Compilation Techniques for Recognition of Parallel Processable Tasks in Arithmetic Expressions

C. V. RAMAMOORTHY, JUONG H. PARK, AND HON F. LI

Abstract—With the developments of parallel computer systems, the techniques for the recognition and representation of parallel task streams in a job (program) have become important to enhance the system performance.

This paper makes a survey of the previous techniques for the recognition and representation of parallelism within an arithmetic expression and introduces some new techniques.

The techniques are classified into three categories: 1) precedent-dependent method; 2) syntax-dependent method; and 3) Polish string-dependent method. This classification is made on the basis of the characteristics of the recognition method. An analysis is also made on them. The advantages and disadvantages of these techniques are discussed.

A new technique introduces a procedure, which generates a homogeneous tree with the minimum path length of a specified degree k. The technique is particularly important for large machines that require more than two operands at a time for an arithmetic operation.

Another new technique introduced recognizes parallelism within an expression and tests "well formation" completely in only one pass.

Index Terms—Explicit approach, implicit approach, maximum parallel form, parallelism, parse tree, Polish string, recursion, "well formation."

I. INTRODUCTION

As computer applications increase, there has appeared many a problem that requires more computational power than can be satisfied by contemporary computers. This requirement can be found in the areas of scientific applications such as space exploration, computer utility, or large complex problems like weather forecasting. The recent architectural developments in multiprocessor computers is one of the responses to the requirement. An important characteristic of a multiprocessor computer is that it contains many identical processors, sharing a common main memory, that execute their own tasks concurrently. However, it is necessary to reorganize the programs in the manner that shows parallel processable task streams to utilize a multiprocessor computer efficiently, since most programs can be executed one statement at a time because of their sequential organization.

This reorganizing work involves two steps: recognition of parallel processable tasks and their representation for execution on a multiprocessor computer system. This can be done at the time when programs are written by programmers indicating explicitly parallel processable tasks by analyzing the algorithms. Alternatively, this can be done by automatic software programs that can recognize parallelism in sequentially organized programs. In general, the latter is referred to as the implicit approach and the former as the explicit approach [1].

In the explicit approach, parallelism in a program is indicated by a set of special instructions like the FORK or the JOIN [2]. The FORK instruction initiates concurrently the execution of several instructions that can be executed in parallel and are synchronized at a certain instruction by the JOIN instruction. Therefore, this approach does not incur any overhead for the recognition of parallel processable tasks, but it imposes on programmers additional duties to analyze algorithms or procedures in detail.

In the implicit approach, an automatic software program accepts sequentially organized programs as its inputs and produces parallel task tables (parallel processable graph) or a syntactic tree that shows parallelism explicitly. This approach involves considerable overhead to recognize parallel tasks in a program although it relieves programmers of additional duties.

The term "task" here, in general, refers to a "self-contained portion of a computation that once initiated can be carried out to its completion without need for additional inputs" [1]. Therefore, parallelism among tasks can exist at many levels, i.e., from subprogram level to the level of microstep. The techniques for the recognition and the representation of parallelism are different depending upon the task level concerned. Bernstein has investigated the conditions for the determination of parallel processability between two tasks [3], and Ramamoorthy and Gonzales have investigated parallelism between instructions [1]. Also, parallelism within an algebraic expression has been studied in [1], [4]–[7].

The explicit approach is advantageous for the recognition and representation of parallelism between blocks of instructions or between instructions, since the analysis of parallelism between tasks at these levels is simple. However, the explicit approach is not advantageous for recognizing and representing parallelism at arithmetic operation (subexpression) or microstep level, because it is tedious and mistake prone.

Parallelism between subexpressions in an expression is important, particularly for array processors (Illiac IV, Pepe) or pipelined computer systems (Star 100 or CDC 6600) since their performance depends upon the parallel processable arithmetic operation streams. Parallelism between blocks of instructions can be more efficiently utilized, however, by multiprocessor computer systems.

Manuscript received September 21, 1972; revised March 22, 1973. This work was supported by NSF Grant GJ-35839.

The authors are with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Calif. 94720.
Table I

<table>
<thead>
<tr>
<th>Classification</th>
<th>Technique</th>
<th>Input</th>
<th>Restriction</th>
<th>Number of Passes</th>
<th>Output</th>
<th>Basic Approach</th>
<th>Test for “Well Formation”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precedence-dependent method</td>
<td>TOMS</td>
<td>arithmetic expression</td>
<td>binary and unary operator</td>
<td>more than the number of tree levels</td>
<td>$R_l$ (operand 1 operator, operand 2, start level, end level)</td>
<td>operator precedence</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>TOAS</td>
<td>arithmetic expression</td>
<td>binary and unary operator</td>
<td>equal to the number of tree levels</td>
<td>$R_l = \text{operand 1}, \text{operand 2}$</td>
<td>operator precedence</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>TWOAS</td>
<td>arithmetic expression</td>
<td>binary and unary operator</td>
<td>one pass</td>
<td>operand 1, operator, $2, R_l$</td>
<td>operator precedence</td>
<td>complete test for “well formation”</td>
</tr>
<tr>
<td>Syntax-dependent method</td>
<td>EODG</td>
<td>reverse Polish string</td>
<td>binary operator only</td>
<td>equal to the number of tree levels</td>
<td>operator, operand 1, operand 2, $-R_l$</td>
<td>property of Polish string</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>EIML</td>
<td>arithmetic expression</td>
<td>binary and unary operator</td>
<td>three passes</td>
<td>[op. 1, op. 2, op. $n$, $R_l$, Level]</td>
<td>property of Polish string</td>
<td>indicate the presence of errors</td>
</tr>
<tr>
<td></td>
<td>EIME</td>
<td>arithmetic expression</td>
<td>binary and unary operator</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In this paper we will consider the implicit approach to parallelism between subexpressions in a compound arithmetic expression. This includes the compilation techniques of arithmetic expressions for the recognition and representation of parallel processable subexpressions. Several techniques in this area already exist. These techniques accept arithmetic expressions and generate syntactic trees indicating parallelism between subexpressions, i.e., subexpressions at the same level in the tree can be executed in parallel. These previous techniques will be reviewed in the following section. Also we shall introduce some new techniques.

New techniques that yield optimal solutions to the generation of minimum height trees and minimum execution time trees are developed and verified. They are useful to machines that require two or more operands at a time for an arithmetic operation. Another new technique introduced recognizes parallelism within arithmetic expression and tests “well formation” in only one pass. Advantages and disadvantages of the new techniques are compared with those of the previous techniques. A consideration is given to possible further research areas in this field.

