https://ntrs.nasa.gov/search.jsp?R=19720023577 2020-03-23T08:37:51+00:00Z

Automatic Recognition of Vector and Parallel Operations in a Higher Level Language

Paul B. Schneck

Institute for Space Studies
Goddard Space Flight Center
New York, N.Y. 10025

(NASA-TM-X-68608) AUTOMATIC RECOGNITION OF VECTOR AND PARALLEL OPERATIONS IN A HIGHER LEVEL LANGUAGE P.B. Schneck (NASA) [1971] 8 p

N72-31227

CSCL 09B

Unclas

G3/08 16202

Reproduced by
NATIONAL TECHNICAL
INFORMATION SERVICE
U S Department of Commerce
Springfield VA 22151

N O T I C E

THIS DOCUMENT HAS BEEN REPRODUCED FROM THE BEST COPY FURNISHED US BY THE SPONSORING AGENCY. ALTHOUGH IT IS RECOGNIZED THAT CERTAIN PORTIONS ARE ILLEGIBLE, IT IS BEING RELEASED IN THE INTEREST OF MAKING AVAILABLE AS MUCH INFORMATION AS POSSIBLE.

A compiler for recognizing statements of a FORTRAN program which are suited for fast execution on a parallel or pipeline machine such as ILLIAC-IV, STAR or ASC is described. The technique employs "interval analysis" to provide flow information to the vector/parallel recognizer. Where profitable the compiler changes scalar variables to subscripted variables. The output of the compiler is an extension to FORTRAN which shows parallel and vector operations explicitly.

KEY WORDS AND PHRASES: parallel computations, vector processing, pipeline, interval, flow analysis, compiler.

CR CATEGORIES: 4.12, 4.22

INTRODUCTION

The very high performance computers being built today for delivery in the next several years (e.g. Texas Instrument's ASC, Control Data's STAR and Burrough's ILLIAC-IV) rely on radically new machine organizations to attain their speed. They are based on pipeline (vector) or parallel processing concepts (1) which, however different they may be, appear quite similar to the user. Each of these computers requires that the same sequence of operations be applied to a large set of data items in a regular fashion. Each operation is specified in turn and it is applied to the entire set of data. Thus, these machines may be thought of as performing an operation simultaneously on all data items. The ILLIAC-IV operates on 64 data items in parallel. The STAR and ASC perform operations on data items sequentially, but with a very high degree of overlap.

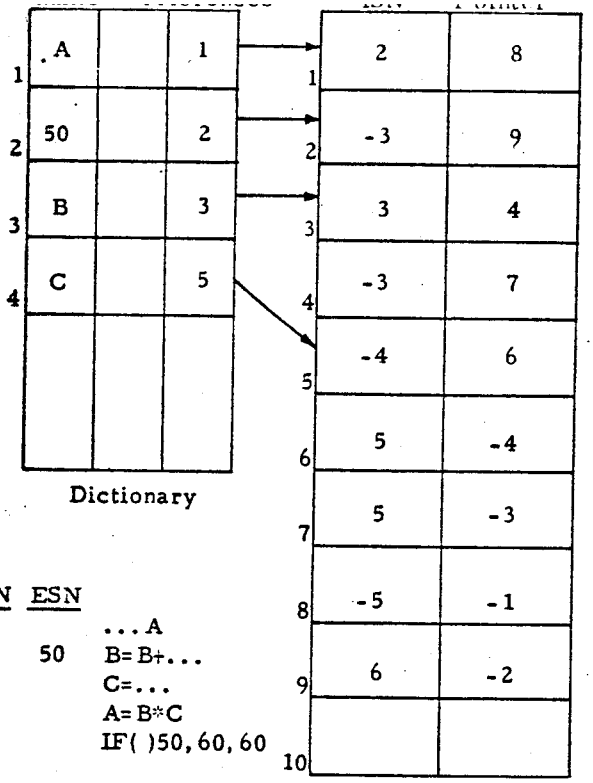
However, programmers are not accustomed to stating problems in the form required by these machines. An effective means of problem statement is transformation of a standard high level language program into one which details parallel operations. This paper describes the techniques used in a compiler to perform this transformation. The compiler accepts a standard FORTRAN program (2), identifies implicit parallel or vector operations and produces a program which performs these operations explicitly. The program output by the compiler is 'almost' in the Burrough's ILLIAC-IV FORTRAN language (3). In this extension to FORTRAN an asterisk appearing in a subscript position indicates that the entire column takes part in an operation. The operation is performed in a manner analagous to the functioning of an I/O statement's "implied DO".

