described in Chapter V, the executive control simulator was designed to simulate the performance of the ACFM in executing various program structures. All operations involved in intertask communication are simulated at the instruction level. Time is handled in the simulation in such a way that interference at the interface with the associative memory subsystem is modeled. The simulator consists of a collection of procedures which implement the various associative memory operations, the five executive microprograms, and the various tasks whose execution is to be simulated. These procedures operate upon data structures which represent processors, the Executive CAM, and the QCAM. The PL/I programming language\(^{(45)}\) was chosen for the simulator primarily because of its capability for specifying data structures.

A structure in PL/I is a hierarchy of variables. An array of structures of dimension N holds the transient data pertaining to N processors. The array is dynamically allocated so that N can be specified at runtime. The Executive CAM and the QCAM are also represented by arrays of structures. In this case, the variables within a
single structure correspond to the fields within a single Executive CAM or QCAM word.

Since the simulator is run on a single processor system (the IBM 370 Model 155), the operation of only one ACFM processor can be simulated at a time. The various procedures which implement executive operations are time-shared among processors. These procedures fall into two groups:

**Associative Control Procedures** implement individual Executive CAM and QCAM operations.

**Executive Microprogram Procedures** implement the five executive microprograms. Each microprogram is subdivided into a number of procedures in such a way that each procedure implements a portion of the microprogram that is executed between associative control operations.

The mnemonics for the Associative Control Procedures are as follows:

- **SETTASK** - Executive CAM order **Set Enable**
- **RENABLE** - Executive CAM order **Reset Enable**
- **TASKSRCH** - Executive CAM order **Task Search**
- **INHSET** - Executive CAM order **Set Inhibit**
- **INHRES** - Executive CAM order **Reset Inhibit**
- **WIWORD** - QCAM order **Write Idle Word**
- **QTASKRD** - QCAM order **Task Read**
Figures A-1 through A-5 illustrate how each of the five executive microprograms are subdivided into Executive Microprogram Procedures.

Once it has selected a particular processor, the simulator continues to simulate the activity of that processor until the processor encounters an Executive CAM or a QCAM operation. When this happens, the simulator enters its main program loop. Before describing this loop, it is first necessary to describe how time is monitored during the simulation.

A single variable called EXECUTIME keeps track of time for the associative control subsystem. (Both the Executive CAM and the QCAM are assumed to be on the same bus; consequently, only one associative control operation can be performed at a time.) Each processor structure contains several variables which monitor time:

PROCTIME is the total elapsed time for the processor.

OVHDTIME is the total time spent in the executive microprograms exclusive of the time spent doing associative memory operations.

XCTIME and XQTIME represent the times spent doing Executive CAM and QCAM operations, respectively, including wait time when conflicts occur at the associate control interface.
<table>
<thead>
<tr>
<th>ENTASK</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>- SET UP EXECUTIVE CAM ACCESS REGISTER FOR SET ENABLE ORDER</td>
<td></td>
</tr>
<tr>
<td>- IF CALLED TASK IS A DIRECT TASK, CALL ENEXCAM</td>
<td></td>
</tr>
<tr>
<td>- ASSEMBLE THE QCAM WORD IN THE QCAM ACCESS REGISTER</td>
<td></td>
</tr>
<tr>
<td>- CAMINDX(X) = 3 (WIPROD)</td>
<td></td>
</tr>
<tr>
<td>- OVHDINDX(X) = 1 (ENEXCAM)</td>
<td></td>
</tr>
<tr>
<td>- GO TO MAINLOOP</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ENEXCAM</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>- CAMINDX(X) = 1 (SETTASK)</td>
<td></td>
</tr>
<tr>
<td>- OVHDINDX(X) = 2 (ENEND)</td>
<td></td>
</tr>
<tr>
<td>- GO TO MAINLOOP</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ENEND</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>- IF CALL IS TWO-WAY, RC = RC + 1</td>
<td></td>
</tr>
<tr>
<td>- IF CALL IS A ONE-WAY NONDISPATCHING CALL, CALL LW2</td>
<td></td>
</tr>
<tr>
<td>- IF CALL IS A TWO-WAY NONDISPATCHING CALL, CALL LW3</td>
<td></td>
</tr>
<tr>
<td>- IF CALL IS A DISPATCHING CALL, CALL TASKDISP</td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE A-1**

SEGMENTATION OF THE ENTASK MICROPROGRAM INTO THREE PROCEDURES
FIGURE A-2
SEGMENTATION OF THE LW MICROPROGRAM INTO EIGHT PROCEDURES
### QENTRY