II. Compiling Techniques

There are several techniques for the recognition and representation of parallelism within an arithmetic expression. They differ in many aspects: input forms, intermediate language produced, recognition method, or number of levels in a tree generated, etc.

Conceptually, these techniques can be classified into three categories: 1) operator precedence-dependent method; 2) syntax-dependent method; and 3) Polish string-dependent method. This classification is made on the basis of the recognition method of each technique, rather than the introducer of each technique. The first method utilizes precedence relationships between operators, the second method the syntax of the input language, and the third method uses the property of Polish string corresponding to arithmetic expression.

In this section, we shall review the previous techniques by category with comments on their advantages and disadvantages. Also, three new techniques will be introduced in this section. One is in the category of operator precedence-dependent method, and the other two are in the category of Polish string-dependent method. The characteristics of these techniques are shown in Table I. The new techniques are also included in the table for the sake of comparison. The usual precedence relationships between operators will be assumed.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>unary minus</td>
<td>4</td>
</tr>
<tr>
<td>exponentiation ($*$)</td>
<td>3</td>
</tr>
<tr>
<td>multiplying operator ($*$,/)</td>
<td>2</td>
</tr>
<tr>
<td>adding operator ($+$,−)</td>
<td>1</td>
</tr>
</tbody>
</table>

The expression $A+B+C+D*E+F+G+H$ will be used to illustrate these techniques.

A. Precedence-Dependent Method

The basis of the operator precedence-dependent method is the precedence relationships between operators. In other words, subexpressions are generated depending upon the determination of the precedence comparison between two or three operators. The objective of the comparison is to identify the executable operator (operators) among the operators involved in the comparison without violating precedence relationships. Therefore, an input arithmetic expression does not need to be transformed into another form in this method. Parallelism among subexpressions is indicated by the execution level, that is, subexpressions at the same execution level can be executed concurrently. This execution level is determined by the level of operands in a subexpression or the $i$th pass of input string. Three techniques are in this category, and they
differ in the number of operators involved in the comparison and the number of operators identified for the generation of subexpressions. One technique compares two operators and identifies multiple operators for subexpression generation, and the other uses three operators for comparison and identifies one operator. A new technique is introduced in this category that compares two operators to select an operator for subexpression generation. These techniques are described in detail in the following paragraph.

1) Two Operators for Multisubexpressions (TOMS): The first function of this technique is to identify the rightmost subexpression that consists of the locally highest precedence operators. This is achieved by comparing two adjacent operators during two passes from right to left and left to right. Therefore, this technique requires multipasses. And the execution level of a subexpression is determined by the level of component operands. This technique was introduced by Squire [4].

Unary minus can be a component of an input arithmetic expression. Two space delimiters \( \dagger \) and \( \dagger \) are introduced at the beginning and end of the input string. The lowest precedence is assumed for these delimiters. The following steps describe the technique in detail.

Step 1—[RIGHT-LEFT]: Scan the input string from the rightmost operator to the left until an operator is found whose precedence is lower than that of the previously scanned operator. For example, this step yields \( D*E*F+G+H \) for the given expression.

Step 2—[LEFT-RIGHT]: Scan the substring resulted from Step 1 from left to right until an operator is found whose precedence is different from that of the leftmost operator of the substring. The substring at the end of Step 2 contains operators of the highest precedence from the right of the input string. For the given expression, this step yields \( D*E*F \).

Step 3—[QUINTUPLE]: Generate quintuples from the substring produced by Step 2 until no more quintuples can be generated. The form of a quintuple is as follows.

\[ R_i \langle \text{operand} \rangle \langle \text{operator} \rangle \langle \text{operand} \rangle \langle \text{start level} \rangle \langle \text{end level} \rangle \]

where \( \langle \text{start level} \rangle = \max \{ \text{end level operand 1; end level operand 2} \} \) and \( \langle \text{end level} \rangle = \langle \text{start level} \rangle + 1 \). The operands 1 and 2 are the operands whose levels are lowest in the substring. The initial operands in the input string are assumed to have end level zero.

Step 4—[REPLACE]: Replace one of the operands with the temporary result \( R_i \) and delete the other together with the operator. For the given example, these two steps produce: \( R_1 \langle D \rangle \langle *, \rangle \langle 0, 1 \rangle \rangle ; R_1+F+G+H \) and \( R_2 \langle F \rangle \langle *, \rangle \langle 1, 1 \rangle \rangle ; R_2+G+H \). The associated procedure is reiterated after Step 4. The string resulting from Step 4 of one iteration becomes the input string to Step 1 of the next iteration—in this example, \( A+B+C+R_2+G+H \). (See Fig. 1.)

In this technique, all temporary results \( R_i \) that have the same start level can be computed in parallel. The number of passes required varies according to the component operators in the input string. For example, the expression given above requires two left and right scans. However, an input string containing only operators of the same precedence will require one left and right pass. However, the search for two operands at lowest level may need more scanning operations. Therefore, this technique may need a number of passes equal to or greater than that of the syntactic tree levels generated. These multiple passes in turn rearrange the input string by grouping together the substrings resulted from a scanning cycle, and produce syntactic trees with the minimum levels in most cases. However, this reordering may result in a modification of the evaluation order of the original input string. The quintuples have a form appropriate for conversion to machine code, although it may be necessary to sort the quintuples according to the start level before conversion. Also, the test for “well formation” of the input string is not taken care of explicitly by the technique even though the power of the technique is suitable for the compilation of production programs.

2) Three Operators for a Subexpression (TOAS): The function of this technique is to recognize, among three operators, an operator to generate a subexpression without violating precedence relationships. The three operators are the two operators immediately to the left and to the right of the operand currently scanned and one of the operators scanned previously, that is, at the top of a stack, in which the operators previously scanned are stored. The actions and operators involved in each comparison are described in Steps 2-4 below.

This technique was introduced by Baer and Bovet [5]. Multipasses are used in this technique from the leftmost to the rightmost symbol of an input string. Each pass is performed
with the comparisons among three operators. Unary minus is permissible in an input string. In the description below, \text{ITEM} represents the operand currently scanned, \text{SCOP} the operator to the right of \text{ITEM}, \text{LSCOP} the operator to the left of \text{ITEM}, \text{STACK} (1, IS) the storage for operands, \text{STACK} (2, IS) the storage for operators, and \(P(\text{variable})\) means the precedence of the operator contained in the variable.