The next three sections of this paper describe three of the four compiler segments:

1. Statement Classification
2. Flow Analysis
3. Recognition of Parallel or Vector Operations
4. Optimization (not included, will be discussed in a later paper).

STATEMENT CLASSIFICATION

As each source statement is read it is classified as one of the approximately forty FORTRAN statement types (4). The appropriate routine is then called to process the statement. Two functions are performed at this time. The source statements are transformed to an intermediate text representation which is convenient for further analysis, and information is gathered which will later be used to determine program flow and data flow. Each appearance of a variable, constant, or label is entered in a "reference table" and dictionary. The format of this key pair of tables is given in Figure 1. When the entire FORTRAN program has been read, control is transferred to the flow analysis routines.



ISN	ESN	
2		...A
3	50	B=B+...
4		C=...
5		A=B*C
6		IF( )50,60,60

*ISN > 0 is a use,
ISN < 0 is a definition.

†a positive number indicates the next reference table entry, a negative number indicates end of reference chain, and points back to the variable in the dictionary.

Figure 1. Reference Table and Dictionary

FLOW ANALYSIS

The first task of the flow analysis routines is to divide the program into a set of basic blocks. A basic block is defined as section of code with only one point of entry, one point of exit, and no internal flow, as shown in Figure 2. A referenced label is a point of entry and causes a basic block to begin, a branch is a point of exit and causes a basic block to end. With an adaptation of an algorithm by Kleir and Ramamoorthy (5) basic blocks are identified by chaining through the reference table to locate entry and exit points.

```
      Q=360./FLOAT(N)
      DO 1 I=1,N

      A(I)=SIN(FLOAT(I)*Q)
1     B(I)=COS(FLOAT(I)*Q)

      Z=A(N)+B(N)
7     Y=A(N)*A(N)-B(N)*B(N)
      DO 2 I=1,N

2     A(I)=A(I)*(Z+B(I)/Y)

      WRITE(6,101)A
      STOP

      END
```

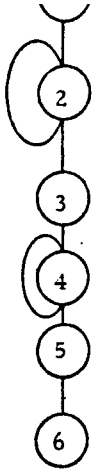


Figure 2. Basic Blocks

It is now possible to characterize the program flow in terms of basic blocks. The elementary flow relationship that we employ is called a predecessor. Block "i" is said to be a predecessor of block "j" if block "j" can be reached from block "i"; it is an immediate predecessor if block "j" can be reached in one step. A set of predecessor lists is used to represent the program flow. Each predecessor list gives all the immediate predecessor blocks, as in Figure 3.

Block	Predecessors
1	None
2	1,2
3	2
4	3,4
5	4
6	5

Figure 3. Predecessor list of sample program in Figure 2

We are ready to begin processing at this time. All that remains is to choose an ordering of the basic blocks. Clearly we wish an ordering that will identify loops, as that is a necessary condition for the existence of parallel or vector operations. The notion of a "Strongly Connected Region" (SCR) which is roughly equivalent to the extended range of a "DO" loop is convenient at this time. An SCR is a set of basic blocks with the property that any pair of basic blocks of the set are predecessors of one another. Nested loops produce a nested set of SCRs when all blocks of inner loops are treated as single blocks. A construct which readily finds SCRs and has other useful ordering relations is called the "Cocke-Allen interval decomposition" (6, 7). By using this technique we will identify SCRs and determine a processing order.

The "interval" construction arises naturally by extension of the concept of basic block. Because a basic block has only sequential flow it is not useful where consideration of flow is necessary. We extend the definition of basic block to permit divergent flow paths. This is still not adequate for handling SCRs so we extend the definition again to permit convergent flow paths from a common predecessor. This development is traced below.

We now define an "Extended Basic Block" as a section of code with one entry and any number of exits. An extended basic block is a set of basic blocks which can be amalgamated to a unit, as shown in Figure 4. The extended basic block permits us to work with larger units of code. For example, Figure 4 illustrates an extended basic block which is a set of basic blocks on three divergent paths. We may consider blocks 1-2-3, blocks 1-2-4 and blocks 1-5 as units, for the order of code is sequential along each path.

