INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. 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 material submitted.

The following explanation of techniques is provided to help you understand markings or notations 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 through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame.

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

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.
WANG, PONG-SHENG

COMPUTER ARCHITECTURE FOR PARALLEL EXECUTION OF HIGH-LEVEL LANGUAGE PROGRAMS

The Ohio State University

University Microfilms International 300 N. Zeeb Road, Ann Arbor, MI 48106

Copyright 1980 by Wang, Pong-Sheng

All Rights Reserved
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark ✓.

1. Glossy photographs ❍
2. Colored illustrations ❍
3. Photographs with dark background ❍
4. Illustrations are poor copy ❍
5. Print shows through as there is text on both sides of page ✓
6. Indistinct, broken or small print on several pages ✓
7. Tightly bound copy with print lost in spine ✓
8. Computer printout pages with indistinct print ❍
9. Page(s) lacking when material received, and not available from school or author
10. Page(s) seem to be missing in numbering only as text follows
11. Poor carbon copy ✓
12. Not original copy, several pages with blurred type ✓
13. Appendix pages are poor copy ✓
14. Original copy with light type ✓
15. Curling and wrinkled pages ✓
16. Other
COMPUTER ARCHITECTURE FOR PARALLEL EXECUTION
OF HIGH-LEVEL LANGUAGE PROGRAMS

DISSERTATION

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

By

Pong-sheng Wang, B.S., M.S.

* * * * *

The Ohio State University
1980

Reading Committee: Approved By
Dr. Ming-Tsan Liu, Chairman
Dr. Kenneth J. Breeding
Dr. Sandra A. Mamrak

Advisor
Department of Computer and
Information Science
ACKNOWLEDGEMENTS

I wish to express my most sincere gratitude to my adviser, Dr. Ming-Tsan Liu, for his guidance in my graduate program and his encouragement and useful suggestions during the course of this research.

I also wish to thank the members of my reading committee for their efforts in making this dissertation more readable.

Thanks are also due to the Department of Computer and Information Science for their financial support as a graduate administrative associate, and to Instruction and Research Computer Center for their financial support as a graduate research associate during my stay at The Ohio State University.

Finally, I would like to thank my loving wife, Ai-chu, for her never ending support and understanding throughout my entire period of graduate study.
VITA

September 13, 1950  Born - Taipei, Taiwan, Republic of China

1972  B.S.E.E., National Taiwan University, Taipei, Taiwan

1974-1976  Associate Instructor, Department of Computer Science, Indiana University, Bloomington, Indiana

1976  M.S., Indiana University, Bloomington, Indiana

1976-1978  Graduate Teaching Associate, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio

1976-1980  Graduate Research Associate, Instruction and Research Computer Center, The Ohio State University, Columbus, Ohio

PUBLICATIONS


FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in Computer Architecture and Organization.
Professor Ming-Tsan Liu

Studies in Computer and Systems Programming.
Professor Sandra A. Mamrak

Studies in Theory and Processing of Programming Languages. Professor Stuart Zweben
TABLE OF CONTENTS

ACKNOWLEDGEMENTS................................. ii
VITA.................................................... iii
LIST OF TABLES.................................... ix
LIST OF FIGURES.................................... x

Chapter

I. INTRODUCTION
   1.1 Background and Motivation.................... 1
   1.2 Scope of Our Work............................. 9
   1.3 Main Results and Contribution............... 11
   1.4 Organization of the Dissertation............ 14

II. SURVEY OF RELATED RESEARCH
   2.1 Recognizing Parallelism in Ordinary Programs............................... 16
   2.2 High-Level Language Computer
       Architecture..................................... 22
### III. PARALLEL EXECUTION STRINGS

3.1 Introduction ........................................ 36  
3.2 Parallel Processing of Expressions .............. 37  
3.3 Parallel Processing of Statements ............... 59  
3.4 Representing Concurrent Statements ............. 66  
3.5 Try-Ahead Processing ............................. 68  
  3.5.1 Try-ahead Processing of IF Statements........ 69  
  3.5.2 Try-ahead Processing of WHILE Statements..... 72  
  3.5.3 Try-ahead Processing of REPEAT Statements... 73  
  3.5.4 Try-ahead Processing of LOOP Statements...... 75  
3.6 Compiling Algorithms ............................. 78  
3.7 Machine Organization ............................. 97  
3.8 Code Generation ................................... 109  
3.9 Code Optimization ................................. 111

### IV. THE ARCHITECTURE OF A PARALLEL EXECUTION HIGH-LEVEL LANGUAGE COMPUTER

4.1 System Overview ................................. 120  
4.2 PES Access Processors ........................... 123  
4.3 Scanner Processor ................................. 125  
4.4 Entry Point Registers ............................ 127
4.5 Syntactic and Semantic Processors........ 130
4.6 Arithmetic Processors...................... 138
4.7 Partial Result Storage...................... 141

V. DESIGN OF A MULTI-MICROPROCESSOR SYSTEM FOR PARALLEL COMPUTATIONS
5.1 System Overview.......................... 143
5.2 Central Sequence Control Unit.............. 148
5.3 Central Interrupt Control Unit.............. 152
5.4 Microprocessor Arrays..................... 154
5.5 Microprogrammed Control Units.............. 158
5.6 Memory System............................. 163

VI. THE SCHEDULING PROBLEM
6.1 The PES Scheduling Problem............... 166
6.2 Comparing Several PES Scheduling Algorithms............................... 167
   6.2.1 Tree Generation....................... 169
   6.2.2 Original-Ordering Schedule.......... 172
   6.2.3 Longest-Processing-Time-First Schedule............................... 173
   6.2.4 Shortest-Processing-Time-First Schedule............................... 174
   6.2.5 Highest-Level-Number-First Schedule............................... 175
   6.2.6 Lowest-Level-Number-First Schedule............................... 176
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Simulation Results of PES scheduling Algorithms for a 5-Processor System</td>
<td>180</td>
</tr>
<tr>
<td>2</td>
<td>Simulation Results of PES scheduling Algorithms for a 4-Processor System</td>
<td>181</td>
</tr>
<tr>
<td>3</td>
<td>Simulation Results of PES scheduling Algorithms for a 3-Processor System</td>
<td>182</td>
</tr>
<tr>
<td>4</td>
<td>Simulation Results of PES scheduling Algorithms for a 2-Processor System</td>
<td>183</td>
</tr>
<tr>
<td>5</td>
<td>Simulation Results of Assembly Line Scheduling Algorithms for a 5-Processor System</td>
<td>197</td>
</tr>
<tr>
<td>6</td>
<td>Simulation Results of Assembly Line Scheduling Algorithms for a 4-Processor System</td>
<td>198</td>
</tr>
<tr>
<td>7</td>
<td>Simulation Results of Assembly Line Scheduling Algorithms for a 3-Processor System</td>
<td>199</td>
</tr>
<tr>
<td>8</td>
<td>Simulation Results of Assembly Line Scheduling Algorithms for a 2-Processor System</td>
<td>200</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Hellerman's Machine Organization</td>
<td>20</td>
</tr>
<tr>
<td>2.2</td>
<td>Bashkow's FORTRAN Machine</td>
<td>24</td>
</tr>
<tr>
<td>2.3</td>
<td>SYMBOL Computer System</td>
<td>27</td>
</tr>
<tr>
<td>2.4</td>
<td>Litton HOL Computer</td>
<td>29</td>
</tr>
<tr>
<td>2.5</td>
<td>Haynes' ALGOL 60 Computer</td>
<td>34</td>
</tr>
<tr>
<td>3.1</td>
<td>Parallel-by-level Approach to executing an Expression in Parallel</td>
<td>39</td>
</tr>
<tr>
<td>3.2</td>
<td>Example of Translating an Expression into PES...</td>
<td>44</td>
</tr>
<tr>
<td>3.3</td>
<td>Example of Translating an Expression into the Internal Language</td>
<td>54</td>
</tr>
<tr>
<td>3.4</td>
<td>Example for Combining Smaller Blocks into Larger Blocks</td>
<td>65</td>
</tr>
<tr>
<td>3.5</td>
<td>Flowchart of a Concurrent Statement</td>
<td>67</td>
</tr>
<tr>
<td>3.6</td>
<td>Flowchart of Algorithm A</td>
<td>84</td>
</tr>
<tr>
<td>3.7</td>
<td>Flowchart of Algorithm B</td>
<td>89</td>
</tr>
<tr>
<td>3.8</td>
<td>Flowchart of the Main Procedure of Algorithm C</td>
<td>92</td>
</tr>
<tr>
<td>3.9</td>
<td>Flowchart of Procedure POP-OPR-STK</td>
<td>93</td>
</tr>
<tr>
<td>3.10</td>
<td>Machine Organization for PES Approach</td>
<td>99</td>
</tr>
</tbody>
</table>
3.11 The State of the Machine before an Execution Stage is Started................................. 105
3.12 The state of the Machine during the execution of a Stage.................................. 106
3.13 The State of The machine When a stage is Just Finished................................. 107
3.14 The State of the Machine When a New Stage is Started................................. 108
3.15 An Example showing the Effects of Reordering PES's.................................... 116
3.16 An Example showing the Longest-Processing-Time-First Schedule is not Optimal........ 118
4.1 System Block Diagram of the High-Level Language Computer................................. 121
4.2 Scanner Processor................................................. 126
4.3 Entry Point Registers................................. 129
4.4 Arithmetic Processors............................................ 140
5.1 Block Diagram of the Multi-microprocessor System 146
5.2 Central Sequence Control Unit................. 149
5.3 Central Interrupt Control Unit..................... 153
5.4 Am2901 Bit-Slice Microprocessor............. 156
5.5 Microprogrammed Control Unit..................... 159
5.6 Alignment Network................................. 165
6.1 Simulation Results of PES Scheduling for a 5-Processor System................................. 184
6.2 Simulation Results of PES Scheduling for a 4-Processor System................................. 185
6.3 Simulation Results of PES Scheduling for a 3-Processor System................................. 186
6.4 Simulation Results of PES Scheduling for a 2-Processor System... 187
6.5 Simulation Results of Assembly Line Scheduling for a 5-Processor System... 201
6.6 Simulation Results of Assembly Line Scheduling for a 4-Processor System... 202
6.7 Simulation Results of Assembly Line Scheduling for a 3-Processor System... 203
6.8 Simulation Results of Assembly Line Scheduling for a 2-Processor System... 204
1.1 Background and Motivation

Among many computer design goals, there are two very common ones: high-speed execution and a good user interface. Parallel processing is one of the many ways to achieve the former, and high-level language computer architecture achieves the latter. The purpose of this dissertation is to investigate some aspects of parallel processing and to bring them together with a high-level language computer architecture. In this section, we will discuss some background and motivation for parallel processing and for high-level language computer architecture.
1.1.1 Parallel Processing

As computer applications increase, there are many problems that require more computation power than can be satisfied by contemporary computers. Such problems can be found in various applications, such as space exploration and weather forecasting. However, the rapid advances in technology have brought us into an era in which speed of computation is limited by factors other than the speed of its components. In order to increase computation speed to satisfy the high computation rate requirement of these applications, the trend has been toward computer systems capable of a high degree of parallelism.

Parallel processing can increase the speed of computation beyond the limit imposed by technology. It can reduce the turnaround time of jobs in batch systems and improve the response time in time-sharing and real-time systems. As a result, the system throughput is improved. Another advantage of parallel processing systems is their potential for reliability and their intrinsic modularity. These systems may operate as pools of resources organized in symmetrical classes and thus provide high availability.
Recent advances in LSI technology have given fresh impetus to research in the area of parallel processing. The once revolutionary concept of interconnecting a large number of processing elements operating in parallel has become much more feasible. In such systems, adding additional processing elements will add only a relatively small incremental cost to the total system. Because of its intrinsic modularity, the only effect of such expansion on the user is improved performance. Since the advances in LSI technology during the past twenty years have provided an exponential increase in logic functions per unit cost, the computer design philosophy is no longer emphasizing the utilization of hardware. Instead, considerations are given to designing systems that have enough processing elements to satisfy the needs of computations.

The importance of parallel processing has led to a great deal of research in this area, ranging from the parallelism at microinstruction level to that at process level. This dissertation is concerned with the parallelism among operations in expressions, the parallelism within a control statement, and the parallelism among a block of statements in high-level language programs.
To exploit the parallelism in algorithms, three approaches can be used. First, language constructs can be provided to allow programmers to specify the parallelism explicitly. For example, Conway [CONW63] has proposed the concepts of FORK, JOIN, and QUIT; the concurrent statement in Concurrent PASCAL allows programmers to specify parallel executable tasks [HANS73].

The second approach is to perform program analysis during the compilation process to automatically recognize parallelism in sequentially organized programs. The resulting object programs will have the instructions showing the parallelism that has been detected [KUCK76].

The third approach is to automatically detect the parallelism among machine instructions by hardware during execution time. For example, the CDC 6600 [THOR64] has several independent arithmetic units that are capable of parallel operations. Machine language instructions are checked for dependency at run time and executed in parallel when possible.

The first approach is referred to as the explicit approach, whereas the second and the third approaches are referred to as implicit approaches [RAMA69]. The explicit approach has the advantage that it allows programmers to
specify the parallelism in the algorithms, which may not be possible for automatic detection during either compile time or execution time. Two disadvantages are that it puts an additional burden on programmers, and errors may occur due to incorrect specification of parallelism by programmers.

The implicit approach has an advantage that it detects parallelism automatically, thereby resulting in a higher performance system without the need for additional effort by programmers. However, the implicit approach also has a disadvantage, depending on whether the compile time approach or the execution time approach is used for automatic detection of parallelism.

The disadvantage of detection at the compile time is that it may not detect all the possible parallelism in a program, since some parallelism can only be detected at run time. The disadvantage of hardware detection at the run time is that the parallelism recognized is limited by the degree of look-ahead built in hardware, and the complexity of hardware could become too great if a high degree of look-ahead is attempted.

The three different approaches to exploiting parallelism are not mutually exclusive. All of them, or any combination of them, can be used in a single system.
However, in this dissertation, we will only look into the implicit approach which automatically recognizes parallelism in programs during the compilation time.

Kuck et al. [KUCK74] have analyzed a set of 86 FORTRAN decks totalling over 4000 cards to determine the parallelism in the programs. In the program analysis process, tree-height reduction techniques are used for expressions, and forward substitution is used for a block of assignment statements. The parallelism being found also includes that in DO loop blocks. The results show that we can expect speedups on the order of 10, using an average of 37 processors with an average efficiency of 35%. This result can lead to the conclusion that multioperation machines could be quite effective in most ordinary FORTRAN computations.

It is obvious that not all programs should be submitted for parallel processing, since program analysis for parallelism could take a significant amount of time compared to the time saved by parallel execution. This is very similar to the situation for code optimization in a compiler: time will be saved only if the optimization takes less time than the time it will save at execution. Therefore, in practice, the user of such a system should be allowed to specify whether the jobs are submitted for
parallel processing or not, as in many compilers programmers can enable or disable the optimization option.

1.1.2 High-Level Language Computer Architecture

In a conventional von Neumann computer, high-level language programs are translated into machine language programs by software compilers, and then the machine language programs are interpreted by machine hardware. There exists a big semantic gap between high-level languages and low-level machine languages. This gap results in many difficulties in user-system interaction, such as awkward run-time error messages and improper run-time debugging aids. To solve these problems, the current trend is to design computer architectures that have high-level machine languages so as to narrow, or even eliminate, the semantic gap. Rapid advances in hardware technology and to recent developments in microprogramming techniques make this approach technically and economically feasible.

Chu [CHU75] has defined a high-level language computer as one that accepts and executes a high-level language program. He has classified high-level language computers into four types:

**Type 1**: von Neumann Architecture
This is the conventional architecture of commercially available computer systems which have a set of machine instructions and compilers that accept and translate high-level language programs into machine language. The machine hardware then interprets the machine instructions.

**Type 2: Syntax-oriented Architecture**

In this type of architecture, the machine instructions are called "syllables." The high-level language programs are translated into reverse Polish strings of syllables by a software translator, and the Polish string programs are then interpreted by the hardware. Procedure library is written in the reverse Polish string language and is brought into the system on demand during the program execution. Burroughs B5500 is of this type.

**Type 3: Indirect Execution Architecture**

In this type of architecture, high-level language programs are translated into reverse Polish strings by a hardware translator, and then the Polish string programs are interpreted by the machine hardware. Burroughs B1700 is of this type.

**Type 4: Direct Execution Architecture**
In this type of architecture, there is no translator at all. The hardware scans the high-level language source and does both the syntactic analysis and semantic processing. The high-level language is the machine language.

Much work has been done in this area. (See Chapter 2 for a survey.) However, the research in high-level language computer architecture is still far from well developed. In particular, parallel processing is rarely discussed in high-level language computer architecture, except by Mooney [MOON77]. This has motivated us to investigate in this dissertation the possibility of incorporating parallel processing as well as try-ahead processing into high-level language computer architectures.

1.2 Scope of Our Work

Much research has been done in the area of parallel processing at various levels in computing. Some results are very practical and have been used in real computer system implementation, whereas others only have theoretical interest. The objective of this dissertation is to propose a parallel processing scheme that is feasible for actual implementation in microprocessor-based systems as well as
high-level language computer systems.

The scope of our work can be broken down as follows:

First, we investigate the implicit approach to recognizing parallelism in programs during the compile time as discussed in Section 1.1.1. This will include two parts: a new scheme called the Parallel Execution String (PES), to recognize parallel processable tasks within a statement and among a block of statements, and the representation of tasks for execution on multiprocessor systems.

Second, we apply the above scheme to parallel processing of high-level language computer architecture. The architecture of such systems will be shown to demonstrate how the scheme can be incorporated into high-level language computer systems. The architecture is based on Haynes' design [HAYN77]. The organization of processors will be described only if they are different from Haynes' designs [HAYN77].

Third, we design a multi-microprocessor system that can perform parallel processing of the object programs with the proposed scheme. This will show the complexity added to a microprocessor-based system when the scheme is adopted for parallel processing.
Finally, we use simulation to compare the performance of several PES scheduling algorithms. The objective of the simulation is to find a simple but still effective scheduling algorithm that will yield near-optimal schedules.

1.3 Main Results and Contribution

A new approach, called Parallel Execution String (PES), is proposed to recognize parallelism in ordinary programs and to represent them in multiprocessor systems for parallel processing. The PES approach can reduce the time before an operation can be started (as the level-by-level approach may have), has minimal intermediate store and fetch of partial results, can minimize the intervention of central control to individual processors, and uses no stacks. The PES approach also has the advantage that it can be used to detect the parallelism among the statements in a block.

We propose the concept of try-ahead processing of control statements, which include: IF, REPEAT, WHILE, and LOOP statements. The representation of these statements for try-ahead processing is presented. The time saving by try-ahead processing could be significant, since the time saving in each iteration of a repetitive statement will be multiplied by the number of iterations.
It should be noted that representing statements in PES's and representing control statements for try-ahead processing can be done in the same program. For example, a block of statements could be part of a WHILE statement. Each statement in the block can be represented in PES, and the scheme for detecting parallelism across statements can be applied to the block of statements. The WHILE statement will have a boolean expression which can also be represented in PES. After this is done, the WHILE statement itself can then be represented for try-ahead processing. Next, looking at a higher level, the WHILE statement itself may possibly be one of the statements in a block, and the scheme for detecting parallelism across statements can be again applied to this outer block.

Based on Haynes' design [HAYN77], we propose the architecture of a parallel execution high-level language computer. In the proposed architecture, the PES approach to parallel processing and the try-ahead processing approach are used. The processor organization to facilitate parallel processing and try-ahead processing is discussed. Independent processors are used for various tasks in program execution. All the processors in the system are operating simultaneously in a pipelined manner to achieve the maximum performance possible. The pipeline effect, parallel
processing, and try-ahead processing result in a system whose performance is much better than that of its individual processors.

The PES approach can also be implemented effectively and efficiently with moderate effort in a low-cost system. This is shown by the design of a multi-microprocessor system using Am2900 based bit-slice microprocessors. The detail design of the central sequence control and the interrupt control are presented.

Simulation is done to compare the average performance of several PES scheduling algorithms. It is found that the "Longest Processing Time" scheduling has the best average performance among the algorithms being tested. Compared to a schedule without any reordering, the longest processing time schedule results in an average improvement of about 10% in completion time.

The most important contribution of this dissertation is that we not only have proposed the new scheme, PES, for arithmetic statements and a block of statements, and the try-ahead processing scheme for control statements, but also have brought them together with a high-level language computer architecture. Very little has been done in this area of parallel processing to date.
1.4 Organization of the Dissertation

Chapter 2 gives a brief survey of the previous work on parallel processing and on high-level language computer architecture. The work on parallel processing discussed in Chapter 2 deals with recognizing parallelism in ordinary high-level language programs.

In Chapter 3, we propose a scheme, called "Parallel Execution String" (PES), to detect the parallelism in high-level language programs at the statement and the block levels. The algorithms for translating assignment statements into PES's and scheduling a block of PES's for parallel execution are presented. The representation of IF, REPEAT, WHILE, and LOOP statements for try-ahead processing is discussed. Code generation and optimization techniques are also discussed. A machine organization for executing the programs compiled with the PES approach is proposed.

Chapter 4 describes the architecture of a parallel execution high-level language computer which has multiple independent processors for performing various language processing tasks simultaneously and multiple identical processors for parallel computations.
Chapter 5 describes the design of a multiple microprocessor system based on the PES approach. The design of a central interrupt control unit is presented. The control sequence of each microprocessor and the central control unit is also described.

Chapter 6 is concerned with the problem of scheduling PES for execution. The simulation model and the results of several PES scheduling algorithms are presented. A more general scheduling problem, the assembly line scheduling problem, is also discussed. Then the simulation results of several assembly line scheduling algorithms are also presented.

Chapter 7 gives a summary of the overall work and some suggestions for further research.

Appendixes A and B give the simulation programs and example outputs of simulating several PES scheduling algorithms and several assembly line scheduling algorithms, respectively. Appendix C gives a list of BNF descriptions of the internal language of the parallel execution high-level language computer. Appendix D lists the PASCAL program that translates an expression into the PES internal language.
CHAPTER 2

SURVEY OF RELATED WORK

In this chapter, we will first survey the research that has been done on recognizing parallelism in ordinary high-level language programs. Next we will survey the work in high-level language computer architecture.

2.1 Recognizing Parallelism in Ordinary Programs

To exploit parallelism within a statement, several methods have been proposed. A statement can be represented by a tree, in which the internal nodes are operators and the external nodes are variables or constants. Assuming that there are as many processors as needed, the execution time of a statement will be equal to the height of the
corresponding tree, provided each operator in the statement takes one unit of time. The tree-height reduction techniques use the commutativity, associativity, and distributivity of arithmetic operators to alter the structure of the tree, thereby resulting in a tree of lower height and hence reduction in total execution time of the statement.

Squire [SQUI63] has proposed an algorithm which adds at the beginning and end of the statement two delimiters which are assigned the lowest priority. The operand, variables, and temporary results are assigned a start and an end level in the tree, and all temporary results which have the same start level can be computed in parallel. The algorithm requires repeated left-to-right and right-to-left scans on the string and a large number of comparisons. It gives a syntactic tree with a minimum number of levels in almost all cases.

Hellerman [HELL66] has proposed an algorithm which assumes that the input string is in reverse Polish notation. The string is scanned from left to right, replacing by temporary results each occurrence of adjacent operands immediately followed by an operator. It will require as many passes as there are levels in the tree. The temporary results will be considered as operands during the next
passes. The operators replaced during a given pass are at the same level and can be executed in parallel. This algorithm will not reorder the operands. The characteristics of his approach include:

1. The generated code will include an instruction count for each level.
2. The intermediate code generated contains three addresses: two for the operands, and one for storing the result.
3. Each operation will be assigned to a processor by the central control. At the completion of each operation, the processor signals its completion to the central control.
4. The operation at a higher level will not be started until all the operations at the lower level are completely finished.

Hellerman has also proposed a machine organization (Figure 2.1) for parallel execution of expressions compiled by the algorithm. His machine has a central control which includes three counters, triggers for holding the completion signals from the processors, and triggers to indicate which processors are assigned. Each processor is capable of fetching an instruction containing two operand addresses, fetching the operands, decoding and executing the
instruction, and then returning the result to storage. Every time a processor finishes an instruction, it sends a "complete" signal to central control. The central control uses the counters to keep track of the number of instructions left to be assigned in the current level, the number of instructions being executed, and the location of the next instruction to be assigned.

Stone [STON67] has proposed a one-pass algorithm to generate a string in reverse Polish notation. It uses a recursive procedure to link, whenever possible, two subtrees of the same level to build a single subtree of a higher level. It will not change the ordering of the operands. The resulting string will not show explicitly which operations can be performed in parallel; another pass is required to show these parallel computations.

Baer and Bovet [BAER69] have proposed an algorithm which produces a minimum number of levels in the tree. The algorithm uses multiple passes, each corresponding to a level. It uses a left-to-right scan so that the same symbol is not scanned more than once during a given pass. All temporary results which can be generated at that level are constructed and inserted appropriately in the output string produced by the corresponding pass. Then this output string becomes the input string for the next level, until the whole
Figure 2.1' Hellerman's Machine Organization
expression has been compiled. Beatty [BEAT72] has shown that the algorithm of Baer and Bovet [BAER68] always produces a minimum number of levels in the tree.

Kuck and Muraoka [KUCK74a] have studied the upper bound on the reduced tree height, assuming only associativity and commutativity are used. For any arithmetic expression with \( n \) operators and depth \( d \) of parenthesis nesting, it has been shown that the expression can be transformed such that the execution time will be less than or equal to \( \text{ceiling}(\log n) + 2d + 1 \), with the number of processors less than or equal to \( \text{ceiling}(n/2-d) \).

Bounds using associativity, commutativity, and distributivity have been studied. In [BREN74], it is proved that by using associativity, commutativity, and distributivity, any arithmetic expression with \( n \) operators can be transformed such that the total execution time is less than or equal to \( \text{ceiling}(4 \log n) \), with the number of processors less than or equal to \( 3n \).

To consider the case where tree-height reduction is desirable but the number of available processors is small, Brent [BREN74] has given an upper bound to the reduction of execution time: given any expression with \( n \) operators and \( p \) processors for its evaluation, by the use of associativity,
commutativity, and distributivity, the expression can be transformed such that the execution time is less than or equal to $4 \log n + 10(n-1)/p$.

2.2 High-Level Language Computer Architecture

In the following we will survey the outstanding features of research and development in high-level language computer architectures.

Burroughs introduced the B5000 in 1961 and the B5500 in 1964. The B5000 series were designed to efficiently handle programs written in ALGOL 60. The Burroughs systems involve the distinctive ideas of hardware stack and descriptor. The stack mechanism is very effective in the ALGOL handling environment. Its use permits efficient evaluation of arithmetic expressions and procedure handling. The descriptor may be regarded as a generalized form of control word. It is used to separate the data definition and control from procedure code. This allows the system to maintain high-level abstraction of the user environment.

Bashkow et al. [BASH67] proposed the system design of a FORTRAN-subset machine, as shown in Figure 2.2. It has two modes: LOAD and EXECUTE. In the LOAD mode, it performs the lexical and syntactical analysis of the FORTRAN program
placing the modified program into the Program Area and certain information into the Symbol Table Area. The modifications to the FORTRAN programs are that variables are replaced by actual data addresses and statement number references by actual addresses of the statement location. When the END statement is encountered, it goes into the EXECUTE mode. Pressing the console START button causes the execution to start at address 100. Although the design was presented in considerable detail, it was not implemented.

Helmut Weber [WEBE67] implemented an EULER processor with microprogramming on an IBM 360/30. EULER is a generalization of ALGOL 60. The system consists of three parts:

1. a translator written in Model 30 microcode. It is a one-pass syntax-driven compiler which translates EULER language programs into a reverse Polish string language.
2. an interpreter interpreting the Polish string language programs.
3. an I/O control program written in 360 machine language.

The storage area is divided into three parts: the program area, the stack area, and the variable area. The data in the stack area and the variable area use descriptors to identify the type of data. The total compiler microprogram space is approximately 500 words of 60 bits. The main
Figure 2.2 Bashkow's FORTRAN Machine
storage required is 1200 bytes. The total space for the Polish string language interpreter is 2500 microwords.

Chu et al. [CHU70] designed an ALGOL 60 computer system, which is classified as Type 4 according to Chu's definition. It consists of a lexical processor, an interpreting processor, an I/O processor, a main memory processor, and various peripheral devices. It does not have any intermediate languages. The lexical processor scans the ALGOL programs and passes a lexical unit with its type and value to the interpreting processor upon its requests. The interpreting processor also controls the semantic processor performing the operators in the ALGOL programs. The procedure library is also written in ALGOL.

R. Rice and W. R. Smith began the development of the SYMBOL computer during the mid-sixties [RICE71]. The system was operational by 1970 and was delivered to the Iowa State University for evaluation in 1971. Its design premises include:

-- direct high-level language implementation
-- interactive system of 32 channels
-- automatic virtual memory management

Since the designers felt that contemporary high-level languages were not suitable candidates for direct high-level
language implementation, they designed the Symbol Programming Language (SPL), which was intended to be a general-purpose language as powerful as PL/I but avoiding the ambiguity and irregularities in the language.

The SYMBOL computer is of type 3 according to Chu's definition. The SPL programs are translated by a hardware translator into a reverse Polish string ready for execution. The system procedure library is also written in SPL source language. The procedures are translated along with the calling programs.

Externally, the SPL language recognizes only one data type: the character string. Internally, however, it utilizes three data types: character string, packed floating point numeric, and integers. Data type and size conversions are done dynamically.

The system has a multiprocessor architecture. It consists of an arithmetic processor, a channel controller, a disk controller, a hardware translator, a hardware text editor, a hardware format processor, a hardware system supervisor, and a hardware virtual memory system. Each processor is heterogeneous and is dedicated to a specific task. Figure 2.3 is a block diagram of the SYMBOL system.
Figure 2.3 SYMBOL Computer System
The Burroughs B1700 was announced in 1972 and 1973. It was designed to support many high-level languages and processing environments. The instruction set of the machine includes the primitives from the set of programming languages and emulation environments. The arithmetic logic unit can be set to a width corresponding to the machine that is being emulated.

S. C. Schroeder and L. E. Vaughn [SCHR73] reported on the Litton HOL computer which is based on the APL machine proposed by Abrams. Figure 2.4 is the block diagram of the HOL computer. The system consists of a Master Processing Unit (MPU) and one or more Satellite Processing Units (SPU). Programs are executed by the MPU, which consists of four functional units, called O-machine, C-machine, D-machine, and E-machine. The O-machine is responsible for job scheduling, resource allocation, library maintenance, diagnostics, and system error procedures. The C-machine translates the APL-like high-level language programs into Polish strings and passes them to the D-machine. The D-machine analyzes the operator-operand sequence passing to it to determine an optimal operator sequence. The E-machine is primarily responsible for the array-processing functions.
Figure 2.4 Litton HOL Computer
Fournier [FOUR75] introduced the concept of "grammar-programming." In his Grammar Programmable Machine (GPM), the system is divided into three levels: the user level, the grammar level, and the hardware/firmware level. The language definition is stored in the grammar level. The interpreter can be reloaded for different languages and extended during execution.

Mooney [MOON77] proposed the design of a variable high-level language computer. The key features of his design are the ability to modify and extend the language definition at any time and the use of a parallel processing philosophy throughout the design. During the translation phase, the Token Processor mechanism is used to explore individual paths in parallel. The programs are translated into Execution Trees. When recognition is complete, the trees represent the semantics of the programs in a form suitable for execution. During the execution, the Operand Evaluators process input operands of a function, and the Execution Processors carry out the operations on the operands. These activities occur in parallel as much as possible.
Yamamoto [YAMA80] reported several high-level language computer implementations in Japan. A LISP machine has been developed by K. Taki et al. at Kobe University. It is organized with 4-bit slice microprocessors (Am2900 family), which have 16-bit data length, 56-bit microinstruction length and 32-bit list cell length. A DEC LSI-11 minicomputer performs initiation and maintenance functions, LISP program loading and I/O operations. A multiprocessor LISP machine has been developed by Yasui et al. at Osaka University. The arguments for a LISP function are evaluated in parallel on multiple processors. The design uses bit-slice microprocessors (Intel 3000). In 1978, Furuya implemented a Concurrent PASCAL Machine. It consists of an interpreter to execute the Concurrent PASCAL instructions, and a Kernel to supervise parallel processes.

Yamamoto et al. [YAMA80B] reported the design and evaluation of a COBOL machine. The COBOL machine runs as a processor attached to a host processor, in which a medium scale conventional commercial computer is used as a base computer. The COBOL machine has many facilities for efficient COBOL execution: many internal data types, highly functional data descriptors, and intensive instructions. It was found that the average statement execution time of the COBOL machine was only 35% of that of the host processor.
Chu [CHU75] has defined a high-level language computer system as "one that can accept and execute a high-level language program." However, this definition does not distinguish which computers are and which are not high-level language computer systems, since every commercial computer or microcomputer can satisfy this definition. Ditzel and Patterson [DITZ80] have proposed another definition which is more useful for that purpose:

"A high-level language computer system is one that:

1. Uses high-level languages for all programming, debugging and other user/system interactions.
2. Discovers and reports syntax and execution time errors in terms of the high-level language source program.
3. Does not have any outward appearance of transformations from the user programming language to any internal languages."

They also introduced the concepts of High-level Language Execution Support Factor (HLLESF), High-level Language Size Support Factor (HLLSSF), and High-Level Language Preparation time Support Factor (HLLPSF) as the measures of evaluation of architectures in terms of high-level language computer systems.
Haynes [HAYN77] has designed an ALGOL-60 computer, as shown in Figure 2.5. In this computer, specific aspects of the translation and execution tasks are implemented by separate, simultaneously operating parallel processors, each optimized to its specific functions and to the syntax, data structure, and semantics of ALGOL-60.

The high-level Polish String language is used as the internal language of the computer. The hardware for the computer consists of a main memory and seven processors. The translation is done as follows. The I/O Processor reads card images from the card reader and stores them into the I/O buffers. The Scanner Processor asks for and receives card images from the I/O Processor. The Scanner then performs a lexical translation on the characters of the source program, storing translated lexical units into the lexical unit buffers. The Syntactic and Semantic Processor asks for and receives lexical units from the Scanner Processor and performs the syntax analysis. It generates and passes Polish String operators to the Polish String Access Processor. The Polish String Access Processor receives Polish String elements and stores them into the main Polish String memory.
Figure 2.5 Haynes' ALGOL 60 Computer
When the translation is complete, the execution phase begins. The Polish String Access Processor reads Polish String word from the main Polish String memory, separates them into Polish String operators, and stores the operators into the Polish String buffers. The Syntactic and Semantic Processor asks for and receives Polish String operators from the Polish String buffers. It issues commands to the I/O Processor to perform I/O operations, to the Arithmetic Processor to perform arithmetic operations, and to the Array Storage Processor to store and retrieve array data.
3.1 Introduction

In this chapter we investigate the problems involved in parallel execution of arithmetic expressions in high-level programming languages. We are not concerned with the tree-height reduction techniques as proposed in [BAER68], [RAMA69], [SQUI63], and [STON67]. Rather, we are dealing with a representation of parallelism, a way the expressions are executed, and an appropriate computer organization for carrying out the execution. The concept and notion of a Parallel Execution String (PES) is introduced as a representation of expressions for parallel execution. The PES approach is then applied to detect the parallelism at both the statement and the block levels in Section 3.2 and
Section 3.3, respectively. Section 3.4 describes the representation of concurrent statements in a high-level language computer for parallel processing. In Section 3.5, we discuss the representation of IF, WHILE, REPEAT, and LOOP statements in the internal language of a high-level language computer for try-ahead processing. Three algorithms are then given to convert expressions into PES's and to schedule them for execution in Section 3.6. A machine organization suitable for carrying out parallel operations with the PES approach is described in Section 3.7. Finally, code generation and code optimization techniques are discussed in Section 3.8 and Section 3.9, respectively.