\text{Step 1–[SCAN]}: Scan next two symbols, \text{ITEM} (an operand), and \text{SCOP} (an operator), and GO TO Step 2.

\text{Step 2–[PUSH DOWN]}: Push down \text{ITEM} and \text{SCOP} on \text{STACK} (1, IS) and \text{STACK} (2, IS), respectively, and GO TO Step 1 if: 1) \text{STACK} is empty, or 2) \(P(\text{SCOP}) > P(\text{LSCOP})\), where \text{LSCOP} is initialized with “+” and later assigned the value of the previous \text{SCOP}. Otherwise GO TO Step 3.

\text{Step 3–[OUTPUT STRING]}: Add \text{ITEM} and \text{SCOP} to the output string and GO TO Step 1, if \(P(\text{LSCOP}) \neq P(\text{STACK} (2, IS))\). Otherwise GO TO Step 4.

\text{Step 4–[GENERATE]}: Generate a temporary result of the form: \(T_k = <\text{operand} 1> <\text{operand} 2> = \text{STACK} (1, IS) \text{STACK} (2, IS) \text{ITEM}, \) and add \(T_k\) and \text{SCOP} to the output string, and GO TO Step 1. In this case \(P(\text{LSCOP}) = (\text{STACK} (2, IS))\). One of the operations above is performed during a pass when the corresponding case arises, and the scanning process is repeated until the whole expression is compiled. For the given example, the output string becomes \(T1 + T2 * F + T3 + H\) at the end of the first pass, and the subexpressions generated during the pass are: \(T1 = A + B, T2 = D * E, \) and \(T3 = C + G\).

Fig. 2 shows the details of the first pass.

This technique requires as many passes as the number of the syntactic tree levels generated. Each pass generates the subexpressions that can be executed in parallel, and the subexpressions generated during the \(n\)th pass have the same execution level \(n\). This technique also produces syntactic trees with the minimum levels by reordering the original input string. Therefore this technique is quite useful for the compilation of production programs even though it may not preserve the evaluation order of the original input string. There is no test for “well formation” and this is a disadvantage of this technique.

3) Two Operators for a Subexpression (TWOAS): A new technique in this category has been developed considering the disadvantages of the previous techniques, i.e., multiple passes, no test for “well formation,” or modification on the original evaluation specification. To achieve the purposes, the new technique has to do the following.

a) Scan input string only once from left to right.

b) Test “well formation” (syntactic correctness) of the input string and recover errors if any occur during the pass.

c) Generate subexpressions indicating the execution level in the pass.

The compilation process is controlled by the precedence relationship between two operators, i.e., the operator currently scanned and operators scanned previously. The technique is described in detail in the following.

Input: The new technique accepts arithmetic expressions as its input. As in the other technique, the expressions can contain unary minus, exponentiation, or parentheses in any

\begin{table}
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
\text{LSCOP} & \text{ITEM} & \text{SCOP} & \text{IS} & \text{STACK} (1, IS) & \text{STACK} (2, IS) & \text{Temporary results} & \text{Output string} \\
\hline
+ & A & + & 1 & A & + & 0 & T1+ \text{A} \text{H} \text{E} + \\
+ & B & + & 0 & 0 & 0 & T1+ & T2+ \\
+ & C & + & 1 & C & + & T2+ & T2+ \\
+ & D & * & 2 & D & * & T2+ & T2+ \\
* & E & * & 1 & C & + & T2+ & T2+ \\
* & F & + & 1 & C & + & T2+ & T2+ \\
+ & G & + & 0 & 0 & 0 & T2+ & T2+ \\
+ & H & + & 0 & 0 & 0 & T2+ & T2+ \\
\hline
\end{tabular}
\caption{Input: \text{AMB4003}\text{G31001199}H}
\end{table}

\[\text{Fig. 2. Compilation result by TOAS.}\]

The operations performed are described. In the following description, \(P(\text{variable})\) also means the precedence of the operator in the variable and the symbol currently scanned is contained in the variable \(N\).

\text{Step 1–[SCAN]}: Scan next symbol (\(N\)), check “operator or operand missing error,” and GO TO Step 2.

\text{Step 2–[OPERAND]}: Push down the operand in \(N\) and the level of the operand on \text{NSTK} and GO TO Step 1, if \(\text{CONTENT}(N)\) is an operand. Otherwise GO TO Step 3.

\text{Step 3–[CONTENT(N) = Operator or (\text{"/"})]}: Push down the operator in \(N\) and \(P(\text{N})\) on \text{OPSTK} and GO TO Step 1, if:

a) \(\text{CONTENT}(N)\) is (\(\text{"/"}\)),

b) \text{OPSTK} is empty; or

c) \(P(\text{N}) > P(\text{OPSTK}(M, 1))\).

where \(P(\text{"/"})\) assumes the value of zero and \(M\) represents the top of \text{OPSTK}. Otherwise GO TO Step 4.
Step 4—\( \text{CONTENT}(N) = "\) : Do “generation” until \( \text{CONTENT}(\text{OPSTK}(M, 1)) \) becomes “(” or \( \text{OPSTK} \) becomes empty, if \( \text{CONTENT}(N) \) is “). Discard “(” in \( \text{OPSTK}(M, 1) \) and “)” in \( N \) and go to Step 1 when the termination conditions above arise. Otherwise go to Step 5.

In this step, unbalanced parenthesis (more right parentheses) is recognized if \( \text{OPSTK} \) becomes empty. And “generation” means that: “Generate a subexpression of the form”:

\[
\langle \text{operand} 1 \rangle <\text{operator}> <\text{operand} 2> = \text{NSTK}(K-1,1) \text{ OPSTK}(M,1) \text{ NSTK}(K,1)
\]

where \( \langle \text{execution level} \rangle = \max(\text{level of op.1}; \text{level of op.2}) + 1 \), i.e.,

\[
R_i - M = M - 1, K = K - 1, \text{and} \text{NSTK}(K, 1) \Rightarrow R_i, \text{NSTK}(K, 2) \Rightarrow n.
\]

Step 5—\( \{ \text{OPSTK}(M, 1) \rangle \Rightarrow \text{P}(N) \} \) : Do “generation” while \( \text{P}(\text{OPSTK}(M, 1)) \Rightarrow \text{P}(N) \) and \( \text{OPSTK} \) is not empty. Push down the operator in \( N \) and \( \text{P}(N) \) on \( \text{OPSTK} \) and go to Step 1, if the condition above fails. Otherwise go to Step 6.

