MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Project MAC

176

[redacted] Structures Group — Memo [redacted]

[redacted] — July [redacted]

A Note on Asynchronous Parallel Processing*

H. Witsenhausen

[redacted]

[redacted]ronous parrallel [redacted]ing within a given job in an [redacted] multi-processor computer is considered in a limited [redacted] items discussed are:

[redacted] The scope of an element of data must be defined in two [redacted] - time and ownership.

[redacted] The operations performed at a fork.

[redacted] The operations that replace the notion of join.

[redacted] The impact of look-ahead.

[redacted] That loss of time due to lock-out is not significant.

[redacted] What a programmers specification of parallelism could be like.

7. Some hardware features that may be helpful.

* [redacted] is a slightly edited version of a term paper submitted for subject [redacted] in January 1964. - J.B.D.

CONTENTS

Types of Parallel Processing

The distinction between sequential, single processor and parallel, multiprocessor computers is in itself not clear.

a) Parallel treatment of several bits and registers within a single processor is not considered parallel processing.

b) Local concurrency in the sense of Codd, which is found in the CDC-6600 and in the Stretch look-ahead scheme does not raise the same type of problems as the operatinn we consider below. (No programmers specification)

c) Synchronous parallel execution, as found in the Solomon computer, is similar to conventional operation in that processors requested are assumed always available and the detailed relative timing of all operations is completely pre-planned. Logically, these are single processor computers with a distributed processor.

d) We shy away from the difficult and very important case where some relations (inequalities) between execution times of some program sequences by two processors are known in advance. In such a case a "fork" does not necessarily require a "join". The processor executing the shorter prong returns unconditionally to the idle processor list while the other processor unconditionally continues the program, secure in the knowledge that the parallel operations have been completed in due time. As opposed to this we assume that no upper limit can be predicted for the time of execution of any program sequence, for instance becuase any processor may be pulled out from under us by a higher priority interrupt from some other program.

e) Parallel operation of processors assigned to different jobs, on disjoint data, does not lead to the same problems (some analogies occur in case of call upon common subroutines). It is essentially a scheduling problem which attempts to satisfy optimization criteria instead of being obligated to satisfy constraints.

Summarizing, we distinguish

1. conventional single processor computers
2. ditto with local cnncurrency
3. distributed processor computers (synchronous multiprocessors)
4. asynchronous multiprocessor computers with interjob concurrenty only.

5. ditto with intrajob concurrency
6. ditto with consideration of known execution time relationships
7. ditto with real time constraints for operation with an environment (strict inequality constraints).

We address ourselves to case 5, and this is a restricted framework.

II. Framework

In order to concentrate on the questions of most interest to the writer, a number of simplifying assumptions will be made, unless otherwise stated, in this note.

1. We seek for a minimum of changes in conventional memories and processors, such that parallel operation becomes easy to implement, rather than try a radically new structure (such as ALPS). In particular one should be able to run any problem sequentially with a single processor in the conventional manner, when desired.

2. We consider a single job. In fact, the multiprogramming concurrency of several jobs is essential to make parallel processing efficient. Idle processors must be able to find employment. This scheduling problem is similiar to that of sequential computers with I/O concurrency and is not our concern.

3. While the main reason for parallel operation within a job is to expedite its completion as required by the final user, we will not consider the problems raised by rigid real time constraints imposed on the production of certain results.

4. I/O operations are not considered; in the parallel processing framework they differ from other program segments solely by the necessity to obtain access to one or more of a special kind of processing units. In all other respects they are implicitly included (cf. Conway).

5. No interrupts or traps are considered explicitly though the possibility of interrupts is one of the factors that make execution time of a program segment a random variable without upper bound and uncorrelated with that of other executions.

6. Allocation problems are xosidered only from the point of view of what should be saved and to <u>whom</u> it is accessible (the 2-dimensional scope problem)

is where it is stored. The latter would be handled by an extension of the dynamic storage allocation techniques for the single processor case (cf. Van Horn).

III. Remarks on Monosequence Execution

1. General

Monosequence execution of a job means its accomplishment by a string of consecutive steps, which include conditional branching. It has been shown by van der Poel that a single instruction is logically sufficient:

$$\left.\begin{array}{l}(A)-(Y) \rightarrow (A);\quad (A)-(Y() \rightarrow (Y)\\ (PC)+1 \rightarrow (PC)\end{array}\right\}\quad \begin{array}{l}\text{1's complement}\\ \text{arithmetic}\end{array}$$