3.2 Parallel Processing of Expressions

It is well known that an expression can be represented by a rooted tree, with its internal nodes denoting operators and its external nodes variables and constants [KUCK76]. The son-nodes of a node are the operands of that node. For two operator nodes in the tree, if neither is the ancestor of the other, these two operators are independent and thus can be executed in parallel. In order to take advantage of the parallelism during the execution of an expression, there should be an intermediate form into which the expression can
be transformed that shows the parallelism explicitly.

One method (i.e., the conventional parallel-by-level approach [HELL66]) is to group together the operations that are at the same level in the tree and then to execute the operations in a group in parallel. The result of each operation is represented by some external symbol which is not in the expression, and the symbol is then used as the operand of some operation on a higher level.

Figure 3.1 is an example of this scheme. The expression \(-\frac{(A+G+B*C)}{(D*(E+I)+F)+H}\) is represented as a binary tree of five levels high. During the first time unit, all the operations at the bottom level are executed simultaneously. The results are stored in T1, T2, and T3. During the second time unit, the two operators at the second level are executed simultaneously. The results are stored in T4 and T5. The execution then goes on to the next level, and so on. The execution ends after five time units.

The implication of this method is that we should have assumed that all operations take an equal length of time; otherwise there will be instances where the executions of some operations are delayed unnecessarily. However, this assumption does not hold for most real computers. Thus we propose another scheme to represent an expression in such a
Figure 3.1 Parallel-by-level Approach to executing an Expression in Parallel
form that is appropriate for parallel executions regardless of the execution time of various operations. Later we will see that this scheme also has some additional advantages. First, let us give some definitions:

**Definition**

In an expression tree, an operator node is called

- **type 1** -- if all of its operands are variables or constants;
- **type 2** -- if exactly one of its operands is an operator; and
- **type 3** -- if more than one of its operands are operators.

If we consider only unary and binary operators, then the definition of type-3 becomes:

- **type 3** -- if it has two operands being operators.

Since all operators in arithmetic expressions are unary and binary, from now on we will only consider unary and binary operators. However, the proposed scheme can be easily extended to handle operators with more than two
If we consider the type-1 nodes as starting points toward the root of the tree, then there are as many paths as type-1 nodes. Each path passes through a sequence of operators and uniquely defines a string of operators and operands, starting at a type-1 node and ending at the root. The string is to be called a Parallel Execution String (PES). These paths merge together on their ways toward the root and eventually converge at the root, where the last operation is performed. Note that each path has a type-1 node at one end and the root of the tree at the other end, and all of the intermediate nodes on each path are of type 2 or type 3. Type 3 nodes are merging points of paths, whereas the others are of type 2.

We observe that all the operations on each path have to be executed sequentially, beginning from the starting node and heading toward the root. However, any two nodes on two different paths prior to the merging point of these two paths (even though they are at different levels) are independent and thus can be executed in parallel. From these observations we can see that there exists more parallelism among the PES's of an expression than what can be exploited by the parallel-by-level approach. In a formal way, we can define a PES as follows:
Definition

A Parallel Execution String (PES) of an expression is a sequence

\[ \text{DO } D_1 \text{ T}_1 \text{ D}_2 \text{ T}_2 \ldots \text{ D}_{(n-1)} \text{ T}_{(n-1)} \text{ D}_n \text{ T}_n \]

where

1. \( T_1, T_2, \ldots, T_n \) are operator nodes in the tree, \( T_1 \) is a type-1 node, \( T_2, \ldots, T_n \) are of type 2 or 3, and \( T_n \) is the root node;

2. for \( i = 1, 2, \ldots, n-1 \), \( T(i+1) \) is the father node of \( T_i \);

3. \( D_0 \) and \( D_1 \) are operands of \( T_1 \);

4. for every \( i \) such that \( T_i \) is of type 2, \( D_i \) is either empty or an operand of \( T_i \), depending upon whether \( T_i \) is an unary or binary operator;

5. for every \( i \) such that \( T_i \) is of type 3, \( D_i \) is represented by "#k", where \( k \) is a number uniquely identifying the node \( T_i \) among all the type-3 nodes; and

6. if \( T_i \) corresponds to a non-commutative operator \( X \), and \( T(i-1) \) is the right son-node of \( T_i \), then \( T_i \) is represented by \( X' \).
Figure 3.2 is an example of representing an expression in PES notations. (In Section 3.6, an algorithm will be described for converting an arithmetic expression into PES's.) In this example, the expression has three type-1 nodes and hence can be converted into three PES's. In the third PES, an operator /' is used to signify that the ordering of its operands is reversed.
The expression \(-\frac{A+G+B*C}{D*(E+I)+F}+H\)
can be represented as a tree:

![Tree Diagram](image)

It can be compiled into PES's as:

\[
\begin{align*}
A \ G \ + \ &\ #1 \ + \ #2 \ / \ H \ + \\
B \ C \ * \ &\ #1 \ + \ - \ #2 \ / \ H \ + \\
E \ I \ + \ &D \ * \ F \ + \ #2 \ / \ H \ +
\end{align*}
\]

Fig. 3.2 Example of translating an expression into PES's
With the definitions above, we thus propose a scheme to transform an expression into PES's and execute them in parallel as follows. Whenever a processor is free, it will pick up one of the PES's that have not yet been taken and start executing that PES from left to right. The first operation is always of type 1, which means all of its operands are variables or constants, so that it can be executed immediately. If it is a unary operator, it performs the operation and keeps the result in the processor for the next instruction. If it is a binary operator, it loads the first operand into the processor, performs the operation on the first and second operands, and keeps the result for the next instruction.

When the processor reaches a type-2 node, nothing will prevent it from performing that operation, because this node has exactly one of its operands as an operator which has already been executed immediately prior to this node by the same processor. Note that the result of the previous operation is still kept in the processor and can be readily used as the operand for this type-2 node. The processor will execute the type-1 and type-2 nodes in the PES one by one, independent of other PES's.
When the processor reaches a type-3 operator, i.e. the operator with a \#k operand, it will either continue executing the PES or save the partial result obtained thus far and then give up the PES, depending upon whether or not any of the other PES's passing through the same type-3 node has been executed up to this node. For each "\#" operand, there is a storage location being used for storing the partial result of the string which is finished earlier than the other string passing through the same type-3 node. One possible machine organization to implement this scheme is described in Section 3.7. Each PES will be executed only by one processor, although the processor may give up that PES before it reaches the end. Even though an arithmetic expression can be executed by several processors simultaneously, only one processor will eventually finish the execution of the expression.

In the example in Figure 3.2, the operation in B*C and the multiplication in D*(E+I) are independent and thus can be executed in parallel. This parallelism is expressed explicitly in the PES representation, whereas the parallel-by-level approach cannot make use of this parallelism.
Note that if \( N \) is the number of PES's in an arithmetic expression. (i.e. there are \( N \) type-1 nodes), then there are \( N-1 \) "#" operands; that is, there are \( N-1 \) type-3 nodes. This can be shown as follows.

To show that there is always one more string than there are type-3 nodes, we will use mathematical induction. Note that there are exactly two strings merging at every type-3 node.

(Basis, \( N=1 \)) If there is one string for an expression, then there is no type-3 nodes. Thus, the statement is true for \( N=1 \).

(Induction step) Suppose the statement is true for \( N=k \), i.e. if there are \( k \) strings for an expression, then there are \( k-1 \) type-3 nodes. For an expression with \( k+1 \) strings, there will be \( k-1 \) type-3 nodes at the merging points of the first \( k \) strings (induction hypothesis). Since the \((k+1)\)th string must merge with one of the other strings at a type-3 node, then the total number of type-3 nodes is \((k-1)+1=k\). That is, for an expression of \( k+1 \) strings, there are \( k \) type-3 nodes. Therefore, the statement is also true for \( N=k+1 \). Hence, it is true for any positive integer \( N \).
The following is an example which shows the results of applying several tree-height reduction techniques to the same expression, their resulting tree representations, corresponding Polish strings, and PES representations. It also compares the execution time required for both Hellerman's machine [HELL66] and our PES approach.

(1) The original expression is:
\[ A + B \times C + (D + E) \times F + G + H \]

The corresponding tree is:

(2) After applying Stone's algorithm [STON67], the tree becomes:

The corresponding Polish string:
\[ ABC*+DE+F*+G+H+ \]
(3) If we apply Baer and Bovet's algorithm [BAER68], the tree becomes:

```
       +
      /\  
     *  +
    /\  /\  
   F  A  G
```

The corresponding Polish string:

\[ AG + BC * + DE + F * + H + \]

The PES are:

\[ AG + \#1 + \#2 + H + \]
\[ BC * \#1 + \#2 + H + \]
\[ DE + F * \#2 + H + \]

(4) If we apply Squire's algorithm [SQUI63], the tree becomes:

```
       +
      /\  
     *  +
    /\  /\  
   A  G  B  C  D  E
```

The corresponding Polish string:

\[ DE + F * BC * A G + H + + + \]

The PES are:

\[ DE + F * \#1 + \]
\[ BC * \#2 + \#1 + \]
\[ AG + H + \#2 + \#1 + \]

(5) Comparison of execution time between Hellerman [HELL66] and PES approaches for applying various tree-height reduction techniques is shown below. When counting the execution time, we assume that each add operation takes one unit of time, each multiplication takes five units of time, and each partial result storing and fetching
takes one unit of time. We also assume that there are always enough processors for execution.

<table>
<thead>
<tr>
<th>Tree-Height Reduction Technique Applied</th>
<th>Execution Time</th>
<th>Hellerman's Approach</th>
<th>PES Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(a)</td>
<td>(b)</td>
<td>(a)</td>
</tr>
<tr>
<td>none</td>
<td>18</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>Stone's</td>
<td>18</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>Baer &amp; Bovet's</td>
<td>16</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>Squire's</td>
<td>16</td>
<td>12</td>
<td>9</td>
</tr>
</tbody>
</table>

Column (a): partial results storing and fetching considered
Column (b): partial results are kept in registers, no STORE and FETCH are needed

From the above comparison, we see that the PES approach outperforms the Hellerman's in every case. This fact is not surprising, since the PES approach makes use of more parallelism than Hellerman's.

As mentioned earlier, the PES scheme has the advantage that it can exploit all the parallelism within an expression regardless of operations that may take unequal length of time. In addition, we also find that it has the following advantages:

1. The execution of a PES is done directly from left to right. No precedence relation between the operators needs to be considered, and it does not use any stacks.
2. If the operations in the expressions are limited to unary and binary operations, only a one-address instruction is needed to perform the operation for each type 2 and type 3 node, and two one-address instructions for each type-1 node.

3. The partial results need not be stored for any type-1 and type-2 nodes, since they remain in the processors to be used in subsequent operations. This will eliminate the time delay created by storing and fetching the partial results between operations, and it will also increase the execution speed by reducing the traffic between processors and memory. Even though the partial result may have to be stored for a type-3 node, this situation occurs only when the other path has not yet reached that node, so that storing the partial result in this case will not substantially increase the total execution time.

4. When a PES is assigned to a processor, a sequence of operations will be performed by the processor without the intervention of others, so that the execution can be done as fast as possible. This also makes it possible to employ some techniques (e.g., pipelining instruction fetching, decoding, and execution) to increase the execution speed further.
5. As will be seen in the next section, the PES scheme allows detection of parallelism between the operations in different statements.

In the above, we have proposed a representation of expressions for parallel execution which can be used as an intermediate form for a compiler to generate machine code. However, in high-level language computers, no conventional machine code will be generated, and the internal language will be executed by hardware directly. Thus, in the following, we will present a slightly modified form of the above concept to represent expressions in the internal language of a high-level language computer.

In the internal language, the entry points of the strings are chained as a linked list by pointers called Parallel Pointers. After two strings merge, they become identical. The identical portion will not appear in the first string. Instead, the first of the two merging strings has a Jump Pointer following the merging point operator and pointing to the location that immediately follows the merging point operator in the second string.
Figure 3.3 is an example of representing an expression in the high-level language computer internal language. Para represents a Parallel Pointer, and Jump a Jump Pointer.
The expression \( J-(A+B)\times(CD+E-F/(G-H)) \) as a tree:

```
     J
    /  
   *   
  /    
A  +   B
 /    /   
C  D  /   
   /   
  G H
```

It can be translated into the internal language as:

\[
\text{Para } A \quad B \quad + \quad \#1 \quad * \quad \text{Jump} \\
\text{Para } C \quad D \quad * \quad E \quad + \quad \#2 \quad - \quad \text{Jump} \\
\text{Para } G \quad H \quad - \quad F \quad / \quad \#2 \quad - \quad \#1 \quad * \quad J \quad -
\]

and in its linear form:

```
Para A B + #1 * Jump Para C D * E + #2 - Jump
\text{Jump } G H - F / \#2 - \#1 * J -
```

Fig. 3.3 Example of Translating an Expression into the Internal Language
In the following, we will show that the PES approach can be used to represent boolean expressions, assignment statements, indexed variables, and conditional statements for parallel processing.

**Boolean Expressions**

Since a boolean expression can be represented by a binary tree, the same approach to representing arithmetic expressions in PES can be used for boolean expressions. For example, the following boolean expression:

\[(A*B)*C + (D/E) \text{ OR } F > (G*H+I/J) \text{ AND } X+Y > W - Z\]

can be represented by the following tree:

By using the PES approach, we can represent the above tree as follows:

\[A\ B\ *\ #1 > \ #5 \text{ OR} \]

\[D\ E\ /\ C + \ #1 < \ #5 \text{ OR} \]
Assignment statement

Since an assignment statement has the following syntax:

```plaintext
<variable> = <expression>
```
we can regard the "=" as a binary operator whose first operand is the expression and the second operand is the variable. An assignment statement can thus be represented as a binary tree in the same way as an expression. Then the binary tree can be transformed into the PES representation. For example, the following assignment statement:

```plaintext
A = B*C + D/E
```
can be represented by a tree as follows:

```
  =
 /   +
A   #1
    / /
   /  C D
  /   /
B   E
```

The PES representation will be:

```plaintext
B C * #1 + A =
D E / #1 + A =
```
Indexed Variables

We can regard the reference to an indexed variable as an operator that has two operands: an index and an array name. The result of the operation is the array element, indexed by the first operand, in the array named by the second operand. The index is calculated as an expression, and the "reference operator" can be treated the same way as other binary operators. Therefore, the PES representation can be used for indexed variables. For example, the following expression:

\[ C \times D + A[(I+1)\times(K+J)] \]

can be represented as a tree:

```
+  #1
  /\   *
 C  D  B  INDX
    /\   *
   #2  A
     /\  +
    I 1 K J
```

Its PES representation will be:

```
C D + #1 +
I 1 + #2 * A INDX B * #1 +
K J + #2 * A INDX B * #1 +
```
IF STATEMENTS

An IF statement has the following syntax:

IF Boolean-expression THEN statement-1 ELSE statement-2

where statement-1 and statement-2 are assignment statements, compound statements, repetitive statements, or IF statements. We can represent the above IF statement in reverse Polish String as follows:

Boolean-expression IF statement-1 statement-2

Similarly, we can represent it in PES notation as follows:

Boolean-expression statement-1(in PES) statement-2(in PES)
in PES

where the "IF" is also considered as an operator. For example, the following IF statement:

IF \((A+B)>(X*Y)\) THEN \(M = (B-C)+X/Y\)
ELSE \(M = (A*C)/(X+Y)\)

can be represented in PES as follows:

\(AB+\#1>\) IF \(XY*\#1<\) IF \(BC-\#2+M = XY/\#2+M = AC*\#3/M = XY+ 3/M = \)

In section 3.5, we will show a modified representation for IF statements in order to perform try-ahead processing.
3.3 Parallel Processing of Statements

In the last section, we looked into the parallelism at the arithmetic expression level and proposed a scheme to make use of the parallelism to increase the execution speed. However, for certain kinds of programs the parallelism at the expression level will not greatly reduce the execution time, namely, the programs with very short expressions. Therefore, we need to go a step further to investigate how the PES scheme can be applied to exploit the parallelism among a block of statements. In this way, more programs can take advantage of the multiple processors available, thereby reducing the program execution time and increasing the system throughput. This will make the proposed system more practical for real application.

To execute two statements in parallel, a condition must be satisfied: neither the input set nor the output set of one statement can have any variable in common with the output set of the other statement. There are two methods that we can use to exploit the parallelism between statements to make the PES scheme more practical. The first method is to make the statements independent of each other by using a technique called forward substitution [KUCK76]. During forward substitution, the statements in a block are
examined one by one. If a variable on the right hand side of an assignment statement also appears on the left hand side of an assignment statement appearing earlier in the same block, then this variable is substituted with the expression which is on the right hand side of that earlier statement. After applying forward substitution, the expressions become independent of each other and usually become more complicated and have more parallelism to be exploited, so that we can use the PES scheme to execute the statements in parallel. The disadvantages of using forward substitution are that there will be many duplicated computations, and the number of processors to execute becomes very large. For example, considering the following four statements:

\[
\begin{align*}
X &= A + B \times C \\
D &= C + A \\
Y &= B / C + D \times X \\
Z &= X + M \times D
\end{align*}
\]

After forward substitution, they become:

\[
\begin{align*}
X &= A + B \times C \\
D &= C + A \\
Y &= B / C + (C + A) \times (A + B \times C) \\
Z &= A + B \times C + M \times (C + A)
\end{align*}
\]
Then they can be translated into PES's as follows:

\[
\begin{align*}
B C \ast A + X &= \\
C A + D &= \\
B C / \#1 + Y &= \\
C A + \#2 \ast Y &= \\
B C \ast A + \#2 \ast Y &= \\
B C \ast A + \#1 + Z &= \\
C A + M \ast \#1 + Z &= 
\end{align*}
\]

It takes seven processors to execute these PES's in parallel.

In this chapter we propose another method to detect the parallelism between the operations in different statements. Our approach is based on the concept introduced in Section 3.2 that an expression can be represented by one or more Parallel Execution Strings (PES's). The statements in a block will be processed as follows. First, each statement is translated into PES's as described in Section 3.2. Next, the PES's of the statements in a block are tested for dependency and scheduled for execution according to the same condition mentioned above for statements, but here the tasks being tested and scheduled are the PES's generated from the statements, instead of the statements themselves. The advantage of this scheme is apparent from the example below.
In the scheduling process, each PES will be assigned to a specific execution stage according to its data dependency on the PES's that have been scheduled. Program execution will be done stage by stage. Those PES's which are assigned to the same stage can be executed in parallel. The algorithm which checks the data dependency and performs the scheduling is the Algorithm B in Section 3.6.

The following is an example for this scheme. The original program consists of three assignment statements:

\[
\begin{align*}
X &= A \times B \times C + D \\
X &= C \times X / (D + E \times (F - G)) \\
A &= D \times E + C \times B \\
\end{align*}
\]

The PES's for these three statements are:

\[
\begin{align*}
A \times B \times C \times D + X &= \\
C \times X \times #1 / X &= \\
F \times G - E \times D + #1 /' X &= \\
D \times E \times #2 + A &= \\
C \times B \times #2 + A &= \\
\end{align*}
\]

These PES's will be scheduled in two stages for execution:

**Stage 1**
A B * C * D + X =
F G - E * D + #1 \div X =
C B * #2 + A =

Stage 2

C X * #1 \div X =
D E * #2 + A =

From the example above we can see that this scheme has
the advantage that, even though Statement 1 and Statement 2
are not independent, a subexpression of Statement 2 can be
executed concurrently with Statement 1. A similar situation
also exists between Statement 1 and Statement 3.

Obviously, the procedure above can be applied only to a
block of statements in which the first statement is the only
entry and the last statement is the only exit. Therefore,
in the sequential execution of such a block, the execution of
any statement is immediately followed by the execution of
the next statement in the block, except the last statement;
the execution of any statement immediately follows the
execution of the previous statement in the block, except the
first statement. However, some parallelism will not be
exploited if we partition programs into blocks in this way,
since the parallelism detection will be done only within each individual block. Therefore we will use the following technique to combine some blocks into larger blocks and then apply the parallelism detection procedure on the resulting blocks. This technique will exploit some of the parallelism among statements in different blocks in the original program.

The first step is to partition the program into simple blocks as described above. The next step is to repeat the following procedure until no modifications can be made in the program: if a Block i has more than one predecessor block of which the last statement in each block does not involve conditions, then append a copy of Block i to each such predecessor block. Appropriate branching statements might have to be added to the end of some copies of Block i to make them all branching to the correct point.

Figure 3.4 is an example of applying this block combining process. Block i has three predecessor blocks: a, b, and c. The resulting program will have three enlarged blocks: a+i, b+i, and c+i. This procedure will enlarge some of the blocks in a program and hence enable the parallelism detection process to exploit more parallelism in the program.
Figure 3.4  Example for combining Smaller Blocks into Larger Blocks
3.4 Representing Concurrent Statements

A concurrent statement [DIJK68] is a set of statements enclosed by a header COBEGIN and a trailer COEND, for example,

```
COBEGIN
Statement 1;
statement 2;
.
.
.
Statement n
COEND
```

The statements in a concurrent statement can be executed simultaneously. A flowchart of the above concurrent statement is shown in Figure 3.5. To execute the concurrent statement, it will be translated into the internal language as follows:

```
COBEGIN [Para Statement 1; Para Statement 2;
Para ...; Statement n COEND
```

The processor that executes Statement n will execute COEND. The effect of executing COEND is that the processor will
Figure 3.5 Flowchart of a Concurrent Statement
halt its execution temporarily until all the other processors become free.

The semicolons in a concurrent statement will be preserved in the internal language. A semicolon indicates the end of a simple statement in a concurrent statement and hence makes the processor which is executing the simple statement free.

3.5 Try-Ahead Processing

In this section, we will discuss try-ahead processing of statements that include boolean expressions in them, such as: IF, REPEAT, WHILE, and LOOP statements. During the sequential execution of these statements, a boolean expression is evaluated and determines which one of the two possible execution paths is to be taken. However, in the try-ahead processing we are proposing, the possible execution paths are executed at the same time as the boolean expression is being evaluated. We will discuss how to handle the problem of side effects and to obtain the same results as in sequential execution. The representation of these statements in a parallel execution high-level language
computer for try-ahead processing will also be presented.

3.5.1 Try-ahead Processing of IF Statements

An IF statement has the following syntax:

IF condition THEN statement-1 ELSE statement-2

where condition is a boolean expression and statement-1 and statement-2 are assignment statements, IF statements, compound statements, repetitive statements, or procedure calls. It will be translated into the internal language as follows:

Para condition IF Para THEN statement-1 THENEND Jump
ELSE statement-2 ELSEND

The processor which starts executing the IF statement will set up the entry to the THEN clause for a second processor, which in turn sets up the entry to the ELSE clause for a third processor. While the first processor is evaluating the conditional expression, both statement-1 and statement-2 are being executed simultaneously. However, any environment changes resulting from the execution of statement-1 and statement-2 are kept in the local storages of the second and the third processors, respectively, and
will have no effect either on the execution of the other or on the evaluation of the conditional expression.

Executing the symbols "THEN" and "ELSE" causes the processors to enter the THEN state and the ELSE state, respectively. When the THEN state processor executes the symbol "THENEND," or when the ELSE state processor executes the symbol "ELSEND," the processor will halt its execution in the WAIT state. However, "THENEND" and "ELSEND" will have no effect on a processor which is in the normal mode of operation.

When the first processor executes the symbol "IF," it interrupts both the second and the third processors. Depending upon the result of the conditional expression, it makes one of the two processors free immediately; it discards any computations the processor has done, any environment changes made by the other processor are copied into the main storage, and the processor becomes free. When that is done, the first processor resumes execution from where the second or third processor was interrupted.
Note that the three parallel execution paths discussed here can be further divided into more parallel executable paths by using the methods we have discussed so far and those we will discuss in the next several subsections. For example, the following IF statement:

\[
\text{IF } A+B>C*D \text{ THEN } X = A*C-B/D+X*Y \\
\text{ELSE } X = A*D-B/C
\]

can be represented as follows:

\[
\begin{align*}
\text{PARA} & \quad \text{PARA} \quad A \ B + \ #1 > \ JUMP \\
& \quad \text{PARA} \quad C \ D * \ #1 \leq \ IF \\
& \quad \text{PARA} \quad \text{THEN} \quad \text{PARA} \quad A \ C * \ #2 \ - \ JUMP \\
& \quad \quad \quad \quad \text{PARA} \quad B \ D / \ #2 \ - \ #3 \ + \\
& \quad \quad \quad \quad \text{PARA} \quad X \ Y * \ #3 \ + \ X = \ THENEND \\
& \quad \quad \quad \quad \text{ELSE} \quad \text{PARA} \quad A \ D * \ #4 \ - \ JUMP \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{PARA} \quad B \ C / \ #4 \ - \ X = \ ELSEEND
\end{align*}
\]

In the above example, the IF statement is divided into seven parallel execution paths: two for the boolean expression, three for the THEN path, and two for the ELSE paths. Therefore, seven processors can execute the IF statement in parallel.
3.5.2 Try-ahead Processing of WHILE Statements

WHILE statements and REPEAT statements will be processed with try-ahead similar to that for IF statements. However, only the repetitive path will be tried in advance.

The WHILE statement

WHILE condition DO statement-1;

will be translated as:

Para condition WHILE WHILEDO statement-1 WHILEND Jump

The processor which executes the conditional expression sets up the entry to the WHILEDO path for a second processor. The conditional expression and the WHILEDO path are then executed simultaneously.

Executing the symbol "WHILEDO" forces the second processor to enter the WHILEDO state. The environment changes made by a WHILEDO state processor do not affect the main storage and are only kept in the local storage of the processor. A WHILEDO state processor will halt its execution in the WAIT state when it executes the symbol "WHILEND." However, the "WHILEEND" will have no effect on a processor which is in the normal mode of operation. The
purpose of adding this symbol is to prevent the second processor, which is doing the try-ahead processing, from going over statement-1 before the first processor finishes executing the conditional expression.

When the first processor executes the symbol "WHILE," it interrupts the second processor. If the result of the conditional expression is FALSE, the second processor becomes free immediately and nothing in its local storage will be used. The first processor then follows the WHILE pointer to execute the next statement.

If the result is TRUE, the environment changes stored in the local storage of the second processor will be copied into the main storage, and the second processor becomes free. The first processor then resumes execution from where the second processor was interrupted.

3.5.3 Try-ahead Processing of REPEAT Statements

The REPEAT statement.

```
REPEAT
  statement-1;
  statement-2;
```
The conditional expression executed simultaneously, forces the second processor environment changes made by a processor to affect the main storage and the storage of the processor. The symbol "REPEATEND" is added to the statement. The processor which executes the conditional expression will be translated as:

```
statement-1 statement-2 ... statement-n REPEATEND
```

The processor which executes the conditional expression sets up the entry to the REPEAT path for a second processor. The conditional expression and the REPEAT path are then executed simultaneously. Executing the symbol "REPEAT" forces the second processor to enter the REPEAT state. The environment changes made by a REPEAT state processor do not affect the main storage and are only kept in the local storage of the processor.

The symbol "REPEATEND" is added to the statement. Execution of "REPEATEND" will have no effect on a processor which is in the normal mode of operation. Once a REPEAT state processor executes the "REPEATEND," it will halt its execution in the WAIT state.
When the first processor executes the "UNTIL," it interrupts the second processor. If the result of the conditional expression is TRUE, the second processor becomes free immediately. The first processor then follows the UNTIL pointer to execute the next statement.

If the result is FALSE, the environment changes stored in the local storage of the second processor will be copied into the main storage, and the second processor becomes free. The first processor then resumes its execution from where the second processor was interrupted.

3.5.4 Try-ahead Processing of LOOP Statements

The LOOP statement

```
LOOP

statement-1;
statement-2;
.
.
.

statement-n
EXIT IF condition;
```
statement n+1;
statement n+2;

statement-m

END

will be translated into the internal language as:

statement-1 statement-2 ... statement-n LOOPEND

Para condition EXITIF

LOOP statement n+1 statement n+2 ... statement-m Jump

The processor which executes the conditional expression sets up the entry to the LOOP path for a second processor. The conditional expression and the LOOP path are then executed simultaneously. Executing the symbol "LOOP" forces the second processor to enter the LOOP state. The environment changes made by a LOOP state processor do not affect the main storage and are only kept in the local storage of the processor.
The symbol "LOOPEND" is added to the end of statement-n. It will have no effect on a processor which is in the normal mode of operation. However, it will halt the execution of a LOOP state processor until the first processor interrupts the halted processor.

When the first processor executes the "EXITIF," it interrupts the second processor. If the result of the conditional expression is TRUE, the second processor becomes free immediately and the computations done by the second processor are ignored, because the try-ahead processing has failed. The first processor then follows the EXITIF pointer to execute the next statement.

If the result of the conditional expression is FALSE, the environment changes made by the second processor and stored in the local storage of the second processor will be copied into the main storage, and the second processor becomes free. The first processor then resumes the execution of the second processor.
3.6 Compiling Algorithms

In this section we will present three algorithms: Algorithm A is to convert an expression into Parallel Execution Strings, and Algorithm B is to detect the parallelism between PES's in a block and to schedule them for execution. Algorithm C is to convert an expression into the internal language form of a high-level language computer. Algorithms A and C require only one pass through the input expression, and each PES corresponds to one of the execution paths described in Section 3.2. Algorithm B is applied to the PES's in a block one by one.

3.6.1 Algorithm A

Algorithm A assumes that the input expressions have been translated into reverse Polish strings. Figure 3.6 is the flowchart of Algorithm A. Since scanning a reverse Polish string corresponds to the post-order traverse of the expression tree, a stack is used in Algorithm A to keep the operands before their operators are scanned. If the operand stored in the stack is an operator, it will be represented as $n$, where $n$ is the highest numbered PES passing through the operator node.
When a variable or a constant is scanned, it is always pushed onto the stack. When an operator is scanned, it will be handled according to its type (definition in Section 3.2). In Step 5 of Algorithm A, binary operators are processed in Case 1 through Case 3, for type-1 through type-3 nodes, respectively. For type-1 nodes (Case 1), a new PES is generated for the operation, and the operands on the stack are replaced by the string number. For type-2 nodes (Case 2), the operator and the constant or variable operand are appended to each of the PES's passing through the operator node, and the operand is deleted from the stack. For type-3 nodes (Case 3), the operator and a "#i" are appended to each of the PES's passing through the operator node, where i is a unique integer for the type-3 node, and the top two elements on the stack are replaced by the larger of the two. Unary operators are processed in Step 6. There are only two possible types for unary operators: type-1 and type-2. For type-1 nodes (Case 1), a new PES is generated for the operation and the operand on the stack is replaced by the string number. For type-2 nodes (Case 2), the operator is appended to each of the PES's passing through the operator node. No changes to the stack will be made.
Algorithm A

1. Convert the expression into a reverse Polish string using the procedure found in Hamblin [HAMB62]. Here we may use the tree-height reduction techniques [BAER68, RAMA69, SQUI63, STON67] to obtain a modified Polish string, and this may reduce the execution time of the resulting strings.

2. Initialize $i \leftarrow 1, j \leftarrow 1$, where
   - $i$ is an index for temporary storage and
   - $j$ is an index for generated strings.

3. From left to right, scan the Polish string for the next symbol $S$.
   - If it is the end of string, the procedure is done, and String 1, String 2, ..., String ($j-1$) are outputs.
   - If the symbol $S$ is an operand, push it onto the stack, then go to Step 3.

4. If the symbol $S$ is a binary operator, the top two elements on the stack have the following possibilities:
   - **Case 1** Both are operands:
     1) Create a new String $j$.
     2) Let the operator $S$ become the third symbol of String $j$; pop the top element off the stack and
let it become the second symbol of String j; pop stack again and make it the first symbol of String j.

3) Push the symbol $j$ onto the stack.

4) $j \leftarrow j+1$.

End case 1

Case 2 One is an operand, and the other is a $k$:

Let $e$ be a number such that $e$ is the next $'$s appearing in the stack below $k$. If no such $e$ exists, $e$ is 0.

1) Pop up the stack twice. Append the one which is an operand to the same set of strings as in 1.

2) If $k$ is on top of the stack and the operator is not commutative, append the "reverse operator" of $S$ to each of: String $k$, String $(k-1)$, ..., String $(e+1)$. Otherwise, append the operator $S$ to each of the same set of strings. (The "reverse operator" means that the order of its operands is reversed.)

3) Push $k$ onto the stack.

End case 2

Case 3 Both of the top two elements on the stack are $'$s:

Let the top one be $n$, the second one be $m$, and
m < n.
Let e be a number such that $e$ is the next $s$'s appearing in the stack below $m$. If no such $e$ exists, $e$ is 0.

1) Append a symbol $\#i$ to each of: String n, String (n-1), ..., String m, ..., String (e+1).
2) Append the operator $S$ to each of: String m, String (m-1), ..., String (e+1).
3) If the operator $S$ is commutative, append it to each of: String n, String (n-1), ..., String (m+1). Otherwise, append its reverse operator to each of these strings.
4) Pop up the stack twice, and push $n$ onto the stack.
5) $i \leftarrow i+1$.

End case 3

Go to Step 3.

6. If the symbol $S$ is a unary operator, there are two possible cases:

Case 1 Top of the stack is an operand:
1) Create a new String $j$.
2) Pop the top element of the stack. Let it become the first symbol of String $j$.
3) Let the operator $S$ become the second symbol of
String $j$.

4) Push $j$ onto the stack.

5) $j \leftarrow j+1$.

End case 1

Case 2 Top of stack is $k$:  
Let $e$ be a number such that $e$ is the next $\$'s appearing in the stack below $k$. If no such $e$ exists, $e$ is 0.

Append the operator $S$ to each of: String $k$, String $(k-1)$, ..., String $(e+1)$.

End case 2

Go to Step 3.