Step 6—\{\text{END}\}: Do “generation” until \( \text{OPSTK} \) becomes empty. Then sort the subexpressions by level number if the end of an input string is recognized. In this step, unbalanced parenthesis (more left parentheses) is recognized if any left parenthesis exists in \( \text{OPSTK} \).

Fig. 3 shows the compilation process of the new technique for the example given earlier. The number below each operator in the expression represents the operator at the corresponding location. The syntactic tree produced is shown at the left together with the subexpressions generated and the contents of the stacks at the right. To illustrate the new technique in more detail, the compilation process of a complicated expression, \( A-B+C*(D/E)**-F+G*H \), is shown in Fig. 4. The recognition of unary minus and syntactic errors is described below.

Unary Minus and Test For “Well Formation”: Unary minus and syntactic errors in input string are recognized by the content of two switches and \( \text{CONTENT}(N) \). The two switches are used to indicate the next syntactic element (ISYN) and the appearance of a left parenthesis or an exponentiation (LPAR). ISYN is set to the value “OPND” if \( N \) contains an operator or a left parenthesis. Otherwise ISYN assumes the value “OPER.” It is initialized with “OPND,” which means that the next syntactic element should be an operand. And LPAR is switched “ON” whenever a left parenthesis or an exponentiation is encountered. Otherwise it is switched “OFF” and initially switched “ON.”

Therefore, the combination of ISYN, LPAR, and \( \text{CONTENT}(N) \) tells what the next syntactic element in the input string should be. When an error is found in Steps 1, 4, or 6 in the procedure above, the corresponding error message is printed and the error is corrected. After the correction, the operation of each step continues until the termination condition at the step arises. The correction involves the insertion or deletion of a syntactic element. Each case is described below.

Unary minus is recognized if LPAR = “ON,” ISYN = “OPND,” and \( \text{CONTENT}(N) = "-" \). A literal constant zero is pushed down on NSTK in this case.

Operator missing is detected between two operands or an operand and a left or right parenthesis when ISYN = “OPER” and \( \text{CONTENT}(N) \) is an operand in Step 1, “(” is inserted between the elements in this case.

Left parenthesis missing is detected in Step 4 when sub-expression generation continues until \( \text{OPSTK} \) becomes empty. In this case, the generation is halted and the right parenthesis is ignored.

Right parenthesis missing error is recognized in Step 6 when a left parenthesis appears in \( \text{OPSTK} \) during the subexpression generation at the end of an input string. The left parenthesis is deleted in this case. Fig. 5 shows the compilation result of arithmetic expressions with syntactic errors.

Implementation: The new technique has been programmed in Fortran and tested on the CDC 6600. The program has about 260 Fortran statements and consists of MAIN program and two subprograms, KOIDE and SCAN. MAIN controls SCAN and KOIDE, tests “well formation,” and manages operand and operator stacks. SCAN simply returns a symbol to MAIN and detects “ill-formed” variable names. And KOIDE generates sub-expressions and determines the level of the subexpression generated.

About 100 arithmetic expressions have been tested. The expressions containing fewer than three operators are not included in the data since they have to be evaluated sequentially. The average number of operators in the data was about six. And the average recognition rate was about 90 percent. This rate is increased proportionally to the number of operators in an arithmetic expression. The recognition rate is relative to the minimum syntactic tree levels.

\[
\text{recognition rate} = \frac{\text{minimum number of levels in a tree}}{\text{number of tree levels generated by the new technique}} \times 100.
\]

As an example, if a syntactic tree of an expression has a minimum of 4 levels and the number of tree levels generated by the new technique is 5, then the recognition rate = \( \frac{4}{5} \times 100 = 80 \text{ (percent)} \).

Comparison: The complexity of the new technique is of the order of \( n \) where \( n \) is the number of symbols in the input expression. To be more explicit, basically the recognition procedure scans the input expression of \( n \) symbols from left to right once and compares instantaneous adjacent operators to decide which should be executed before the other. Therefore, the number of comparisons and temporary result generations is \( (m-1) \) where \( m \) is the number of operators. Since \( m \) is smaller than \( n \), therefore, the claim that the complexity is of
the order of $n$ is justified. (There are altogether $n$ scan operations, $m-1$ comparisons, and $m-1$ temporary result generations.)

Thus this method is relatively simpler than the previous techniques that require multipasses (scans) or recursion. Also, "well formation" is easily tested by some additional comparisons within the same pass. Unfortunately, its recognition rate is not 100 percent. (Recall that for a multipass technique, the number of passes needed is not fewer than the tree height.)

The subexpressions generated are suitable for execution on multiprocessor computers or pipeline computers. The new technique is applicable in real-time environments because the one pass it requires is just as illustrated above, and is also easily adaptable to compilers that use operation precedence relationships for the compilation of arithmetic expressions, since the new technique uses the same relationships. One disadvantage of the new technique is that it does not generate optimal trees for some input expressions, though an experiment showed that it had a high recognition rate of 90 percent.

### B. Syntax-Dependent Method

This method is a variation of the syntax-directed compilation technique, which is based on a formal description of the syntax of the input language. Therefore, the compiling processes are intimately connected with the syntax of the arithmetic expressions of the input language.

There is only one technique in this category, which is introduced by Stone [6]:

The language accepted by the technique is described in modified Backus normal form (BNF) which uses metasymbols: «::=» (is defined to be), «< >» (bracket for syntactic unit), «'X''Y''» (means "empty | X | XX | XXX | ...", (means "empty | X | XX | XXX | ..."), (means "empty | X | XX | XXX | ...")
where the adding operators are “+” and “-,” the multiplying operators are “*” and “/”, and the ↑ denotes exponentiation.

In the language description, the syntax of expression and term is essential for the technique. In other words, the compilation processes of the technique are directed by the syntax of an expression or a term. The technique attempts to form an expression when the syntactic unit currently scanned is an adding operator, or a term when the syntactic unit currently scanned is a multiplying operator. The precedence relationships between operators are defined by the grammar: in this case, factors will be recognized before terms, terms before expressions. Whenever the technique recognizes an expression or a term, it produces reverse Polish string corresponding to the expression or the term. The other syntactic units like variable, primary, or factor are added to the output string in reverse Polish form.

To represent the parallelism in an input string, the technique keeps track of the tree level in the output string. For example, when an adding operator is scanned, the technique attempts to form an expression at the tree level n by combining two subexpressions (operands) at the tree level n - 1, with the adding operator. Therefore, to form an expression at the tree level n the technique uses recursive calls on the procedure that recognizes another subexpression (operand) at the tree level n - 1 for the subexpression (operand) at the same level compiled up to now. The basic idea of the technique can be explained more intuitively with the parse tree generated by the technique. First let us take a simple example A + B + C + D + E + F + G + H.