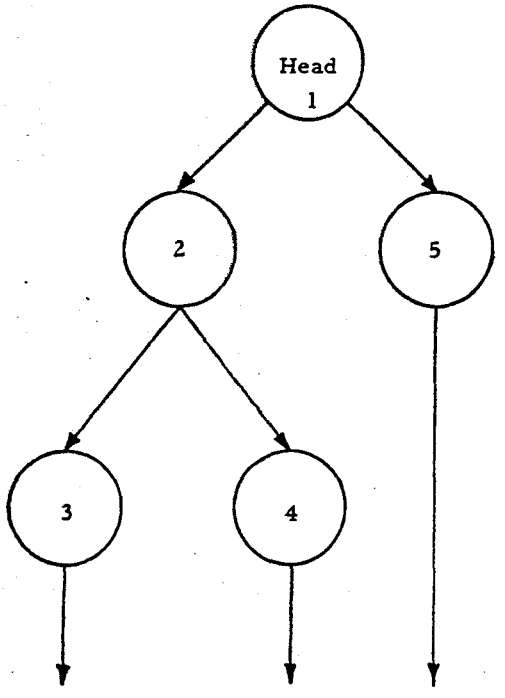


Figure 4. Extended Basic Block

To construct an extended basic block we begin by initializing it to a given basic block called the "head". A basic block may be added to the extended basic block if and only if it has a single immediate predecessor which is already a member of the extended basic block. This definition immediately rules out many common cases of program flow, as shown in Figure 5.

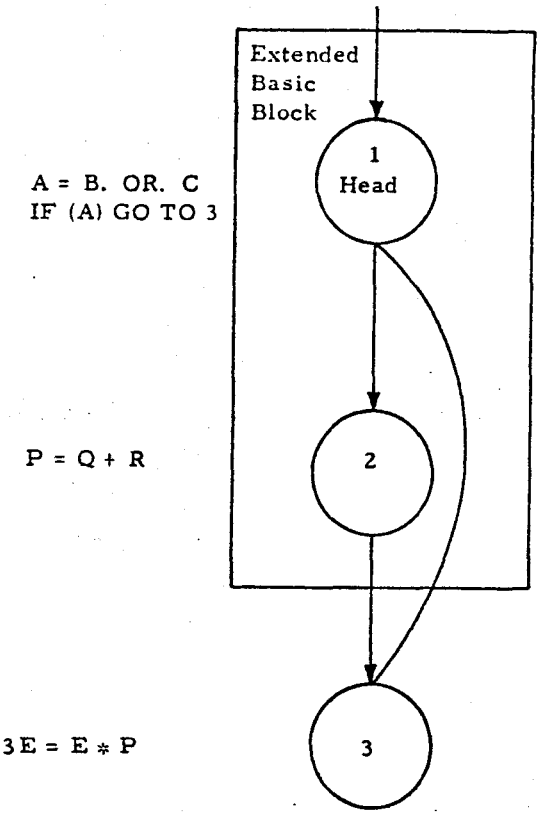


Figure 5. Flow which is not an Extended Basic Block

The definition is now extended to allow convergent paths, and so yields an "interval". An interval is the maximal set of basic blocks containing a distinguished basic block, called the "interval head", with the properties that:

a) all predecessors of blocks in the interval, except the interval head, must belong to the interval.
b) any SCR in the interval must include the head.

In constructing an interval we again begin by setting the interval equal to a given basic block, the interval head. A basic block may be addded to the inteval if and only if all of its immediate predecessors are already in the interval. This definition yields a partial order among the blocks of the interval, shown in Figure 6.

Since a basic block can be made part of an interval only if all its predecessors are already in the interval, it is clear that an SCR cannot be added to an interval. However, an interval will contain an SCR when a block of the SCR which could not be added to a prior interval becomes the head of the interval. Thus, intervals can be used to identify SCRs.