End of Algorithm A
Figure 3.6 Flowchart of Algorithm A
3.6.2 Algorithm B

Figure 3.7 shows the flowchart of Algorithm B. Algorithm B will detect the parallelism across statements based on the PES scheme. It uses a symbol table to keep track of the variable names used in the block. For each variable in the symbol table, there are two fields associated with it: LAST-FETCHED and LAST-STORED. We will use LF(X) and LS(X) to denote the two fields associated with variable name X. These two fields contain the stage numbers at which a variable name was last fetched and last stored, respectively. Any PES's changing the variable X will be scheduled for a stage later than the larger of LF(X) and LS(X), because they cannot change the value of X until the prior fetching and storing operations of X are completed. Case 1 in Step 1 is designed to handle this situation. Any PES's using the variable X as input will be scheduled for a stage later than LS(X), because they have to wait until the storing operation of X is completed. This situation is handled by Case 2 in Step 1.
In Algorithm B, we use an array TMP whose size is at least the maximum number of temporary storage elements used in a block. TMP(K) keeps the largest stage number of the scheduled PES's that contain the symbol #k. TMP is used for eliminating unnecessary conflict checking and for assigning sub-expressions to the earliest stage possible. During the scanning of a PES, if we find any temporary storage symbol which has been scheduled before, there is no need to continue on the current PES. This situation is handled by Case 3 in Step 1. During the scanning of a PES, a variable STG is updated to the largest stage number in which variable conflicts will prohibit the execution of the current PES. When it comes to Step 2, the current PES has been determined not to be executed in or before stage STG. Therefore, the PES is scheduled for stage STG+1. In Step 3, the table entries of LF, LS, and TMP are updated to reflect the results of scheduling up to the current PES.
Algorithm B

Clear the TMP array.

Apply the following steps to each of the PES in the block one by one.

0) Set STG to 0.

1) Scan the PES from left to right. If it reaches the end of the PES, go to Step 2. Otherwise, get the next symbol S.

If S is an operator, ignore it and get the next symbol.

If it is an operand, there are three possible cases:

Case 1 If S is the output variable of the assignment:

1.1) STG ← max[STG, LF(S), LS(S)]

1.2) go to Step 1.

End_case 1

Case 2 If S is an input variable of the assignment statement:

1.3) STG ← max[STG, LS(S)]

1.4) go to Step 1.

End_case 2
Case 3 If $S$ is a temporary storage for partial result, say $#k$:

If $\text{TMP}(k) = 0$, go to Step 1.

If $\text{TMP}(k) > \text{STG}$, go to Step 2.

If $0 < \text{TMP}(k) \leq \text{STG}$, then

$S \leftarrow \text{end of string},$

go to Step 2.

End case 3

2) \text{STG} \leftarrow \text{STG}+1.

3) For each operand $T$ to the left of $S$ in the string, do the following:

If $T$ is the output variable of the assignment,

$\text{LS}(T) \leftarrow \text{STG}.$

If $T$ is an input variable of the assignment,

$\text{LF}(T) \leftarrow \max[\text{STG}, \text{LF}(T)].$

If $T$ is a temporary storage for the partial result, say $#j$,

$\text{TMP}(j) \leftarrow \text{STG}.$

End of Algorithm B
START

get a new PES. STG = 0

get next symbol S

Output Variable S?

Input Variable

If STG is less than either LF(S) or LS(S), update STG to be the larger of the two

If STG is less than LS(S), update STG to be LS(S).

Temporary Storage

STG + 1 is the stage number for the current PES

S appeared before?

Yes

No

update LF, LS, TMP entries for the variables appearing in the current PES

End of block

STOP

End of PES

Figure 3.7 Flowchart of Algorithm B
3.6.3 Algorithm C

Algorithm C is designed to translate an expression into the internal language form of a high-level language computer as described in Section 3.2. During the translation process, two stacks will be used: OPR-STK and OPN-STK, for storing operators and operands, respectively. Dollar signs ($) will also be used in OPN-STK. LC is the Location Counter which contains the address of the location for storing the next output. Two variables are used: TEMP-COUNTER is for the number of temporary storage locations used, and PES-BEGIN is the starting address of the string currently being generated. An array TEMP-POINTER (TP) is used in the algorithm. TP(i) stores the address of the first of the two #i's in the output, so that when the second #i is generated, a Jump Pointer pointing to the second #i can be generated at the location following the first #i.

The algorithm is similar to that of translating an expression into a reverse Polish string, except that operands are not written out immediately and its operator output procedure is more complicated. A hardware translator can be easily implemented in the Syntactic and Semantic Processor (see Chapter 4) [LALI73]. Figure 3.8 is the
flowchart of the main procedure of the algorithm. Figure 3.9 is the flowchart of the procedure POP-OPR-STK.

In the main procedure of the algorithm, the first step initializes the TEMP-POINTER array, TEMP-COUNTER, and PES-BEGIN. In step 2, the next symbol is read in. Since it is expecting an operand, if the symbol read is neither an operand or a left parenthesis, then it is an error. If it reads a left parenthesis, then it goes back to step 2 and repeats again until it reads an operand. After it reads an operand, it is expecting an operator or a right parenthesis. In step 5, the operators stored in the stack are popped until it encounters an operator on the stack that has a lower priority than the one just read in. If the operator just read is an arithmetic operator, then it goes back to step 2 to read an operand. If the symbol read is a right parenthesis, then the stack should have been popped to where the matched left parenthesis is. If this is the case, then it goes back to step 4 to read an operator. When the input has ended, the symbol read is represented by a special "end-of-expression" symbol, which has the lowest priority among all the other operators.
Figure 3.8 Flowchart of the Main Procedure of Algorithm C
Figure 3.9 Flowchart of Procedure POP-OPR-STK
The POP-OPR-STK procedure is called when the operator stack is to be popped in the main procedure. Depending upon the operator being popped, there are two cases to be considered: unary and binary; these are handled in Case 1 and Case 2 in the procedure, respectively.

For a unary operator, there are two cases to be considered (case 1.1 and case 1.2), depending upon the top element on the top of the operand stack. If the element is a variable or constant (case 1.1), then a new PES is started. If the element is a "$" item (case 1.2), then it is in the middle of a PES and it simply outputs the operator.

For a binary operator, there are three possibilities: type 1, type 2, and type 3. They are handled in the procedure by case 2.1, case 2.2, and case 2.3, respectively. The type of operator can be identified by examining the top two elements on the top of the operand stack. If both of the two elements are variables or constants, then the operator is of type 1. If both of the two elements are "$" items, then the operator is of type 3. If one is a variable or constant and the other is a "$" item, then the operator is of type 2. In case 2.1, since the operator is of type 1,
a new PES is started. In case 2.2, since the operator is of type 2, the variable operand is popped out, and then the operator is popped out in the middle of a PES. In case 2.3, the operator is of type 3, so a "#" item is generated, and a Jump pointer is generated for the PES which also passes the same operator node.

Main Procedure

1. Clear TP array. Initialize TEMP-COUNTER <- 0;
   PES-BEGIN <- LC.
2. S <- next input symbol.
3. If S is a '(', then push OPR-STK('(') and go to 2,
   else if S is a variable, then push OPN-STK(S),
   else ERROR.
4. S <- next input symbol.
5. WHILE Priority(OPR-TOP) > Priority(S) DO POP-OPR-STK.
6. If S is a ')' and OPRTOP='(' , then pop OPR-STK and go to 4.
   If S is a ')' and OPRTOP is not '(', ERROR.
7. If S is an arithmetic operator, then push OPR-STK(S) and go to 2.
8. If S is 'end-of-expression' and OPR-TOP is not a '(',
   then DONE else ERROR.
Procedure POP-OPR-STK

Case 1  The OPR being popped is a unary operator:

Case 1.1  OPN-STK(TOP) is a variable:
1. FINISH-PREVIOUS-PES.
2. Pop OPN-STK, and output it.
3. Output OPR.
4. Push $(TEMP-COUNTER + 1)$ onto OPN-STK.

Case 1.2  OPN-STK(TOP) is a $k$:
1. Output OPR.

Case 2  The OPR being popped is a binary operator. Depending upon the top two elements on OPN-STK, there are three cases:

Case 2.1  Both of the two elements are variables:
1. FINISH-PREVIOUS-PES.
2. Output the top two elements from OPN-STK.
3. Replace the top two elements on OPN-STK by $(TEMP-COUNTER + 1)$.
4. Output OPR.

Case 2.2  One element is a variable, and the other is a $i$:
1. Output the variable.
2. If OPR is non-commutative and OPN-STK(TOP) is $i$, then output OPR', else output OPR.
3. Replace the top two elements on OPN-STK by $i$. 
Case 2.3 Both of the two elements are $'s. Let OPN-STK(TOP-1) be $k$, and OPN-STK(TOP) be $j$:
1. Output #k.
2. If OPR is non-commutative, then output OPR', else output OPR.
3. Output OPR to the location pointed to by TP(K).
4. Output a Jump Pointer with the content of LC to the location pointed to by TP(K)+1.
5. Replace the top two elements on OPN-STK by $j$.

Procedure FINISH-PREVIOUS-PES

1. TEMP-COUNTER ← TEMP-COUNTER + 1.
2. Output # (TEMP-COUNTER).
3. TL(TEMP-COUNTER) ← LC; increment LC by 2.
4. Output a Parallel Pointer with the content of LC to the location addressed by PES-BEGIN.
5. PES-BEGIN ← LC.

3.7 Machine Organization
One possible machine organization for parallel processing of PES’s is shown in Figure 3.10. In this system, there is a variable number of identical microprocessors. Each microprocessor has its own program counter PC, accumulator AC, busy bit indicator B, and ALU. It is capable of fetching, decoding, and executing the instructions stored in the main memory. At the completion of a non-branching instruction, the microprocessor increments its own program counter and starts executing the next instruction fetched from the main memory.

The instruction set of each microprocessor will have a special "operate-or-store" type of instruction, which is an ordinary operator except that the execution will depend upon the condition of the operand. If the operand in the PR memory (to be explained below) shows a "not ready" condition, the execution will not proceed any longer, the microprocessor will become free, and its B indicator will be reset to 0. Instead of performing the operation, it simply stores the content of its AC into the storage location addressed by the operand field of the instruction. If the operand is ready, the instruction will be performed as an ordinary operation. Instructions of this type are used for binary operators of which both operands are the results of some other operators, i.e. the type-3 nodes in the tree
Figure 3.10 Machine Organization for PES Approach
There is a Partial-Result (PR) memory in the system. Its purpose is to store the partial result obtained by the microprocessor that reaches a Type-3 node first. Each location in the PR memory has a status bit which indicates the availability of the partial result. The status bits are set initially. The first "operate-or-store" instruction accessing a PR location will store its partial result and set the status bit. The second "operate-or-store" instruction accessing the same location will use its content and reset the status bit. While a microprocessor is executing the "operate-or-store" instructions, no other microprocessors will be allowed to access the same location in the PR memory. This is to insure that only one of the two operands for a type-3 node will be stored in the PR memory.

In the system, there is an "Entry-Point-List" (EPL) memory, which consists of pointers pointing to the starting points of each PES. There is a pair of registers, "Front" (F) and "Rear" (R), and an indicator "Need-Processor" (NP). F and R contain pointers to the EPL memory. Before an execution stage starts, F points to the beginning of the EPL
for that stage, and R points to the beginning of the EPL for the next stage. F is incremented by one when a PES is taken by a processor for execution. The indicator NP indicates 1 when F is not equal to R, and 0 when F = R. Thus NP indicates whether or not the stage needs any more processors for execution.

There is an FR memory to store the pointers that will be loaded into F and R throughout the execution. The "Central-Program-Counter" (CPC) register is a pointer pointing to the FR memory for the current execution stage. Changing the execution sequence is done by changing the content of the CPC register.

In order to detect the completion of an execution stage, a "Number-of-Parallel-Strings" (NPS) register is used. When new values are load into F and R, NPS is set to a value equal to the difference between the values of R and F. It is then decremented at the completion of each PES. The completion of an execution stage is indicated by a zero in the NPS.
The control sequence for each microprocessor can be summarized as follows:

0. Idles. When NP = 1, go to 1.
1. PC ← EPL(F), F ← F+1, B ← 1.
2. Fetch, decode, and execute instructions. Repeat until an IDLE instruction is encountered or the operand of an "operate-or-store" instruction is not ready.
3. NPS ← NPS-1, B ← 0, go to 0.

The control sequence for CPC, FR, F, R, and NPS can be summarized as follows:

0. R ← FR(CPC), CPC ← CPC+1.
1. F ← R, R ← FR(CPC), CPC ← CPC+1.
2. NPS ← R-F.
3. When NPS = 0, go to 1.

Figures 3.11 through 3.14 show an example of the execution of a program in such a computer. Figure 3.11 shows the beginning of an execution stage, where the CPC register points to the FR entry for the next stage. The F register, which contains "a," points to the EPL entry for the first Parallel Execution String in the first stage. The R register, which contains "a+5," points to the EPL entry
for the beginning of the next execution stage. Initially, the NPS register contains the difference between the R register and the F register, i.e., the number of PES's in the current execution stage. In this example, there are five PES's in the first execution stage. Hence the NPS register contains 5.

Figure 3.12 shows the case in which two of the five execution strings have been taken for execution and one of the two has been finished. This is indicated by the NPS, which is 4, and by the difference between the R register and the F register, which is the number of PES's waiting for execution (in this case, 3). When a PES is taken for execution by a processor, the F register is incremented by one. When the execution of a PES is finished, the NPS register is decremented by one.

Figure 3.13 shows the case in which the first execution stage is just finished. The F register and the R register, both containing "a+5," point to the beginning of the second stage. NPS is 0, indicating that all PES's in the first execution stage have been finished. NP, which is 0 because the F register is equal to the R register, indicates that there are no PES's waiting for execution. This causes the state of the machine to be changed to that shown in Figure
In Figure 3.14, the FR entry pointed to by the CPC register is loaded into the R register, and the CPC register is incremented by one. The R register, now containing "a+8," points to the beginning of the third execution stage, and the F register points to the beginning of the second execution stage. In this example, there are three PES's in the second stage. This is indicated by the difference between the R register and the F register, which is 3. The state of the machine is similar to that at the beginning of the first execution stage, which is shown in Figure 3.11.

The design of a multi-microprocessor system using Am2900 bit-slice microprocessors is described in Chapter 5. The system is capable of performing parallel operations with the PES approach and multi-processing sequential programs. It also has an efficient and flexible interrupt handling mechanism. The memory system is also designed to match the high throughput of the multi-microprocessor system.
Figure 3.11 The State of the Machine Before an Execution Stage is Started
Figure 3.12 The state of the machine during the execution of a stage
Figure 3.13  The state of the machine when a stage is just finished
Figure 3.14 The state of the machine when a new stage is started
3.8 Code Generation

For each expression, several sequences of instructions will be generated. Each sequence of instructions corresponds to one of the PES's generated by Algorithm A described above, which also corresponds to one path in the tree representation. The last instruction of each sequence is always an IDLE instruction, which will set the processor free. However, if there is only one sequence in each of two consecutive execution stages, the first sequence will not have the IDLE instruction at its end. This is to eliminate the overhead in rescheduling processors in the case that only one processor is needed in each of the two consecutive execution stages.

For each sequence of instructions, an EPL entry, which is the address of the first instruction in the sequence, is generated in the EPL memory. For each execution stage, an FR entry, which is the address of the first EPL in the execution stage, is generated in the FR memory. Again, if there is only one sequence in each of two consecutive execution stages, there will be no FR entry generated for the second stage. This code generation scheme will insure
that a strictly sequential program can be executed by a multi-processor computer system as fast as by a uni-processor computer system, while the programs with parallelism exploited by the PES scheme will be executed by a multi-processor computer system faster than by a uni-processor computer system.

The translation from the PES's into machine instructions are straightforward. It can be summarized as follows:

1. The first symbol in the PES is always an operand. Generate a LOAD instruction to load that operand into the accumulator. Scan the PES from left to right and do the following steps.

2. If it is a unary operator, generate an instruction to perform that operation on the content of the accumulator. The result of the operation is stored in the accumulator.

3. If it is an operand, generate an instruction to perform the operation which is the next symbol following the operand. The operation has an implied operand which is in the accumulator, and the result is stored in the accumulator.
4. In step 3, if the operand is a numerical symbol preceded by a #, then the instruction generated is a special "operate or store" instruction.

For example, the PES: A G + #1 * #2 / H + will be translated into:

```
LDA A ; load A into accumulator
ADD G ; add G to accumulator
MOS #1 ; multiply #1 to ACC or
; store ACC to #1 then
; idle
DOS #2 ; divide ACC by #2 or
; store ACC to #2 then
; idle
ADD H ; add H to ACC
IDLE ; end of PES, free
; the processor
```
The proposed scheme in Section 3.8 will not inhibit the compiler from performing architecture-independent code optimization techniques, such as operator folding, moving invariant expressions out of loops, and removing calculation for dead variables, [AH077, WULF75]. In addition to the usual techniques, there are three types of code optimization that can be done with the proposed compiling scheme. The first is to eliminate the redundant portion of a PES to save space. When the PES's generated from an expression are scheduled to be executed in more than one execution stage, the redundant portions of some PES's can be eliminated as follows. If a PES contains a partial result symbol (i.e. the symbol preceded by a "#") which also appears in some PES's scheduled for a later execution stage, then the substring to the right of that symbol can be eliminated. That substring is redundant because the execution of the PES will never proceed beyond that partial result symbol.

The following is an example showing how the redundant portion of a PES can be eliminated. The two adjacent assignment statements

\[
\begin{align*}
A &= B \ast C + D \\
X &= (E \ast F \ast G + A \ast X) / (A - B)
\end{align*}
\]

can be represented in PES notation as follows:
The four PES's can be scheduled for two execution stages. There are two PES's in the first execution stage:

\[ BC * D + A = \]
\[ EF * G * \#1 + \#2 / X = \]
\[ AX * \#1 + \#2 / X = \]
\[ AB - \#2 / X = \]

and there are two PES's in the second stage:

\[ AX * \#1 + \#2 / X = \]
\[ AB - \#2 / X = \]

The second PES in the first stage can be shortened to

\[ EF * G * \#1 + \]

Since the other PES's that pass through the operator with \#1 are scheduled for the second stage, the processor executing the PES with \#1 in the first stage will store the partial result and become free as soon as it reaches the \#1 operator. Therefore, the portion to the right of the \#1 symbol of the PES in the first stage can be eliminated. With this technique, the program space requirement can be reduced. Note that this type of code optimization is
machine independent.

The second type of code generation optimization is to generate code so that the different PES's in the same stage will be stored in different memory modules during execution. This optimization is advantageous only if the memory system consists of multiple memory modules and different processors can access different memory modules at the same time. In this way, the memory access contention will be eliminated; hence delay caused by memory contention is also eliminated. It should be noted that since the entry point of each PES is stored in the EPL memory, it is not necessary to store the PES's of the same stage into adjacent memory locations, and this makes this type of code optimization possible. However, it is necessary to know the machine configuration before this optimization technique can be applied during compilation. Therefore, this code optimization technique is machine-dependent.

The third type of optimization is to reorder the PES's to minimize the total execution time. If the number of processors is equal to or greater than the number of PES’s in an execution stage, the reordering of the PES's will not affect the total execution time. However if there are more PES’s than processors, it might be advantageous to reorder
the PES's. This can be shown by the example in Figure 3.15. We assume that each multiplication or division will take 10 time units and each addition and subtraction will take 1 time unit. The expression

\[-(A*G+B*C)/(D*(E+J)*F)+H\]

is expressed as a binary tree and translated into the PES representation. If there are two processors available, it will take 42 time units to complete the execution of the expression. However, if we change the ordering of the PES's so that the third PES will be taken for execution first, then the execution of the three PES's by two processors will take only 33 time units. These schedules are shown by the Gantt chart in Figure 3.15. The operators in the Gantt chart are represented by numbers as labeled in the tree representation.

In this problem, the tasks to be scheduled are the PES's in an execution stage. The goal is to find a optimal non-preemptive m-processor schedule for the PES's in an execution stage to minimize the total execution time. However, the problem of finding the optimal non-preemptive schedule for n independent tasks of unequal length executed by m processors is NP-complete even for m=2 [GARE78, ULLM75]. The problem is complicated even more by the fact
Expression

\[
\frac{5}{4 - \frac{1}{3 + \frac{2 + 9}{8 + 7 + 6}}}
\]

PES's

\[
\begin{align*}
A & \ast \#1 + \#2 / H + \\
B & \ast \#1 + \#2 / H + \\
E & J + D \ast F \ast \#2 / H + 
\end{align*}
\]

Schedule with Original Ordering

Processor

1

11111111134

11111111111

5

10

15

20

25

30

35

40

45

2

22222222267777777888888885555555559

Schedule after Reordering

1

11111111122222222345555555559

1111111111122222222345555555559

5

10

15

20

25

30

35

40

45

2

67777777788888888888

Figure 3.15 An example showing the effects of reordering PES's
that reordering the PES's will vary the execution time for each PES. Therefore, we can only find some near optimal scheduling strategies. A simple and intuitively sound strategy for this problem is the longest-processing-time-first scheduling, which gives the tasks with the longest processing time the highest priority. The execution time for each PES can be estimated by adding up the execution time for all the operators in the PES. Simulations based on randomly generated PES's show that the longest-processing-time-first schedules have near-optimal performance, despite the fact that the estimated execution time for a PES is usually not its actual execution time.

Sometimes, however, the longest-processing-time-first scheduling does produce non-optimal schedules. Figure 3.16 shows an example in which the longest-processing-time-first schedule is not optimal. With two processors, the longest-processing-time-first schedule will take 14 time units, whereas the optimal schedule takes only 13 time units. In this example, each node of the tree represents a sequence of operations. The number inside each node is the execution time for the sequence of operations represented by that node.
Figure 3.16 An example showing the longest-processing-time-first schedule is not optimal
A more detailed investigation of the scheduling problem is described in Chapter 6. The simulation results of several scheduling strategies are also included.
4.1 System Overview

The architecture of a high-level language computer which can execute the internal language as described in Chapter 3 is shown in Figure 4.1. It consists of a PES memory, a main memory, a partial result storage, a scanner, an I/O processor, and a number of PES access processors, entry point registers, syntactic and semantic processors, local storage, and arithmetic processors. The various kinds of processors operate simultaneously in a pipelined manner to achieve maximum performance, and multiple identical processors are used for parallel computations.
Figure 4.1 System Block Diagram of the High-Level Language Computer
The PES memory is the program memory which stores the program represented in the internal language as described in Chapter 3. The main memory is the data memory which stores the environment during the program execution. The PES Access Processors are between the PES memory and the Syntactic & Semantic Processors. The PES Access Processors store the programs in the internal language form into the PES memory during the translation phase and read the programs out of the PES memory, preprocess them, and send them to the Syntactic & Semantic Processors during the execution phase. PES Access Processors are discussed in Section 4.2. The Scanner Processor is an independent processor which performs lexical analysis and sends results to the Syntactic & Semantic Processor for syntactic analysis. It is discussed in Section 4.3. The Entry Point Registers store the starting address of the PES's which are ready for execution at that moment. These registers are discussed in Section 4.4. The Syntactic & Semantic Processors receive lexical units from the scanner processor, perform syntactic analysis, and translate them into PES internal language form during the translation phase. During the execution phase, they read the program in PES's, send commands to the arithmetic processors, and perform all program operations. These processors are discussed in Section 4.5. The arithmetic Processors are under the
control of the Syntactic & Semantic Processors to perform all arithmetic operations and any necessary type conversions. They are discussed in Section 4.6. The partial result storage stores the partial results obtained during the execution of expressions. It is discussed in Section 4.7.

6.2 PES Access Processors

The PES memory stores the internal PES representation of the source programs. The PES Access Processors are independent processors that provide the communication between the PES memory and the Syntactic & Semantic Processors.

During the translation phase, the PES Access Processor receives program tokens in the internal form from the associated Syntactic and Semantic Processor, assembles them, and stores them into the PES Memory.

During the execution phase, each PES Access Processor reads the program from the PES Memory, separates the symbols, and delivers them to the associated Syntactic and Semantic Processor that it is attached to.
A free PES Access Processor will start accessing a parallel execution string by using a non-empty value from one of the Entry Point Registers as a starting address in the PES memory for execution. After that, the Entry Point Register is cleared. The PES Access Processor can continue reading from the PES Memory at its maximum rate until either its buffers are full or it has read a semicolon, which indicates the end of a simple statement in a concurrent statement.

The PES's read from the PES Memory are decoded by the PES Access Processors and delivered to the Syntactic & Semantic Processors. However, some operators, such as Jump Pointers, GOTO labels, and procedure calls, can be preprocessed by the PES Access Processors. When a PES Access Processor reads a Parallel Pointer, it puts the pointer value and its processor identification into one of the Entry Point Registers and continues its processing. When a PES Access Processor reads a Jump Pointer, it simply alters its program counter and reads the program from the new location. When a PES Access Processor reads a GOTO label or a procedure call, it searches the label chain to determine the location of the label or the procedure.
4.3 Scanner Processor

The Scanner Processor is an independent asynchronous processor. It reads the source program from the I/O processor, performs the lexical analysis, and delivers the lexical units and their type information to the Syntactic & Semantic Processors for syntax analysis. The interprocessor communication between the Scanner Processor and the Syntactic & Semantic Processors is done through the lexical unit buffers and semaphores. The Scanner Processor can operate at its maximum rate until its buffers are full. The Syntactic & Semantic Processors can obtain the next lexical unit from the buffers immediately after they request it, until the buffers are empty.

The Scanner Processor will assemble the character strings into lexical units and determine their lexical types, such as identifiers, real numbers, integers, arithmetic operators, logical operators, reserved words, and delimiters. Figure 4.2 is the design of a Scanner Processor for PASCAL lexical units.
Figure 4.2 Scanner Processor
4.4 Entry Point Registers

To save the entry points of the available PES's when all processors are busy, a number of Entry Point Registers are used. Each of the Entry Point Registers consists of four fields: entry point address, identification of the processor that initiated this register (PPID), state of the processor which initiated this register (SPP), and Ready Indicator (RI).

When a Parallel Pointer is executed, the pointer value is stored into the entry point address field of one of the free Entry Point Registers at that moment. The PPID field is set to the processor identification of the processor executing the Parallel Pointer. The SPP field is set to the same state as the processor which is executing the Parallel Pointer. The READY Indicator is set according to the state of the processor as well as the PES that the Entry Point Register is pointing to. If the processor executing the Parallel Pointer is in a normal state, the Ready Indicator of the Entry Point Register will be set to "ready." If the processor is in a try-ahead state, it will look ahead to see whether the string which the Parallel Pointer is pointing to starts with try-ahead operators. If it does, the Ready
Indicator is set to indicate "not ready." Otherwise, it is set to indicate "ready."

When a processor is free, it will look for a non-empty Entry Point Register to get the starting address of available PES's. Only the Entry Point Registers with "ready" RI's will be used for starting executing PES's.

The purpose of the different fields in the Entry Point Registers will become clear in Section 4.5, in which we discuss the Syntactic & Semantic Processors. Figure 4.3 is the design of Entry Point Registers. Since entries will be added and deleted from the entry point registers during the execution, the entry point registers are organized and accessed as a circular queue. Two pointers to the registers are used: the FRONT pointer points to the one that will be taken first, and the REAR pointer points the register that a new entry will be placed. When an entry is added to or taken from the entry point registers, the pointers REAR and FRONT will be incremented accordingly.
Entry Point Register

<table>
<thead>
<tr>
<th>RI</th>
<th>PPID</th>
<th>SPP</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.3 Entry Point Registers
4.5 Syntactic and Semantic Processors

Each PES Access Processor is attached to a Syntactic and Semantic Processor. The PES Access Processor and its associated Syntactic and Semantic Processor operate concurrently and asynchronously. The communication between them is carried out by the buffers in the PES Access Processor and counting semaphore.

During translation, the Syntactic and Semantic Processor receives program tokens from the scanner, performs syntax analysis, translates the program into the internal language, and delivers the resulting program to the PES Access Processor.

During the execution phase, the Syntactic and Semantic Processor executes various types of operators sent by its PES Access Processor, such as IF, THEN, ELSE, BEGIN, WHILE, and REPEAT. It also sends commands to its Arithmetic Processor and the I/O Processor to perform the computations. Each Syntactic and Semantic Processor has its own local memory to temporarily store the environment changes during try-ahead processing.
The Syntactic and Semantic Processors are interconnected so that when a processor has finished a logical expression, it can interrupt the processors that are doing the try-ahead processing.

There are two basic states that a Syntactic & Semantic Processor can be in: the normal state and the try-ahead state. The try-ahead state includes THEN state, ELSE state, WHILEDO state, REPEAT state, and LOOP state.

Each Syntactic & Semantic Processor has two registers: a Parent Processor Identification register (PPID), which stores the identification of the processor which generated the entry point register entry for the current processor; and a State Register (SR), which stores the current state of the processor, i.e. the normal state, THEN state, ELSE state, WHILEDO state, REPEAT state, or LOOP state. The purpose of these registers is that, during the try-ahead processing, the try-ahead processors can use them to determine whether to respond to the interrupt from a processor which just finished a logical expression. It is possible that several try-ahead operations are being carried out at the same time, and this determination enables the correct processors to be interrupted when the try-ahead logical expression is complete. For example, suppose there
are two IF statements in a concurrent statement:

```
COBEGIN IF cond1 THEN stmt1 ELSE stmt2;
   IF cond2 THEN stmt3 ELSE stmt4;
COEND
```

in the internal language, it becomes:

```
COBEGIN Para Para cond1 IF
   Para THEN stmt1 THENEND Jump
   ELSE stmt2 ELSEEND
   Para cond2 IF
   Para THEN stmt3 THENEND Jump
   ELSE stmt4 ELSEEND
COEND
```

Suppose that there are six processors executing the six strings simultaneously. The contents of the PPID register and the State Registers of these processors will be:

<table>
<thead>
<tr>
<th>PPID</th>
<th>Processor</th>
<th>Register</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>normal</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td>normal</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>THEN</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>ELSE</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>THEN</td>
<td></td>
</tr>
</tbody>
</table>
When Processor 1 is finished, it sends out the interrupt request to all other processors, but only Processor 3 will respond since its PPID register is 1, and that is the identification of the processor which is executing the logical expression.

When a processor is free, it will get the starting address of a available PES from an Entry Point Register. If all Entry Point Registers are empty, the processor will be idle until something has appeared in the entry point registers.

When an Entry Point Register is taken by a processor, the PPID field of the entry point register is copied into the PPID register of the processor, and the SPP field of the entry point register into the state register of the processor. Notice that only the entry point registers with "ready" Ready Indicator will be taken by processors.

The purpose of the Ready Indicators in the Entry Point Registers is to prevent multiple levels of try-ahead processing. For example, the IF statement:
IF cond1 THEN stmt1
ELSE IF cond2
    THEN stmt2
    ELSE stmt3

will be represented in the internal language as:

Para cond1 IF
    Para THEN stmt1 THENEND
    ELSE Para cond1 IF
        Para THEN stmt2 THENEND
        ELSE stmt3 ENDEND

Only three processors will be executing the first three strings simultaneously. Their PPID registers and State Registers are:

<table>
<thead>
<tr>
<th>Processor</th>
<th>Register</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>normal</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>THEN</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>ELSE</td>
</tr>
</tbody>
</table>

The entry point register for
    THEN stmt2 THENEND
indicates "not ready," because it has been set up by
processor 3, which is in a try-ahead state, the ELSE state, and the string itself is also a try-ahead string. Therefore, the Ready Indicator of the entry point register is set to "not ready" to prevent multiple levels of try-ahead processing.

However, multiple parallel strings in a try-ahead path are permitted. For example,

**IF** \( a+b > c+d \)**

**THEN** \( z := m*n + x/y \)**

**ELSE** \( s := t/u + v*w \)

will become:

1. **Para** \( a+b + #1 > \text{Jump} \)**
   \( c+d + #1 > \text{IF} \)**

2. **Para** **THEN** **Para** \( m*n * #2 + \text{Jump} \)**
   \( x/y / #2 + z := \text{THENEND} \)**

3. **Para** **ELSE** **Para** \( t/u / #3 + \text{Jump} \)**
   \( v*w * #3 + s := \text{END} \)**

and six processors can be used for execution in parallel.

The processor states will be:
When a try-ahead processing is complete, the entry point registers initiated by the "wrong path" try-ahead processor are also discarded. When a try-ahead state processor enters the normal state, i.e., it has been determined to be the "right path," it will set the entry point registers initiated by it to be "ready."

The reason why the SPP field and the PPID field of an entry point register are copied to the State Register and the PPID registers, respectively, of the processor which is taking the string, is that multiple parallel strings may exist in a try-ahead path. All of these strings must be interrupted appropriately when the try-ahead processing is complete.
No processor in a try-ahead state can become free until it enters the normal state. When a normal state processor becomes free, the content of its PPID register is copied to the PPID register of its child processor.

If an error condition such as arithmetic overflow occurs in a processor in a try-ahead state, the condition will not be attended to immediately. Instead, the processor will halt its execution temporarily until the other processor executing the logical expression interrupts. When that happens, if the try-ahead processor is the "wrong path" processor, i.e. the try-ahead processor to be discarded, then the error condition is ignored and nothing will be done.

If the try-ahead processor is the "right path" processor, i.e. the one to become active, the error condition will then be handled as in a normal state. The reason why the error condition is ignored in the former case is that the error was caused by some computation that should not have been done. For example, in the IF statement:

\[
\text{IF } x=0 \text{ THEN } y := a + b \\
\text{ELSE } y := a / x
\]

When \( x \) is 0 during the actual execution, the ELSE state
processor will have a "divide by zero" error condition; but that error condition will never occur if no try-ahead processing is attempted. Therefore, the error should not get any attention before the processor executing the logical expression is finished and has determined what to do about the try-ahead processors.

4.6 Arithmetic Processors

An arithmetic processor is connected to each of the Syntactic and Semantic Processors. It operates independently and asynchronously with other processors. When a Syntactic and Semantic Processor receives an operand from its PES Access Processor, it saves the type and value of the operand in its operand registers. When it receives an arithmetic operator, it directs its arithmetic processor to perform the operation on the operands stored in its operand registers. The Arithmetic Processor will check the types of the operands, determine if a type error has occurred, and perform any type conversions necessary. The results of an arithmetic operation, including the type and the value, are stored into the operand registers of the Syntactic and Semantic Processor which sent the operator. The scheme used here will not require any stack for
arithmetic expression executions, and, at any time, no more than two operands will be in the operand registers of a Syntactic and Semantic Processor. Stack is used in the main storage only to allocate space when a block or procedure is entered.