Fig. 6 shows the parse tree of the expression produced by Stone's algorithm. In the figure, T(E) means a term (an expression).

The technique produces a term A at level 0 at the beginning of compilation since A is a variable and scans “+”. Since “+” is an adding operator, the technique attempts to form an expression at level 1 which is one level higher than the term compiled up to now. To form an expression at level 1, the technique searches another term at level 0, which is B. At this point, the technique produces an expression at level 1 in reverse Polish form, i.e., AB+. Now the next syntactic unit scanned is an adding operator; the technique again attempts to form an expression at level 2 by searching a subexpression at level 1. To search a subexpression at level 1 the technique uses the same procedure to generate the subexpression AB+. Therefore, the procedure outputs C at level 0 and scans “+.” And it again searches D at level 0 and combines C and D to form a subexpression CD+ at level 1. At this point, the technique combines these two subexpressions to form an expression at level 2, that is, AB+CD++. Now the technique scans the fourth operator “+,” and it attempts to form an expression at level 3 by finding another subexpression at level 2. Therefore, the subexpression EF+GH++ obtained by the same procedure, is combined with AB+CD++ to form an expression at level 3, AB+CD++EF+GH+++

In the description of the technique, the procedure EXPRESSION combines subexpressions (operands) at level n - 1 to form an expression at level n, and increases the tree level by 1. The subexpression at level n - 1 is compiled by the recursive procedure TERM, which is called by EXPRESSION. To form a term including multiplying operators at level m, TERM calls another recursive procedure FACTOR to find a subterm (a term at one level lower) at level m - 1. The relationship between TERM and FACTOR is similar to that between EXPRESSION and TERM.

Now let us consider the example given earlier in this section, A + B + C + D + E + F + G + H. The parse tree of the expression is given again in Fig. 7 for easy understanding. The technique begins to compile by producing A, a term at level 0. Now the syntactic unit scanned is “+,” EXPRESSION increases the level by 1 and calls TERM to compile another term at level 0. TERM outputs B, a term at level 0, and returns control to EXPRESSION after combining these two terms to form an expression at level 1. Then EXPRESSION increases the tree level by 1. Therefore, we have AB+ and the level 2. Now EXPRESSION attempts to form an expression at level 2 by calling TERM to find another subexpression at level 1.

\[1\] An expression at one level lower.
expression at level 1 with another term D at level 0. However, TERM cannot combine C and D to form a subexpression at level 1 since the next syntactic unit is "*". Therefore, TERM calls FACTOR to find a factor at level 0 to form a subterm at level 1 before combining C and D with "*". Now FACTOR finds another factor E at level 0, combines D and E to form a subterm at level 1, i.e., AE+CDE*. And FACTOR returns control to TERM. Then TERM increases the subterm level by 1 for the next term and checks the next syntactic unit. Then the next syntactic unit is "*", TERM calls FACTOR to find a subterm at level 1 to form a term at level 2. Now FACTOR outputs F and scans "*", the sixth operator in the expression. At this point FACTOR outputs "*" and returns control to TERM. Now TERM combines C and DE*F*, a term at level 0, to form a subexpression CDE*F** at level 1 and returns control to EXPRESSION after combining two subexpressions at level 1, AB+ and CDE*F**, to form an expression at level 2, AB+CDE*F**++. Now EXPRESSION increases the tree level by 1 for the next expression. Then TERM which is called by EXPRESSION combines G and H to form a subexpression at level 1. Now the input string is processed to the end, EXPRESSION finishes compilation by combining two subexpressions, AB+CDE*F**++ and GH+ to form a final expression at level 3, AB+CDE*F**++GH++.

The syntactic tree corresponding to the output string is shown in Fig. 8. This technique uses only one pass and it produces a maximally parallel syntax tree. However, it is slow due to the recursion. As an example, it requires four recursive calls on TERM for a full binary tree like \( A+B+C+D+E+F+G+H \).

In general, it requires \( \Sigma_{i=1}^{l} (M_{RI} - 1) \) recursive calls on TERM for an expression containing \( 2^n \) terms, and the total number of calls on TERM including recursive calls are \( \eta + \Sigma_{i=1}^{l} (M_{RI} - 1) + V \), where \( l \) is the number of root nodes in a tree, \( M_{RI} \) is the level of the root node \( RI \), \( V \) is the number of variables in the input expression, and \( \eta \) is the level of the root node of the tree generated. For the expression above, \( \eta = 3, l = 7, \Sigma_{i=1}^{l} (M_{RI} - 1) = 4, \) and \( V = 8 \). This technique does not test "well formation" explicitly, although it can be modified to accommodate it. Likewise, the technique can be modified so that unary minus is acceptable.

C. Polish String-Dependent Method

The idea of this method is to utilize the property of reverse Polish form of an arithmetic expression. Therefore, arithmetic expressions have to be translated into reverse Polish form.

In reverse Polish form, it is possible to recognize two related operands for a given operator in the form. Two related operands are adjacent to the given operator, but can be explicit or implicit. As an example in "AB*C+", "AB" is explicit to "*", but "AB" is not explicit to "+". In brief, an explicit operand can be defined as an original operand adjacent to an operator, and an implicit operand as a temporary result generated by previous computation.

There are two techniques in this category. One technique detects two explicit operands for an operator and generates a subexpression. The new technique in this category recognizes two explicit or implicit operands for an operator and rearranges the operands to form a maximum parallel form. In this form original operands related to the same operator are combined together.

These techniques are described in the following paragraphs. The translation process of an arithmetic expression into reverse Polish form is explained first as a background.

1) Translation of Arithmetic Expression into Reverse Polish Form: The translation process is facilitated by the use of two stacks: a stack for Polish form and another for operators. During translation, operands pass directly from an input expression to Polish stack, but operators are added to Polish stack through operator stack. The steps below describe the translation process of parenthesis-free expression.

Step 1—[SCAN]: Scan next element and GO TO Step 2.
Step 2—[OPERAND]: Push down the operand on the Polish stack and GO TO Step 1 if the scanned element is an operand. Otherwise GO TO Step 3.
Step 3—[OPERATOR]: Pop up the operator at the top of the operator stack and push it down on the Polish stack while the precedence of the operator at the top of the operator stack is equal to or greater than that of the operator scanned. If the opposite case arises, push down the operator scanned on the operator stack and GO TO Step 1. Otherwise GO TO Step 4.
Step 4—[END]: Pop up the operators in the operator stack and push down on the Polish stack until the operator stack becomes empty if the end of an arithmetic expression is recognized. As an example, the expression given above is converted to the reverse Polish string by the above steps: AB+C*DE*F++G+H+.