- MOVE FIELDE AND FIELDA FROM QCAM ACCESS REGISTER INTO TASK REGISTER OF ACTIVE REGISTER GROUP
- IF THIS IS A TWO-WAY CALL, CALL CONTTASK
- \text{CAMINDEX}(X) = 5 \text{ (CLRWRD)}
- \text{OVHDINDEX}(X) = 12 \text{ (CONTTASK)}
- GO TO MAINLOOP

### CONTTASK

CALL TASKDISP

---

**Figure A-3**

SEGMENTATION OF QENTRY MICROPROGRAM INTO TWO PROCEDURES
**QRET**

- Using the RGI from FielDC of the QCAM Access Register, find the suspended register group and make it active
- Load FieldA of the QCAM Access Register into R1 of the active register group
- CAMINDX(X) = 5 (CLRWRD)
- OVDINDX(X) = 14 (QRETA)
- Go to Mainloop

<table>
<thead>
<tr>
<th>QRETA</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Set up return address to calling task</td>
</tr>
<tr>
<td>- Call TaskDisp</td>
</tr>
</tbody>
</table>

**FIGURE A-4**

Segmentation of QRET Microprogram into two procedures
FIGURE A-5
SINGLE PROCEDURE FOR WRET MICROPROGRAM

<table>
<thead>
<tr>
<th>WRET</th>
</tr>
</thead>
<tbody>
<tr>
<td>- SET UP QCAM ACCESS REGISTER FOR A RETURN WRITE</td>
</tr>
<tr>
<td>- CAMINDX(X) = 6 (REWRITE)</td>
</tr>
<tr>
<td>- OVHDINDX(X) = 10 (LW2)</td>
</tr>
<tr>
<td>- GO TO MAINLOOP</td>
</tr>
</tbody>
</table>
WORKTIME is the portion of the processor's time which is spent executing tasks. The difference between PROCTIME and WORKTIME is the total overhead time for the processor.

When the main loop is entered (see Figure A-6), that processor with minimum value in its PROCTIME field is selected for simulation. Suppose processor X is selected. Two variables in the data structure for processor X, CAMINDX(X) and OVHDINDX(X), were set up by the executive microprogram procedure that processor X was executing at the time it last encountered an associative control operation and entered the main loop. CAMINDX(X) specifies which associative control procedure is to be executed on behalf of processor X. This associative control procedure sets both EXECTIME and PROCTIME(X) to the value K given by:

\[ K = \text{MAX(EXECTIME,PROCTIME(X))} + \text{OPTIME} \]

where OPTIME is the time it takes to perform the associative operation. The updated time, K, takes into account interference at the associative memory interface. If EXECTIME is greater than PROCTIME(X) at the time processor X is selected, processor X has to wait (EXECTIME-PROCTIME(X)) time units for the associative control subsystem to become available. If EXECTIME is less than PROCTIME(X), then the associative control subsystem is available when processor X needs it. In this case, EXECTIME
- Select processor with minimum proctime
  - X is the index of the selected processor

Execute the associative control procedure identified by CAMINDX(X)

Execute the executive microprogram procedure identified by OVHDINDX(X)

Continue simulating the activity of processor X until an associative operation is encountered

When an associative operation is encountered:
  - Set CAMINDX(X) = index of associative control procedure
  - Set OVHDINDX(X) = index of executive microprogram procedure to which control is to be returned following the associative operation

FIGURE A-6
MAIN LOOP OF EXECUTIVE CONTROL SIMULATOR
must be updated by \((\text{PROCTIME}(X) - \text{EXECTIME})\) time units to bring it up to the time of the request by processor \(X\). The function \(\text{MAX}(\text{EXECTIME}, \text{PROCTIME}(X))\) covers both situations. The associative control procedure pointed to by \(\text{CAMINDX}(X)\) also updates either \(\text{XCTIME}(X)\) or \(\text{XQTIME}(X)\) depending upon whether the associative operation was an Executive CAM or a QCAM operation.

After the desired associative operation is performed for processor \(X\), control is passed to the executive microprogram procedure identified by \(\text{OVHINDX}(X)\). Each executive microprogram procedure updates \(\text{OVHDTIME}(X)\) and \(\text{PROCTIME}(X)\) by the time it takes to perform the executive operations. The activity of processor \(X\) is simulated until another associative operation is encountered. Once processor \(X\) reaches a point in the executive microprograms where it is ready to begin execution of a task program, control is passed to the appropriate Task Procedure.