Figure 4.4 shows the block diagram of an arithmetic processor. There are five inputs from the Syntactic & Semantic Processor: the operand 1 type, the operand 1 value, the operand 2 type, the operand 2 value, and the operator. The two outputs to the Syntactic and Semantic Processor are the result type and result value. The types of the operands and the operator are the inputs to the type checker, which determines the type of the result. The type checker also generates commands to the type converter for operand values to perform any necessary conversions. The arithmetic logic unit performs the operation specified by the operator from the type checker on the converted operands and generates the result value. The result type and value are sent back to the operand registers in the associated Syntactic & Semantic Processor.
Figure 4.4 Arithmetic Processor
4.7 Partial Result Storage

The Partial Result Storage is designed to temporarily store the partial results obtained during the execution of an expression. Each location in the Partial Result Storage has two fields: the type field and the value field. The type field indicates the data type of the value in the associated data field. The size of the type field is such that it can represent all possible data types as well as a special type called "empty." All type fields are cleared initially to indicate "empty." When a Syntactic and Semantic Processor receives a partial result operand, i.e., an operand of the form $#i$, from its PES Access Processor, it will check the type field of location $i$ in the Partial Result Storage. If it indicates "empty," the Syntactic and Semantic Processor will save the contents of its operand registers, including its type, into location $i$ of the Partial Result Storage; hence the type field of the location $i$ will no longer indicate "empty." The Syntactic and Semantic Processor then becomes free. If the type field indicates "full," i.e., anything other than "empty," the Syntactic and Semantic Processor will read the contents of location $i$ into its operand registers, use them as the operand for the next operation, and reset the type field of
the location $i$ to indicate "empty" so that the same location can be reused by some other operations.
CHAPTER 5

DESIGN OF A MULTI-MICROPROCESSOR SYSTEM
FOR PARALLEL COMPUTATIONS

5.1 System Overview

In Chapter 3 we describe the Parallel Execution String approach to parallel execution of ordinary programs. One possible machine organization for parallel processing PES's is given in Section 3.10, although there are many ways of implementing the machine organization. In this chapter, we will design a multi-microprocessor system using Am2900 family bit-slice microprocessors [ADVA78] to implement the organization for parallel computations.
A major consideration of this design is to limit the complexity of control units. Since the main reason for using parallel execution is to increase the execution speed, a complicated control unit design runs the risk of high overhead, which would defeat the purpose of parallel execution.

Another consideration of the design is modularity. The system should be so designed that adding one more processor will require no changes in the compiler and only the incremental cost of the processor. The design goals of the system can be summarized as follows:

- programs written in high-level languages can be compiled and executed with the proposed scheme to exploit the parallelism in programs at the statement and the block levels;
- several sequential programs can be executed at the same time by different microprocessors;
- simultaneous access to different memory modules is provided;
- an interrupt can be serviced by a microprocessor at the same time as the other microprocessors are executing background programs or servicing other interrupts.
The system consists of five main components: the central sequence control unit, the central interrupt control unit, AM2901 bit-slice microprocessor arrays, the microprogrammed control unit for each bit-slice microprocessor array, and the random access memory system. A block diagram of the system is shown in Figure 5.1.

The Central Sequence Control Unit directs the Am2901 bit-slice microprocessor arrays to the starting points of the parallel executable program streams for execution. It keeps track of the number of tasks waiting for execution and the number of tasks being executed. It also keeps track of the number of tasks having been interrupted and suspended for execution so that it can direct the processors to resume execution of the interrupted tasks. The Central Sequence Control Unit is described in Section 5.2.

The Central Interrupt Control Unit receives interrupt requests from various devices and directs them to the free processors, or to the processors running the lowest priority tasks if there are no free processors. It also maintains a system stack pointer register so that an interrupted task can be saved in a system stack area for later execution. The Central Interrupt Control Unit is described in Section 5.3.
Figure 5.1 Block Diagram of the Multi-microprocessor System
The Am2900 bit-slice microprocessor arrays are the components that execute the program instructions. They are under the control of the Microprogrammed Control Units. The Am2900 bit-slice Microprocessor arrays are described in Section 5.4.

The Microprogrammed Control Units receive directions from the Central Sequence Control Unit and the Central Interrupt Control Unit, and they execute the micro-operations which direct the Am2900 Bit-slice Microprocessor Arrays to realize the program instructions. The Microprogrammed Control Units are described in Section 5.5.

The memory system stores the program instructions and data compiled by the scheme described in Chapter 3. To make simultaneous access possible, a Memory Access Control Unit is used as the interface between the microprocessors and the memory modules. The Memory Access Control Unit is described in Section 5.6.
5.2 Central Sequence Control Unit

Figure 5.2 is a block diagram of the central sequence control unit. It consists of two registers, four register/counters, three comparators, one subtractor, one sequence counter logic, and two RAM's. One of the two RAM's, called EPL memory, contains address pointers pointing to the beginning addresses of the PES's. Therefore, there are as many entries in the EPL memory as PES's in the program. The other RAM, called FR memory, contains address pointers pointing to the EPL entries that point to the first PES in each execution stage. Therefore, there are as many entries in the FR memory as execution stages in the program.

The F register and R register obtain their contents from the FR memory. When they are loaded from the FR memory, F contains the address of the EPL entry that is the beginning address of the first PES in a new execution stage, while R contains the address of the EPL entry that is the beginning address of the first PES in the next execution stage. During the progress of an execution stage, F is used as the address register of the EPL memory, and the data output from the EPL memory directs the free processors to fetch the PES's stored in the main memory. Every time a PES is taken by a processor, F is incremented by one until F is
Figure 5.2 Central Sequence Control Unit
equal to \( R \).

The CPC register, used as the address register of the FR memory, contains the address of the FR entry corresponding to the next execution stage. At the completion of an execution stage, CPC is incremented by one and thus causes the next FR entry to be fetched from the FR memory. The CPC can be directed by each processor to change the execution sequence of a program, such as at a GOTO statement.

The NPS register contains, at any moment, the number of PES's in an execution stage that have not been completed, including those that are being executed. At the beginning of an execution stage, NPS is loaded with a value equal to the difference between \( R \) and \( F \). The completion of a PES execution will decrement the NPS by one. When NPS equals 0, it causes the execution of the next stage to start.

The INTJ counter records the number of tasks that are interrupted and waiting for processors to resume execution. The tasks interrupted can be PES's or interrupt service routines. It is incremented by one when a task is interrupted and decremented by one when a free processor
resumes the execution of an interrupted task. The purpose of this INTJ counter is to remember whether there are any interrupted tasks waiting for execution, so that as soon as a processor becomes free, it can use the information saved in the system stack area to resume the execution of a task that has been interrupted.

There are several input control lines: LFRSA for loading the CPC register, LFR for writing the FR memory, LEPLSA for loading the F register, LEPL for writing the EPL memory, NPSD for decrementing the NPS, INTJU for incrementing the INTJ, INTJD for decrementing the INTJ, and START for starting the control sequence. These lines are under the control of the microprograms in the Microprogrammed Control Units. There are two output control lines: INTJO to indicate that there are interrupted tasks waiting for free processors, and NP to indicate that there are PES's in the current execution stage waiting for free processors.

The control sequence can be summarized as follows:

CS0: load FR and EPL under the control of LFRSA, LFR, LEPLSA, LEPL. When START=1, CS ← 1.
CS1: R ← FR(CPC), CPC ← CPC+1, CS ← 2;
CS2: $F \leftarrow R$, $R \leftarrow FR(CPC)$, $CPC \leftarrow CPC+1$, $CS \leftarrow 3$;
CS3: $NPS \leftarrow R-F$, $CS \leftarrow 4$;
CS4: When $NPS=0$, $CS \leftarrow 2$.

5.3 Central Interrupt Control Unit

Figure 5.3 shows the design of the central interrupt control unit for a system with four microprocessors and eight-level interrupt. The interrupt requests are handled as follows. If there is a free processor when an interrupt request occurs, the free processor will receive the request and execute the proper service routine. If there is no free processor, the interrupt request will be serviced by one of the processors that are running background programs. If no processor is running a background program, the processor which is running the lowest-level interrupt service will be interrupted and switched to run the service for the incoming interrupt. When the interrupt service is finished, the processor becomes free whether or not it was interrupted while processing a task. There are four interrupt status registers, $I_1$ through $I_4$, one for each processor. Let $I_{ij}$ denote the $j$-th bit of register $I_i$. Then $I_{ij}$ is set to 1 when processor $i$ is servicing the $j$-th level interrupt; otherwise it is set to zero. When processor $i$ is running
Figure 5.3 Central Interrupt Control Unit
the background program, I10 will be set to 1. The four bit inputs, B1 through B4, are the busy bit indicators coming from the control units of microprocessors 1 through 4, respectively.

The four control outputs, INTP1 through INTP4, are connected to the next-address control of the microprograms for microprocessors 1 through 4, respectively, to interrupt one of the microprocessors, according to the criteria mentioned above.

When a task is interrupted, the current state of the processor will be saved in a stack under the microprogrammed control. To facilitate this process, a stack pointer register (STKPTR) is used. It can be loaded by a microprocessor to specify the starting address of the stack and incremented by the microprogrammed control when the state of a microprocessor is saved. When a free processor resumes the execution of an interrupted task, the processor state is restored from the memory locations specified by the STKPTR; then the STKPTR is decremented accordingly.

5.4 Microprocessor Arrays
Because of the microprogrammability and flexibility needed in this design, the Am2901 [ADV78] bipolar bit-slice microprocessors are chosen for the CPU’s. Each AM2901 is a four-bit CPU slice, and four Am2901’s are cascaded to form a 16-bit CPU. In this design, sixteen Am2901’s are used to form four 16-bit CPU’s. The connections of the four CPU’s are identical. If more CPU’s are desired, they can be connected in the same way.

Each Am2901 microprocessor slice is a low-power Schottky 40-lead LSI chip. Figure 5.4 shows a block diagram of an Am2901 device. Each Am2901 consists of a 16-word by 4-bit two port RAM, a high-speed ALU, and the associated shifting, decoding and multiplexing circuitry. Data in any of the 16 words of the RAM can be read from the A-port of the RAM as controlled by the 4-bit A-address field inputs. Similarly, data in any of the 16 words of the RAM, as defined by the B-address field input, can be simultaneously read from the B-port of the RAM. The same code can be applied to the A select field and the B select field, in which case the identical file data will appear at both the RAM A-port and B-port outputs simultaneously.
Figure 5.4 Am2901 3-bit-Slice Microprocessor
The direct data input (D) is used to input data into the working registers inside the device. Likewise, this input can be used in the ALU to modify any of the internal data file.

The Q register is a separate 4-bit register intended primarily for multiplication and division, but it can also be used as an accumulator or holding register for some applications.

The ALU is a high-speed arithmetic logic unit which can perform three binary arithmetic and five logic operations on the two 4-bit input words. The two 4-bit input words are selected from the RAM A-port, RAM B-port, direct data input (D), Q register, and "0" inputs. The microinstruction input used to select the ALU source operands are the I0, I1, and I2 inputs. There are eight possible combinations that can be selected. The ALU function is selected by the I3, I4, and I5 microinstruction inputs.

The ALU data output can be routed to several destinations. It can be a data output (Y) of the device, and it can also be stored in the RAM or the Q register. Eight possible combinations of ALU destination functions are
available as defined by the I6, I7, and I8 microinstruction inputs.

The device output Y can be selected from either the RAM A-port or the ALU output F. This is controlled by the I6, I7, and I8 microinstruction inputs.

The RAM inputs are driven from a three-input multiplexer. This allows the ALU outputs to be entered non-shifted, shifted-up one position, or shifted-down one position. The Q register is also driven from a three input multiplexer. In the non-shift mode, the multiplexer enters the ALU data into the Q register. In the shift-up or shift-down mode, the multiplexer selects the Q register data appropriately shifted up or down.

5.5 Microprogrammed Control Units

Figure 5.5 shows the design of a microprogrammed control unit. There is one microprogrammed control unit for each Am2901 Bit-slice Microprocessor array. When a CPU is free, it checks the INTP input from the central interrupt control unit. If there is an interrupt request sent to this CPU, it will start executing the interrupt service routine
Figure 5.5 Microprogrammed Control Unit
immediately. At the completion of the interrupt service routine, the processor becomes free automatically.

If the INTP is 0, the CPU checks whether the INTJO is 1. If the INTJO=1, i.e., if there are interrupted tasks waiting for processors to resume executions, the free processor will restore the state (including the I register) of the interrupted task from the stack pointed to by the STKPTR, update the INTJ and STKPTR, and resume its execution.

If both the INTP and the INTJO are zero, the processor will idle until NP becomes 1. When that happens, it indicates that the background program needs processors, and the starting address will appear on the EPL bus. The processor will load its program counter from the EPL bus and start executing the background program.

When a busy processor detects that the INTP has become 1, it will save the state of the current task in the stack pointed to by the STKPTR, update the INTJ and the STKPTR, obtain the state for the interrupting program, load its program counter from a predetermined location in the memory, and start executing the interrupting program.
Note that the status indicators B and I are used not only by the central sequence control unit and the central interrupt control unit to dispatch the processors and the interrupt requests, but also by the processors themselves to determine the actions to take when they encounter the IDLE instructions. The control sequence of the i-th processor can be summarized as follows:

**PS0**: Idles. When INTP=1, go to PS1. When INTJ0=1, go to PS2. When NP=1, go to PS3.

**PS1**: (the free processor receives interrupts)

**PS1.1**: Disable interrupt, IRPT-reg ← P exor IRPT-reg, II ← P.

**PS1.2**: PC ← service routine address,
Enable interrupt.

**PS1.3**: Fetch, decode, and execute instructions. Repeat until INTP=1 or it encounters an IDLE instruction or the operand of an "operate-or-store" instruction is not ready.
If INTP=1, go to PS4.

**PS1.4**: B ← 0, I ← 0, go to PS0.

**PS2**: (the free processor resumes the execution of an interrupted task)

**PS2.1**: INTJ ← INTJ-1, B ← 1.
PS2.2: restore the state of the interrupted task by using the STKPTR. Update the STKPTR.
PS2.3: Fetch, decode, and execute instructions. Repeat until INTP=1 or it encounters an IDLE instruction or the operand of an "operate-or-store" instruction is not ready. If INTP=1, go to PS4.
PS2.4: B ← 0, I ← 0, go to PS0.

PS3: (the free processor starts executing the background program)
PS3.1: PC ← EPL(F), F ← F+1, B ← 1, IiO ← 1.
PS3.2: Fetch, decode, and execute instructions. Repeat until INTP=1 or it encounters an IDLE instruction or the operand of an "operate-or-store" instruction is not ready. If INTP=1, go to PS4.
PS3.3: NPS ← NPS-1, B ← 0, I ← 0, go to PS0.

PS4: (the busy processor is interrupted)
PS4.1: Disable interrupt.
IRPT-reg ← P xor IRPT-reg,
save the state of the current task in the stack pointed to by the STKPTR. Increment the STKPTR properly.
PS4.2: INTJ ← INTJ+1, B ← 1,
IiO ← 0, Ii ← P.

PS4.3: PC ← service routine address.
    Enable interrupt.

PS4.4: Fetch, decode, and execute instructions. Repeat until INTP=1 or it encounters an IDLE instruction or the operand of an "operate-or-store" instruction is not ready.
    If INTP=1, go to PS4.

PS4.5: B ← 0, I ← 0, go to PS0.

5.6 Memory System

In order to avoid main memory bottleneck and to achieve high throughput, the memory system is designed to be 32-way interleaved. That is, the adjacent addresses are located in different memory modules, and every 32nd word is located in the same memory module.

Any of the four CPU's can access any of the 32 memory modules at the same time as the other CPU's are accessing the other memory modules. When two or more CPU's try to access the same memory module at the same time, access conflicts occur. An alignment network is used between the CPU's and the memory modules. Conflict resolution is done
by the control logic in the alignment network. The lower-order 5 address lines from each CPU are used as the selection controls in the alignment network. When there is an access conflict, the priority logic will select only one of the requests at one time.

Figure 5.6 shows the design of the alignment network for the memory system.
Figure 5.6 Alignment Network
6.1 The PES Scheduling Problem

In the PES scheduling problem, we are given a set of tasks to be executed by a system with m identical processors. Each task assignment is non-preemptive, i.e. once a task is taken by a processor for execution, the processor may not leave it until the task is complete. Each task is also indivisible, i.e. a task can be executed by only one processor at one time. The precedence relation between tasks can be represented by a binary tree. A task cannot be started by a processor until all of its predecessor tasks have been completed. In addition, once a task is finished by a processor and if all the other predecessors of its successor are also finished, its
successor will be started by the same processor immediately. Each task requires a fixed, known time to complete. In our case, the tasks are arithmetic operators, so their execution times usually are known and fixed. With the above constraints, we are to find a schedule that will minimize the total completion time of the tasks.

Since the assembly line scheduling problem with tasks of arbitrary length is an NP-complete problem and can be reduced to the PES scheduling problem, the PES scheduling problem is also NP-complete. The purpose of this chapter is to compare several PES scheduling algorithms by simulation and to obtain some near-optimal scheduling algorithms. Five scheduling algorithms will be simulated, each of which has time complexity of \( O(n) \).

6.2 Comparing Several PES Scheduling Strategies

In this section, we will define five different scheduling strategies for PES and describe the method of simulating these five strategies and the results of such simulation. The five scheduling strategies are: Original-Ordering, Longest-Processing-Time-First, Shortest-Processing-Time-First, Highest-Level-Number-First, and Lowest-Level-Number-First. Each strategy will be
defined below.

The **Original-Ordering (00)** scheduling will not change the ordering of the input PES's. The PES which is closer to the beginning of the expression, and hence closer to the left in the tree representation, will be taken earlier by a processor for execution than one farther from the beginning of the expression.

The **Longest-Processing-Time-First (LPTF)** scheduling gives the PES with the longest estimated execution time the highest priority for being taken by a processor for execution. The estimated execution time of a PES is calculated by adding the sum of the execution time of each operator in the PES. It should be noted that the estimated execution time usually is not the actual execution time for a PES, because for each expression, only one of its PES's will be completely executed; all the others will terminate before the processors reach the end.

The **Shortest-Processing-Time-First (SPTF)** scheduling is the reverse of LPTF. It gives the PES that has the shortest estimated execution time the highest priority for being taken by a processor for execution.
The **Highest-Level-Number-First** (HLNF) scheduling gives the PES whose starting operator node has the highest level number the highest priority for execution. The level number of an operator node is defined as the number of operator nodes between it and the root node, including the root node itself. With this scheduling method, the execution time of each individual operator is not considered as a factor for determining the priority.

The **Lowest-Level-Number-First** (LLNF) scheduling is the opposite of HLNF. It gives the PES whose starting node has the lowest level number the highest priority for execution.

A program has been written in PASCAL to simulate the five PES scheduling strategies in order to determine their performance and to choose the best scheduling method for PES. The program listing is included in Appendix A. The simulation program consists of three main components: tree generation, scheduling with various strategies, and simulation of execution.

6.2.1 Tree Generation
Tree generation is a procedure which takes two inputs: the number of external nodes of the binary tree it will generate, and the maximum execution time of a node in the tree. It will generate a binary tree in which each node has either two son-nodes or none. Each node will have a node number to identify it and be given an integer number representing its execution time. Each external node in the generated binary tree represents a sequence of type-1 and type-2 operators in an expression tree starting from a type-1 node and ending before the first type-3 node. The execution time of the external node corresponds to the sum of the execution times of the operators in such sequence.

Each internal node in the generated binary tree represents a sequence of type-3 and type-2 operators in an expression tree starting from a type-3 node and ending before the next type-3 node. The execution time of the internal node corresponds to the sum of the execution times of the operators in such sequence.

The reason for this scheme is that in PES execution, the sequence of operations corresponding to a node in the generated tree will be executed without interruption once it is taken by a processor. Therefore, the sequence of operations is represented by a single node in the simulation model.
This also explains why each node in the generated tree has either two son-nodes or none. When a node has no son-nodes, it corresponds to a type-1/type-2 operator sequence; when a node has two son-nodes, it corresponds to a type-3/type-2 operator sequence.

The binary tree is generated randomly. The execution time associated with each node is generated by a random number generator. The shape of the tree is also random. A random number generator generates a sequence of integer numbers and these numbers are inserted into a binary tree to become a binary sort tree. Every time the tree generator procedure is called, it generates a random binary tree to be used by the various scheduling algorithms. The binary tree will be scheduled with various PES scheduling algorithms and computed for the execution times of various schedules.

The simulation program will draw a diagram of the tree it generates. In the diagram, a node is represented by a vertical line. At the top end of the line, there is a number representing the execution time of the node. At the bottom end of the line is the node number. The length of the vertical line is proportional to the execution time of the node. There is a horizontal line connecting the node number of a node and the execution time numbers of its sons.
The scheduling will be simulated as follows. Each of the five scheduling algorithms will create a priority list for the nodes in the generated tree. The node closer to the beginning of the list has the higher priority than the other nodes to its right. However, the nodes will not always be taken for execution exactly according to their order in the list, because the nodes in the tree represent operations which are not independent. Instead, the precedence relation among the nodes must be taken into consideration during execution. The priority list only indicates the order of the nodes to be taken when there is more than one node ready for execution.

6.2.2 Original-Ordering Schedule

In the Original-Ordering scheduling, the generated binary tree is traversed recursively. Each internal node will be assigned a weight of 0. Each external node will be assigned a weight equal to $2 \times (\text{number of external node}) - 1 - (\text{node number of its parent node}) - (0 \text{ or } 1, \text{ depending upon whether it is a left or right son of another node, respectively})$. Note that the internal nodes of the tree form a binary sort tree by their node numbers. Having been assigned weights as described above, the nodes will be
sorted into descending order of weight. The resulting list will have all the external nodes assigned higher priority than the internal nodes, and each external node will have higher priority than all the external nodes to its right in the original tree.

Once this is done, the priority list is reordered to comply with the PES execution scheme as follows. For each node in the list, its successor node in the tree is moved up to the next position following it. The process is done iteratively until no nodes can be moved up any more. In this way, when the predecessors of a node are all finished, the node will have a higher priority than any nodes without predecessors. The resulting list will correspond to a PES Original-Ordering schedule.

6.2.3 Longest-Processing-Time-First Schedule

In the Longest-Processing-Time-First scheduling, each node of the tree is assigned a weight equal to the sum of the execution times of all the nodes from the node to the root node. The procedure is a recursive one. When a node is visited, its son node is assigned a weight equal to the sum of the weight of the node and the execution time of the son. Then the left and the right sub-trees of the node are
traversed recursively.

After the weight assignment process is done, the nodes are sorted in descending order of weight. The resulting node list is then reordered to simulate PES execution. For each node in the list, its successor node in the tree is moved up to the position in the list immediately following it. This process is done repeatedly until no nodes can be moved up any more. The resulting list is the priority list for the Longest-Processing-Time-First scheduling. The nodes corresponding to the operators in the PES which has the longest estimated execution time will have the highest priority in the list.

6.2.4 Shortest-Processing-Time-First Schedule

In the Shortest-Processing-Time-First scheduling, the weight assignment procedure is first done exactly the same way as that in the Longest-Processing-Time-First scheduling, i.e. the weight of a node is the sum of the execution times of all the nodes from the node to the root node. However, an additional process is needed before the nodes are sorted: the weights of the external nodes are negated and the weights of the internal nodes are set to a large negative number. The reason for doing this is that after the nodes
are sorted into descending order of weight, the external nodes will have higher priority than the internal nodes, and the external nodes with a shorter distance to the root node will have higher priority than those with longer ones.

After the negation procedure, the nodes are sorted into descending order of weight, and then the same reordering process as in the Longest-Processing-Time-First scheduling is applied. The resulting node list is the priority list for the Shortest-Processing-Time-First scheduling.

6.2.5 Highest-Level-Number-First Schedule

To simulate the Highest-Level-Number-First scheduling, each node in the tree is given a weight equal to the number of nodes between it and the root node. The weight of the root node is set to zero. A recursive procedure is called to assign weights to the nodes. The weight of a node is calculated by 1 plus the weight of its parent node.

After the weight assignment procedure is done, the nodes are sorted into descending order of weight. The node list is then reordered so that each internal node will immediately follow its higher priority son node in the list without changing the relative order of the external nodes.
The resulting list is the priority list of the Highest-Level-Number-First schedule. The node with the highest level number and its successor nodes have the highest priority.

6.2.6 Lowest-Level-Number-First Schedule

First, the procedure for assigning weight in Highest-Level-Number-First scheduling is used here for the Lowest-Level-Number-First scheduling. Next, the negation procedure used in the Shortest-Processing-Time-First scheduling is applied to the nodes in the tree.

After the weight for each node is assigned, the nodes are sorted into descending order of weight. The node list is then reordered by the procedure described in the Highest-Level-Number-First scheduling. The resulting list is the priority list of the Lowest-Level-Number-First schedule.

6.2.7 Simulation of Multi-processor Execution
In this part, the simulation program asks for the number of processors and then performs the simulation of execution of the tree generated above, with the five different scheduling algorithms. A variable called "TIMER" is used to keep track of the progress in time. At the beginning of execution, or when a node is finished execution by a processor, each free processor in turn searches the priority list to find a node which has not been taken and is ready for execution, i.e. all of its predecessor nodes are finished. If more than one such node exists, the one with the highest priority will be taken.

After all free processors have taken some nodes, or when there are no nodes ready to be taken, the remaining execution time of the nodes being executed is compared. The timer is then incremented by the smallest remaining execution time among the nodes being executed. This shows that, in the simulation, time has elapsed and some processor has completed the execution of some node. Then the process described above for assigning free processors to nodes is applied again.

The above procedure is repeated until all processors have become free. At that time, the timer contains the total execution time for the binary tree. In the simulation program, the execution of the binary tree is simulated for
the five different scheduling algorithms.

The output of the simulation program shows the execution time of the tree by the five scheduling strategies. It also shows the Gantt charts for the five different schedules. In the Gantt charts, the execution of a node is represented by its node number, followed by a number of hyphens representing the length of its execution. A sample output of the simulation program is shown in Appendix A.

6.2.8 Results of Simulation

The simulation program described above will generate a binary tree and perform the simulation of the execution of the tree. The program was modified so that it would input a number and repeat the above simulation as many times as specified by the input.

There are two varying parameters in the simulation runs: the number of nodes in the tree and the number of processors. Simulation runs were done with the number of processors ranging from 2 to 5, and the number of external nodes from 3 to 10. For each pair of these parameter values, 3000 binary trees were generated and simulated for
execution with five different scheduling algorithms.

The average of the execution time for each set of 3000 cases is used to compare the performance of the scheduling algorithms. The results of the simulation runs are summarized in Tables 1 through 4. The results are plotted in Figures 6.1 through 6.4. The average execution time is plotted against the number of external nodes in the tree. Figures 6.1 through 6.4 are simulation results for 5-processor through 2-processor systems, respectively.

From the simulation results, we can conclude that the Longest-Processing-Time-First scheduling is the best among the five, and the Shortest-Processing-Time-First is the worst. The five scheduling algorithms can be ranked from lowest execution time (best performance) to highest (worst performance):

\[ \text{LPTF} < \text{HLNF} < \text{OO} < \text{LLNF} < \text{SPTF} \]

Intuitively, we know that delaying the task which is farthest from completion will definitely delay the total completion. Therefore, assigning the highest priority to such tasks should result in a better schedule than otherwise. The simulation results have confirmed this.
Table 1  Simulation Results of PES Scheduling Algorithms for a 5-processor System

<table>
<thead>
<tr>
<th>Number of External Nodes</th>
<th>PES Scheduling Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ORG</td>
</tr>
<tr>
<td>10</td>
<td>28.7409</td>
</tr>
<tr>
<td>9</td>
<td>26.5553</td>
</tr>
<tr>
<td>8</td>
<td>24.0893</td>
</tr>
<tr>
<td>7</td>
<td>21.5296</td>
</tr>
<tr>
<td>6</td>
<td>19.0710</td>
</tr>
<tr>
<td>3</td>
<td>10.4243</td>
</tr>
<tr>
<td>Number of External Nodes</td>
<td>PES Scheduling Algorithms</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td></td>
<td>ORG</td>
</tr>
<tr>
<td>10</td>
<td>31.5153</td>
</tr>
<tr>
<td>8</td>
<td>25.7326</td>
</tr>
<tr>
<td>5</td>
<td>16.8686</td>
</tr>
<tr>
<td>3</td>
<td>10.3916</td>
</tr>
</tbody>
</table>
Table 3  Simulation Results of PES Scheduling Algorithms for a 3-processor System

<table>
<thead>
<tr>
<th>Number of External Nodes</th>
<th>PES Scheduling Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ORG</td>
</tr>
<tr>
<td>10</td>
<td>36.4809</td>
</tr>
<tr>
<td>9</td>
<td>32.8490</td>
</tr>
<tr>
<td>8</td>
<td>29.3503</td>
</tr>
<tr>
<td>7</td>
<td>25.6056</td>
</tr>
<tr>
<td>5</td>
<td>18.2213</td>
</tr>
<tr>
<td>3</td>
<td>10.3463</td>
</tr>
</tbody>
</table>
Table 4 Simulation Results of PES Scheduling Algorithms for a 2-processor System

<table>
<thead>
<tr>
<th>Number of External Nodes</th>
<th>PES Scheduling Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ORG</td>
</tr>
<tr>
<td>10</td>
<td>47.3100</td>
</tr>
<tr>
<td>9</td>
<td>42.2406</td>
</tr>
<tr>
<td>8</td>
<td>37.4553</td>
</tr>
<tr>
<td>7</td>
<td>32.2059</td>
</tr>
<tr>
<td>6</td>
<td>27.3129</td>
</tr>
<tr>
<td>5</td>
<td>22.2466</td>
</tr>
<tr>
<td>4</td>
<td>17.1273</td>
</tr>
<tr>
<td>3</td>
<td>11.6913</td>
</tr>
</tbody>
</table>
Figure 6.1 Simulation Results of PES Scheduling for a 5-Processor System
Figure 6.2 Simulation Results of PES Scheduling for a 4-Processor System
Figure 6.3 Simulation Results of PES Scheduling for a 3-Processor System
Figure 6.4 Simulation Results of PES Scheduling for a 2-Processor System
6.3 More on Scheduling Problems

6.3.1 The Assembly Line Scheduling Problem

In this section, we consider a less restrictive scheduling problem than the PES scheduling problem, namely, the assembly line scheduling problem. In this problem, we are given a set of tasks to be executed by a system with \( m \) identical processors. Each task assignment is non-preemptive, and each task is indivisible. The precedence relation between tasks can be represented by a tree. A task cannot be started until all of its predecessors have finished. Each task has a fixed, known execution time. The above conditions are the same as those in the PES scheduling problem, but we do not have the constraint of assigning a processor to the successor of a task as soon as the task is finished.

Examples of actual problems which fit this formulation include scheduling operations for the parallel execution of arithmetic expressions, scheduling of real-time tasks in an environment where task execution times can be predicted, and the scheduling of independent tasks [GRAH72, BRUN73]. In particular, we are interested in the scheduling for the parallel execution of arithmetic expressions.
In [HU61], an optimal scheduling algorithm was published for the assembly line scheduling problem in which all tasks are the same length. It uses the "longest path" algorithm scheduling algorithm. The algorithm assigns free processors to those available tasks which are farthest from the root of the tree at the time of assignment. Hu showed that the algorithm is optimal, i.e. the execution time of the schedule is minimal. However, it is not optimal for tasks that are of different length.

In [KAUF74], it is showed that the longest path algorithm remains "almost" optimal when arbitrary times are allowed for each task. In particular, the following relations hold:

Let $W$ be the total time required to process all of the tasks by the longest path algorithm.

Let $W_0$ be the minimal time in which all of the tasks can be processed by any nonpreemptive algorithm.

Let $W_p$ be the minimal time to process all of the tasks if arbitrary preemption of processors is allowed.

Then $W_p \leq W_0 \leq W \leq W_p + k - k/m$

where $m$ is the number of processors used by the algorithm and $k$ is an upper bound on task time.
In this section, however, we will use simulation to compare several assembly line scheduling algorithms for tasks with arbitrary length. The algorithms that we are considering all have the same order of time complexity as the longest path algorithm.

6.3.2 Comparing Several Assembly Line Scheduling Algorithms

The longest path algorithm (LP) discussed in [KAUF74] schedules tasks of which the precedence relations form a tree. The precedence tree is such that the leaf-nodes are starting nodes, i.e. those without predecessors, and the root node is the final node, i.e. that without successors.

The simulation done in this section is particularly interested in comparing the "backward longest path" (BLP) scheduling algorithm, which considers the root node as the starting node and the leaf-nodes as the final nodes, with the LP scheduling algorithm. The BLP algorithm can be used even if the actual precedence relations among tasks are the same as those used by [KAUF74]. This can be shown as follows.

Suppose that there are n tasks: J1, J2, ..., Jn to be scheduled. Let Ji < Jj mean that Ji must be complete before Jj can be started. Let T1, T2, ..., Tn be the
execution time for tasks J1, J2, … , Jn. A schedule of J1, J2, … , Jn can be specified by a list of times ST1, ST2, … , STn, where STi is the starting time of task Ji, for 1 ≤ i ≤ n. With the above definition, the ending time of task Ji will be STi + Ti. A schedule for J1, J2, … , Jn is called valid if and only if, for any i, j such that Ji < Jj, STi + Ti ≤ STj.

i.e. in a valid schedule, the successors of a task can not be started until the task is complete.

Suppose that the relation < on J1, J2, … , Jn forms a tree, and if Ji < Jj & Ji < Jk, then either Jj < Jk or Jk < Jj. This specifies a tree structure precedence relation of which the leaf-nodes are starting nodes and the root node is the final node.

If we use the "backward longest path" algorithm to schedule J1, J2, … , Jn, we will get a valid backward schedule, ST1', ST2', … , STn', which satisfies the condition: STi + Ti ≤ STj, for any Jj < Ji. To get a valid actual schedule for the tasks, we can use the time difference between the ending time of a task in the backward schedule and the completion time of the whole backward schedule, as the starting time of that task in the "normal" schedule. More specifically, the actual schedule will be T-(ST1'+T1), T-(ST2'+T2), … , T-(STn'+Tn),
i.e. \( ST_i \) will be \( T - (ST_i' + Ti) \) for \( 1 \leq i \leq n \), where \( T \) is the completion time of the whole schedule.

To prove that this yields a valid actual schedule, we will show that for any \( i, j \), such that \( J_i < J_j \), we have \( ST_i + Ti < ST_j \).

**Proof** Suppose \( J_i < J_j \) in the original tree for some \( i, j \).

Then in the backward schedule, we have \( ST_j' + T_j < ST_i' \) by definition. Negating both sides, we get

\[-ST_i' < -(ST_j' + T_j).\]

Adding \( T \) to both sides, we get

\[T - (ST_i' + Ti - Ti) < T - (ST_j' + T_j).\]

Since \( T - (ST_j' + T_j) \) is used as \( ST_j \), by definition, the right hand side becomes \( ST_j \). The left hand side becomes

\[T - (ST_i' + Ti)] + Ti \]

by regrouping the terms, and hence

\[ST_i + Ti.\]

This means that \( ST_i + Ti < ST_j \) for \( J_i < J_j \).

Therefore, the schedule

\[T - (ST_1' + T_1), T - (ST_2' + T_2), \ldots, T - (ST_n' + T_n)\]

is valid for tasks \( J_1, J_2, \ldots, J_n \). Q.E.D.

Although the worst case analysis for the LP algorithm for tasks with arbitrary times is done in [KAUF74], the average performance of the algorithm is hard to analyze. The analysis for the BLP algorithm is even harder than that of the LP, because in a LP schedule, the number of processors being used at any moment is a decreasing function.
of time, whereas in the BLP schedule it changes irregularly. This is the motivation for using simulation to compare the average performance of the algorithms.