2) Explicit Operands for Direct Generation (EODG): This technique assumes reverse Polish form corresponding to an arithmetic expression as its input and that the expression does not contain unary minus.

During scanning of reverse Polish form, the technique searches two explicit operands when an operator is encountered. If it finds them, a subexpression is generated.
Otherwise the scanning proceeds to the next operator. The following steps describe the operations performed during a scan. For the example given earlier, this technique assumes \( AB+CD+DE*F+G+H^+ \) as its input.

**Step 1—[SUBSTRING]:** Find the substring of the form: 
\(<\text{operand}\> <\text{operand}> <\text{operator}>.\)

**Step 2—[GENERATE]:** Generate a subexpression of the following form with the components of the substring, i.e., 
\(<\text{temporary result } Ri> <\text{operand } 1> <\text{operator}> <\text{operand } 2> <\text{execution level}>\)

**Step 3—[REPLACE]:** Replace the substring with the temporary result. For the given example, these steps yield

\[
\begin{align*}
R1 & \ A + B \ 1; \ R1 \ C + DE*F* + G + H^+ \\
R2 & \ D + E \ 1; \ R1 \ C + R2 F* + G + H^+.
\end{align*}
\]

The string resulting from the pass, \( R1 \ C + R2 F* + G + H^+ \), becomes the input string for the next pass and the temporary results in the string are considered as operands. Fig. 9 shows the compilation process of the example given above. The scanning operation of this technique is simple. However, this technique cannot detect parallel processable subexpressions in the expression containing only operators of the same precedence, although it uses as many passes as the number of tree levels generated. Each pass generates the subexpressions that can be executed in parallel. Therefore, the subexpressions generated at the \( n \)th pass have execution level \( n \). The intermediate language is suitable for the execution on a multiprocessor computer or pipelined computer since all subexpressions have their execution level and they are already sorted by the level. But the test for “well formation” is not performed in this technique.

3) **Explicit or Implicit Operands for Minimum Level Tree Generation (EIML):** A new technique is introduced in this paragraph. The goal of the new technique is to generate a tree with minimum path length. The basis of the technique is to interchange two explicit or implicit operands of an operator in such a manner that original operands related to the same operator in Polish string are combined. The second pass (Step 2) below corresponds to this operation.

This technique assumes arithmetic expression as its input. Unary minus and parentheses are permissible. Three passes are used in this technique. The first pass produces reverse Polish form and checks “well formation” of the expression. The second pass rearranges the Polish string into a maximum parallel form. The third pass generates the syntactic tree showing the executable parallel processable subexpressions with a minimum number of levels.

**Step 1—[POLISH]:** Rewrite arithmetic expression in reverse Polish form and reverse its order. For the example given above, this step yields: \(+H+G++F*ED+C+BA.\) During this process, “well formation” is checked by computing the weight of each symbol of the string to its corresponding member on the basis of the following procedure.

Assign to symbol \( S_i \) the value \( V_i = V_{i-1} + R_i, i = 1, 2, \cdots n \) where \( R_i = 1 - (S_i) \) and

\[
\begin{align*}
&\begin{cases}
0, & \text{if } S_i \text{ is a variable} \\
1, & \text{if } S_i \text{ is a unary operator} \\
2, & \text{if } S_i \text{ is a binary operator.}
\end{cases}
\end{align*}
\]

Therefore, \( V_{i-1} = V_{i-2} + R_{i-1} \cdots \) such that \( V_1 = R_1 \) and \( V_0 = 0. \) For the example, this procedure yields

\[
\begin{align*}
i & \quad 15 \quad 14 \quad 13 \quad 12 \quad 11 \quad 10 \quad 9 \quad 8 \quad 7 \quad 6 \quad 5 \quad 4 \quad 3 \quad 2 \quad 1 \\
S_i & \quad + \quad H \quad + \quad G \quad + \quad * \quad F \quad * \quad E \quad D \quad + \quad C \quad + \quad B \quad A \\
V_i & \quad 1 \quad 2 \quad 1 \quad 2 \quad 1 \quad 2 \quad 3 \quad 2 \quad 3 \quad 2 \quad 1 \quad 2 \quad 1 \quad 2 \quad 1.
\end{align*}
\]

For a “well-formed” expression of \( n \) symbols, \( V_n \) equals 1. This computation is certainly not necessary if the test for “well formation” is not needed.

Before describing Step 2, the following definitions and equivalent relations are used.

**Definition:**

1) The degree of an operator is the number of operands directly related to it (though the number of operands required for each operation is less than or equal to the degree).

2) \( \odot_2 \) is the commutative operator of degree 2. \( \odot_i \) is the commutative operator of degree \( i \); the “*” operator can, for example, be “*” or “*.”

3) \( f = V_0 \odot f_1 f_2 \cdots f_i, \) where \( V \) means a variable and \( f \) an operand.

**Equivalence Relation:** Between the same commutative operators

\[
\begin{align*}
1) & \quad \odot_2 f_1 \odot_2 f_2 f_3 \cdots f_{i+1} = \odot_2 f_1 f_2 f_3 \cdots f_{i+1} \quad \text{commutativity} \\
2) & \quad \odot_2 f_1 f_2 f_3 f_{i+1} = \odot_2 f_1 f_2 f_3 f_{i+1} \quad \text{equivalence.}
\end{align*}
\]

In general,

3) \( \odot_2 f_1 f_2 f_3 f_{i+1} = \odot_2 f_1 f_2 f_3 f_{i+1} \cdot f_{i+1}. \)

Between different commutative operators

4) \( \odot_2 f_1 f_2 f_{i+1} = \odot_2 f_1 f_2 f_{i+1} \cdots f_i. \)

**Examples:** \( \odot_2 A \odot_2 B C = \odot_2 B \odot_2 A \odot_2 C = \odot_2 B \odot_2 A \odot_2 C \) by 1), 2), 3) \( \odot_2 B \odot_2 A \odot_2 C = \odot_2 B \odot_2 A \odot_2 C \) by 4).