Task Procedures implement the application programs being simulated. They interface with executive microprogram procedures. Figure A-7 illustrates the coding of a task procedure called TASKA.

When TASKA is selected for execution and a QCAM entry has been read, an executive microprogram procedure called CONTTASK (CONTTASK is part of the QENTRY microprogram) transfers control to TASKA via a call to a procedure called TASKDISP. The TASKDISP procedure has two arguments; the name of the task procedure being called, and an index when specifies at what entry point it is to be entered. When TASKA is first entered, the index is zero; consequently control is passed to location START. \(\text{PROCTIME}(X)\) and \(\text{WORKTIME}(X)\) are updated by TASKAT1
FIGURE A-7

CODING FOR A TASK PROCEDURE, TASKA

TASKA: PROCEDURE (INDEX);

DECLARE (INDEX,D1,D2,D3) FIXED BINARY:

IF INDEX=0 THEN GO TO START;
IF INDEX=1 THEN GO TO AD1;
IF INDEX=2 THEN GO TO AD2;

START:
PUT EDIT('TASKA ENTERED')(A);
PUT SKIP(2);
PROCTIME(X) = PROCTIME(X) + TASKAT1;
WORKTIME(X) = WORKTIME(X) + TASKAT1;
/* TASKAT1 REPRESENTS THE EXECUTION TIME FOR TASKA PRIOR TO ITS CALL TO TASKB*/
D1=1; /* SPECIFIES TWO-WAY CALL */
D2=0; /* SPECIFIES NON-DISPATCHING CALL */
D3=1; /* SPECIFIES RETURN ADDRESS AD1 */
CALL ENTASK ('TASKB',D1,D2,D3); /* CALL MACRO */
/* TASKA IS SUSPENDED FOLLOWING THE ABOVE CALL */

AD1:
D1=0; /* SPECIFIES ONE-WAY CALL */
D2=1; /* SPECIFIES DISPATCHING CALL */
D3=0; /* NO RETURN ADDRESS */
TASKINDX(X)=2; /* SPECIFIES THAT CONTROL IS TO BE RETURNED TO TASKA AT ADDRESS AD2 FOLLOWING THE DISPATCHING CALL */
CALL ENTASK ('TASKC', D1, D2, D3); /* CALL MACRO */

AD2:
PROCTIME(X) = PROCTIME(X) + TASKAT2;
WORKTIME(X) = WORKTIME(X) + TASKAT2;

/* TASKAT2 IS THE EXECUTION TIME FOR TASKA FOLLOWING
THE DISPATCHING CALL TO TASKC */

CALL LW2; /* END MACRO */

END TASKA;
time units to represent processing done by TASKA. A two-way call is then placed to TASKB. Parameters D1, D2, and D3 correspond to the α, β, and γ fields of a CALL macro. When the return to TASKA from TASKB is later picked up, the return address field is used to set up a call to TASKDISP with INDEX=1. This returns control to address AD1 in TASKA. Prior to the dispatching call to TASKC, a variable called TASKINDX(X) is set to 2. It is used as an index by TASKDISP to return control to address AD2 in TASKA following the dispatching call. A complete trace of the procedures involved in the execution of TASKA on processor X is given in Figure A-8. The vertical dashes indicate periods when the simulator may be simulating the activity of some other processor. Refer to Figures A-1 through A-5 for the functions of the various executive microprogram procedures.

A flowchart of the entire simulator is shown in Figure A-9. When the simulator is started, the number of processors, N, is inputted and the array of processor structures is allocated. Other input parameters are read in including the processor names, the circuit delays for various operations, the execution times for tasks being simulated, the Executive CAM words for the tasks being simulated, and QCAM words for any requests which are assumed to already exist. Initialization consists of zeroing certain variables and setting the CAMINDX of every processor to point to the TASKSRCH associative control procedure. The main loop, which has already been described, is then executed until either the simulated program structure is completed or a COUNT (which is supplied as an input parameter) is exceeded. The results are then
TASKSRCH------EXECUTIVE CAM ORDER (SELECT TASKA)
LWC
LWB
QTASKRD------QCAM ORDER (READ QCAM WORD FOR TASKA)
LWD
REENABLE------EXECUTIVE CAM ORDER (RESET ENABLE OF TASKA)
QENTRY
CLRWRD------QCAM ORDER (CLEAR QCAM WORD FOR TASKA)
CONT TASK
TASKDISP
TASKA--------ENTRY AT ADDRESS START
ENTASK--------CALL TO TASKB PLACED
WIWORD------QCAM ORDER (WRITE QCAM WORD FOR TASKB)
ENEXCAM
SETTASK------EXECUTIVE CAM ORDER (ENABLE TASKB)
ENEND
LW3---------SUSPEND TASKA
LW4
TASKSRCH------EXECUTIVE CAM ORDER (SEARCH FOR OTHER WORK)
OTHER WORK
LW1
RET SRCH------QCAM ORDER (RETURN FROM TASKB PICKED UP)
LWA
QRET
CLRWRD------QCAM ORDER (ERASE QCAM WORD USED FOR RETURN)
QRETA
TASKDISP
TASKA--------ENTRY AT ADDRESS AD1
ENTASK--------DISPATCHING CALL TO TASKC
WIWORD------QCAM ORDER (WRITE QCAM WORD FOR TASKC)
ENEXCAM
SETTASK------EXECUTIVE CAM ORDER (ENABLE TASKC)
ENEND
TASKDISP
TASKA--------ENTRY AT ADDRESS AD2
LW2---------END TASKA
LW1---------BEGIN SEARCH FOR MORE WORK