In the simulation runs, we also test several other algorithms: the highest level algorithm (HL), the backward highest level algorithm (BHL), the lowest level algorithm (LL), the backward lowest level algorithm (BLL), the shortest path algorithm (SP), and the backward shortest path algorithm (BSP).

The level number used in the HL algorithm and the LL algorithm is defined as the number of intermediate nodes between a node and the root node. The level number used in the BHL algorithm and the BLL algorithm is defined as the largest number of nodes between the node and the leaf-nodes. The HL and the BHL algorithms assign free processors to those available tasks which have the highest level number at the time of assignment. The LL and the BLL algorithms assign free processors to those available tasks which have the lowest level number at the time of assignment.

The trees generated for simulation are similar to those in the simulation of PES scheduling. However, each node in the trees may have two, one, or no son nodes, whereas in the PES scheduling simulation, each node has either two sons or
In the simulation, each node is assigned a weight, and the nodes are sorted into either ascending or descending order of weight. The resulting ordering of nodes represents the priority list of the schedule. The weight assigned to a node is calculated according to the algorithms. In the LP and the SP algorithms, the weight of a node is equal to the sum of its execution time and the weight of its parent node. This recursive definition is used by the simulation program to calculate the weight recursively. In the HL and the LL algorithms, the weight of a node is equal to one plus the weight of its parent node. In the BLP and BSP algorithms, the weight of a node is equal to the sum of its execution time and the largest of the weight of its son nodes. In the BHL and BLL algorithms, the weight of a node is equal to one plus the largest of the weight of its son nodes. The LP, BLP, HL, and BHL algorithms sort the nodes into descending order of weight. The SP, BSP, LL, and BLL algorithms sort the nodes into ascending order of weight.

In order to use the same generated tree for simulation of execution by different algorithms, the LP, SP, HL, and LL algorithms use the original connectivity matrix to schedule the execution, and the BLP, BSP, BHL, and BLL algorithms use the transposed connectivity matrix to schedule the
The same execution simulation procedure used in the PES simulation is used here, since the scheduling is already done by the priority list.

6.3.3 Results of Simulation

The simulation program listing and some sample output are included in Appendix B. Simulation runs were done with the number of processors ranging from 2 to 5, and the number of nodes from 10 to 19. For each pair of these parameter values, 3000 trees were generated and simulated for execution by the eight different scheduling algorithms.

The average of the execution time for each set of 3000 cases simulated is shown in Tables 5 - 8. Figures 6.5 - 6.8 show the average execution time plotted against the number of nodes with the eight scheduling algorithms. Figures 6.5 through 6.8 are for 5-processor through 2-processor configurations, respectively.

The results of simulation show that the backward longest path algorithm is the best, and the shortest path algorithm is the worst. The algorithms can be ranked from lowest average execution time (best performance) to highest (worst performance):
Table 5  Simulation Results of Assembly Line Scheduling Algorithms for a 5-processor System

<table>
<thead>
<tr>
<th>Number of Nodes</th>
<th>Scheduling Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LP</td>
</tr>
<tr>
<td>19</td>
<td>29.73</td>
</tr>
<tr>
<td>18</td>
<td>29.17</td>
</tr>
<tr>
<td>16</td>
<td>27.38</td>
</tr>
<tr>
<td>14</td>
<td>25.32</td>
</tr>
<tr>
<td>12</td>
<td>23.33</td>
</tr>
<tr>
<td>10</td>
<td>20.91</td>
</tr>
</tbody>
</table>
Table 6  Simulation Results of Assembly Line Scheduling Algorithms for a 4-processor System

<table>
<thead>
<tr>
<th>Number of Nodes</th>
<th>Scheduling Algorithms</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>LP</td>
<td>BLP</td>
<td>SP</td>
<td>BSP</td>
<td>HL</td>
<td>BHL</td>
<td>LL</td>
<td>BLL</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>29.98</td>
<td>29.98</td>
<td>36.17</td>
<td>31.68</td>
<td>30.55</td>
<td>30.30</td>
<td>35.29</td>
<td>31.38</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>29.19</td>
<td>29.20</td>
<td>34.87</td>
<td>30.66</td>
<td>29.67</td>
<td>29.49</td>
<td>33.91</td>
<td>30.41</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>28.33</td>
<td>28.34</td>
<td>33.21</td>
<td>29.40</td>
<td>28.68</td>
<td>28.55</td>
<td>32.28</td>
<td>29.19</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>27.59</td>
<td>27.59</td>
<td>31.75</td>
<td>28.37</td>
<td>27.88</td>
<td>27.75</td>
<td>30.92</td>
<td>28.21</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>25.57</td>
<td>25.57</td>
<td>28.30</td>
<td>25.89</td>
<td>25.73</td>
<td>25.66</td>
<td>27.58</td>
<td>25.81</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>24.43</td>
<td>24.43</td>
<td>26.35</td>
<td>24.64</td>
<td>24.54</td>
<td>24.49</td>
<td>25.70</td>
<td>24.58</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>23.24</td>
<td>23.24</td>
<td>24.46</td>
<td>23.34</td>
<td>23.32</td>
<td>23.29</td>
<td>24.02</td>
<td>23.30</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>22.07</td>
<td>22.07</td>
<td>22.67</td>
<td>22.11</td>
<td>22.10</td>
<td>22.09</td>
<td>22.41</td>
<td>22.09</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>20.74</td>
<td>20.74</td>
<td>20.99</td>
<td>20.76</td>
<td>20.76</td>
<td>20.75</td>
<td>20.85</td>
<td>20.75</td>
</tr>
</tbody>
</table>
Table 7  Simulation Results of Assembly Line Scheduling Algorithms for a 3-processor System

<table>
<thead>
<tr>
<th>Number of Nodes</th>
<th>LP</th>
<th>BLP</th>
<th>SP</th>
<th>BSP</th>
<th>HL</th>
<th>BHL</th>
<th>LL</th>
<th>BLL</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>31.88</td>
<td>31.71</td>
<td>41.00</td>
<td>36.20</td>
<td>33.20</td>
<td>32.76</td>
<td>40.23</td>
<td>35.80</td>
</tr>
<tr>
<td>18</td>
<td>30.52</td>
<td>30.41</td>
<td>38.99</td>
<td>34.48</td>
<td>31.79</td>
<td>31.34</td>
<td>38.17</td>
<td>34.07</td>
</tr>
<tr>
<td>17</td>
<td>29.54</td>
<td>29.46</td>
<td>37.34</td>
<td>32.91</td>
<td>30.63</td>
<td>30.28</td>
<td>36.50</td>
<td>32.53</td>
</tr>
<tr>
<td>16</td>
<td>28.19</td>
<td>28.12</td>
<td>35.18</td>
<td>31.10</td>
<td>29.15</td>
<td>28.80</td>
<td>34.38</td>
<td>30.74</td>
</tr>
<tr>
<td>15</td>
<td>27.07</td>
<td>20.02</td>
<td>33.40</td>
<td>29.52</td>
<td>27.88</td>
<td>27.64</td>
<td>32.59</td>
<td>29.14</td>
</tr>
<tr>
<td>14</td>
<td>25.89</td>
<td>25.87</td>
<td>31.44</td>
<td>27.90</td>
<td>26.59</td>
<td>26.39</td>
<td>30.61</td>
<td>27.55</td>
</tr>
<tr>
<td>12</td>
<td>23.65</td>
<td>23.64</td>
<td>27.58</td>
<td>24.75</td>
<td>24.06</td>
<td>23.93</td>
<td>26.79</td>
<td>24.53</td>
</tr>
<tr>
<td>11</td>
<td>22.34</td>
<td>22.34</td>
<td>25.37</td>
<td>23.06</td>
<td>22.68</td>
<td>22.54</td>
<td>24.65</td>
<td>22.86</td>
</tr>
<tr>
<td>10</td>
<td>20.89</td>
<td>20.89</td>
<td>23.00</td>
<td>21.31</td>
<td>21.10</td>
<td>21.03</td>
<td>22.41</td>
<td>21.18</td>
</tr>
<tr>
<td>Number of Nodes</td>
<td>Scheduling Algorithms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LP</td>
<td>BLP</td>
<td>SP</td>
<td>BSP</td>
<td>HL</td>
<td>BHL</td>
<td>LL</td>
<td>BLL</td>
</tr>
<tr>
<td>19</td>
<td>41.64</td>
<td>41.24</td>
<td>50.60</td>
<td>46.30</td>
<td>42.94</td>
<td>42.65</td>
<td>49.86</td>
<td>45.95</td>
</tr>
<tr>
<td>18</td>
<td>39.35</td>
<td>38.96</td>
<td>47.87</td>
<td>43.70</td>
<td>40.61</td>
<td>40.35</td>
<td>47.18</td>
<td>43.39</td>
</tr>
<tr>
<td>17</td>
<td>37.07</td>
<td>36.72</td>
<td>45.51</td>
<td>41.42</td>
<td>38.25</td>
<td>38.09</td>
<td>44.75</td>
<td>41.07</td>
</tr>
<tr>
<td>16</td>
<td>35.06</td>
<td>34.74</td>
<td>42.84</td>
<td>39.21</td>
<td>36.24</td>
<td>36.02</td>
<td>42.23</td>
<td>38.88</td>
</tr>
<tr>
<td>15</td>
<td>33.03</td>
<td>32.75</td>
<td>40.52</td>
<td>36.89</td>
<td>34.17</td>
<td>33.97</td>
<td>39.81</td>
<td>36.55</td>
</tr>
<tr>
<td>14</td>
<td>30.79</td>
<td>30.57</td>
<td>37.81</td>
<td>34.48</td>
<td>31.85</td>
<td>31.68</td>
<td>37.17</td>
<td>34.05</td>
</tr>
<tr>
<td>13</td>
<td>28.74</td>
<td>28.49</td>
<td>35.28</td>
<td>32.00</td>
<td>29.68</td>
<td>29.50</td>
<td>34.58</td>
<td>31.63</td>
</tr>
<tr>
<td>12</td>
<td>26.71</td>
<td>26.53</td>
<td>32.79</td>
<td>29.65</td>
<td>27.61</td>
<td>27.47</td>
<td>32.17</td>
<td>29.31</td>
</tr>
<tr>
<td>11</td>
<td>24.54</td>
<td>24.41</td>
<td>29.94</td>
<td>27.08</td>
<td>25.35</td>
<td>25.23</td>
<td>29.33</td>
<td>26.78</td>
</tr>
<tr>
<td>10</td>
<td>22.69</td>
<td>22.61</td>
<td>27.37</td>
<td>24.83</td>
<td>23.36</td>
<td>23.26</td>
<td>26.76</td>
<td>24.51</td>
</tr>
</tbody>
</table>
Figure 6.5 Simulation Results of Assembly Line Scheduling for a 5-Processor System
Figure 6.6 Simulation Results of Assembly Line Scheduling for a 4-Processor System
Figure 6.7 Simulation Results of Assembly Line Scheduling for a 3-Processor System
Figure 6.8 Simulation Results of Assembly Line Scheduling for a 2-Processor System

NUMBER OF PROCESSORS = 2
7.1 Summary

In this dissertation, we have proposed the PES scheme in detecting parallelism in ordinary high level language programs at the statement and the block levels. The distinguishing features of this approach are:

- each PES is executed by an independent processor asynchronously at its maximum speed, and no unnecessary delay will occur;
- the intermediate store and fetch of partial results is kept to a minimum;
- parallelism among statements is detected and utilized, even if the statements are not independent of each other;
- the approach is flexible, because the number of processors in the system does not have to be known at
  the compilation time; and
- it is easy to implement.

The PES compiling algorithm and the block level parallelism detection algorithm are presented in Chapter 3.
The former takes an expression as input and generates the corresponding PES’s for it. The latter takes a block of
statements as input, checks the dependencies among the PES’s of the statements, and schedules the PES’s for parallel
execution. Both algorithms are of moderate complexity and can be implemented without too much effort.

Code generation techniques are discussed. Several ways to eliminate some unnecessary overhead are described. With
the proposed code generation techniques, programs that have little parallelism will not be slowed down by the overhead,
while programs that have more parallelism can be completed faster.

Three different code optimization techniques have been discussed. The first method is to eliminate the redundant
portions of PES’s to reduce the program space requirement. The second is to distribute PES’s into different memory
modules to reduce memory access conflicts. The third is to
reorder the PES's in an execution stage to reduce the total completion time.

We have shown that the PES approach can be used on high-level language computers for parallel processing, which is very rarely mentioned in high-level language computer architecture except [MOON77], and we have shown the internal language constructs for representing expressions and concurrent statements to allow parallel processing. We have also discussed the internal language constructs for IF, REPEAT, WHILE, and LOOP statements to facilitate try-ahead processing.

In chapter 4, we presented the architecture of a high level language computer using parallel processing and try-ahead processing. The high-level language computer consists of independent processors operating asynchronously at their own maximum speed for language processing and multiple identical processors for parallel and try-ahead processing. The increase in speed is achieved by the pipelining effect of the independent processors, as well as by the parallel computations by the multiple processors.

To implement the PES approach in microprocessor-based systems, we have proposed a machine organization that can execute PES's efficiently and effectively. In chapter 5,
the design of a multi-microprocessor system was presented. The system uses Am2900 family bit-slice microprogrammable microprocessors. This design suggests that the PES approach to parallel processing can be implemented on a low cost system with moderate effort.

In chapter 6, we used simulation to show the effect of the third code optimization technique on the total execution time. The longest-processing-time scheduling algorithm has been shown to be a simple but still effective algorithm. The simulation was then performed on a related but more general scheduling problem, the assembly line scheduling with tasks of arbitrary times. The main interest was to compare the performance of the leaves-to-root algorithms with the root-to-leaves algorithms. We have shown that both algorithms can be used for scheduling whether the actual task precedence relations form a leaves-to-root tree or a root-to-leaves tree. The simulation results showed that on the average the backward (root-to-leaves) algorithms perform better than the normal (leaves-to-root) algorithms, provided that the scheduling criteria are the same. The simulation programs are included in Appendix A and Appendix B, for the PES scheduling and the assembly line scheduling, respectively.
7.2 Suggestions for Future Work

We have proposed the PES approach to exploiting parallelism in ordinary high-level language programs, designed a multi-microprocessor system, proposed the try-ahead processing in high-level language computers, and presented the architecture of a parallel execution high-level language computer.

Although we have used simulations to show the performance of several PES scheduling algorithms, the analysis of these algorithms about their average, worst case, and best case performance will provide us a better means for predicting the results.

A complete design and simulation of the multi-microprocessor system will be valuable. The simulator can test the real high level language programs compiled with the PES approach and will be able to obtain some realistic measures of the speed increase due to parallel execution and the system overhead that may occur.

The parallel execution high-level language computer architecture has been presented largely at the function level. The detailed design of each processor should be done. A complete design of the computer may uncover some problems
that are not obvious.

Also, a simulator of the complete high-level language computer will be valuable in detecting design errors and measuring the performance enhancement due to parallel processing and try-ahead processing.

The simulator should run a variety of real application programs in order to get realistic measures. The simulation should also be performed to find the optimal speed configuration of each processor to avoid the possibility that some processors are unnecessarily fast and others are too slow and become the bottleneck of the system.

Finally, the multi-microprocessor system and the parallel execution high-level language computer should be implemented with actual hardware.
APPENDIX A

SIMULATION PROGRAMS FOR SEVERAL PES SCHEDULING ALGORITHMS

The listing of simulation program PESSCH.PAS:

(*$T-*

(* This is a simulation program to simulate five different PES scheduling algorithms. The input to this program includes: the number of external nodes of the tree to be generated, the number of processors for execution, and the upper bound of the task execution time. A binary tree will be generated and the nodes of the tree correspond to tasks to be executed.

The output of the program includes the drawing of the binary tree, the completion time of each scheduling algorithm, and the Gantt charts for the different schedules.

The five PES scheduling algorithms to be simulated are: the longest-processing-time-first algorithm, the shortest-processing-time-first algorithm, the highest-level-number-first algorithm,
the lowest-level-number-first algorithm, and the original-ordering algorithm.

```
class

const

line_num=120;
line_len=60;
numpes=30;
star='*';
horiz=' -';
vert='!';
maxintg=9999999;

var

tree:array[1..line_num,1..line_len] of char;
pes_num:integer;
pesgen,nodegen,nodeind:integer;
pesptr:array[1..numpes] of integer;
type3:array[1..numpes] of integer;
node_num:array[1..numpes] of integer;
leng:array[1..numpes] of integer;
lptr:array[1..numpes] of integer;
rptr:array[1..numpes] of integer;
lw,rw,lw2,rw2:array[1..numpes] of integer;
weight:array[0..numpes] of integer;
pr1lst:array[1..numpes] of integer;
unsort:array[1..numpes] of integer;
length:array[0..numpes] of integer;
i,j,k,ii:integer;
umline:integer;
termnode:integer;
pes_num_2_1:integer;
busy,jtaken:array[1..numpes] of boolean;
job,ptime:array[1..numpes] of integer;
nextime,timer,jdonenum,jtakennum:integer;
conn,tconn:array[1..numpes,1..numpes] of boolean;
pjob,ppleng:array[1..numpes,1..numpes] of integer;
pindx:array[1..numpes] of integer;
num_of_proc,leng_max:integer;
```

Function chrr(num: integer): char;
    (** convert a numeric digit to ascii code ***)
    begin
        chrr:=chr(num+48);
    end;

Function rannode(seed: integer): integer;
    (** random number generator for node number ***)
    var
ran: integer;
ran2: real;
i,j: integer;

begin
ran:=trunc(time*random(time));
ran:=ran - ran div 1000 * 1000;
i:=ran-ran div 10 * 10;
for j:=1 to i do
ran2:=random(time);
ran2:=random(time);
ran2:=(i/10) * ran2 + random(time) * (1-ran2);
ran:=trunc(ran2 * (pes_num - 1) + 1);
rannode:=ran;
end;

Function ranleng(seed:integer):integer;
(*** random number generator for node length ***)
begin
ranleng:=trunc(random(time)*leng_max + 1);
end;

Procedure insert(node:integer);
(* insert a new node with number 'node' into the tree *)
var
i,j,k: integer;
begin
i:=1;

(*** traverse the tree to find the place to insert ***)
while (node > node_num[i]) and (rptr[i] # 0) or
(node < node_num[i]) and (lptr[i] # 0) do
begin
if node > node_num[i] then
(*** go right ***)
i:=rptr[i]
else
(*** go left ***)
i:=lptr[i];
end;

(*** insert ***)
nodeind:=nodeind+1;
if node > node_num[i] then
begin
rptr[i]:=nodeind;
end;
(* * * set up for the new node * * *)
node_num[nodeind]:=node;
leng[nodeind]:=ranleng(time);
lptr[nodeind]:=0;
rptr[nodeind]:=0;

Function visit(nodein:integer):integer;
(* * * traverse the subtree with root nodein, and compute its left and right subtrees' width * * *)
begin
if node_num[nodein] >= pes_num
then
(* * * terminal nodes * * *)
begin
visit:=1;
end
else
(* * * not terminal node, recursive call * * *)
begin
lw[nodein]:=visit(lptr[nodein]);
rw[nodein]:=visit(rptr[nodein]);
visit:=lw[nodein]+rw[nodein]+1;
end;
end;

Function visit2(nodein:integer):integer;
(* * * traverse the subtree with root nodein, and compute its left and right subtrees' width * * *)
begin
if node_num[nodein] >= pes_num
then
(* * * terminal nodes * * *)
begin
visit2:=1;
end
else
(* * * not terminal node, recursive call * * *)
begin
lw2[nodein]:=visit2(lptr[nodein]);
rw2[nodein]:=visit2(rptr[nodein]);
visit2:=lw2[nodein]+rw2[nodein];
Procedure draw(nodein, row, col: integer);
    (** draw the tree.**) 
var 
i, j, left, right, lh, rh: integer;
begin
    tree[row, col] := chr(node_num[nodein]);
    left := lptr[nodein];
    right := rptr[nodein];
    lh := (rw[left] + 1) * 2;
    rh := (lw[right] + 1) * 2;

    (** draw the horizontal line to the left of nodein ***)
    for i := col - lh to col - 1 do
        tree[row, i] := horiz;

    (** draw the horizontal line to the right of nodein *)
    for i := col + 1 to col + rh do
        tree[row, i] := horiz;

    (** print the length of left and right branch ***)
    tree[row + 1, col - lh] := chr(leng[left]);
    tree[row + 1, col + rh] := chr(leng[right]);

    (** draw the vertical line for the left branch ***)
    for i := row + 2 to row + leng[left] * 2 - 1 do
        tree[i, col - lh] := vert;

    (** draw the vertical line for the right branch ***)
    for i := row + 2 to row + leng[right] * 2 - 1 do
        tree[i, col + rh] := vert;

    (** draw left subtree ***)
    if node_num[left] >= pes_num
        then
            (** terminal node ***)
            begin
                tree[row + leng[left] * 2, col - lh] :=
                    chr(node_num[left] - node_num[left] div 10 * 10);
                if (node_num[left] div 10 > 0)
                    then
                        tree[row + leng[left] * 2, col - lh - 1] :=
                            chr(node_num[left] div 10);
                        if (row + leng[left] * 2 > numline)
                            then
                                numline := row + leng[left] * 2;
            end
        else
            end
Procedure out;
  (** output the tree ***)
  var
    i,j,k:integer;
  begin
    (** output the tree ***)
    rewrite(output,'output ','');
    for i:=1 to numline do
      begin
        for j:=1 to linelen do
          begin
            write(tree[i,j]);
          end;
        writeln;
      end;
    page;
  end;

Procedure connvisit(node:integer);
(* set up an entry in the connectivity matrix for
"node" and its left and right sons *)

var
  l_node, r_node: integer;

begin
  if lptr[node] # 0
  then
    | l_node := lptr[node];
    | conn[node_num[l_node], node_num[node]] := true;
    | connvisit(l_node);
  end;

  if rptr[node] # 0
  then
    | r_node := rptr[node];
    | conn[node_num[r_node], node_num[node]] := true;
    | connvisit(r_node);
  end;
end;

Procedure connection;
  (* set up the connectivity matrix *)
begin
  for i := 1 to pes_num_2_l do
  for j := 1 to pes_num_2_l do
    | conn[i, j] := false;
  end;
  connvisit(1);
end;

Procedure gentree;
  (* generate a binary tree and
draw it on the output file *)
  (** Procedure generate ***)
begin
  (** initialization for tree drawing ***)
  numline := 0;

  for i := 1 to linenum do
  for j := 1 to linelen do
    | tree[i, j] := ' ';
  rewrite(ttyoutput);
  write(tty, 'number of external nodes:');
  break;
reset(tty);
read(tty,pes_num);

write(tty,'number of processors:');
break;
read(tty,num_of_proc);

write(tty,'maximum duration of each task:');
break;
read(tty,leng_max);

pes_num_2_1 := pes_num*2-1;
pesgen:=0;
nodegen:=0;
nodeind:=1;

(*** set up the first node ***)
type3[1]:=rannode(2);

node_num[1]:=type3[1];
leng[1]:=0;
lptr[1]:=0;
rptr[1]:=0;

(*** set up the rest nodes in the tree ***)
for i:=2 to pes_num-1 do
begin
  (** check if any duplicate type 3 node ***)
  repeat
    type3[i]:=rannode(time);
    j:=1;
    while (j < i-1) and (type3[i] # type3[j]) do
      j:=j+1;
    until (type3[i] # type3[j]);

    insert(type3[i]);
  end;

(*** generate the terminal nodes ***)
termnode:=pes_num;
for ii:=1 to pes_num-1 do
begin
  if lptr[ii] = 0
    then
      begin
        nodeind:=nodeind+1;
        node_num[nodeind]:=termnode;
        termnode := termnode+1;
        leng[nodeind]:=ranleng(time);
      end;

begin...
Procedure cpweigh(node:integer);
(* assigning the weight of the the left and the right
sons of "node" *)
(* the weight of a node is equal to its execution time
plus the weight of its parent *)
var
  l_node,r_node:integer;
begin
  if (lptr[node] # 0)
  then
    (* left son exists *)

  if rptr[node] = 0
  then
    begin
      nodeind:=nodeind+1;
      node_num[nodeind]:=termnode;
      termnode:=termnode+1;
      leng[nodeind]:=ranleng(time);
      lptr[nodeind]:=0;
      rptr[nodeind]:=0;
      rptr[ii]:=nodeind;
    end;
end;

(** dummy call to 'visit' to compute
 the width at each node ***)
i:=visit(1);
i:=visit2(1);

(** draw the tree ***)
draw(l,l,linslen div 2);

(** output the tree ***)
out;

(** create connection matrix ***)
connection;
end;
begin
| l_node:=lptr[node];
| weight[node_num[l_node]]:=weight[node_num[node]]+
| leng[l_node];
| length[node_num[l_node]]:=leng[l_node];
| (* recursive call *)
| cpweigh(l_node);
| end;
| length[node_num[l_node]]:=leng[l_node];
| (* recursive call *)
| cpweigh(l_node);
| end;
| if (rptr[node] # 0)
| then
| (* right son exists *)
| begin
| r_node:=rptr[node];
| weight[node_num[r_node]]:=weight[node_num[node]]+
| leng[r_node];
| length[node_num[r_node]]:=leng[r_node];
| (* recursive call *)
| cpweigh(r_node);
| end;
| end;

Procedure pes_org_weigh(node:integer);
(* assigning weight to nodes for original ordering
pes scheduling *)
var
  l_node,r_node:integer;
begin
  l_node := lptr[node];
  r_node := rptr[node];
  weight[node_num[node]] := 0;
  length[node_num[node]] := leng[node];
  if (lptr[l_node] = 0)
  then
    (* left son is a leaf node *)
    begin
      weight[node_num[l_node]] := pes_num_2_1 -
      node_num[node] * 2;
      length[node_num[l_node]] := leng[l_node];
    end
  else
    (* left son is not a leaf node, recursive on left son *)
    pes_org_weigh(l_node);
  if (lptr[r_node] = 0)
  then
    (* right son is a leaf node *)

begin
  weight[node_num[r_node]] := pes_num_2_l -
  node_num[node] * 2 - 1;
  length[node_num[r_node]] := leng[r_node];
end
else
  (* right son is not a leaf node, recursive on right son *)
pes_org_weigh(r_node);
end;

Procedure pes_org_weigh(r_node);
(* right son is not a leaf node, recursive on right son *)

Procedure neg_weight(node:integer);
(* negate the weight of terminal nodes and set the weight of non-terminal nodes to a large negative number *)
var
  l_node, r_node: integer;
begin
  l_node := lptr[node];
  r_node := rptr[node];
  if (l_node = 0) and (r_node = 0)
  then
    (* leaf node *)
    weight[node_num[node]] := - weight[node_num[node]]
  else
    (* not a leaf node *)
    begin
      weight[node_num[node]] := - maxint;
      neg_weight(l_node);
      neg_weight(r_node);
    end;
end;

Procedure pes_hlnf_weigh(node:integer);
(* assigning weight for hlnf pes scheduling *)
var
  l_node, r_node: integer;
begin
  l_node := lptr[node];
  r_node := rptr[node];
  if l_node # 0
  then
    (* left son exists *)
    begin
      weight[node_num[l_node]] := weight[node_num[node]] +1;
    end;
length[node_num[1_node]] := leng[1_node];
(* recursive on left son *)
pes_hlnf_weigh(1_node);
end;

if (r_node # 0)
then
(* right son exists *)
begin
weight[node_num[r_node]] := weight[node_num[node]] +1;
length[node_num[r_node]] := leng[r_node];
(* recursive on right son *)
pes_hlnf_weigh(r_node);
end;
end;

Procedure dsort;
(* sort nodes into descending order of weight *)
(* if two nodes have same weight, sort into ascending 
order of length *)
(* bubble sort is used *)
var
i,j,temp1,temp2,temp3:integer;
begin
weight[0]:=100000;
length[0]:=0;

for i:=2 to pes_num_2_1 do 
begin 
| temp1:=weight[i];
| temp2:=unsort[i];
| temp3:=length[i];
| j:=i-1;

| while (weight[j] < temp1) or ((weight[j] = temp1) and 
| (length[j] > temp3)) do 
| begin 
| weight[j+1] := weight[j];
| unsort[j+1]:=unsort[j];
| length[j+1]:=length[j]; 
| j:=j-1;
| end;
| weight[j+1]:=temp1;
| unsort[j+1]:=temp2;
| length[j+1]:=temp3;
end;
end;
Procedure pes_org_sort;
 (* sort nodes for pes original ordering scheduling *)
 var
   i,j,temp1,temp2,temp3:integer;
 begin
   weight[0]:=100000;
   length[0]:=0;
   for i:=2 to pes_num_2_l do
     begin
       temp1:=weight[i];
       temp2:=unsort[i];
       temp3:=length[i];
       j:=i-1;
       while (weight[j] < temp1) do
         begin
           weight[j+1] := weight[j];
           unsort[j+1]:=unsort[j];
           length[j+1]:=length[j];
           j:=j-1;
         end;
       weight[j+1]:=temp1;
       unsort[j+1]:=temp2;
       length[j+1]:=temp3;
     end;
 end;

Procedure pes_lptf;
 (* longest processing time first scheduling *)
 var
   i,j,k,templ,1,m,mm:integer;
 begin
   for i:=1 to pes_num_2_l do
     unsort[i]:=i;

   weight[node_num[1]]:=0;
   length[node_num[1]]:=0;
   cpweigh(1);
   dsort;

   for i:=1 to pes_num_2_l do
     begin
       if unsort[i] >= pes_num
         then
           begin
             j:=i;
             while j#0 do
               begin
                 k:=1;
               end;
           end;
while not(conn[unsort[j],k]) and k<pes_num_2_l do
k:=k+1;
if not(conn[unsort[j],k])
then j:=0
else
begin
l:=k;
m:=j;
while (unsort[m]#1) and (m<pes_num_2_l) do
m:=m+1;
if (unsort[m]=1)
then
begin
templ:=length[m];
for mm:=m-1 downto j+1 do
begin
length[mm+1]:=length[mm];
unsort[mm+1]:=unsort[mm];
end;
unsort[j+1]:=1;
length[j+1]:=templ;
j:=j+1;
end
else
j:=0;
end;
end;

(* output the priority list *)
writeln;
write('pes lptf schedule ');
writeln('(longest-processing-time-first) ');
write(' priority list: ' :20);
for i:=1 to pes_num_2_l do
begin
prtlst[i]:=unsort[i];
write(prtlst[i]:3);
end;
writeln;
end;

Procedure pes_sptf;
(* shortest-processing-time-first scheduling *)
var
  i, j, k, temp1, l, m, mm: integer;
begin
  for i := 1 to pes_num_2_l do
    unsort[i] := i;
    weight[node_num[1]] := 0;
    length[node_num[1]] := 0;
    cpweigh(1);
    neg_weight(1);
    dsort;
  for i := 1 to pes_num_2_l do
    begin
      if unsort[i] >= pes_num
        then
          begin
            j := i;
            while j # 0 do
              begin
                k := 1;
                while not(conn[unsort[j], k]) and
                  (k < pes_num_2_l) do
                  begin
                    k := k + 1;
                    if not(conn[unsort[j], k])
                      then
                        j := 0
                      else
                      begin
                        begin
                          l := k;
                          m := j;
                          while (unsort[m] != 1) and (m < pes_num_2_l) do
                            begin
                              m := m + 1;
                              if (unsort[m] = 1)
                                then
                                  begin
                                    templ := length[m];
                                    for mm := m - 1 downto j + 1 do
                                      begin
                                        length[mm + 1] := length[mm];
                                        unsort[mm + 1] := unsort[mm];
                                      end;
                                    unsort[j + 1] := 1;
                                    length[j + 1] := templ;
                                    j := j + 1;
                                  end
                                else
                                  j := 0;
                            end
                      end
              end
        end;
(* output the priority list *)
writeln;
write('pes_sptf_schedule ');
writeln('shortest-processing-time-first');
write(' priority list: ' :20);
for i:=1 to pes_num_2_1 do
begin
  prtlst[i]:=unsort[i];
  write(prtlst[i]:3);
end;
writeln;
end;

Procedure pes_hlnf;
(* highest-level-number-first scheduling *)
var
  i,j,k,temp,1,m,mm:integer;
begin
  for i:=1 to pes_num_2_1 do
    unsort[i]:=i;

  weight[node_num[1]]:=0;
  length[node_num[1]]:=0;
  pes_hlnf weigh(1);
  dsort;

  for i:=1 to pes_num_2_1 do
  begin
    if unsort[i] > pes_num
    then
      begin
        j:=i;
        while j#0 do
        begin
          k:=1;
          while not(conn[unsort[j],k]) and
          (k<pes_num_2_1) do
            k:=k+1;
          if not(conn[unsort[j],k])
          then
            j:=0
          else
            begin
              l:=k;
              m:=j;
            end;

          end;
        end;
      end;
  end;
end;
while (unsort[m] ≠ 1) and (m < pes_num_2_1) do
m := m + 1;
if (unsort[m] = 1)
then
begin
  begin
    temp1 := length[m];
    for mm := m - 1 downto j + 1 do
      begin
        length[mm + 1] := length[mm];
        unsort[mm + 1] := unsort[mm];
      end;
    unsort[j + 1] := 1;
    length[j + 1] := temp1;
    j := j + 1;
  end
else
  j := 0;
end;
end;

(* output the priority list *)
writeln;
write('pes_hlnf schedule ');
writeln('(highest-level-number-first)');
write('priority list:');
for i := 1 to pes_num_2_1 do
begin
  prtlst[i] := unsort[i];
  write(prtlst[i]:3);
end;
writeln;