Then Step 2 can be executed as follows.
Step 2—[DEVELOP MAXIMUM PARALLEL FORM]: Scan Polish string resulting from Step 1 from the rightmost symbol to the leftmost. Whenever an operator is encountered, examine the relations described above. If relations 1), 2), and 3) are applicable, combine the operators according to 3). If only the relation 4) holds, interchange two operands according to 4). For the given example, this yields $+_{s}HGCBAO_{s}FED.\textsuperscript{2}

Noncommutative Operator and Unary Minus: In order that the previous step works properly, noncommutative operators and unary minus should be replaced in the following manner.

1) Noncommutative Operator:

$$a) -f_{1}f_{2} = +f_{1}f_{2}$$

$$b) lf_{1}f_{2} = +f_{1}f_{2}.$$  

During code generation, the operators changed should be redefined in the reverse way.

2) Unary Minus:

$$\langle \text{unary minus} \rangle f_{1} = \overline{f_{1}}.$$  

It is assumed that the negative value of a variable can be loaded. Also any temporary result of $T_{i}$ means $-T_{i}$.

Step 3—[GENERATE PARALLEL TREE WITH THE MINIMUM LEVEL]: For a given maximum parallel form we can construct various trees with different maximum path length (tree level). As an example, we can construct a tree with a maximum parallel form, $\otimes_{a}ABCD$, as shown in Fig. 10. In this example, the tree with maximum path length 2 shows two parallel operations, but the other tree does not. In brief, sometimes it is important in parallel processing of arithmetic expression to construct a tree in such a way that the maximum path length is minimized.

This problem is not that simple when a maximum parallel form has multilevels. In this case the maximum path length of a tree depends upon the strategy attaching the subtrees constructed at the $i$-th level to nodes of the trees constructed from the $i$th level of a maximum parallel form.

The procedure given in this step constructs a tree such that its maximum path length is minimized. Part of the procedure is analogous to the Huffman encoding technique which minimizes the average path length of the code words [9].

To generalize the application, the case is not restricted to binary operators only. Hence this method may prove valuable to machines that require more than two operands at a time for an operation. In fact, a combination of operators requiring different numbers of operands are allowed in this algorithm. The following steps describe the procedure.

Step p1: Assign a weight of 1 to each operand in the maximum parallel form; this designates its position in the lowest level, namely level 1. Initialize $i = 0$.

Step p2: $i = i + 1$. Let $m$ = number of operands required for each operation for the rightmost operator. Let $(x_{1}, x_{2}, \ldots, x_{m})$ be the levels of the $m$ least weight operands related to the current rightmost operator. Form a subtree by combining these $m$ operands into a node $T_{i}$. Compute the sum of the weight of the $m$ operands and assign it to the node $T_{i}$. Define $L_{i} = \text{level of } T_{i} = \text{max}(x_{1}, x_{2}, \ldots, x_{m}) + 1$. If the number of operands left for the rightmost operator after the replacement is 1, go to Step p4.

Step p3: Rearrange the operands remained in order of decreasing weight and repeat Step p2.

Step p4: Delete the rightmost operator and replace the weight of $T_{i}$ by $L_{i}$. If the deleted operator is the leftmost one also, the complete tree is generated and halt. Otherwise, go to Step p2.

Fig. 11 shows the details of Steps 2 and 3 for the example given earlier. The optimality of this procedure will be proved later.

This method can also be applied to expressions with parentheses. However, it cannot then be guaranteed that trees of minimum height will be obtained because this technique has ignored the improvement that can be achieved by distribution examined in [10]. Nonetheless, if it is desired that the compiler does not disturb the precedence of subexpression as indicated by the user, this point can be overlooked here.

Finally, it can be observed that the procedure can be simplified when all the operators in the expression require the same number of operands in each execution simply because the weight associated with each node of the tree will be subject to the same function of its level. Therefore, the normalization procedure of replacing the weight by the level when deleting the rightmost operator in Step p2 is not necessary.

Definition: A set of nodes $X$ with weight (total) = $(x_{1} + x_{2} + \cdots + x_{n})$ is replaceable by the set $Y$ with weight $= (y_{1} + y_{2} + \cdots + y_{n})$ if $\text{max}(x_{1}, x_{2}, \cdots, x_{n}) \geq \text{max}(y_{1}, y_{2}, \cdots, y_{n})$.

Lemma: Using the algorithm, the final tree for the expression is optimal if and only if the individual subtrees associated with each operator in the maximum parallel form is optimal.

Proof: Obvious from the properties of the maximum parallel form.

Theorem: The algorithm just outlined yields at least one optimal solution, that is, one minimum height tree for the expression.

Proof: An inductive proof is given as follows.
Theorem. Assume the operator requires $m$ operands in each execution.

Basis: For the case of $k = m$, it is certainly true.

Inductive Step: Assume the assertion is true for $k = n$.

Then for the case of $k = n+m-1$: if $u_i$ is the weight associated with the operand $i$, the initial listing can be represented by $(u_1, u_2, \ldots, u_n, \ldots, u_n+m-1)$ in descending order of weight.

If, for the first combination of $m$ operands, an arbitrary set $S = (u_{i_1}, u_{i_2}, \ldots, u_{i_m})$ is chosen with the largest weight in the set $u_{i_1}$ for some $i_1 = p < n$, then the rearranged operands after combination will be of the form

$$A = (u_1, \ldots, u_{i_1}, u_{i_2} + \cdots + u_{i_m}, \ldots, u_{i_1}, \ldots, u_{i_m})$$

where $(u_{i_1}, \ldots, u_{i_m})$ is the $m$ least weight operands for the next operation.

On the other hand, if the $m$ least weight operands are chosen for the first combination according to the algorithm, the resulting list will be of the form

$$B = (u_1, \ldots, u_{i_1} + u_{i_2} + \cdots + u_{i_m}, \ldots, u_{i_1}, \ldots, u_{i_m})$$

If it can be proved that any optimal solution of $A$ is guaranteed by the algorithm (the inductive step, since $A$ has only $n$ elements) is a solution of $B$ regardless of the initial choice of $S$, then clearly our assertion is valid. This can be proved as follows.

By applying the algorithm to the $A$ list, an optimal solution can be obtained. In particular, $(u_1, \ldots, u_{i_m})$ will be the second set combined.