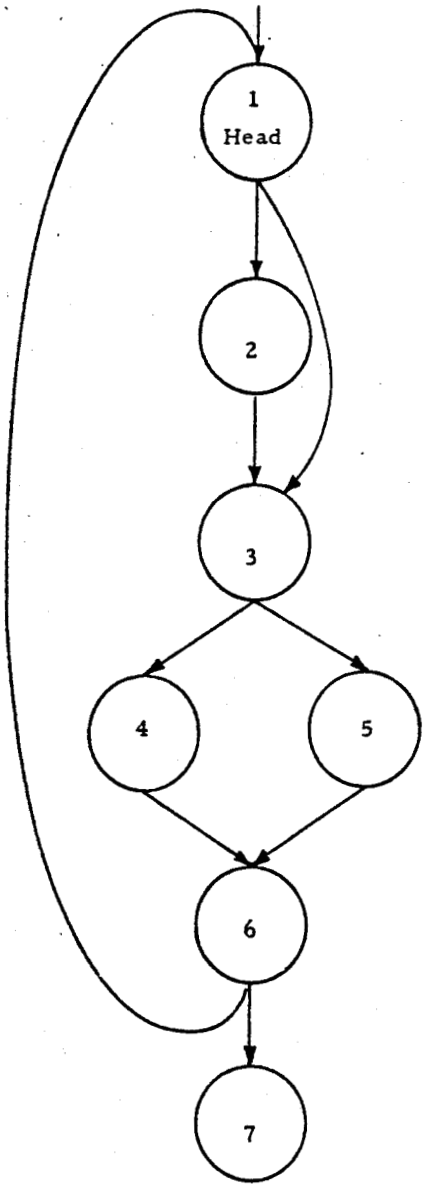


Figure 6. Interval

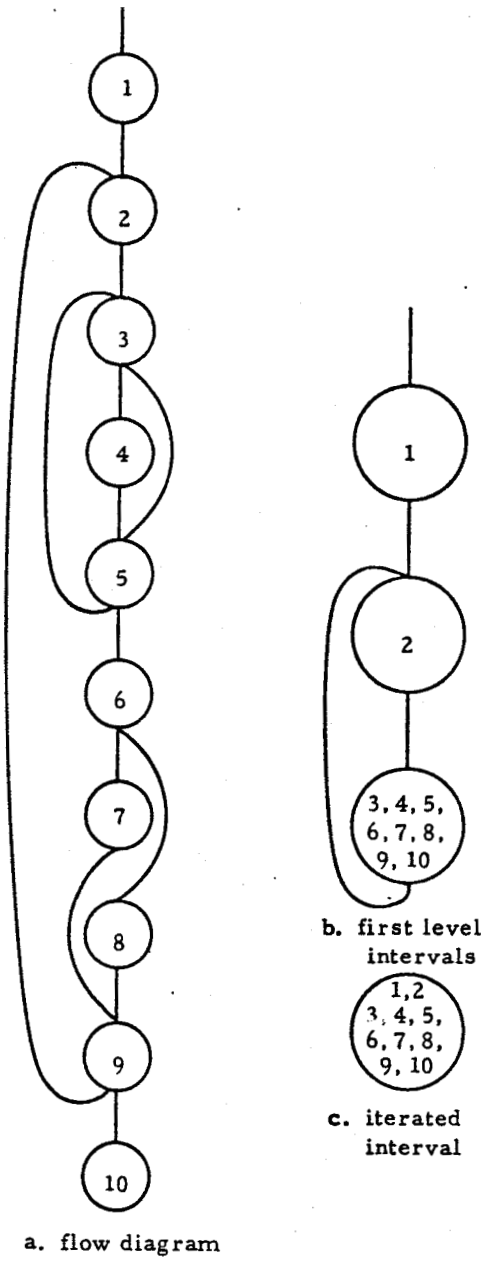


Figure 7. Iterated Intervals

The interval construction uniquely partitions a program flow-graph (6). The resulting intervals are then treated as basic blocks and the process repeated as in Figure 7. The set of iterated intervals defines a processing order (with some blocks processed several times) for analysis of the source program. The iterated interval sequence either converges to a single block or takes on an irreducible form. By transformation of the source program the irreducible form could be eliminated (8). Currently either condition signals the completion of processing.