Procedure pes_llnf;
(* lowest-level-number-first scheduling *)
var
  i, j, k, templ, l, m, mm: integer;
begin
  for i := 1 to pes_num_2_1 do
  unsort[i] := i;
  weight[node_num[1]] := 0;
  length[node_num[1]] := 0;
  pes_hlnf_weigh(1);
  neg_weight(1);
dsort;
for i:=1 to pes_num_2_l do
begin
  if unsort[i] > pes_num
  then
    begin
      j:=i;
      while j#0 do
        begin
          k:=1;
          while not(conn[unsort[j], k]) and
            (k<pes_num_2_l) do
            k:=k+1;
          if not(conn[unsort[j], k])
            then
              j:=0
            else
              begin
                l:=k;
                m:=j;
                while (unsort[m]#1) and (m<pes_num_2_l) do
                  m:=m+1;
                if (unsort[m]=1)
                  then
                    begin
                      templ:=length[m];
                      for mm:=m-1 downto j+1 do
                        begin
                          length[mm+1]:=length[mm];
                          unsort[mm+1]:=unsort[mm];
                        end;
                      unsort[j+1]:=1;
                      length[j+1]:=templ;
                      j:=j+1;
                    end
                  else
                    j:=0;
                end;
              end;
    end;
(* output the priority list *)
writeln;
write('pes llnf schedule ');
writeln('(lowest-level-number-first) ');
write(' priority list:' :20);
for i:=1 to pes_num_2_l do
begin
Procedure pes_org;
   (* pes original ordering schedule *)
   var
     i,j,k,templ,1,m,mm:integer;
   begin
     for i:=1 to pes_num_2_1 do
       unsort[i]:=i;
     weight[node_num[1]]=0;
     length[node_num[1]]=0;
     pes_org_weigh(1);
     pes_org_sort;
     for i:=1 to pes_num_2_1 do
       if unsort[i] >= pes_num
       then
         j:=i;
         while j#0 do
           begin
             k:=1;
             while not(conn[unsort[j],k]) and
             (k<pes_num_2_1) do
               k:=k+1;
             if not(conn[unsort[j],k])
             then
               j:=0
             else
               begin
                 l:=k;
                 m:=j;
                 while (unsort[m]#1) and (m<pes_num_2_1) do
                   m:=m+1;
                 if (unsort[m]=1)
                 then
                   begin
                     templ:=length[m];
                     for mm:=m-1 downto j+1 do
                       begin
                         length[mm+1]:=length[mm];
                         unsort[mm+1]:=unsort[mm];
                       end;
                     end;
                   end;
                 end;
               end;
             end;
           end;
         end;
       end;
     end;
   end;
(* output the priority list *)
write ln;
write ln (pes org schedule (original ordering));
write ('priority list: ' :20);
for i:=1 to pes_num_2_l do
  begin
    prtlst[i]:= unsort[i];
    write (prtlst[i]:3);
  end;
write ln;
end;

Procedure exec(numproc:integer);
(* simulation of multi-processor execution *)
(* conn is the connectivity matrix for tasks *)
(* prtlst is the priority list of tasks *)
(* numproc is the number of processors *)
(* length is the execution time of tasks *)
begin
  for i:=1 to pes_num_2_l do
    for j:=1 to pes_num_2_l do
      begin
        tconn[i,j]:= conn[i,j];
        end;
  for i:=1 to pes_num_2_l do
    begin
      jtaken[i]:= false;
      end;
  timer:=0;
  jtakennum:=0;
jdonenum:=0;
for i:=1 to numproc do
  begin
    pindx[i]:=0;
    busy[i]:=false;
  end;
loop
i:=1;
while (i<=numproc) and (jtakennum < pes_num_2_1) do
  begin
    if not(busy[i])
      then
        begin
          for j:=1 to pes_num_2_1 do
            begin
              if not(jtaken[j])
                then
                  begin
                    for k:=1 to pes_num_2_1 do
                      begin
                        if tconn[k,prtlst[j]]
                          then goto 1;
                      end;
                    busy[i]:=true;
                    jtaken[j]:=true;
                    jtakennum:=jtakennum+1;
                    job[i]:=prtlst[j];
                    ptime[i]:=length[j];
                    pindx[i]:=pindx[i]+1;
                    pjob[i,pindx[i]]:=job[i];
                    pleng[i,pindx[i]]:=ptime[i];
                    goto 2;
                  end;
            end;
        end;
    i:=i+1;
  end;
for i:=1 to numproc do
  if not(busy[i])
    then
      begin
        i=i+1;
      end;
pindx[i] := pindx[i] + 1;
pjob[i,pindx[i]] := 0;
pleng[i,pindx[i]] := 0;
end;

nextime := maxintg;
for i := 1 to numproc do
  if busy[i] then
    if (nextime > ptime[i]) then
      begin
        nextime := ptime[i];
      end;
  end;

exit if jdonenum = pes_num_2_1;
timer := timer + nextime;

for i := 1 to numproc do
  if busy[i] then
    ptime[i] := ptime[i] - nextime;

for i := 1 to numproc do
  begin
    if busy[i] then
      begin
        if ptime[i] = 0 then
          begin
            busy[i] := false;
            jdonenum := jdonenum + 1;
            for j := 1 to pes_num_2_1 do
              tconn[job[i], j] := false;
          end;
        end else
          begin
            pleng[i, pindx[i]] := nextime;
          end;
      end;
  end;
Procedure outtime;
 (* output the total completion time *)
| begin
| writeln;
| write(' total execution time: ', timer:5);
| writeln;
| writeln;
| end;

Procedure gantt(numproc:integer);
 (* output the gantt chart of the schedule *)
| var
| i, j: integer;
| begin
| writeln;
| writeln(' gantt chart: ');
| for i:= 1 to numproc do
| begin
| writeln;
| write(' p', i: 2, ' : ');
| for j:= 1 to pleng[i] do
| begin
| | if pjob[i, j] = 0
| | then
| | | begin
| | | | for k:= 1 to pleng[i, j] do
| | | | | write(' - ');
| | | | end
| | | else
| | | begin
| | | | if pleng[i, j] # 0
| | | | then write(pjob[i, j]: 2);
| | | | for k:= 1 to pleng[i, j] - 1 do
| | | | | write(' - ');
| | | | end;
| | | end;
| | end;

(*** main program ***)
| begin
| writeln;write(' number of external nodes: ', pes_num: 2);
writeln;write('number of processors: ',num_of_proc:2);
writeln;
write('maximum duration of each task: ',leng_max:2);
writeln;
gentree;
pes_org;
exec(num_of_proc);
gantt(num_of_proc);
outtime;
pes_lptf;
exec(num_of_proc);
gantt(num_of_proc);
outtime;
pes_sptf;
exec(num_of_proc);
gantt(num_of_proc);
outtime;
pes_11nf;
exec(num_of_proc);
gantt(num_of_proc);
outtime;
pes_11nf;
exec(num_of_proc);
gantt(num_of_proc);
outtime;
end.
An example of the output of program PESSCH.PAS:

--- 4 ---
7 3
1 1
1 1
1 1
1 1
1 1

--- 6 ---
1 6 1
1 1 7
1 1 8
1 1 9

--- 7 ---
1 4 1
1 1 5
1 1 6
1 1 7
1 1 8

--- 8 ---
1 5 6
1 1 1
1 1 1
1 1 1
1 1 1
1 1 1

--- 9 ---
5 5 1
1 1 1
1 1 1
1 1 1
1 1 1
1 1 1

--- 10 ---
3 3 1
1 1 1
1 1 1
1 1 1
1 1 1
1 1 1

--- 11 ---
9 1 1
1 1 1
1 1 1
1 1 1
1 1 1
1 1 1
1 1 1
1 1 1
1 1 1
### PES ORG Schedule (Original Ordering)
**Priority List:** 10 1 3 4 11 2 12 9 13 5 6 14 15 7 16 8 17

**Gantt Chart:**

<table>
<thead>
<tr>
<th>Process</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>7 6</td>
</tr>
<tr>
<td>P2</td>
<td>9 6</td>
</tr>
<tr>
<td>P3</td>
<td>8 7 6 5 15</td>
</tr>
</tbody>
</table>

**Total Execution Time:** 30

### PES LPTF Schedule (Longest-Processing-Time-First)
**Priority List:** 11 2 1 3 4 12 13 5 6 14 10 9 17 9 7 16 15

**Gantt Chart:**

<table>
<thead>
<tr>
<th>Process</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>7 6</td>
</tr>
<tr>
<td>P2</td>
<td>9 6</td>
</tr>
<tr>
<td>P3</td>
<td>8 7 6 5 15</td>
</tr>
</tbody>
</table>

**Total Execution Time:** 28

### PES SPTF Schedule (Shortest-Processing-Time-First)
**Priority List:** 15 7 6 4 16 8 17 9 3 10 1 13 5 14 12 2 11

**Gantt Chart:**

<table>
<thead>
<tr>
<th>Process</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>1 5 7 8 14 5</td>
</tr>
<tr>
<td>P3</td>
<td>10 12 16 13</td>
</tr>
</tbody>
</table>

**Total Execution Time:** 33

### PES HLNF Schedule (Highest-Level-Number-First)
**Priority List:** 12 2 1 3 4 16 8 7 6 11 17 10 15 13 5 14 9

**Gantt Chart:**

<table>
<thead>
<tr>
<th>Process</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>17 6 5 15 3 2 7 14 16 11 12 1 3 4 8 7 6 10 9</td>
</tr>
<tr>
<td>P2</td>
<td>11 10 15 14 5 11 17 8 7 6 10 9 15 13 5 14 12 2 11</td>
</tr>
</tbody>
</table>

**Total Execution Time:** 39

### PES LLNF Schedule (Lowest-Level-Number-First)
**Priority List:** 9 3 4 17 1 15 7 6 13 5 14 12 2 15 8 11 17

**Gantt Chart:**

<table>
<thead>
<tr>
<th>Process</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>17 6 5 15 3 2 7 14 16 11 12 1 3 4 8 7 6 10 9</td>
</tr>
<tr>
<td>P2</td>
<td>11 10 15 14 5 11 17 8 7 6 10 9 15 13 5 14 12 2 11</td>
</tr>
</tbody>
</table>

**Total Execution Time:** 31
The listing of simulation program PESCMP.PAS:

(*$t-,d-*)

(* This is a simulation program to simulate five
different pes scheduling algorithms. The input
to this program include: the number of external
nodes of the tree to be generated, the number
of processors for execution, the upper bound
of the task execution time, and the number of
cases to be simulated. For each case, a binary
tree will be generated and the nodes of the
tree correspond to tasks to be executed.

Five scheduling algorithms will be simulated on
the generated tree and average completion time
of each algorithm is calculated.
The five pes scheduling algorithms to be simulated
are: the longest-processing-time-first algorithm,
the shortest-processing-time-first algorithm,
the highest-level-number-first algorithm,
the lowest-level-number-first algorithm, and
the original-ordering algorithm. *)

const
  linenum=120;
  linelen=60;
  numpes=30;
  star='*';
  horiz='-';
  vert='!';
  maxintg=9999999;

var
  tree:array[1..linenum,1..linelen] of char;
  pes_num:integer;
  pesgen,nodegen,nodeind:integer;
  pesptr:array[1..numpes] of integer;
  type3:array[1..numpes] of integer;
  node_num:array[1..numpes] of integer;
  leng:array[1..numpes] of integer;
  lptr:array[1..numpes] of integer;
  rptr:array[1..numpes] of integer;
  lw,rw,lw2, rw2:array[1..numpes] of integer;
  weight:array[0..numpes] of integer;
  prtlst:array[1..numpes] of integer;
  unsort:array[1..numpes] of integer;
  length:array[0..numpes] of integer;
i,j,k,ii:integer;
numline:integer;
termnode:integer;
pes_num_2_l:integer;
busy,jtaken:array[1..numpes] of boolean;
job,ptime:array[1..numpes] of integer;
nextime,timer,jdonenum,jtakennum:integer;
conn,tconn:array[1..numpes,1..numpes] of boolean;
pjob,pleng:array[1..numpes,1..numpes] of integer;
pindx:array[1..numpes] of integer;
num_of_proc,leng_max:integer;

org_time,lptf_time,sptf_time,hlnf_time,llnf_time:integer;
total_org_time,total_lptf_time,total_sptf_time,
total_hlnf_time,total_llnf_time:integer;
avg_org_time,avg_lptf_time,avg_sptf_time,avg_hlnf_time,
avg_llnf_time:real;
repet:integer;
um_of_cases:integer;

Procedure init_procedure;
|begin
|  total_org_time := 0;
|  total_lptf_time := 0;
|  total_sptf_time := 0;
|  total_hlnf_time := 0;
|  total_llnf_time := 0;
|end;

Function chrr(num:integer):char;
  (*** convert a numeric digit to ascii code ***)
|begin
|  chrr:=chr(num+48);
|end;

Function rannode(seed:integer):integer;
  (*** random number generator for node number ***)
var
  ran:integer;
  ran2:real;
  i,j:integer;
|begin
|  ran:=trunc(time*random(time));
|  ran:=ran - ran div 1000 * 1000;
|  i:=ran - ran div 10 * 10;
|  for j:=1 to i do
|    ran2:=random(time);
|    ran2:=random(time);
|    ran2:=(i/10)*ran2 + random(time)*(1-ran2);
ran:= trunc(ran2 * (pes_num - 1) + 1);
| rannode:=ran;
|end;

Function ranleng(seed:integer):integer;
  (** random number generator for node length ***)
|begin
| ranleng:=trunc(random(time)*leng_max + 1);
|end;

Procedure insert(node:integer);
  (** insert a new node with number 'node' into the tree **) var
  i,j,k:integer;
|begin
| i:=1;
| (** traverse the tree to find the place to insert **) while (node > node_num[i]) and (rptr[i] # 0) or
| (node < node_num[i]) and (lptr[i] # 0) do
| begin
| if node > node_num[i]
| then
| (** go right **) i:=rptr[i]
| else
| (** go left **) i:=lptr[i];
| end;
| (** insert **)
| nodeind:=nodeind+1;
| if node > node_num[i]
| then
| begin
| rptr[i]:=nodeind;
| end
| else
| begin
| lptr[i]:=nodeind;
| end;
| (** set up for the new node **)
| node_num[nodeind]:=node;
| leng[nodeind]:=ranleng(time);
| lptr[nodeind]:=0;
| rptr[nodeind]:=0;
Procedure connvisit(node:integer);
 (* set up an entry in the connectivity matrix for
 "node" and its left and right sons *)
 var
 l_node, r_node: integer;
 begin
 if lptr[node] # 0
 then
 (* left son exists *)
 begin
 l_node:= lptr[node];
 conn[node_num[l_node], node_num[node]] := true;
 connvisit(l_node);
 end;
 if rptr[node] # 0
 then
 (* right son exists *)
 begin
 r_node:= rptr[node];
 conn[node_num[r_node], node_num[node]] := true;
 connvisit(r_node);
 end;
 end;

Procedure connection;
 (* set up the connectivity matrix *)
 begin
 for i := 1 to pes_num_2_1 do
 for j := 1 to pes_num_2_1 do
 conn[i,j] := false;
 connvisit(1);
 end;

Procedure gentree;
 (** Procedure generate ***)
 (* generate a binary tree and
   draw it on the output file *)
 begin
 pesgen:= 0;
 nodegen:= 0;
nodeind:=1;

(*** set up the first node ***)
type3[1]:=rannode(2);
node_num[1]:=type3[1];
leng[1]:=0;
lptr[1]:=0;
rptr[1]:=0;

(*** set up the rest nodes in the tree ***)
for i:=2 to pes_num-1 do
begin
 (** check if any duplicate type 3 node ***)
 repeat
 | type3[i]:=rannode(time);
 | j:=1;
 | while (j < i-1) and (type3[i] # type3[j]) do
 | j:=j+1;
 | until (type3[i] # type3[j]);
 | insert(type3[i]);
end;

(*** generate the terminal nodes ***)
termnode:=pes_num;
for ii:=1 to pes_num-1 do
begin
 if lptr[ii] = 0
 then
 begin
 | nodeind:=nodeind+1;
 | node_num[nodeind]:=termnode;
 | termnode:=termnode+1;
 | leng[nodeind]:=ranleng(time);
 | lptr[nodeind]:=0;
 | rptr[nodeind]:=0;
 | lptr[ii]:=nodeind;
end;

 if rptr[ii] = 0
 then
 begin
 | nodeind:=nodeind+1;
 | node_num[nodeind]:=termnode;
 | termnode:=termnode+1;
 | leng[nodeind]:=ranleng(time);
 | lptr[nodeind]:=0;
 | rptr[nodeind]:=0;
end;
Procedure cpweigh(node:integer);
(* assigning the weight of the the left and the right
sons of "node" *)
(* the weight of a node is equal to its execution time
plus the weight of its parent *)
var
  l_node, r_node: integer;
begin
  if (lptr[node] # 0)
  then
    (* left node exists *)
  begin
    l_node := lptr[node];
    weight[node_num[l_node]] := weight[node_num[node]] +
    length[l_node];
    length[node_num[l_node]] := length[l_node];
    (* recursive on left son *)
    cpweigh(l_node);
  end;

  if (rptr[node] # 0)
  then
    (* right son exists *)
  begin
    r_node := rptr[node];
    weight[node_num[r_node]] := weight[node_num[node]] +
    length[r_node];
    length[node_num[r_node]] := length[r_node];
    (* recursive on right son *)
    cpweigh(r_node);
  end;
end;

Procedure pes_org_weigh(node:integer);
(* assigning weight to nodes for original ordering
pes scheduling *)
var
  l_node, r_node: integer;
begin
  l_node := lptr[node];
  r_node := rptr[node];

  weight[node_num[node]] := 0;
  length[node_num[node]] := leng[node];

  if (lptr[l_node] = 0)
    then
      (* left son is a leaf node *)
      begin
        weight[node_num[l_node]] := pes_num_2_1 -
          node_num[node] * 2;
        length[node_num[l_node]] := leng[l_node];
      end
    else
      (* left son is not a leaf node, recursive on left son *)
      pes_org_weigh(l_node);

  if (lptr[r_node] = 0)
    then
      (* right son is a leaf node *)
      begin
        weight[node_num[r_node]] := pes_num_2_1 -
          node_num[node] * 2 - 1;
        length[node_num[r_node]] := leng[r_node];
      end
    else
      (* right son is not a leaf node, recursive on right son *)
      pes_org_weigh(r_node);
end;

Procedure neg_weight(node:integer);
  (* negate the weight of terminal nodes and set the
  weight of non-leaf nodes to a large negative number *)
var
  l_node, r_node: integer;
begin
  l_node := lptr[node];
  r_node := rptr[node];

  if (l_node = 0) and (r_node = 0)
    then
      (* node is a leaf node *)
      weight[node_num[node]] := -weight[node_num[node]]
    else
Procedure pes_hlnf_weigh(node:integer);
 (* assigning weight for hlnf pes scheduling *)
 var
   l_node, r_node: integer;
 begin
   l_node := lptr[node];
   r_node := rptr[node];
   if l_node # 0
     then
       (* left son exists *)
       begin
       weight[node_num[l_node]] := weight[node_num[node]]+1;
       length[node_num[l_node]] := leng[l_node];
       (* recursive on left son *)
       pes_hlnf_weigh(l_node);
       end;
     if (r_node /= 0)
       then
       (* right son exists *)
       begin
       weight[node_num[r_node]] := weight[node_num[node]]+1;
       length[node_num[r_node]] := leng[r_node];
       (* recursive on right son *)
       pes_hlnf_weigh(r_node);
       end;
   end;
 end;

Procedure dsort;
 (* sort nodes into descending order of weight *)
 (* if two nodes have same weight, sort into ascending order of length *)
 (* bubble sort is used *)
 var
   i, j, templ, temp2, temp3: integer;
 begin
   weight[0] := 100000;


length[0] := 0;
for i := 2 to pes_num_2_l do
begin
  temp1 := weight[i];
  temp2 := unsort[i];
  temp3 := length[i];
  j := i - 1;
  while (weight[j] < temp1) or ((weight[j] = temp1) and (length[j] > temp3)) do
  begin
    weight[j + 1] := weight[j];
    unsort[j + 1] := unsort[j];
    length[j + 1] := length[j];
    j := j - 1;
  end;
  weight[j + 1] := temp1;
  unsort[j + 1] := temp2;
  length[j + 1] := temp3;
end;

Procedure pes_org_sort;
(* sort nodes for pes original ordering scheduling *)
var
  i, j, temp1, temp2, temp3: integer;
begin
  weight[0] := 100000;
  length[0] := 0;
  for i := 2 to pes_num_2_l do
  begin
    temp1 := weight[i];
    temp2 := unsort[i];
    temp3 := length[i];
    j := i - 1;
    while (weight[j] < temp1) do
    begin
      weight[j + 1] := weight[j];
      unsort[j + 1] := unsort[j];
      length[j + 1] := length[j];
      j := j - 1;
    end;
    weight[j + 1] := temp1;
    unsort[j + 1] := temp2;
Procedure pes_lptf;
(* longest processing time first scheduling *)
var
  i, j, k, temp1, l, m, mm: integer;
begin
  for i := 1 to pes_num_2_l do
    unsort[i] := i;
  weight[node_num[1]] := 0;
  length[node_num[1]] := 0;
  cpweig(1);
  dsort;
  for i := 1 to pes_num_2_l do
    begin
      if unsort[i] >= pes_num
        then
          begin
            j := i;
            while j # 0 do
              begin
                k := 1;
                while not (conn[unsort[j], k]) and
                  (k < pes_num_2_l) do
                  k := k + 1;
                if not (conn[unsort[j], k])
                  then
                    j := 0
                  else
                    begin
                      l := k;
                      m := j;
                      while (unsort[m] # 1) and (m < pes_num_2_l) do
                        m := m + 1;
                      if (unsort[m] = 1)
                        then
                          begin
                            temp1 := length[m];
                            for mm := m - 1 downto j + 1 do
                              begin
                                length[mm + 1] := length[mm];
                                unsort[mm + 1] := unsort[mm];
                              end;
                            unsort[j + 1] := 1;
                            length[j + 1] := temp1;
                          end;
                        end;
          end;
        end;
      end;
end;
PROCEDURE pes_sptf;
(* shortest-processing-time-first scheduling *)
VAR
  i,j,k,templ,1,m,mm:integer;
BEGIN
  FOR i:=1 TO pes_num_2_1 DO
    unsort[i]:=i;
    weight[node_num[1]]:=0;
    length[node_num[1]]:=0;
    cpweigh(1);
    neg_weight(1);
    dsort;
    FOR i:=1 TO pes_num_2_1 DO
      BEGIN
        IF unsort[i] >= pes_num THEN
          BEGIN
            j:=i;
            WHILE j#0 DO
              BEGIN
                k:=1;
                WHILE not(conn[unsort[j],k]) AND
                  (k<pes_num_2_1) DO
                  k:=k+1;
                IF not(conn[unsort[j],k]) THEN
                  j:=0
                ELSE
                  BEGIN
                    l:=k;
                    m:=j;
                    WHILE (unsort[m]#1) AND (m<pes_num_2_1) DO
                      l:=l+1;
                      IF not(conn[unsort[m],l]) AND
                        (l<pes_num_2_1) THEN
                        k:=l;
                      m:=m-1
                  j:=j+1;
                  END
              END
      END
END
m:= m+1;
if (unsort[m]=1)
then
begin
begin
length[m]: =length[m];
for mm:=m-1 downto j+1 do
begin
length[mm+1]: =length[mm];
unsort[mm+1]: =unsort[mm];
end;
unsort[j+1]: =1;
length[j+1]: =temp1;
j:=j+1;
end;
else
j:=0;
end;
end;

for i:=1 to pes_num_2_l do
begin
prt1st[i]:=unsort[i];
end;

Procedure pes_hlnf;
(* highest-level-number-first scheduling *)
var
i,j,k,temp1,1,m,mm:integer;
begin
for i:=1 to pes_num_2_l do
unsort[i]:=i;
weight[node_num[i]]:=0;
length[node_num[i]]:=0;
pes_hlnf_weigh(1);
dsort;

for i:=1 to pes_num_2_l do
begin
if unsort[i] >= pes_num
then
begin
j:=i;
while j#0 do
begin
k:=j;
while not(conn[unsort[j],k]) and
Procedure pes_linf;
  (* lowest-level-number-first scheduling *)
var
  i,j,k,temp1,l,m,mm:integer;
begin
  for i:=1 to pes_num_2_1 do
    unsort[i]:=i;
  weight[node_num[1]]:=0;
  length[node_num[1]]:=0;
  pes_hlnf_weigh(1);
  neg_weight(1);
end;
dsort;

for i:=1 to pes_num_2_1 do
begin
  if unsort[i] >= pes_num
  then
  begin
    j:=i;
    while j#0 do
      begin
        k:=1;
        while not(conn[unsort[j],k]) and
        (k<pes_num_2_1) do
          k:=k+1;
        if not(conn[unsort[j],k])
          then
            j:=0
          else
            begin
              l:=k;
              m:=j;
              while (unsort[m]#1) and (m<pes_num_2_1) do
                m:=m+1;
              if (unsort[m]=1)
                then
                  begin
                    templ:=length[m];
                    for mm:=m-1 downto j+1 do
                      begin
                        length[mm+1]:=length[mm];
                        unsort[mm+1]:=unsort[mm];
                      end;
                    unsort[j+1]:=1;
                    length[j+1]:=templ;
                    j:=j+1;
                  end
                else
                  j:=0;
                end;
            end;
  end;
end;

for i:=1 to pes_num_2_1 do
begin
  prtlst[i]:=unsort[i];
end;
end;
Procedure pes_org;
(* pes original ordering schedule *)
var
i,j,k,templ,1,m,mm:integer;
begin
for i:=1 to pes_num_2_l do
unsort[i]:=i;
weight[node_num[1]]:=0;
length[node_num[1]]:=0;
pes_org_weigh(1);
pes_org_sort;
for i:=1 to pes_num_2_l do
begin
if unsort[i] >= pes_num
then
begin
j:=i;
while j#0 do
begin
k:=1;
while not(conn[unsort[j],k]) and
(k<pes_num_2_l) do
k:=k+1;
if not(conn[unsort[j],k])
then
j:=0
else
begin
l:=k;
m:=j;
while (unsort[m]#1) and (m<pes_num_2_l) do
m:=m+1;
if (unsort[m]=1)
then
begin
templ:=length[m];
for mm:=m-1 downto j+1 do
begin
length[mm+1]:=length[mm];
unsort[mm+1]:=unsort[mm];
end;
unsort[j+1]:=1;
length[j+1]:=templ;
j:=j+1;
end
else
j:=0;
end;
end;
Procedure exec(numproc:integer);
(* simulation of multi-processor execution *)
(* conn is the connectivity matrix for tasks *)
(* prtlst is the priority list of tasks *)
(* numproc is the number of processors *)
(* length is the execution time of tasks *)
begin
for i:=1 to pes_num_2_1 do
  for j:=1 to pes_num_2_1 do
    tconn[i,j] := conn[i,j];
end;
for i:=1 to pes_num_2_1 do
  jtaken[i] := false;
end;
timer := 0;
jtakennum := 0;
donenum := 0;
for i:=1 to numproc do
  pindx[i] := 0;
  busy[i] := false;
end;
loop
  i:=1;
  while (i<=numproc) and (jtakennum < pes_num_2_1) do
    begin
      if not(busy[i])
        then
          begin
            begin
              for j:=1 to pes_num_2_1 do
                begin
                  if not(jtaken[j])
                    then
                      begin
                      end;
                  end;
              end;
            end;
          end;
    end;
  end;
end;
for k:=1 to pes_num_2_l do
begin
if tconn[k,prtlst[j]]
then goto 1;
end;

busy[i]:=true;
jtaken[j]:=true;
jtakennum:=jtakennum+1;
job[i]:=prtlst[j];
ptime[i]:=length[j];
pindx[i]:=pindx[i]+1;
pjob[i,pindx[i]]:=job[i];
pleng[i,pindx[i]]:=ptime[i];
goto 2;
end;

1:
end;
end;

2:
i:=i+1;
end;

for i:=1 to numproc do
if not(busy[i])
then
begin
pindx[i]:=pindx[i]+1;
pjob[i,pindx[i]]:=0;
pleng[i,pindx[i]]:=0;
end;

nextime:=maxintg;
for i:=1 to numproc do
if busy[i]
then
if (nextime > ptime[i])
then
begin
nextime:=ptime[i];
end;
exit if jdonenum = pes_num_2_l;
timer:=timer+nextime;
for i:=1 to numproc do 
if busy[i] then 
ptime[i] := ptime[i] - nextime;

for i:=1 to numproc do 
begin 
if busy[i] then 
begin 
if ptime[i] = 0 then 
begin 
busy[i] := false; 
jdonenum := jdonenum + 1;
end;
end;
end;

else 
begin 
pleng[i, pindx[i]] := nextime; 
end;
end;
end;

(*** main program ***)
begin 
init_procedure;
rewrite(ttyoutput);
write(tty, 'number of external nodes:'); 
break;
reset(tty);
read(tty, pes_num);
write(tty, 'number of processors:'); 
break;
read(tty, num_of_proc);
write(tty, 'maximum duration of each task:'); 
break;
read(tty,len_max);
write(tty,"number of cases simulated: ");
beg
read(tty,num_of_cases);
write(tty,num_of_cases);
write(tty,"number of external nodes: ",pes_num:2);
write(tty,"number of processors: ",num_of_proc:2);
write(tty,"maximum duration of each task: ",leng_max:2);
write(tty,"number of cases simulated: ",num_of_cases:4);
write(tty,

pes_num_2_1 := pes_num*2-1;

(* simulate "num_of_cases" different cases *)
for repet := 1 to num_of_cases do
| (* generate a binary tree *)
gentree;

| (* simulate the original-ordering scheduling *)
pes_org;
exec(num_of_proc);
org_time := timer;
total_org_time := total_org_time + org_time;

| (* simulate longest-processing-time-first schedule *)
pes_lptf;
exec(num_of_proc);
lptf_time := timer;
total_lptf_time := total_lptf_time + lptf_time;

| (* simulate shortest-processing-time-first schedule *)
pes_sptf;
exec(num_of_proc);
sptf_time := timer;
total_sptf_time := total_sptf_time + sptf_time;

| (* simulate the highest-level-number-first schedule *)
pes_hlnf;
exec(num_of_proc);
hlnf_time := timer;
total_hlnf_time := total_hlnf_time + hlnf_time;
(* simulate the lowest-level-number-first schedule *)
pes_llnf;
exec(num_of_proc);
llnf_time := timer;
total_llnf_time := total_llnf_time + llnf_time;
end;

(* calculate the average completion time for each scheduling algorithm *)
avg_org_time := total_org_time / num_of_cases;
avg_lptf_time := total_lptf_time / num_of_cases;
avg_sptf_time := total_sptf_time / num_of_cases;
avg_hlnf_time := total_hlnf_time / num_of_cases;
avg_llnf_time := total_llnf_time / num_of_cases;

write('average time:':15,avg_org_time:10:4,
    avg_lptf_time:10:4,avg_sptf_time:10:4,
    avg_hlnf_time:10:4,avg_llnf_time:10:4);
writeln;writeln;
end.
An example of the output of program PESCMP.PAS:

NUMBER OF EXTERNAL NODES: 10
NUMBER OF PROCESSORS: 5
MAXIMUM DURATION OF EACH TASK: 8
NUMBER OF CASES SIMULATED: 3000

<table>
<thead>
<tr>
<th>ORG</th>
<th>LPTF</th>
<th>SPTF</th>
<th>HLNF</th>
<th>LLNF</th>
</tr>
</thead>
</table>
APPENDIX B

SIMULATION PROGRAMS FOR SEVERAL ASSEMBLY LINE SCHEDULING ALGORITHMS

The listing of simulation program ALSSCH.PAS:

(*$t-,d-*)

(* This program simulates eight different assembly line scheduling algorithms:
the longest path algorithm,
the backward longest path algorithm,
the shortest path algorithm,
the backward shortest path algorithm,
the highest level algorithm,
the backward highest level algorithm,
the lowest level algorithm, and
the backward lowest level algorithm.

The inputs of this program are:
the number of nodes in the tree to be generated,
the number of processors, and
the upper bound of the task length.