FIGURE A-8
ACTIVITY OF PROCESSOR X DURING SIMULATION OF TASKA
BEGIN

INPUT AND STORAGE ALLOCATION

INITIALIZATION

COUNT EXCEEDED?

YES

OUTPUT RESULTS

END

NO

SELECT PROCESSOR WITH MINIMUM PROCOTIME

EXECUTE ASSOCIATIVE OPERATION IDENTIFIED BY CAMINDX(X)

EXECUTE EXECUTIVE MICROPROGRAM PROCEDURE IDENTIFIED BY OVHDINDX(X)

CONTINUE SIMULATING PROCESSOR X UNTIL NEXT ASSOCIATIVE OPERATION

END OF SIMULATED WORK

FIGURE A-9

COMPLETE FLOCHART OF EXECUTIVE CONTROL SIMULATOR
The simulator output consists of:

<table>
<thead>
<tr>
<th><strong>PROCTIME</strong></th>
<th>The total execution time for each processor.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OVHDTIME</strong></td>
<td>The time spent by each processor executing executive microprograms.</td>
</tr>
<tr>
<td><strong>XCTIME</strong></td>
<td>The time spent by each processor doing Executive CAM operations including wait times due to interference.</td>
</tr>
<tr>
<td><strong>XQTIME</strong></td>
<td>The time spent by each processor doing QCAM operations including wait times due to interference.</td>
</tr>
<tr>
<td><strong>WORKTIME</strong></td>
<td>The time spent by each processor executing task programs.</td>
</tr>
<tr>
<td><strong>XCCOUNT</strong></td>
<td>The number of Executive CAM operations performed by each processor.</td>
</tr>
<tr>
<td><strong>XQCOUNT</strong></td>
<td>The number of QCAM operations performed by each processor.</td>
</tr>
<tr>
<td><strong>IDLECNT</strong></td>
<td>The number of times each processor idles because it has no work available.</td>
</tr>
<tr>
<td><strong>UTILIZATION</strong></td>
<td>The percentage of time spent doing useful work for each processor ( \left( \frac{\text{WORKTIME}}{\text{PROCTIME}} \right) \times 100 ).</td>
</tr>
<tr>
<td><strong>Percent Executive CAM Time</strong></td>
<td>The percentage of time which each processor spends doing Executive CAM operations ( \left( \frac{\text{XCTIME}}{\text{PROCTIME}} \right) \times 100 ).</td>
</tr>
<tr>
<td><strong>Percent QCAM Time</strong></td>
<td>The percentage of time spent by each processor doing QCAM operations ((XQTIME/PROCTIME) \times 100).</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Percent Executive</strong></td>
<td>The percentage of time spent by each processor executing the executive microprograms (((OVHDTIME/PROCTIME) \times 100)).</td>
</tr>
<tr>
<td><strong>Microprogram</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Execution Time</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Total Overhead</strong></td>
<td>The total percentage of overhead time for each processor ((100 - UTILIZATION)).</td>
</tr>
<tr>
<td><strong>RUNTIME</strong></td>
<td>The amount of ACFM time that has been simulated.</td>
</tr>
</tbody>
</table>

The most useful output data are the utilizations and the runtime. When all processors are supplied with enough work so that they do not have idle time, \((100 - UTILIZATION)\) is a measure of how much overhead is required for intertask communication. In simulations of programs which have internal parallel branches, processor utilization is relatively unimportant. In this case, **RUNTIME** is the important performance indicator. **RUNTIME** is taken as the maximum of **EXECUTE TIME** and **PROCTIME** for all processors.
The purpose of this appendix is to justify the assumptions that were made concerning what circuit delays to use in the simulations. The times chosen for executive microprogram operations reflect the capability of current semiconductor memory and logic technology. The times chosen for associative memory operations are somewhat conservative compared to the anticipated performance forecast by reference 16.