RECOGNITION OF PARALLEL OR VECTOR OPERATIONS

The input to this segment of the compiler is the ordered set of blocks of an interval. The interval may contain the entire program or an SCR. The SCR is the item of interest and must be located within the interval. As observed earlier the target of the backward branch creating the SCR must be the interval head. The SCR consists of all interval predecessors of the interval head. It is in the SCR that parallel or vector operations can be performed when sucessive passages through the SCR are independent of one another.

In order to execute computations within the SCR in a parallel or vector mode of operation, the number of iterations must not be dependent on computations within the SCR. Equivalently, the SCR must have only one exit, controlled by the variable used to count the number of iterations called the induction variable.

The induction variable must now be identified. To accomplish this it is necessary to distinguish "relative constants", those variables (all constants are relative constants) whose values do not change within the SCR. This is done iteratively, by first marking all variables which are not defined within the SCR, as they are clearly relative constants. Next, all variables which have only a single unconditional definition that precedes all uses and is defined in terms of relative constants are marked relative constants. This process is repeated until no new relative constants are found.

If the exit branch of the SCR depends upon precisely one variable that is not a relative constant then that variable is the induction variable candidate. In order for the induction variable candidate to be the induction variable it may only be defined by addition or subtraction of a relative constant expression to itself within unconditional arithmetic statements. In an SCR corresponding to a "DO" loop the induction variable is the do-variable. Identification of the induction variable means it is possible to determine the number of iterations of the SCR before entry. If no induction variable can be identified then parallel or vector calculations cannot be performed and the next interval is processed.

When the induction variable has been identified it is used to determine which subscripted variables may participate in vector and parallel operations. Currently the compiler considers only subscripted variables in which the induction variable makes precisely one appearance in the same position in every subscript. Any other variable which appears in the subscript must be a relative constant, so that its value is available for the entire parallel or vector operation (corresponding to each iteration through the SCR). A subscripted variable may participate in vector and parallel operations only if there is no "feedback" between different iterations of the SCR, as shown in Figure 8.

In order to determine whether feedback can occur we must know in which direction the induction variable is changing. If the induction variable is not incremented by a constant, we must examine the exit branch of the SCR. If the loop exit occurs on "greater" (>,≥) condition the induction variable is increasing, if the loop exit occurs on "less" (<,≤) condition, the induction variable is decreasing. Other conditions make it impossible to determine the direction of change of the induction variable, therefore impossible to check for feedback, and so parallel and vector operations cannot be performed.

The condition for feedback is now described for an increasing induction variable. Let I(S) be the multiplier of the induction variable in subscript S, and let J(S) be the constant term of the induction variable in subscript S. For example, if "L" is the induction variable, then

$$I(3 * L + 2, M - 7, 2 * N + 4) = 3$$

and $J(3 * L + 2, M - 7, 2 * N + 4) = 2.$

Let I_L be the maximum of I(S) over all subscripts appearing in definitions of the variable and let I_R be the minimum of I(S) over all subscripts appearing in uses of the variable. Define J_L and J_R in a similar fashion. Feedback potentially exists if

$$I_L > I_R \quad \text{or} \quad J_L > J_R$$

As illustrated in Figure 8, potential feedback does not always result in feedback. Potential feedback implies the existence of subscripts "s" in a definition and "t" in a use of the variable with I(s) > I(t) or J(s) > J(t). In the case of potential feedback all definition/use pairs of subscripts must be checked to determine whether feedback actually occurs. The necessary information is contained in the reference table. For each definition/use pair of subscripts causing potential feedback, if the definition occurs before the use along all possible paths (i.e. the definition back dominates the use) the pair does not cause feedback. If all such pairs of subscripts do not cause feedback then the variable is a candidate for parallel or vector execution.

```
        = ...A(I)
A(I+1) = ...
```

a. Feedback

```
A(I+1) = ...
       = ... A(I)
```

b. potential feedback, no feedback

```
       =...A(I+1)...
       =...A(I) ...
A(I-1) = ...
 A(I)  = ...
```