The outputs of this program are:
the drawing of the tree generated,
the total completion times for each schedule, and
the gantt charts of the schedules. *)

const
  linenum=120;
  linelen=60;
  numpes=30;
  star= '*';
  horiz= '-';
  vert= '!';
  maxintg=9999999;

var
  tree: array[1..linenum,1..linelen] of char;
  pes_num: integer;
  pesgen,nodegen,nodeind: integer;
  pespstr: array[1..pesgen] of integer;
  type3: array[1..pesgen] of integer;
  node_num: array[1..pesgen] of integer;
  leng: array[0..pesgen] of integer;
  lptr: array[0..pesgen] of integer;
  rptr: array[0..pesgen] of integer;
  lw, rw, lw2, rw2: array[0..pesgen] of integer;
  weight: array[0..pesgen] of integer;
  prtlist: array[1..pesgen] of integer;
  unsort: array[1..pesgen] of integer;
  length: array[0..pesgen] of integer;

  i, j, k, ii: integer;
  numline: integer;
  terminode: integer;
  pes_num_2_1: integer;
  busy, j_taken: array[0..pesgen] of boolean;
  job, ptime: array[1..pesgen] of integer;
  nextime, timer, jdone_num, jtaken_num: integer;
  conn, tconn: array[0..pesgen, 0..pesgen] of boolean;
  pjob, pleng: array[1..pesgen, 1..pesgen] of integer;
  pindx: array[1..pesgen] of integer;
  num_of_proc, leng_max: integer;

Function chrr(num: integer): char;
  (** convert a numeric digit to ascii code ***)
  begin
    chrr := chr(num+48);
  end;

Function rannode(seed: integer): integer;
  (** random number generator for node number ***)
  var
    ran: integer;
ran2: real;
i, j: integer;
begin
ran := trunc(time * random(time));
ran := ran - ran div 1000 * 1000;
i := ran - ran div 10 * 10;
for j := 1 to i do
ran2 := random(time);
ran2 := random(time);
ran2 := (i / 10) * ran2 + random(time) * (1 - ran2);
ran := trunc(ran2 * pes_num + 1);
rannode := ran;
end;

Function ranleng(seed: integer): integer;
(* random number generator for node length ***)
begin
ranleng := trunc(random(time) * leng_max + 1);
end;

Procedure insert(node: integer);
(* insert a new node with number 'node' into the tree *)
var
i, j, k: integer;
begin
i := 1;

(** traverse the tree to find the place to insert ***)
while (node > node_num[i]) and (rptr[i] # 0) or
(node < node_num[i]) and (lptr[i] # 0) do
begin
if node > node_num[i]
then
(** go right ***)
i := rptr[i]
else
(** go left ***)
i := lptr[i];
end;

(** insert ***)
nodeind := nodeind + 1;

if node > node_num[i]
then
begin
rptr[i] := nodeind;
end
else
| begin
| lptr[i]:=nodeind;
| end;

(*** set up for the new node ***)
node_num[nodeind]:=node;
leng[nodeind]:=ranleng(time);
lptr[nodeind]:=0;
rptr[nodeind]:=0;
end;

Function visit(nodein:integer):integer;
(*** traverse the subtree with root nodein, and
compute its left and right subtrees' width ***)
begin
if nodein = 0
then visit:=0
else
(** not terminal condition, recursive call ***)
begin
| lw[nodein]:=visit(lptr[nodein]);
| rw[nodein]:=visit(rptr[nodein]);
| visit:=lw[nodein]+rw[nodein]+1;
| end;
end;

Function visit2(nodein:integer):integer;
(*** traverse the subtree with root nodein, and
compute its left and right subtrees' width ***)
begin
if node_num[nodein] >= pes_num
then
(** terminal nodes ***)
begin
| visit2:=1;
| end
else
(** not terminal node, recursive call ***)
begin
| lw2[nodein]:=visit2(lptr[nodein]);
| rw2[nodein]:=visit2(rptr[nodein]);
| visit:=lw2[nodein]+rw2[nodein];
| end;
end;

Procedure draw(nodein,row,col:integer);
(*** draw the tree ***)

var
  i, j, left, right, lh, rh: integer;
begin
  left := lptr[nodein];
  right := rptr[nodein];
  lh := (rw[left] + 1) * 2;
  rh := (lw[right] + 1) * 2;

  (** draw the horizontal line to the left of nodein ***)
  if left # 0
    then
      for i := col - lh to col - 1 do
        tree[row, i] := horiz;

  (* draw the horizontal line to the right of nodein *)
  if right # 0
    then
      for i := col + rh to col + lh do
        tree[row, i] := horiz;

  (** print the length of left and right branch ***)
  if left # 0
    then
      tree[row + 1, col - lh] := chrr(leng[left]);
  if right # 0
    then
      tree[row + 1, col + rh] := chrr(leng[right]);

  (** draw the vertical line for the left branch ***)
  for i := row + 2 to row + leng[left] * 2 - 1 do
    tree[i, col - lh] := vert;

  (** draw the vertical line for the right branch ***)
  for i := row + 2 to row + leng[right] * 2 - 1 do
    tree[i, col + rh] := vert;

tree[row, col] := chrr(node_num[nodein] -
  node_num[nodein] div 10 * 10);
if (node_num[nodein] div 10 > 0)
  then
    tree[row, col - 1] := chrr(node_num[nodein] div 10);
(*** draw left subtree ***)
if left = 0
  then
    (** no left tree ***)
else
  if (lptr[left] = 0) and (rptr[left] = 0)
    then
      (** terminal node ***)
      begin
        tree[row + leng[left] * 2, col - lh] :=
        chrr(node_num[left] - node_num[left] div 10 * 10);
if (node_num[left] div 10 > 0)
then
tree[row+leng[left]*2, col-lh-1]:= chrr(node_num[left] div 10);
if (row+leng[left]*2 > numline)
then
numline:=row+leng[left]*2;
end
else
(*** not terminal node, recursive call ***)
draw(left, row+leng[left]*2, col-lh);
end;

(*** draw the right subtree ***)
if right = 0
then
(*** no right tree ***)
el se
if (lptr[right]=0) and (rptr[right]=0)
then
(*** terminal node ***)
begin
  tree[row+leng[right]*2, col+rh]:=
    chrr(node_num[right] - node_num[right] div 10 * 10);
  if (node_num[right] div 10 > 0)
  then
    tree[row+leng[right]*2, col+rh-1]:= chrr(node_num[right] div 10);
    if (row+leng[right]*2 > numline)
    then
      numline:=row+leng[right]*2;
    end
  else
    (*** not terminal node, recursive call ***)
    draw(right, row+leng[right]*2, col+rh);
  end;
end;

Procedure'out;
    (** output the tree ***)
var
    i,j,k:integer;
begin

    (** output the tree ***)
    rewrite(output, 'output ');
    for i:=1 to numline do
for $j : = 1$ to $\text{len}$ do
| begin
| | write($\text{tree}[i,j]$);
| | end;
| | writeln;
| | end;

begin

Procedure connvisit($\text{node}: \text{integer}$);
(* set up the entry in the connectivity matrix for
"node" and its left and right sons *)

var
inode, rnode: integer;

begin
if $\text{lptr}[\text{node}] \neq 0$
then
(* left son exists *)
begin
inode := $\text{lptr}[\text{node}]$;
| conn[\text{node_num}[\text{lnode}],\text{node_num}[\text{node}]] := true;
| (* recursive on left son *)
| connvisit(\text{lnode});
| end;
if $\text{rptr}[\text{node}] \neq 0$
then
(* right son exists *)
begin
rnode := $\text{rptr}[\text{node}]$;
| conn[\text{node_num}[\text{rnode}],\text{node_num}[\text{node}]] := true;
| (* recursive on right son *)
| connvisit(\text{rnode});
| end;
end;

Procedure connection;
(* set up the connectivity matrix *)

begin
for $i := 1$ to $\text{pes_num}_2$ do
for $j := 1$ to $\text{pes_num}_2$ do
| conn[$i,j]$ := false;
Procedure gentree;
  (* generate a tree and draw it on output *)
begin
  (** initialization for tree drawing ***)
  numline:=0;
  for i:=1 to linenum do
    for j:=1 to linelen do
      tree[i,j]:=" ";
      rewrite(ttyoutput);
  write(tty,'number of nodes:');
  break;
  reset(tty);
  read(tty,pes_num);
  write(tty,'number of processors:');
  break;
  read(tty,num_of_proc);
  write(tty,'maximum duration of tasks:');
  break;
  read(tty,leng_max);
  pes_num_2_1 := pes_num;
  pesgen:=0;
  nodegen:=0;
  nodeind:=1;
  rw[0]:=0;
  lw[0]:=0;
  leng[0]:=0;

  (** set up the first node ***)
  type3[1]:=rannode(2);
  node_num[1]:=type3[1];
  leng[1]:=0;
  lptr[1]:=0;
  rptr[1]:=0;

  (** set-up the rest nodes in the tree ***)
  for i:=2 to pes_num do
    begin
      (** check if any duplicate type 3 node ***)
repeat
|    type3[i]:=rannode(time);
|    j:=1;
|    while (j < i-1) and (type3[i] # type3[j]) do
|        j:=j+1;
|    until (type3[i] # type3[j]);
|    insert(type3[i]);
|end;

(*** dummy call to 'visit' to compute
    the width at each node ***)
i:=visit(1);

(*** draw the tree ***)
draw(1,1,1inelen div 2);

(*** output the tree ***)
oout;

(*** create connection matrix ***)
connection;
end;

Procedure lpweigh(node:integer);
    (* assigning weight for longest path schedule *)
var
    lnode,rnode:integer;
begin
    if (lptr[node] # 0)
    then
        (* left son exists *)
        |begin
        |  lnode:=lptr[node];
        |  weight[node_num[lnode]]:=weight[node_num[node]]+
          |  leng[lnode];
        |  length[node_num[lnode]]:=leng[lnode];
        |  (* recursive on left son *)
        |  lpweigh(lnode);
        |end;
        
    if (rptr[node] # 0)
    then
        (* right son exists *)
        |begin
        |  rnode:=rptr[node];
        |  weight[node_num[rnode]]:=weight[node_num[node]]+
          |  leng[rnode];
        |  length[node_num[rnode]]:=leng[rnode];
        |  (* recursive on right son *)
        |  lpweigh(rnode);
        |end;
Procedure hlweigh(node:integer);
(* assigning weight for highest level schedule *)
var
  lnode,rnode:integer;
begin
  if (lptr[node] # 0) then
    (* left son exists *)
    begin
      lnode:=lptr[node];
      weight[node_num[lnode]]:=weight[node_num[node]]+1;
      length[node_num[lnode]]:=length[lnode];
      (* recursive on left son *)
      hlweigh(lnode);
    end;
  if (rptr[node] # 0) then
    (* right son exists *)
    begin
      rnode:=rptr[node];
      weight[node_num[rnode]]:=weight[node_num[node]]+1;
      length[node_num[rnode]]:=length[rnode];
      (* recursive on right son *)
      hlweigh(rnode);
    end;
end;

Procedure blpweigh(node:integer);
(* assigning weight for backward longest path algorithm *)
var
  lnode,rnode:integer;
begin
  lnode:=lptr[node];
  rnode:=rptr[node];
  if (lnode=0) and (rnode=0)
then
(* leaf node *)
begin
weight[node_num[node]] := leng[node];
length[node_num[node]] := leng[node];
end
else
if lnode = 0
then
(* no left son, has right son *)
begin
(* recursive on right son *)
blpweight(rnode);
weight[node_num[node]] := leng[node] +
weight[node_num[rnode]];
length[node_num[node]] := leng[node];
end
else
if rnode = 0
then
(* no right son, has left son *)
begin
(* recursive on left son *)
blpweight(lnode);
weight[node_num[node]] := leng[node] +
weight[node_num[lnode]];
length[node_num[node]] := leng[node];
end
else
(* both left and right sons exist *)
begin
(* recursive on left son *)
blpweight(lnode);
(* recursive on right son *)
blpweight(rnode);
(* weight equals its length plus the larger of
the left and the right son weight *)
if weight[node_num[lnode]] >
weight[node_num[rnode]]
then
weight[node_num[node]] := leng[node] +
weight[node_num[lnode]]
else
weight[node_num[node]] := leng[node] +
weight[node_num[rnode]];
length[node_num[node]] := leng[node];
end;
end;
Procedure bhlweigh(node:integer);
    (* assigning weight for backward highest level algorithm *)
var
    lnode, rnode: integer;
begin
    lnode := lptr[node];
    rnode := rptr[node];

    if (lnode=0) and (rnode=0)
    then
        (* leaf node *)
        begin
            weight[node_num[node]] := 1;
            length[node_num[node]] := length[node];
        end
    else
        if lnode=0
        then
            (* no left son, has right son *)
            begin
                (* recursive on right son *)
                bhlweigh(rnode);
                weight[node_num[node]] := 1 + weight[node_num[rnode]];
                length[node_num[node]] := length[node];
            end
        else
            if rnode=0
            then
                (* no right son, has left son *)
                begin
                    (* recursive on left son *)
                    bhlweigh(lnode);
                    weight[node_num[node]] := 1 + weight[node_num[lnode]];
                    length[node_num[node]] := length[node];
                end
            else
                (* both left and right sons exist *)
                begin
                    (* recursive on left son *)
                    bhlweigh(lnode);
                    (* recursive on right son *)
                    bhlweigh(rnode);
                    (* weight equals one plus the larger of the left and the right son weight *)
                    if weight[node_num[lnode]] >
                        weight[node_num[rnode]]
                    then
                        weight[node_num[node]] := 1 + weight[node_num[lnode]]
else
    weight[node_num[node]] := l + weight[node_num[1node]];  
    length[node_num[node]] := leng[node];
end;

Procedure dsort;
(* sort nodes into descending order of weight *)
var
  i, j, templ, temp2, temp3: integer;
begin
  weight[0] := 100000;
  length[0] := 0;
  for i := 2 to pes_num_2_l do 
    begin
      templ := weight[i];
      temp2 := unsort[i];
      temp3 := length[i];
      j := i - 1;
      while (weight[j] < templ) or ((weight[j] = templ) and
      (length[j] > temp3)) do 
        begin
          weight[j + 1] := weight[j];
          unsort[j + 1] := unsort[j];
          length[j + 1] := length[j];
          j := j - 1;
        end;
      weight[j + 1] := templ;
      unsort[j + 1] := temp2;
      length[j + 1] := temp3;
    end;
end;

Procedure asort;
(* sort nodes into ascending order of weight *)
var
  i, j, templ, temp2, temp3: integer;
begin
  weight[0] := 0;
  length[0] := 1000000;
  for i := 2 to pes_num_2_l do
begin
    temp1:=weight[i];
    temp2:=unsort[i];
    temp3:=length[i];
    j:=i-1;
    while (weight[j] > temp1) or ((weight[j] = temp1) and
        (length[j] < temp3)) do
        begin
            weight[j+1] := weight[j];
            unsort[j+1]:=unsort[j];
            length[j+1]:=length[j];
            j:=j-1;
        end;
    weight[j+1]:=temp1;
    unsort[j+1]:=temp2;
    length[j+1]:=temp3;
end;

Procedure lp;
    (* simulate the longest path algorithm *)
    begin
    for i:=1 to pes_num_2_1 do
        unsort[i]:=i;
    weight[node_num[1]]:=0;
    length[node_num[1]]:=0;
    lpweigh(l);
    dsort;
    writeln;
    writeln('lp schedule ');
    writeln('(longest path algorithm)');
    writeln(' priority list: ' :20);
    for i:=1 to pes_num_2_1 do
        begin
            prtlst[i]:=unsort[i];
            write(prtlst[i]:3);
        end;
    writeln;
    end;

Procedure sp;
    (* simulate the shortest path algorithm *)
begin
  for i : - l  to pes_num_2 - 1 do
    unsort[i] := i;
    weight[node_num[1]] := 0;
    length[node_num[1]] := 0;
    lpweigh(1);
    asort;
    writeln;
    write('sp schedule ');
    writeln('(shortest path algorithm)');
    write(' priority list: ' :20);
    for i := 1 to pes_num_2 - 1 do
      begin
        prtlst[i] := unsort[i];
        write(prtlst[i]:3);
      end;
      writeln;
  end;

Procedure blp;
  (* simulate the backward longest path algorithm *)
  begin
    for i := 1 to pes_num_2 - 1 do
      unsort[i] := i;
      weight[node_num[1]] := 0;
      length[node_num[1]] := 0;
      blpweigh(1);
      dsort;
      writeln;
      write('blp schedule ');
      writeln('(backward longest path algorithm)');
      write(' priority list: ' :20);
      for i := 1 to pes_num_2 - 1 do
        begin
          prtlst[i] := unsort[i];
          write(prtlst[i]:3);
        end;
        writeln;
    end;

Procedure bsp;
(* simulate the backward shortest path algorithm *)
begin
for i:=1 to pes_num_2_1 do
  unsort[i]:=i;
weight[node_num[i]]:=0;
length[node_num[i]]:=0;
blpweigh(1);
asort;
writeln;
write('bsp schedule ');
writeln('(backward shortest path algorithm)');
write(' priority list: ' :20);
for i:=1 to pes_num_2_1 do
  begin
    prtlst[i]:=unsort[i];
    writeln(prlst[i]:3);
  end;
writeln;
end;

Procedure hl;
(* simulate the highest level algorithm *)
begin
for i:=1 to pes_num_2_1 do
  unsort[i]:=i;
weight[node_num[i]]:=0;
length[node_num[i]]:=0;
hlweigh(1);
dsort;
writeln;
write('hl schedule ');
writeln('(highest level algorithm)');
write(' priority list: ' :20);
for i:=1 to pes_num_2_1 do
  begin
    prtlst[i]:=unsort[i];
    writeln(prtlst[i]:3);
  end;
writeln;
end;
Procedure 11;
(* simulate the lowest level algorithm *)
begin
for i:=1 to pes_num_2_l do
unsort[i]:=i;

weight[node_num[1]]:=0;
length[node_num[1]]:=0;
hlweigh(1);
asort;

writeln;
write('11 schedule ');
writeln(' (lowest level algorithm)');
write(' priority list: ' :20);
for i:=1 to pes_num_2_l do
begin
| prtls[t][i]:=unsort[i];
| write(prtlst[i]:3);
| end;

writeln;

end;

Procedure bhl;
(* simulate backward highest level algorithm *)
begin
for i:=1 to pes_num_2_l do
unsort[i]:=i;

weight[node_num[1]]:=0;
length[node_num[1]]:=0;
bhlweigh(1);
dsort;

writeln;
write('bhl schedule ');
writeln(' (backward highest level algorithm)');
write(' priority list: ' :20);
for i:=1 to pes_num_2_l do
begin
| prtlst[i]:=unsort[i];
| write(prtlst[i]:3);
| end;

writeln;
end;
Procedure bll;
  (* simulate backward lowest level algorithm *)
  begin
  for i:=1 to pes_num_2_l do
    unsort[i]:=i;

    weight[node_num[i]]:=0;
    length[node_num[i]]:=0;
    bhlweigh(l);
    asort;

    writeln;
    write('bll schedule ');
    writeln('backward lowest level algorithm');
    write('priority list:');
    for i:=1 to pes_num_2_l do
      begin
      prtlst[i]:=unsort[i];
      write(prtlst[i]:3);
      end;

    writeln;
  end;

Procedure exec(numproc:integer);
  (* simulation of multi-processor execution *)
  (* conn is the connectivity matrix for tasks *)
  (* prtlst is the priority list of tasks *)
  (* numproc is the number of processors *)
  (* length is the execution time of tasks *)
  begin
  for i:=1 to pes_num_2_l do
    for j:=1 to pes_num_2_l do
      begin
      tconn[i,j]:=conn[i,j];
      end;

    (* jtaken stores the status of tasks *)
    (* clear the array jtaken *)
    for i:=1 to pes_num_2_l do
      begin
      jtaken[i]:=false;
      end;

    timer:=0;
    jtakennum:=0;
jdonenum := 0;

(* initialize processor status *)
for i := 1 to numproc do
| begin
| pindx[i] := 0;
| busy[i] := false;
| end;

(* execution loop *)
| loop
| (* assign free processors to tasks *)
| i := 1;
| while (i <= numproc) and (jtakennum < pes_num_2_l) do
| begin
| if not(busy[i])
| then
| | begin
| | for j := 1 to pes_num_2_l do
| | | begin
| | | if not(jtaken[j])
| | | then
| | | | begin
| | | | for k := 1 to pes_num_2_l do
| | | | | begin
| | | | | | if tconn[k, prtlst[j]]
| | | | | | then goto 1;
| | | | | | end;
| | | | busy[i] := true;
| | | | jtaken[j] := true;
| | | | jtakennum := jtakennum + 1;
| | | | job[i] := prtlst[j];
| | | | ptime[i] := length[j];
| | | | pindx[i] := pindx[i] + 1;
| | | | pjob[i, pindx[i]] := job[i];
| | | | pleng[i, pindx[i]] := ptime[i];
| | | | goto 2;
| | | | end;
| | | end;
| | end;
| end;
|
| i := i + 1;
| end;
(* update processor status *)
for i:=1 to numproc do
if not(busy[i])
then
| begin
| pindx[i]:=pindx[i]+1;
| pjob[i,pindx[i]]:=0;
| pleng[i,pindx[i]]:=0;
| end;

(* find the earliest next event time *)
nextime:=maxintg;
for i:=1 to numproc do
if busy[i]
then
  if (nextime > ptime[i])
  then
    | begin
    | nextime:=ptime[i];
    | end;

(* execution is done, if all tasks are complete *)
exit if jdonenum = pes_num_2_l;

(* increment timer to the next event *)
timer:=timer+nextime;

(* update processor status *)
for i:=1 to numproc do
if busy[i]
then
  ptime[i]:=ptime[i]-nextime;

for i:=1 to numproc do
| begin
  | if busy[i]
  | then
  |   | begin
  |   |   if ptime[i] = 0
  |   |     then
  |   |     | begin
  |   |     |   if jdonenum = pes_num_2_l
  |   |     |     then
  |   |     |     | begin
  |   |     |     |   for j:=1 to pes_num_2_l do
  |   |     |     |     | tconn[job[i],j]:=false;
  |   |     |   | end;
  |   |   | end;
  |   | end;
  | end;
end;
end;
Procedure bwexec(numproc:integer);
(* simulation of multi-processor execution
  for backward scheduling algorithms *)
(* conn is the connectivity matrix for tasks *)
(* prtlst is the priority list of tasks *)
(* numproc is the number of processors *)
(* length is the execution time of tasks *)
begin
for i:=1 to pes_num_2_l do
  for j:=1 to pes_num_2_l do
    begin
      jconn[j,i]:=conn[i,j];
    end;

for i:=1 to pes_num_2_l do
  begin
    jtaken[i]:=false;
  end;

timer:=0;
jtakennum:=0;
jdonenum:=0;
for i:=1 to numproc do
  begin
    pindx[i]:=0;
    busy[i]:=false;
  end;

loop
  i:=1;
  while (i<=numproc) and (jtakennum < pes_num_2_l) do
    begin
      if not(busy[i])
        then
          begin
            for j:=1 to pes_num_2_l do
              begin
                if not(jconn[j,i])
                  then
                    begin
                      pindx[i]:=pindx[i]+1;
                      jdonenum:=jdonenum+1;
                      if jdonenum = pes_num_2_l
                        then
                          begin
                            num做了遍，程序结束。
                            break;
                          end;
                      if jdonenum < pes_num_2_l
                        then
                          begin
                            for j:=1 to pes_num_2_l do
                              begin
                                if (jconn[j,i])
                                  then
                                    begin
                                      busy[j]:=true;
                                      timer:=timer+length[j];
                                      jdonenum:=jdonenum+1;
                                      if jdonenum = pes_num_2_l
                                        then
                                          begin
                                            num做了遍，程序结束。
                                            break;
                                          end;
                                    end;
                              end;
                          end;
                    end;
                  end;
          end;
    end;
  end;
end;
\begin{verbatim}
begin
if not(jtaken[j])
then
  begin
    for k:=1 to pes_num_2_1 do
      begin
        if tconn[k,prtlst[j]]
        then goto 1;
      end;
    busy[i]:=true;
    jtaken[j]:=true;
    jtakennum:=jtakennum+1;
    job[i]:=prtlst[j];
    ptime[i]:=length[j];
    pindx[i]:=pindx[i]+1;
    pjob[i,pindx[i]]:=job[i];
    pleng[i,pindx[i]]:=ptime[i];
    goto 2;
  end;
1:
  end;
end;
end;
2:
i:=i+1;
end;
for i:=1 to numproc do
  if not(busy[i])
  then
    begin
      pindx[i]:=pindx[i]+1;
      pjob[i,pindx[i]]:=0;
      pleng[i,pindx[i]]:=0;
    end;
nextime:=maxintg;
for i:=1 to numproc do
  if busy[i]
  then
    if (nextime > ptime[i])
    then
      begin
        nextime:=ptime[i];
      end;
end;
\end{verbatim}
exit if jdonenum = pes_num_2_l;

    timer:=timer+nextime;

    for i:=1 to numproc do
        if busy[i] then
            ptime[i]:=ptime[i]-nextime;

    for i:=1 to numproc do
        begin
            if busy[i] then
                begin
                    if ptime[i] = 0 then
                        begin
                            busy[i]:=false;
                            jdonenum:=jdonenum+1;
                            for j:=1 to pes_num_2_l do
                                tconn[job[i], j]:=false;
                        end;
                    end;
                else
                    begin
                        pleng[i, pindx[i]]:=nextime;
                    end;
                end;
        end;
end;
end;

Procedure outtime;
(* output the total completion time *)
begin
    writeln;
    write(' total execution time: ', timer:5);
    writeln;
end;

Procedure gantt(numproc:integer);
(* output the gantt chart of the schedule *)
var
    i, j: integer;
begin
writeln;
writeln(' gantt chart:');
for i:=1 to numproc do
begin
writeln;
write(' p', i:2, ': '); 
for j:=1 to pindx[i] do
begin
if pjob[i,j] = 0 then
begin
for k:=1 to pleng[i,j] do
write(' '); 
end;
else
begin
if pleng[i,j] # 0 then write(pjob[i,j]:2);
for k:=1 to pleng[i,j]-1 do
write('-');
end;
end;
end;

Procedure bwgantt(numproc:integer);
(* the gantt charts of backward algorithms *)

var
i, j:integer;

begin
writeln;
writeln(' gantt chart:');
for i:=1 to numproc do
begin
writeln;
write(' p', i:2, ': '); 
for j:=pindx[i] downto 1 do
begin
if pjob[i,j] = 0 then
begin
for k:=1 to pleng[i,j] do
write(' '); 
end;
else
begin
if pleng[i,j] # 0
\begin{verbatim}
|  |  |  then write(pjob[i,j]:2);  
|  |  |  for k:=1 to pleng[i,j]-1 do  
|  |  |  write(' -');  
|  |  |  end;  
|  |  end;  
|  end;  

(** main program **)  
begin  
gentree;  
writeln;write('number of nodes: ',pes_num:2);  
writeln;write('number of processors: ',num_of_proc:2);  
writeln;write('maximum duration of each task: ',leng_max:2);  
writeln;  
(* simulate the longest path algorithm *)  
lp;  
exec(num_of_proc);  
gantt(num_of_proc);  
outtime;  

(* simulate backward longest path algorithm *)  
blp;  
bwexec(num_of_proc);  
bwgantt(num_of_proc);  
outtime;  

(* simulate the shortest path algorithm *)  
sp;  
exec(num_of_proc);  
gantt(num_of_proc);  
outtime;  

(* simulate backward shortest path algorithm *)  
bsp;  
bwexec(num_of_proc);  
bwgantt(num_of_proc);  
outtime;  

(* simulate highest level algorithm *)  
hl;  
exec(num_of_proc);  
gantt(num_of_proc);  
outtime;  

(* simulate backward highest level algorithm *)
\end{verbatim}
bhl;
bwexec(num_of_proc);
bwgantt(num_of_proc);
outtime;

(* simulate lowest level algorithm *)
ll;
exec(num_of_proc);
gantt(num_of_proc);
outtime;

(* simulate backward lowest level algorithm *)
bll;
bwexec(num_of_proc);
bwgantt(num_of_proc);
outtime;

end.
An example of the output of program ALSSCH.PAS:

```plaintext
-----7-----
  1  2
-----4-----
  4   3
  1   1
  !   !

-----12-----
  1   3  3
  !   1   1
  !   1   !
  !   5-- 1   !
  !   2   !

1--   !   --9--  13--
  4   !  2   3  3
  !   6   !   !
  !   1   1   !
  !   8   !   !
  !    !   !
  !    -11  14--
  !    5    1

2--   !    15
  5
  !
  !
  !
  !
  !
  !
  !
  !  10
  !

3
```
NUMBER OF NODES: 15
NUMBER OF PROCESSORS: 2
MAXIMUM DURATION OF EACH TASK: 5

LP SCHEDULE (LONGEST PATH ALGORITHM)
PRIORITY LIST: 3 10 15 2 11 14 8 6 9 13 1 5 12 4 7

GANTT CHART:

P 1 : 3 6 11 14 9 1 12 4 7
P 2 : 10 2 8 6 13 5 4
TOTAL EXECUTION TIME: 21

BLP SCHEDULE (BACKWARD LONGEST PATH ALGORITHM)
PRIORITY LIST: 7 4 12 1 9 2 11 13 5 3 10 14 6 8 15

GANTT CHART:

P 1 : 3 6 10 5 2 1 4 9 2 11 13 5 3 10 14 6 8 15
P 2 : 15 4 13 9 12
TOTAL EXECUTION TIME: 21

SP SCHEDULE (SHORTEST PATH ALGORITHM)
PRIORITY LIST: 7 4 12 5 1 9 13 6 8 11 14 2 15 10 3

GANTT CHART:

P 1 : 6 5 10 11 9 12 1 4
P 2 : 8 15 14 13 3 2
TOTAL EXECUTION TIME: 23

BSP SCHEDULE (BACKWARD SHORTEST PATH ALGORITHM)
PRIORITY LIST: 15 6 8 14 3 10 5 13 11 2 9 1 12 4 7

GANTT CHART:

P 1 : 3 2 1 8 15 14 13 9 6 5 4
P 2 : 10 11 9 6 5 4
TOTAL EXECUTION TIME: 24

HL SCHEDULE (HIGHEST LEVEL ALGORITHM)
PRIORITY LIST: 15 3 10 6 8 11 14 2 5 9 13 1 4 12 7

GANTT CHART:

P 1 : 15 10 8 14 5 9 1 4
P 2 : 3 6 11 2 13 12
TOTAL EXECUTION TIME: 22
The listing of simulation program ALSMMP.PAS:

(*$t-,d-*

(* This program simulates eight different assembly line scheduling algorithms:
  the longest path algorithm,
  the backward longest path algorithm,
  the shortest path algorithm,
  the backward shortest path algorithm,
  the highest level algorithm,
  the backward highest level algorithm,
  the lowest level algorithm, and
  the backward lowest level algorithm.

The inputs of this program are:
  the number of nodes in the tree to be generated,
  the number of processors,
  the upper bound of the task length, and
  the number of cases to generate.

The outputs of this program are the average completion times for each schedule algorithm. *)

const
  linenum=120;
  linelen=60;
  numpes=30;
  star=’*’;
  horiz=’-’;
  vert=’!’;
  maxintg=9999999;

var
tree:array[1..linenum,1..linelen] of char;
pes_num:integer;
pesgen,nodegen,nodeind:integer;
pesptr:array[1..numpes] of integer;
type3:array[1..numpes] of integer;
node_num:array[1..numpes] of integer;
leng:array[0..numpes] of integer;
lptr:array[0..numpes] of integer;
rptr:array[0..numpes] of integer;
lw,rw,lw2,rw2:array[0..numpes] of integer;
weight:array[0..numpes] of integer;
prtlst:array[1..numpes] of integer;
unsort:array[1..numpes] of integer;
length:array[0..numpes] of integer;
i, j, k, li, li: integer;
numline: integer;
termnode: integer;
pes_num_2_l: integer;
bet, jtaken: array[0..numpes] of boolean;
job, ptime: array[1..numpes] of integer;
nextime, timer, jdonenum, jtakennum: integer;
conn, tconn: array[0..numpes, 0..numpes] of boolean;
pjob, pleng: array[1..numpes, 1..numpes] of integer;
pindx: array[1..numpes] of integer;
num_of_proc, leng_max: integer;
lp_time, blp_time, sp_time, bsp_time, hl_time,
bl_l_time, bll_time, bhl_time: integer;
total_lp_time, total_blp_time, total_sp_time,
total_bsp_time, total_hl_time, total_bhl_l_time,
total_ll_time, total_bll_time: integer;
avg_lp_time, avg_blp_time, avg_sp_time,
avg_bsp_time, avg_hl_time, avg_bhl_time,
avg_ll_time, avg_bll_time: real;
repet: integer;
num_of_cases: integer;

Procedure init_procedure;
| begin
| total_lp_time := 0;
| total_blp_time := 0;
| total_sp_time := 0;
| total_bsp_time := 0;
| total_hl_time := 0;
| total_bhl_time := 0;
| total_ll_time := 0;
| total_bll_time := 0;
| end;

Function rannode(seed: integer): integer;
(* *** random number generator for node number ***)
var
ran: integer;
ran2: real;
i, j: integer;
| begin
| ran := trunc(time * random(time));
| ran := ran - ran div 1000 * 1000;
| i := ran - ran div 10 * 10;
| for j := 1 to i do
| ran2 := random(time);
| ran2 := random(time);
| ran2 := (i / 10) * ran2 + random(time) * (1 - ran2);
| ran := trunc(ran2 * pes_num + 1);
Function ranleng(seed:integer):integer;
    (** random number generator for node length ***)
begin
    ranleng:=trunc(random(time)*leng_max + 1);
end;

Procedure insert(node:integer);
    (* insert a new node with number 'node' into the tree *)
var
    i,j,k:integer;
begin
    i:=1;

    (** traverse the tree to find the place to insert ***)
    while (node > node_num[i]) and (rptr[i] # 0) or
        (node < node_num[i]) and (lptr[i] # 0) do
        begin
            if node > node_num[i]
            then
                (** go right ****)
                i:=rptr[i]
            else
                (** go left ****)
                i:=lptr[i];
        end;

    (** insert ****)
    nodeind:=nodeind+1;

    if node > node_num[i]
    then
        begin
            rptr[i]:=nodeind;
        end
    else
        begin
            lptr[i]:=nodeind;
        end;

    (** set up for the new node ****)
    node_num[nodeind]:=node;
    leng[nodeind]:=ranleng(time);
    lptr[nodeind]:=0;
    rptr[nodeind]:=0;
Function visit(nodein:integer):integer;
    (** traverse the subtree with root nodein, and compute its left and right subtrees' width ***)
    begin
        if nodein = 0
            then visit:=0
        else
            (** not terminal condition, recursive call ***)
            begin
                lw[nodein]:=visit(lptr[nodein]);
                rw[nodein]:=visit(rptr[nodein]);
                visit:=lw[nodein]+rw[nodein]+1;
            end;
    end;

Function visit2(nodein:integer):integer;
    (** traverse the subtree with root nodein, and compute its left and right subtrees' width ***)
    begin
        if node_num[nodein] >= pes_num
            then (** terminal nodes ***)
            begin
                visit2:=1;
            end
        else (** not terminal node, recursive call ***)
            begin
                lw2[nodein]:=visit2(lptr[nodein]);
                rw2[nodein]:=visit2(rptr[nodein]);
                visit:=lw2[nodein]+rw2[nodein];
            end;
    end;

Procedure connvisit(node:integer);
    (* set up the entry in the connectivity matrix for "node" and its left and right sons *)
    var
        lnode,rnode:integer;
    begin
        if lptr[node] ≠ 0
            then (** left son exists ***)
            begin
                lnode:=lptr[node];
                conn[node_num[lnode],node_num[node]]:=true;
            end;
        if rptr[node] ≠ 0
            then (** right son exists ***)
            begin
                rnode:=rptr[node];
                conn[node_num[rnode],node_num[node]]:=true;
            end;
    end;
I (* recursive on left son *)
  | connvisit(lnode);
  | end;
  |
if rptr[node] # 0 then
  (* right son exists *)
  | begin
  | rnode:=rptr[node];
  | conn[node_num[rnode],node_num[node]]:=true;
  | (* recursive on right son *)
  | connvisit(rnode);
  | end;
end;

Procedure connection;
  (* set up the connectivity matrix *)
  | begin
  | for i:=1 to pes_num_2_l do
  | for j:=1 to pes_num_2_l do
  | conn[i,j]:=false;
  |
  | connvisit(1);
  |
end;

Procedure gentree;
  (* generate a tree *)
  (* *** initialization for tree drawing ***)
  | begin
  | numline:=0;
  |
  | for i:=1 to linenum do
  | for j:=1 to linelen do
  | tree[i,j]:=' ';
  |
  | pes_num_2_l := pes_num;
  | pesgen:=0;
  | nodegen:=0;
  |
  | nodeind:=1;
  |
  | rw[0]:=0;
  | lw[0]:=0;
  | leng[0]:=0;
(*** set up the first node ***)

```
type3[1] := rannode(2);
```

```
node_num[1] := type3[1];
leng[1] := 0;
lptr[1] := 0;
rptr[1] := 0;
```

(*** set up the rest nodes in the tree ***)

```
for i := 2 to pes_num do
begin
  (** check if any duplicate type 3 node ***)
  repeat
    type3[i] := rannode(time);
    j := 1;
    while (j < i - 1) and (type3[i] # type3[j]) do
      j := j + 1;
  until (type3[i] # type3[j]);
  insert(type3[i]);
end;
```

(*** dummy call to 'visit' to compute the width at each node ***)

```
i := visit(1);
```

(*** create connection matrix ***)

```
connection;
```

```
Procedure lpweigh(node:integer);
(* assigning weight for longest path schedule *)