Observe the following “replaceable” operations: Either

$$(u_1 + \cdots + u_{i_m}) = S_1$$

is replaceable by

$$(u_n + u_{n+1} + \cdots + u_{n+m-1}) = S'_1$$

and

$$(u_1 + \cdots + u_{i_m}) = S_2$$

is replaceable by

$$(u_1 + \cdots + u_{i_m} + u_{i_1} + \cdots + u_{i_m} - u_n - \cdots - u_{n+m-1}) = S'_2$$

or $S_2$ is replaceable by $S'_1$ and $S_1$ is replaceable by $S'_2$. The “...” is used here to cancel repeated operands. But $S_1$ and $S_2$ are either a terminal node (initial operands) or an intermediate node (temporary results) in the solution of $A$ while $S'_1$ is a node in any solution of $B$ and $S'_2$ may be grouped in $B$ to obtain such a solution. What is more important: $S_1 + S_2 = S'_1 + S'_2$. Together with the obvious property of the operation “replaceable,” remembering that each $u_i$ as the weight in the initial step is the same as the level of the operand, clearly any optimal solution to $A$ using the algorithm is also solution to $B$. Hence, by induction the assertion is validated.

Q.E.D.

4) Explicit or Implicit Operands for Minimum Execution Time Tree Generation (EIME): This technique is a modification of the previous one and provides a better criterion to obtain parallel tasks in an expression. Basically it involves the same three passes of operation.

Take the expression $A + B + C + F + G + D + E$ as an example. The maximum parallel form for this expression is $\oplus E D C B A \oplus F G$ and trees with maximum path lengths of 3 or 4 can be constructed as in Fig. 12.

Suppose in the example, the addition operation takes 1 unit of time and multiplication takes 4 units of time. It can be shown that the tree with maximum path length of 4 yields a faster execution procedure than the tree of length 3 when sufficient processors are available. It is in this respect that our method will try to tackle the problem. That is to say, it is of utmost importance in parallel processing to minimize the total execution time when the operations have different time durations.

The algorithm can be programmed as follows.

Steps 1 and 2: Same as the previous case (EIML).

Step 3: It is modified as follows.

**Step 1:** Assign a weight of 0 to all operands; this means they all start at the initial time of $t = 0$. Initialize $i = 0$.

**Step 2:** $i = i + 1$. For the rightmost operator, combine the $m$ least weight operands of weights $(u_1, u_2, \ldots, u_{i_m})$ and replace them by an intermediate node $T_i$. Let $w_i$ be the weight of the node $T_i$ and $t_{exec}$ be the time needed for the operation.
Define $w_i = \max (u_1, u_2, \cdots, u_m) + t_{\text{exec}}$. Obviously, $w_i$ is the total execution time required to obtain the intermediate result $T_i$. If the number of operands related to the current rightmost operator after the replacement is one, go to Step p4.

Step p3: Rearrange the operands in descending order of weight and go to Step p2.

Step p4: Delete the current rightmost operator. If the deleted operator is also the leftmost one, the tree is generated and halt. Otherwise to to Step p2.

An example to illustrate this procedure is worked out in Fig. 13.

The preceding procedure yields an optimal solution. This can be proved in a similar way as in the previous theorem, using the following definition.

Definition: A set of nodes $X = (x_1, x_2, \cdots, x_n)$ can be substituted by another set $Y = (y_1, y_2, \cdots, y_n)$ if $\max(X) \geq \max(Y)$, where $x_i$ and $y_i$ are weights of the nodes.

Theorem: The preceding algorithm yields an optimal solution.

Proof: Follow the same method as the previous proof, and use “substitute” instead of “replace” wherever applicable.

Q.E.D.

A few words can be said about the computational complexity involved in Step 3 of the algorithm. In the case of binary operations with $n$ operands, the complexity is $n \log_2 n$. If not all of the operators are binary (for example, some adder takes care of three operands at a time), clearly this bound can be further reduced. But this measure of complexity is meaningful only when the value of $n$ is big compared to the constant associated with this order of magnitude ($n \log_2 n$). Therefore, it can be concluded that this optimal algorithm is more complex than the new “nonoptimal” algorithm introduced in Section II-A3.

III. CONCLUSION

The previous techniques have been reviewed and some new techniques introduced according to three categories: 1) operator precedence-dependent method; 2) syntax-dependent method; and 3) Polish string-dependent method. These categories are classified on the basis of the recognition method of parallelism. The disadvantages of the previous techniques are multiple passes or recursiveness. A new technique has been introduced to take care of these disadvantages. The test results show that it is efficient. The recognition rate of the technique was high (90 percent), nevertheless it uses only one pass. Another new technique has been introduced and proved that it generates a tree whose maximum path is minimized. With some modifications, minimum execution time trees can also be generated. This technique is particularly important for large machines that require more than two operands at a time for an arithmetic operation.

More parallelism can be exploited from arithmetic expressions if we consider more than two-dimensional subscripted variables or a task of arithmetic expressions. As an example, the index terms of more than two-dimensional subscripted variables in an expression, like $B(i,j,k) + C(k,L)$, can be computed in parallel. Similarly, more parallelism can be recognized from a subtask that is a group of arithmetic expressions to be executed in sequence.

Memory conflict or subexpression sequencing problems should be further studied for more efficient utilization of super computers like CDC Star 100 or Illiac IV.
REFERENCES


C. V. Ramamoorthy (M’57) received the undergraduate degrees in physics and technology from the University of Madras, India, the M.S. degree and the professional degree of Mechanical Engineer, both from the University of California, Berkeley, and the M.A. and Ph.D. degrees in applied mathematics and computer theory from Harvard University, Cambridge, Mass. He was associated with Honeywell’s Electronic Data Processing Division, Waltham, Mass. He was a Professor in the Departments of Computer Science and Electrical Engineering at the University of Texas, Austin. He is currently a Professor in the Department of Electrical Engineering and Computer Science at the University of California, Berkeley.

Juong H. Park (S’71) was born in Choong-Buk, Korea, on May 28, 1940. He received the LL.B. degree from the School of Law, Seoul National University, Seoul, Korea, in 1964, and the M.A. degree in computer sciences from the University of Texas, Austin, in 1971. From 1970 to 1972 he participated in the development of the automated software evaluation system (ASES) at the Electronic Research Center, University of Texas, Austin. In 1972 he joined the Information Research Associates, Inc., Berkeley, Calif., and Austin, Tex. His present work involves the development of a computer system validation complex.

Hon F. Li (S’73) was born in Hong Kong, China, on August 22, 1950. He received the B.S. and M.S. degrees from the University of California, Berkeley, in 1972 and 1973, respectively, and since that time he has been doing graduate work at the same university under a Chancellor’s Science Fellowship. He is currently a Research Assistant in the Electronic Research Laboratory, University of California, Berkeley. Mr. Li is a member of Tau Beta Pi and Eta Kappa Nu.