c. no feedback, no potential feedback

Figure 8. Testing for Feedback

It is not sufficient merely to recognize vector and parallel operations among subscripted variables. It is desirable to replace sequential processes with vector and parallel processes whenever possible. Figure 9a illustrates a loop where temporary variables are used to hold common subexpressions; Figure 9b illustrates the same loop as transformed by the compiler. A scalar in an SCR is transformed into a vector when at least one of its unconditional definitions is in terms of a vector quantity.

```
      DO 3 I=1,IM
      IM1=MOD(I+IMM2,IM)+1
      DO 3 J=2,JM
      ALPH=FXCO*(P(J,I)+P(J-1,I))
     1 *(FD(J,I)+FD(J-1,I))
      UT(J,I,L)=UT(J,I,L)+ALPH*V(J,I,L)
      UT(J,IM1,L)=UT(J,IM1,L)+ALPH*V(J,IM1,L)
      VT(J,I,L)=VT(J,I,L)-ALPH*U(J,I,L)
    3 VT(J,IM1,L)=VT(J,IM1,L)-ALPH*U(J,IM1,L)
```

a. Source Program with Scalar Variables.

```
      DO 3 I=1,IM
      IM1=(MOD((I+IMM2),IM)+1)
      ALPHV(*)=((FXCO*(P(*,I)+P(*-1,I)))
     1 *(FD(*,I)+FD(*-1,I)))
      UT(*,I,L)=(UT(*,I,L)+(ALPHV(*)
     1 *V(*,I,L)))
      UT(*,IM1,L)=(UT(*,IM1,L)+(ALPHV(*)
     1 *V(*,IM1,L)))
      VT(*,I,L)=(VT(*,I,L)-(ALPHV(*)
     1 *U(*,I,L)))
      VT(*,IM1,L)=(VT(*,IM1,L)-(ALPHV(*)
     1 *U(*,IM1,L)))
    3 CONTINUE
```

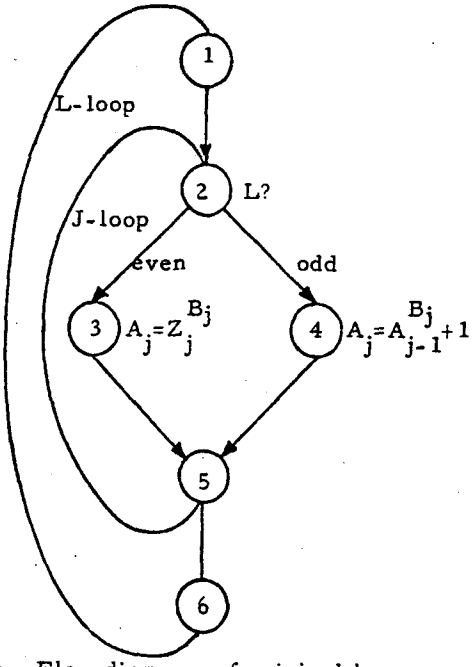
b. Compiler Output with Subscripted Variables.

Figure 9. Change Scalar to Vector.

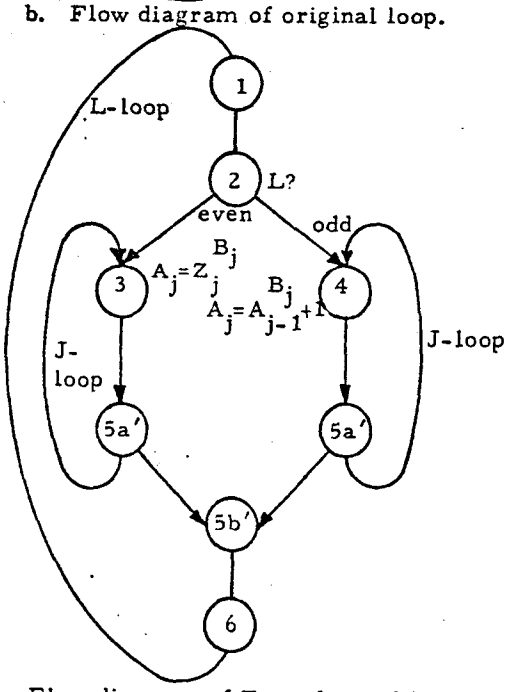
Any calculation which is conditional, and thus loop dependent, cannot be performed in parallel or as a vector. This would at first seem to eliminate a great many operations but in fact does not. Even though flow information indicates that a calculation is conditional it may not be so in the context of an SCR. As illustrated in Figure 10, if a conditional calculation depends on a loop independent test then either the calculation will always or never be executed during the loop, as the test result does not vary within the loop. In this case both the loop independent test as well as the parallel and vector operations may be moved out of the loop as described below.

```
      DO 1 L=1, N
      DO 2 J=2, 100

      IF (MOD(L, 2).EQ. 0) A(J)=Z(J)**B(J)
      IF (MOD(L, 2).EQ. 1) A(J)=A(J-1)**B(J)+1

    2 CONTINUE

    1 CONTINUE
```

a. Loop Independent Tests.



b. Flow diagram of original loop.



c. Flow diagram of Transformed loop.