The times for performing operations within the five executive microprograms are based upon a basic microinstruction cycle time of 50 nanoseconds and a register transfer time of 25 nanoseconds. These numbers were obtained through private communication with D. M. Rouse (32) who is building an experimental high-speed processor. The simulated times for performing operations within executive microprogram procedures of the executive control simulator are listed below.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Operation</th>
<th>Time (nanoseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENTASK</td>
<td>One-way call</td>
<td>175</td>
</tr>
<tr>
<td></td>
<td>Two-way call</td>
<td>300</td>
</tr>
<tr>
<td>LW1</td>
<td></td>
<td>100</td>
</tr>
</tbody>
</table>
The time assumed for setting up a FORK count is 100 nanoseconds. The time assumed for decrementing and testing the count via a JOIN macro is 150 nanoseconds.

In the simulations, associative write operations were assumed to take 300 nanoseconds and associative read operations were assumed to take 350 nanoseconds. These numbers are conservative compared to those listed in Figure B-1. In Figure B-1, the time for performing the basic operations of read, write, or equality search is assumed to be 40 nanoseconds for the Executive CAM and 50 nanoseconds for the QCAM. The difference takes into account the fact that the QCAM is a larger memory. These assumptions are consistent with reference 16. The time for performing a maximum or minimum priority search in the Executive CAM is assumed to be 50 nanoseconds. The bus transmission time is assumed to be 75 nanoseconds; and, decoding is assumed to take 25 nanoseconds.
The more conservative set of numbers assumed for associative operations in the simulations allows for unknown delays which may have been overlooked.
### FIGURE B-1

**ANTICIPATED TIMES FOR ASSOCIATIVE CONTROL OPERATIONS**

<table>
<thead>
<tr>
<th>Priority Task Search</th>
<th>Return Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Bus</td>
<td>Input Bus</td>
</tr>
<tr>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>Decode</td>
<td>Decode</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Equality Search and Set Management Bits</td>
<td>Equality Search</td>
</tr>
<tr>
<td>40</td>
<td>50</td>
</tr>
<tr>
<td>Maximum Priority Search</td>
<td>Read</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Read Task Word</td>
<td>Output Bus</td>
</tr>
<tr>
<td>40</td>
<td>75</td>
</tr>
<tr>
<td>Reset C Bit</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
</tr>
<tr>
<td>Output Bus + Reset Management Bits</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>345 nsec</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set Enable, Reset Enable, Set Inhibit, or Reset Inhibit</th>
<th>QCAM Task Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Bus</td>
<td>Input Bus</td>
</tr>
<tr>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>Decode</td>
<td>Decode</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Equality Search</td>
<td>Equality Search</td>
</tr>
<tr>
<td>40</td>
<td>50</td>
</tr>
<tr>
<td>Write</td>
<td>Read</td>
</tr>
<tr>
<td>40</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td><strong>180 nsec</strong></td>
</tr>
<tr>
<td></td>
<td>Output Bus + Set K Bit</td>
</tr>
<tr>
<td></td>
<td><strong>275 nsec</strong></td>
</tr>
</tbody>
</table>
**FIGURE B-1 - Continued**

<table>
<thead>
<tr>
<th>Process</th>
<th>Time (nsec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Bus</td>
<td>75</td>
</tr>
<tr>
<td>Decode</td>
<td>25</td>
</tr>
<tr>
<td>Search</td>
<td>50</td>
</tr>
<tr>
<td>Write</td>
<td>50</td>
</tr>
</tbody>
</table>

**Write Idle Word, Clear Word, or Return Write**

200 nsec
REFERENCES

1. Riley, W. B.: "Minicomputer Networks - A Challenge to Maxi-


3. Martin, James: *Programming Real-Time Computer Systems*, Prentice-

FJCC*, 1963, pp. 139-147.

5. Wegner, Peter: *Programming Languages Information Structures and

6. Bell, C. G., and A. Newell: *Computer Structures: Readings and

7. Katzan, Harry, Jr.: *Computer Organization and the System/370,

8. Thornton, J. E.: *Design of a Computer-Control Data 6600*, Scott
Foresman, and Company, 1970.


42. Orcutt, S. E.: Internal Bell Telephone Laboratories Document.
44. Saad, M. W.: Private Communication.