```
var
  lnode, rnode: integer;
begin
  if (lptr[node] # 0) then
  (* left son exists *)
  begin
    lnode := lptr[node];
    weight[node_num[lnode]] := weight[node_num[node]] +
    leng[lnode];
    length[node_num[lnode]] := leng[lnode];
    (* recursive on left son *)
    lpweigh(lnode);
  ```
Procedure hlweigh(node:integer);
(* assigning weight for highest level schedule *)
var
lnode, rnode: integer;
begin
if (lptr[node] # 0)
then
(* left son exists *)
begin
lnode := lptr[node];
weight[node_num[lnode]] := weight[node_num[node]] +
leng[lnode];
length[node_num[lnode]] := leng[lnode];
(* recursive on left son *)
hlweigh(lnode);
end;
endif;
if (rptr[node] # 0)
then
(* right son exists *)
begin
rnode := rptr[node];
weight[node_num[rnode]] := weight[node_num[node]] + 1;
leng[rnode];
length[node_num[rnode]] := leng[rnode];
(* recursive on right son *)
hlweigh(rnode);
end;
end;

Procedure blpweigh(node:integer);
(* assigning weight for backward longest path algorithm *)
var
lnode, rnode: integer;
begin
lnode := lptr[node];
rnode := rptr[node];
if (lnode = 0) and (rnode = 0)
then (% leaf node *)
| begin
| weight[node_num[node]] := leng[node];
| length[node_num[node]] := leng[node];
| end
else
| if lnode = 0
| then
| (* no left son, has right son *)
| begin
| (* recursive on right son *)
| blp weigh(rnode);
| weight[node_num[node]] := leng[node] +
| weight[node_num[rnode]];
| length[node_num[node]] := leng[node];
| end
| else
| if rnode = 0
| then
| (* no right son, has left son *)
| begin
| (* recursive on left son *)
| blp weigh(lnode);
| weight[node_num[node]] := leng[node] +
| weight[node_num[lnode]];
| length[node_num[node]] := leng[node];
| end
| else
| (* both left and right sons exist *)
| begin
| (* recursive on left son *)
| blp weigh(lnode);
| (* recursive on right son *)
| blp weigh(rnode);
| (* weight equals its length plus the larger of
| the left and the right son weight *)
| if weight[node_num[lnode]] >
weight[node_num[rnode]]
| then
| weight[node_num[node]] := leng[node] +
| weight[node_num[lnode]]
| else
| weight[node_num[node]] := leng[node] +
| weight[node_num[rnode]];
Procedure bhlweight(node:integer);
(* assigning weight for backward highest level algorithm *)
var
    lnode, rnode: integer;
begin
    lnode := lptr[node];
    rnode := rptr[node];
    if (lnode = 0) and (rnode = 0) then (* leaf node *)
    begin
        weight[node_num[node]] := 1;
        length[node_num[node]] := leng[node];
    end
    else
        if lnode = 0 then (* no left son, has right son *)
        begin
            (* recursive on right son *)
            bhlweight(rnode);
            weight[node_num[node]] := 1 + weight[node_num[rnode]]; 
            length[node_num[node]] := leng[node];
        end
        else
            if rnode = 0 then (* no right son, has left son *)
            begin
                (* recursive on left son *)
                bhlweight(lnode);
                weight[node_num[node]] := 1 + weight[node_num[lnode]]; 
                length[node_num[node]] := leng[node];
            end
            else (* both left and right sons exist *)
            begin
                (* recursive on left son *)
                bhlweight(lnode);
                (* recursive on right son *)
                bhlweight(rnode);
(* weight equals one plus the larger of *)
if weight[node_num[lnode]] > weight[node_num[rnode]] then
    weight[node_num[node]] := 1 + weight[node_num[lnode]]
else
    weight[node_num[node]] := 1 + weight[node_num[rnode]];
length[node_num[node]] := length[node];
end;
end;

Procedure dsort;
(* sort nodes into descending order of weight *)
var
    i, j, temp1, temp2, temp3: integer;
begin
    weight[0] := 100000;
    length[0] := 0;
    for i := 2 to pes_num_2_1 do
    begin
        temp1 := weight[i];
        temp2 := unsort[i];
        temp3 := length[i];
        j := i - 1;
        while (weight[j] < temp1) or ((weight[j] = temp1) and (length[j] > temp3)) do
        begin
            weight[j+1] := weight[j];
            unsort[j+1] := unsort[j];
            length[j+1] := length[j];
            j := j - 1;
        end;
        weight[j+1] := temp1;
        unsort[j+1] := temp2;
        length[j+1] := temp3;
    end;
end;

Procedure asort;
(* sort nodes into ascending order of weight *)
var
i,j,temp1,temp2,temp3:integer;
begin
  weight[0]:=0;
  length[0]:=1000000;

  for i:=2 to pes_num_2_l do
    begin
      temp1:=weight[i];
      temp2:=unsort[i];
      temp3:=length[i];
      j:=i-1;

      while (weight[j] > temp1) or ((weight[j] = temp1) and (length[j] < temp3)) do
        begin
          weight[j+1] := weight[j];
          unsort[j+1]:=unsort[j];
          length[j+1] :=length[j];
          j:=j-1;
        end;

      weight[j+1] :=temp1;
      unsort[j+1]:=temp2;
      length[j+1] :=temp3;
    end;

  end;

Procedure lp;
(* simulate the longest path algorithm *)
begin
  for i:=1 to pes_num_2_l do
    unsort[i]:=i;

  weight[node_num[1]] :=0;
  length[node_num[1]] :=0;
  lpweigh(1);
  dsort;

  for i:=1 to pes_num_2_l do
    begin
      prt1st[i]:=unsort[i];
    end;
end;

Procedure sp;
(* simulate the shortest path algorithm *)
begin
Procedure blp;
(* simulate the backward longest path algorithm *)
begin
for i:=1 to pes_num_2_l do
unsort[i]:=i;

weight[node_num[1]]:=0;
length[node_num[1]]:=0;
lpweigh(1); asort;

for i:=1 to pes_num_2_l do
|begin
| prtlst[i]:=unsort[i];
|end;
end;

Procedure bsp;
(* simulate the backward shortest path algorithm *)
begin
for i:=1 to pes_num_2_l do
unsort[i]:=i;

weight[node_num[1]]:=0;
length[node_num[1]]:=0;
blpweigh(1); asort;

for i:=1 to pes_num_2_l do
|begin
| prtlst[i]:=unsort[i];
|end;
end;
Procedure h1;
 (* simulate the highest level algorithm *)
  begin
  for i:=1 to pes_num_2_1 do
   unsort[i]:=i;
   weight[node_num[1]]:=0;
   length[node_num[1]]:=0;
   hlweight(1);
   dsort;
   for i:=1 to pes_num_2_1 do
    |begin
    | prtlst[i]:=unsort[i];
    | end;
    end;

Procedure l1;
 (* simulate the lowest level algorithm *)
  begin
  for i:=1 to pes_num_2_1 do
   unsort[i]:=i;
   weight[node_num[1]]:=0;
   length[node_num[1]]:=0;
   hlweight(1);
   asort;
   for i:=1 to pes_num_2_1 do
    |begin
    | prtlst[i]:=unsort[i];
    | end;
    end;

Procedure bhl;
 (* simulate backward highest level algorithm *)
  begin
  for i:=1 to pes_num_2_1 do
   unsort[i]:=i;
weight[node_num[1]] := 0;
length[node_num[1]] := 0;
bhlweigh(1);
dsort;
for i := 1 to pes_num_2 1 do
| begin
| prtlst[i] := unsort[i];
| end;
end;

Procedure bll;
    (* simulate backward lowest level algorithm *)
| begin
| for i := 1 to pes_num_2 1 do
| unsort[i] := i;
| weight[node_num[1]] := 0;
| length[node_num[1]] := 0;
| bhlweigh(1);
| asort;
| for i := 1 to pes_num_2 1 do
| begin
| | prtlst[i] := unsort[i];
| | end;
| end;

Procedure exec(numproc:integer);
    (* simulation of multi-processor execution *)
    (* conn is the connectivity matrix for tasks *)
    (* prtlst is the priority list of tasks *)
    (* numproc is the number of processors *)
    (* length is the execution time of tasks *)
| begin
| for i := 1 to pes_num_2 1 do
| for j := 1 to pes_num_2 1 do
| | begin
| | | tconn[i, j] := conn[i, j];
| | | end;
| | (* jtaken stores the status of tasks *)
| | (* clear the array jtaken *)
| | for i := 1 to pes_num_2 1 do
begin
  jtaken[i]:=false;
end;

timer:=0;
jtakennum:=0;
donenum:=0;

(* initialize processor status *)
for i:=1 to numproc do
begin
  pindx[i]:=0;
  busy[i]:=false;
end;

(* execution loop *)
loop

(* assign free processors to tasks *)
i:=1;
while (i<=numproc) and (jtakennum < pes_num_2_l) do
begin
  if not(busy[i])
  then
  begin
    for j:=1 to pes_num_2_l do
    begin
      if not(jtaken[j])
      then
      begin
        for k:=1 to pes_num_2_l do
        begin
          if tconn[k,prtlst[j]]
          then goto 1;
        end;
        busy[i]:=true;
        jtaken[j]:=true;
        jtakennum:=jtakennum+1;
        job[i]:=prtlst[j];
        ptime[i]:=length[j];
        pindx[i]:=pindx[i]+1;
        pjob[i,pindx[i]]:=job[i];
        pleng[i,pindx[i]]:=ptime[i];
        goto 2;
      end;
    end;
  end;
end;
(* update processor status *)
for i:=1 to numproc do
if not(busy[i])
then
  pindx[i] := pindx[i] + 1;
  pjob[i, pindx[i]] := 0;
  pleng[i, pindx[i]] := 0;
end;

(* find the earliest next event time *)
nextime := maxintg;
for i:=1 to numproc do
if busy[i]
then
  if (nextime > ptime[i])
  then
    begin
    nextime := ptime[i];
    end;

(* execution is done, if all tasks are complete *)
exit if jdonenum = pes_num_2_l;

(* increment timer to the next event *)
timer := timer + nextime;

(* update processor status *)
for i:=1 to numproc do
if busy[i]
then
  ptime[i] := ptime[i] - nextime;

for i:=1 to numproc do
  begin
  if busy[i]
  then
    begin
    if ptime[i] = 0
    then
      ...
Procedure bwexec(numproc:integer);
(* simulation of multi-processor execution
for backward scheduling algorithms *)
(* conn is the connectivity matrix for tasks *)
(* prtlst is the priority list of tasks *)
(* numproc is the number of processors *)
(* length is the execution time of tasks *)
begin
for i:=1 to pes_num_2_l do
for j:=1 to pes_num_2_l do
|begin
| tconn[j,i]:=conn[i,j];
|end;

for i:=1 to pes_num_2_l do
|begin
| jtaken[i]:=false;
|end;

timer:=0;
jtakennum:=0;
jdonenum:=0;
for i:=1 to numproc do
|begin
| pindx[i]:=0;
| busy[i]:=false;
|end;

|loop
i := 1;
while (i <= numproc) and (jtakenum < pes_num_2_1) do
  begin
    if not(busy[i])
      then
        begin
          for j := 1 to pes_num_2_1 do
            begin
              if not(jtaken[j])
                then
                  begin
                    for k := 1 to pes_num_2_1 do
                      begin
                        if tconn[k, prtlst[j]]
                          then goto 1;
                    end;
                    busy[i] := true;
                    jtaken[j] := true;
                    jtakenum := jtakenum + 1;
                    job[i] := prtlst[j];
                    ptime[i] := length[j];
                    pindx[i] := pindx[i] + 1;
                    pjob[i, pindx[i]] := job[i];
                    pleng[i, pindx[i]] := ptime[i];
                    goto 2;
                  end;
            end;
          i := i + 1;
        end;
  end;
end;

for i := 1 to numproc do
  if not(busy[i])
    then
      begin
        pindx[i] := pindx[i] + 1;
        pjob[i, pindx[i]] := 0;
        pleng[i, pindx[i]] := 0;
      end;

nextime := maxintg;
for i := 1 to numproc do
  if busy[i]
then
  if (nextime > ptime[i])
  then,
    |begin
    |  nextime:=ptime[i];
    |end;

exit if jdonenum = pes_num_2_1;

  timer:=timer+nextime;

for i:=1 to numproc do
  if busy[i]
  then
    ptime[i]:=ptime[i]-nextime;

for i:=1 to numproc do
  begin
    if busy[i]
    then
      begin
        if ptime[i] = 0
        then
          begin
            busy[i]:=false;
            jdonenum:=jdonenum+1;
          end;
          for j:=1 to pes_num_2_1 do
            tconn[job[i],j]:=false;
        end;
      end
    else
      begin
        pleng[i,pindx[i]]:=nextime;
      end;
  end;
end;

(* * * main program * * *)
begin
  init_procedure;
  rewrite(ttyoutput);
  write(tty,'number of nodes:');
break;
reset(tty);
read(tty, pes_num);
write(tty, 'number of processors:');
break;
read(tty, num_of_proc);
write(tty, 'maximum duration of each task:');
break;
read(tty, leng_max);
write(tty, 'number of cases simulated:');
break;
read(tty, num_of_cases);
writeln;write('number of external nodes: ', pes_num:2);
writeln;write('number of processors: ', num_of_proc:2);
writeln;
write('maximum duration of each task: ', leng_max:2);
writeln;
write('number of cases simulated: ', num_of_cases:4);
writeln;
writeln;
writeln;
writeln('':15, 'l p':7, 'b l p':7, 's p':7, 'b s p':7, 
'':7, 'b h l':7, 'b l l':7);
writeln;

(* simulate "num_of_cases" different cases *)
for repet := 1 to num_of_cases do
begin
  (* generate a binary tree *)
gentree;

  (* simulate the longest path algorithm *)
lp;
  exec(num_of_proc);
lp_time := timer;
total_lp_time := total_lp_time + lp_time;

  (* simulate backward longest path algorithm *)
blp;
bwexec(num_of_proc);
blp_time := timer;
total_blp_time := total_blp_time + blp_time;

  (* simulate the shortest path algorithm *)
sp;
  exec(num_of_proc);
sp_time := timer;
total_sp_time := total_sp_time + sp_time;

(* simulate backward shortest path algorithm *)
bsp;
bwexec(num_of_proc);
bsp_time := timer;
total_bsp_time := total_bsp_time + bsp_time;

(* simulate highest level algorithm *)
hl;
exec(num_of_proc);
hl_time := timer;
total_hl_time := total_hl_time + hl_time;

(* simulate backward highest level algorithm *)
bhl;
bwexec(num_of_proc);
bhl_time := timer;
total_bhl_time := total_bhl_time + bhl_time;

(* simulate lowest level algorithm *)
ll;
exec(num_of_proc);
ll_time := timer;
total_ll_time := total_ll_time + ll_time;

(* simulate backward lowest level algorithm *)
bll;
bwexec(num_of_proc);
bll_time := timer;
total_bll_time := total_bll_time + bll_time;

end;

(* calculate the average completion time for each scheduling algorithm *)
avg_lp_time := total_lp_time / num_of_cases;
avg_blp_time := total_blp_time / num_of_cases;
avg_sp_time := total_sp_time / num_of_cases;
avg_bsp_time := total_bsp_time / num_of_cases;
avg_hl_time := total_hl_time / num_of_cases;
avg_bhl_time := total_bhl_time / num_of_cases;
avg_ll_time := total_ll_time / num_of_cases;
avg_bll_time := total_bll_time / num_of_cases;

write(‘average time:’ :15, avg_lp_time:7:2,
avg_blp_time:7:2, avg_sp_time:7:2,
avg_bsp_time:7:2, avg_hl_time:7:2,
avg_bhl_time:7:2, avg_ll_time:7:2,
avg_bll_time:7:2);
| writeln; writeln;
| end.
An example of the output of program ALSCMP.PAS:

NUMBER OF EXTERNAL NODES: 10
NUMBER OF PROCESORS: 2
MAXIMUM DURATION OF EACH TASK: 8
NUMBER OF CASES SIMULATED: 3000

<table>
<thead>
<tr>
<th>LP</th>
<th>BLP</th>
<th>SP</th>
<th>BSP</th>
<th>HL</th>
<th>BHL</th>
<th>LL</th>
<th>BLL</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.69</td>
<td>22.61</td>
<td>27.37</td>
<td>24.83</td>
<td>23.36</td>
<td>23.26</td>
<td>26.76</td>
<td>24.51</td>
</tr>
</tbody>
</table>
APPENDIX C

BNF DESCRIPTION OF THE PES LANGUAGE

<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<special symbol> ::= + | - | * | / | and | or | not | = | <> | <= | >= | ( | ) | [ | ] | { | } | ; | : | , | : | ’ | div | mod | nil | in | if | then | else | case | of |
repeat | until | while | do | for | to | downto | then | end | else | send |
while | do | whileend | repeat | end | loopend | exit | begin | end | with |
goto | var | type | array | record | powerset | file | class | loop |
function | procedure | const

IDENTIFIERS AND NUMBERS

<identifier> ::= <letter> <letter or digit>*
<letter or digit> ::= <letter> | <digit>
<number> ::= <integer> | <real number>
<integer> ::= <digit>"<sign><digit>"
<real number> ::= <digit>".\<digit>"|<digit>"E\<scale factor>
<scale factor> ::= <digit>"\<sign><digit>"
<sign> ::= + | -

CONSTANT DEFINITIONS

<unsigned constant> ::= <number>|\"<character>" |<identifier>|nil
<constant> ::= <unsigned constant> | <sign><number>
<constant definition> ::= <identifier>=<constant>
DATA TYPE DEFINITIONS

<type> ::= <scalar type>|<sub-range type>|<array type>|
<record type>|<powerset type>|<file type>|<class type>|
<pointer type>|<type identifier>
<type definition> ::= <identifier>=<type>

<scalar type> ::= (identifier>{,<identifier>}")
<sub-range type> ::= <constant>..<constant>

<array type> ::= array[<index type>{,<index type>}"]of
<component type>
#index type> ::= <scalar type>|<sub-range type>|
<type identifier>
<component type> ::= <type>

<record type> ::= record<field list>end
<field list> ::= <fixed part>|<fixed part>;<variant part>|
<variant part>
<fixed part> ::= <record section>{;<record section>"}
<record section> ::= <field identifier>{,<field identifier>}
";<type>
<variant part> ::= case<tag field>:<type identifier>of
<variant>{;<variant>"}
<variant> ::= {<case label>:"(<field list>)|(<case label>)"}
<case label> ::= <unsigned constant>
<tag field> ::= <identifier>

<powerset> ::= powerset<type identifier>|
powerset<sub-range type>

<file type> ::= file of <type>
<class type> ::= class<maxnum>of<type>
<maxnum> ::= <integer>

<pointer type> ::= ^<class variable>
<class variable> ::= <variable>

DECLARATIONS AND DENOTATIONS OF VARIABLES

<variable declaration> ::= <identifier>{,<identifier>}":
<type>
<variable> ::= <entire variable>|<component variable>

<entire variable> ::= <variable identifier>
<variable identifier> ::= type dynamic-address

<component variable> ::= <indexed variable>|
<field designator>|
<current file component>|
<referenced component>

<indexed variable> ::= <array variable><expression>{, <expression>}"]ary
<array variable> ::= <variable>

'field designator' ::= <record variable>..<field identifier>
<record variable> ::= <variable>
'field identifier' ::= <identifier>

<current file component> ::= <file variable>
<file variable> ::= <variable>

<referenced components> ::= <pointer variable>
<pointer variable> ::= <variable>

EXPRESSIONS

<factor> ::= <variable>|<unsigned constant>|<function designator>|<set>|#<integer>
<pes> ::= <factor> {<factor><operator>|<operator>}"jump
<sub-tree> ::= <pes>jump|<sub-tree>}
{<factor><operator>|<operator>}|<sub-tree>jump|<sub-tree>
<expression> ::= <sub-tree>|<sub-tree>#<integer>
<relational operator> ::= =|<>|>|<|<=|=|in

OPERATORS

<operator> ::= +|-|*|/|and|or|div|mod|-'|-'|/'|div'|mod'
<relational operator> ::= =|<>|>|<|<=|=|in

FUNCTION DESIGNATORS

<function designator> ::= <function identifier>
(function identifier,<function identifier>* fjump
<function identifier> ::= <identifier>

STATEMENTS

<statement> ::= <simple statement>|<structured statement>

<simple statement> ::= <assignment statement> |<procedure statement>|<goto statement>

<assignment statement> ::= <variable> <expression> := |<function identifier> <expression> :=

<procedure statement> ::= <procedure identifier> Pjump |<procedure identifier>(<actual parameter>
STRUCTURED STATEMENTS

<structure statement> ::= <compound statement> |<conditional statement>|<repetitive statement>|
  <with statement>

<compound statement> ::= begin <component statement>
  <component statement>* end
<component statement> ::= <statement>|<label definition>
  <statement>
<label definition> ::= <label>:

<conditional statement> ::= <if statement>|<case statement>
<if statement> ::= para. <expression>if then <statement>
  thenend | para. <expression> if then <statement> thenend,
  else <statement> endif
<case statement> ::= case <expression> of <case list
  element>{;<case list element}>* end
<case list element> ::= {<case label>:}"<statement>|
  {<case label>}"

<repetitive statement> ::= <while statement>|<repeat
  statement>|<for statement>|<loop statement>

<while statement> ::= "para. <expression> while<statement> whendol
<repeat statement> ::= <statement> repeatend
  para. <expression> until<repeat>
<for statement> ::= for <control variable>::=<for list> do
  <statement>
<for list> ::= <initial value>to<final value>|<initial value>downto<final value>
<control variable> ::= <identifier>
<initial value> ::= <expression>
<final value> ::= <expression>

<loop statement> ::= <statement> loopend
  para. <expression> exitif loop <statement>" jump
<with statement> ::= with <record variable>do<statement>
PROCEDURE DECLARATIONS

<procedure declaration> ::=  
  <procedure heading>  
  <constant definition part><type definition part>  
  <variable declaration part>  
  <procedure and function declaration part>  
  <statement part>  
  <procedure heading> ::= procedure <identifier> ; |  
  procedure <identifier> (formal parameter section>  
  {;<formal parameter section> }*);  
  <formal parameter section> ::= <parameter group> |  
  const <parameter group>{; <parameter group> }* |  
  var <parameter group>{; <parameter group> }* |  
  function <parameter group> |  
  procedure <identifier> {, <identifier> }*  
  <parameter group> ::= <identifier> {, <identifier> }*:  
  <type identifier>  
  <constant definition part> ::= <empty> |  
  const <constant definition> {, <constant definition> }*;  
  <type definition part> ::= <empty> |  
  type <type definition> {, <type definition> }*;  
  <variable declaration part> ::= <empty> |  
  var <variable declaration> {; <variable declaration> }*;  
  <procedure or function declaration part> ::=  
  {<procedure or function declaration> ; }*  
  <procedure or function declaration> ::=  
  <procedure declaration> | <function declaration>  
  <statement part> ::= <compound statement>

FUNCTION DECLARATIONS

<function declaration> ::=  
  <function heading>  
  <constant definition part><type definition part>  
  <variable declaration part>  
  <procedure and function declaration part>  
  <statement part>  
  <function heading> ::= function <identifier> (<formal 
  parameter section>{; <formal parameter section> }*):  
  <result type>;  
  <result type> ::= <type identifier>

PROGRAMS

<program> ::= <constant definition part><type definition 
part><variable declaration part><procedure and 
function declaration part><statement part>
APPENDIX D

PES TRANSLATION PROGRAM

The listing of program REVPES.PAS:

(*$L+,T-,D-*)

(* This program accepts an arithmetic expression as input, and generates reverse PES's that are ready for parallel processing. *)

const
strmax 10; (* maximum number of pes_s *)
lenmax 20; (* maximum length of a pes *)
stksize 20; (* stack size *)
outmax 100; (* size of output area *)
bufmax 150; (* size of input buffer *)
alfaleng 10;
lc_init 1;
space ;
quote ;
gt > ;
ge >>=
lt <
le <=
eq =

314
ne = '#';
leftparen = '(';
rightparen = ')';
add = '+';
subtract = '-';
multiply = '*';
divide = '/';
exponentiation = '^';
end_of_exp = '\';
beg_of_exp = '$';
negate = '_';
assign = ':=';

type
(* alfa = packed array [1..10] of char; *)
token_type = (gt_type,ge_type,lt_type,le_type,eq_type,
  ne_type,leftp_type,rightp_type,
  add_type,subtract_type,multiply_type,divide_type,
  exponentiation_type,negate_type,assign_type,
  eoe_type,boe_type,var_type);

var
alphanum : set of char;
token : token_type;
.tp : array [0..strmax] of integer;
temp_counter,lc,pes_begin : integer;
.s, opr : alfa;
opn_stk_ptr,opr_stk_ptr:integer;
opn_stk : array [1..stksize] of alfa;
opr_stk : array [0..stksize] of alfa;
out_area : array [1c_init..outmax] of alfa;
buffer : array [1..bufmax] of char;
buffer_ptr : integer;

Procedure initproc;
var
  i,j,k:integer;
begin
  rewrite(ttyoutput);
  for i:=1 to strmax do
  tp[i]:=0;
  temp_counter := 0;
  lc := 1;
  pes_begin := lc;
  opr_stk_ptr := 0;
  opn_stk_ptr := 0;
  opr_stk[0] := beg_of_exp;
  alphanum := ['a'..'z'] or ['0'..'9'];
end;
Function operand(s:alpha):boolean; forward;

Function t_type(token:alpha):token_type;
 (* t_type receives a token and returns its type *)
| begin
| if operand(token)
| then t_type := var_type
| else
| if token = leftparen
| then t_type := leftp_type
| else
| if token = rightparen
| then t_type := rightp_type
| else
| if token = add
| then t_type := add_type
| else
| if token = subtract
| then t_type := sub_type
| else
| if token = multiply
| then t_type := mul_type
| else
| if token = divide
| then t_type := div_type
| else
| if token = assign
| then t_type := ass_type
| else
| if token = exponentiation
| then t_type := expo_type
| else
| if token = negate
| then t_type := neg_type
| else
| if token = gt
| then t_type := gt_type
| else
| if token = ge
| then t_type := ge_type
| else
| if token = lt
| then t_type := lt_type
| else
| if token = le
| then t_type := le_type
| else
| if token = eq
| then t_type := eq_type
el se
if token = ne
then t_type := ne_type
else
  if token = end_of_exp
    then t_type := eoe_type
  else
    if token = beg_of_exp
    then t_type :=
      boe_type;
end;

Function priority(s:alfa):integer;
(* it receives a token and returns its priority *)
begin
  case t_type(s) of
    eoe_type :
      priority := 0;
    assg_type :
      priority := 0;
    leftp_type :
      priority := 1;
    rightp_type :
      priority := 2;
    ge_type, gt_type, le_type, lt_type, eq_type, ne_type :
      priority := 3;
    add_type, sub_type :
      priority := 4;
    mul_type, div_type :
      priority := 5;
    expo_type, neg_type :
      priority := 6;
    boe_type :
      priority := -1;
    var_type :
      priority := 7
  end;
end;

Procedure get_next(var s:alfa);
(* obtains the next token from input buffer *)
var
  i, j, k:integer;
  next_char : char;
begin
  for i := 1 to alfaleng do
    s[i] := space;
  while buffer [buffer_ptr] = space do
buffer_ptr := buffer_ptr + 1;

i := 1;
if buffer[buffer_ptr] in alphanum then
repeat
s[i] := buffer[buffer_ptr];
buffer_ptr := buffer_ptr + 1;
i := i + 1;
until not (buffer[buffer_ptr] in alphanum)
else
begin
s[1] := buffer[buffer_ptr];
buffer_ptr := buffer_ptr + 1;
case s[1] of
'>' :
if buffer[buffer_ptr] = '=' then
begin
s[2] := '=';
buffer_ptr := buffer_ptr + 1;
end;
'<' :
if buffer[buffer_ptr] in ['=','>'] then
begin
s[2] := buffer[buffer_ptr];
buffer_ptr := buffer_ptr + 1;
end;
'.' :
if buffer[buffer_ptr] = '=' then
begin
s[2] := '=';
buffer_ptr := buffer_ptr + 1;
end end;
end;
end;

Procedure fill_buffer;
(* reads in one line of input *)
var
i:integer;
input_char : char;
begin
i := 1;
repeat
read(input_char);

case s[1] of
'>' :
if buffer[buffer_ptr] = '=' then
begin
s[2] := '=';
buffer_ptr := buffer_ptr + 1;
end;
'<' :
if buffer[buffer_ptr] in ['=','>'] then
begin
s[2] := buffer[buffer_ptr];
buffer_ptr := buffer_ptr + 1;
end;
'.' :
if buffer[buffer_ptr] = '=' then
begin
s[2] := '=';
buffer_ptr := buffer_ptr + 1;
end end;
end;
end;
buffer[i] := input_char;
i := i + 1;
until eoln(input);
buffer[i] := '\';
buffer_ptr := 1;
end;

Procedure echo_input;
(* print out the input line *)
var
i:integer;
begin
write('input expression: ');
i := 1;
while buffer[i] # '\' do
begin
write(buffer[i]);
i := i + 1;
end;
writeln; writeln; writeln;
end;

Procedure push_opr(s:alpha);
(* push a token onto operator stack *)
begin
opr_stk_ptr := opr_stk_ptr + 1;
opr_stk [opr_stk_ptr] := s;
end;

Procedure push_opn(s:alpha);
(* push a token onto operand stack *)
begin
opn_stk_ptr := opn_stk_ptr + 1;
opn_stk [opn_stk_ptr] := s;
end;

Procedure pop_opn(var s:alpha);
(* pop out one element from operand stack *)
begin
s := opn_stk [opn_stk_ptr];
opn_stk_ptr := opn_stk_ptr - 1;
end;

Procedure pop_opr(var s:alpha);
(* pop out one element from operator stack *)

begin
| s := opr_stk [opr_stk_ptr];
| opr_stk_ptr := opr_stk_ptr - 1;
end;

Procedure pop_opr_stk(var s:alfa);
 (* Procedure for outputing an operator from stack *)
var
   opr, pointer, outword : alfa;
   k : integer;
   temp, dummy : alfa;

Function ascii(number:integer):char;
 (* translate a digit into ascii character *)
| begin
|   ascii := chr(number+48);
| end;

Function dollars(opn:alfa):boolean;
 (* check whether the token starts with $ *)
| begin
|   if opn[1] = '$'
|     then
|     dollars := true
|     else
|     dollars := false;
| end;

Procedure outputpes(word:alfa);
 (* a token is sent to output *)
| begin
|   out_area[lc] := word;
|   lc := lc + 1;
| end;

Procedure back_output(addr:integer; word:alfa);
 (* output a token to a specific location *)
| begin
|   out_area[addr] := word;
| end;

Procedure jump_ptr(addr:integer; var pointer:alfa);
 (* output a jump pointer *)
I  begin
|  pointer := 'jump';
|  pointer[5] := ascii(addr div 1000);
|  pointer[6] := ascii((addr - addr div 1000 * 1000) div 100);
|  pointer[7] := ascii((addr - addr div 100 * 100) div 10);
|  pointer[8] := ascii(addr - addr div 10 * 10);
|  end;

Procedure finish_prev;

Procedure gen_hash(number:integer ; var word:alfa);
(* output #number *)
begin
|  word := '#';
|  word[2] := ascii(number);
end;

Procedure para_ptr(addr:integer; var pointer:alfa);
(* output a parallel pointer *)
begin
|  pointer := 'para';
|  pointer[5] := ascii(addr div 1000);
|  pointer[6] := ascii((addr - addr div 1000 * 1000) div 100);
|  pointer[7] := ascii((addr - addr div 100 * 100) div 10);
|  pointer[8] := ascii(addr - addr div 10 * 10);
end;

(* finish outputing the previous pes *)
begin
|  temp_counter := temp_counter + 1;
|  gen_hash(temp_counter,outword);
|  outputpes(outword);
|  tp[temp_counter] := lc;
|  lc := lc + 2;
|  para_ptr(lc,pointer);
|  back_output(pes_begin,pointer);
|  pes_begin := lc;
end;

Procedure gen_dollar(number:integer; var word:alfa);
(* output $number *)
I begin
word := '$';
word[2] := ascii(number);
end;

(* output an element from operator stack *)
begin (* pop_opr_stk *)
pop_opr(opr);
if dollars(opn_stk[opn_stk_ptr]) and
dollars(opn_stk[opn_stk_ptr - 1])
then
(* type 3 operator *)
begin
outword := opn_stk[opn_stk_ptr-1];
outword [1] := '№';
outputpes(outword);
outword := opr;
case t_type(opr) of
  div_type, sub_type, expo_type :
gt_type :
  outword := le;
ge_type :
  outword := lt;
lt_type :
  outword := ge;
le_type :
  outword := gt
end;
outputpes(outword);
k := ord(opn_stk[opn_stk_ptr-1][2])-48;
back_output(tp[k], opr);
jump_ptr(lc:pointer);
back_output(tp[k]+1, pointer);
pop_opn(temp);
pop_opn(dummy);
push_opn(temp);
end
else
  if not(dollars(opn_stk[opn_stk_ptr])) and
      not(dollars(opn_stk[opn_stk_ptr - 1]))
  then
    (* type 1 node *)
    begin
      if (lc # lc_init)
        then finish_prev;
      lc := lc + 1;
      outputpes(opn_stk[opn_stk_ptr - 1]);
outputpes(opn_stk[opn_stk_ptr]);
pop_opn(dummy);
pop_opn(dummy);
generate_dollar(temp_counter + 1, outword);
push_opn(outword);
outputpes(opr);
end
else
(* type 2 node *)
begin
if dollars(opn_stk[opn_stk_ptr])
then
outputpes(opn_stk[opn_stk_ptr - 1])
else
outputpes(opn_stk[opn_stk_ptr]);
outword := opr;
if dollars(opn_stk[opn_stk_ptr])
then
case t_type(opr) of
  div_type, sub_type, expo_type :
    gt_type : outword := le;
    ge_type : outword := lt;
    lt_type : outword := ge;
    le_type : outword := gt
  end;
outputpes(outword);
pop_opn(temp);
if dollars(temp)
then
begin
  pop_opn(dummy);
push_opn(temp);
end;
end;
end;

Function of_arith_opr(opr:alpha):boolean;
(* check if a token is an operator *)
begin
case t_type(opr) of
  add_type, mul_type, div_type,
  sub_type, expo_type, assg_type,
  neg_type, gt_type, ge_type, lt_type,
  le_type, eq_type, ne_type :
    of_arith_opr := true;
others:
    of_arith_opr := false
Function operand;
(* check if the token is an operand *)
begin
if s[1] in alphanum
then operand := true
else operand := false;
end;

Procedure error;
(* error handling routine *)
var
  i: integer;
begin
  writeln(tty,"*****error in translation*****");
  writeln; writeln;
  writeln("***** opr stack dump *****");
  for i := opr_stk_ptr downto 1 do
    writeln(opr_stk[opr_stk_ptr]);
  writeln; writeln;
  writeln("***** opn stack dump *****");
  for i:= opn_stk_ptr downto 1 do
    writeln(opn_stk[opn_stk_ptr]);
end;

Procedure printout;
(* print the results *)
var
  i:integer;
begin
  writeln('address',space:3,'contents');
  writeln;
  for i := lc_init to lc do
    begin
    | writeln(i:6,space:4,out_area[i]);
    | break;
    | end;
end;

(* main Procedure *)
begin
  initproc;
  fill_buffer;

echo_input;

(* loop until it is not a left parenthesis *)
loop
get_next(s);
exit if (s # leftparen);
push_opr(s);
end;

(* it should be an operand *)
if not(operand(s))
then error;
push_opn(s);

(* output the higher priority operators from stack *)
loop
get_next(s);
while (priority(opr_stk[opr_stk_ptr]) >=
  priority(s)) and (opr_stk_ptr >= 1) do
  pop_opr_stk(opr);

exit if (s # rightparen);
if opr_stk[opr_stk_ptr] # leftparen
  then error;
pop_opr(opr);
end;
exit if not(of_arith_opr(s));
push_opr(s);
end;

if (opr_stk[opr_stk_ptr] = leftparen) or (s # end_of_exp)
then error;
printout;
end.