Figure 10. Loop Independent Tests

Now the individual statements are processed in interval order. If a statement is in a block which is executed conditionally on a loop dependent test it cannot be executed as a parallel or vector operation. Other statements are processed by the arithmetic analyzer to find and delineate parallel and vector operations. Operands are categorized as one of three types:

- V - vector quantities which may appear in parallel and vector operations.
- R - relative constants; scalars or single items of a vector.
- M - mongrel quantities; neither of the above.

Pairs of operands combine in the following manner:

•	M	V	R
M	M	M	M
V	M	V	V
R	M	V	R

The arithmetic analyzer moves 'V' subexpressions out of the loop to enable them to be executed in parallel or vector mode. It moves 'R' subexpressions to the next outer loop where they are again processed. On this second processing a former 'R' subexpression can assume any of the three possible types. 'M' subexpressions are left unchanged and are flagged so that they need not be re-examined since an 'M' subexpression cannot take on a different type in an outer loop. Figure 11 illustrates the action of the compiler on a sample program.

CONCLUSION

This paper has described the internal organization of a compiler used to recognize, create and translate vector or parallel executable operations into a form useful by the appropriate hardware.

ACKNOWLEDGEMENT

The author wishes to thank Ellinor Angel who designed and programmed the arithmetic analyzer.


```
      DO 135 J=1,JM
      ZM(J)=0.0
      DO 136 I=1,IM
  136 ZM(J)=ZM(J) + P(J,I)
  135 ZM(J)=ZM(J)*FIM
      WTM=0.
      ZMM=0.
      DO 137 J=1,JM
      WTM=WTM + ABS(DXYP(J))
  137 ZMM=ZMM + ZM(J)*ABS(DXYP(J))
      ZMM=ZMM/WTM +PTROP
      DELTAP = PSF - AMM
      DO 301 I=1,IM
      DO 301 J=1,JM
  301 P(J,I)=P(J,I)+DELTAP
```

a. Source Program.

```
      ZM(*)=0.
      ZM(*)=ZM(*)*FIM
      DO 135 J=1,JM
      DO 136 I=1,IM
      ZM(J)=(ZM(J)+P(J,I))
  136 CONTINUE
  135 CONTINUE
      WTM=0.
      ZMM=0.
      T00002(*)=ABS(DXYP(*))
      T00004(*)=ZM(*)*T00002(*)
      DO 137 J=1,JM
      WTM=(WTM+T00002(J))
  137 ZMM=(ZMM+T00004(J))
      ZMM=((ZMM/WTM)+PTROP)
      DELTAP=(PSF-ZMM)
      DO 301 I=1,IM
      P(*,I)=(P(*,I)+DELTAP)
  301 CONTINUE
```

b. Compiler Output.

Figure 11. Sample Program

1. Graham, W. R., "The Parallel and the Pipeline Computers," Datamation, 16, No. 4, 60, 1970.

2. American National Standards Institute, USA Standard FORTRAN, USAS X3.9-1966.

3. Burroughs Corp., Array Processing System FORTRAN IV Reference Manual, #66106C, 1971.

4. Sale, A. H. J., "The Classification of FORTRAN Statements," The Computer Journal, 14, 10, 1971.

5. Kleir, R. L. and Ramamoorthy, C. V., "Optimization Strategies for Microprograms," IEEE Trans. Computers, C-20, 783, 1971.

6. Cocke, J. and Schwartz, J. T., Programming Languages and Their Compilers, New York University, Courant Institute of Mathematical Sciences, 1969.

7. Allen, F. E. and Cocke, J., "Graph-Theoretic Constructs for Program Control Flow Analysis," to be published.

8. Cocke, J. and Miller, R. E., "Some Analysis Techniques for Optimizing Computer Programs," Proc., Second Hawaii International Conference of System Sciences, 1969.