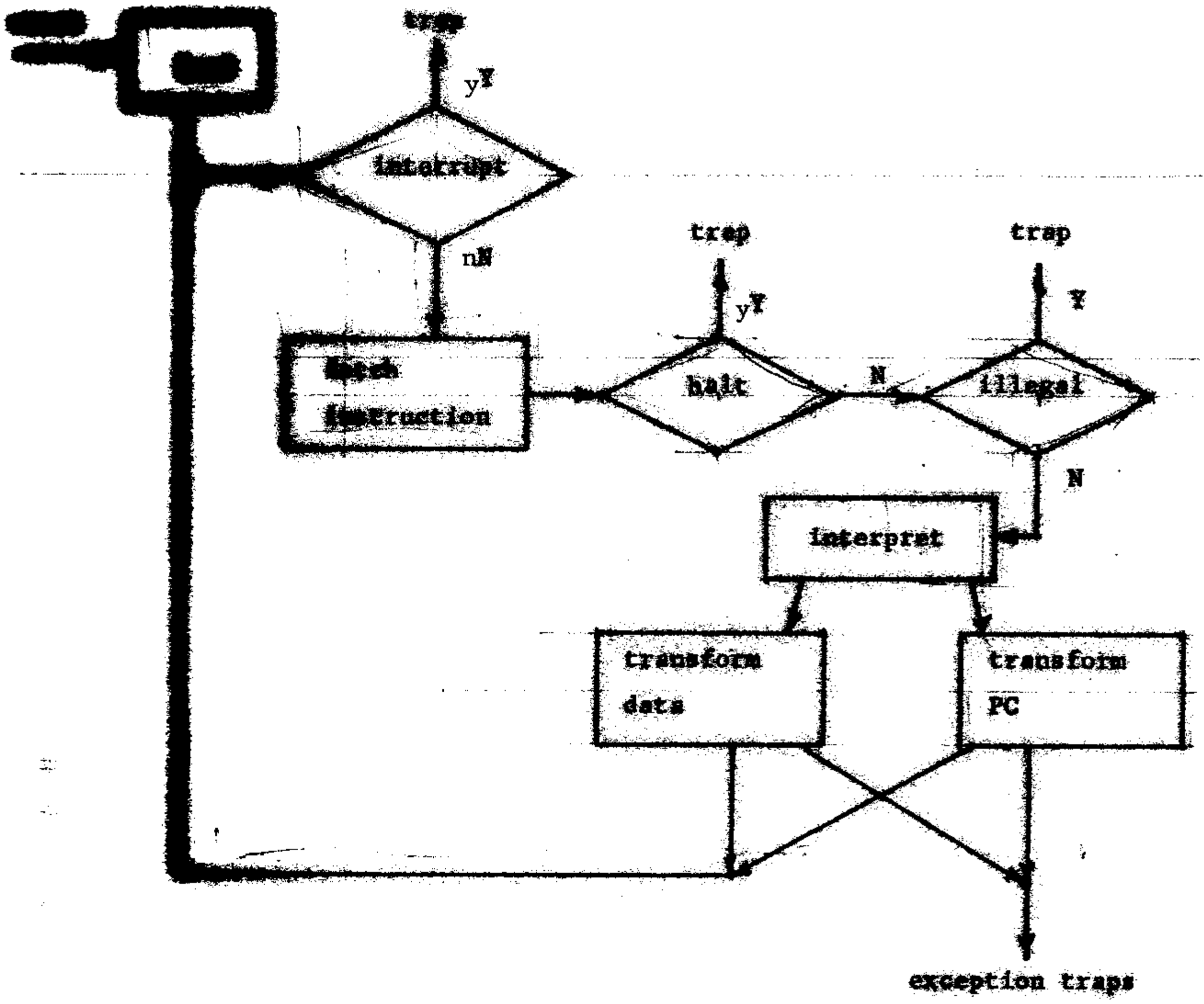
A distinction between processor state and other data is not essential. With multiple address instructions the state of the processor between instructions is characterized solely by the program counter. The latter could be a specific storage location. The notion of state-word is unnecessary as long as there are no interrupts. Even with interrupts, provision for multiple program counters (TX2, Honeywell 800) does away with the state-word concept. In other machines this concept occurs only because of technological structure: the special location of AC, MQ, XR's. . . makes a difference as to <u>how</u> they are specified (implicitly, tags, . . .) and how fast they are accessible, and nothing more.

The specification of the algorithm carried out by the machine can be made at various levels.

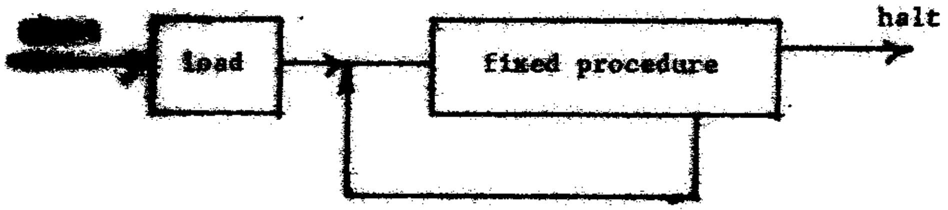
Representation instruction by instruction is shown in Fig. 1. Without interrupts and exception traps,and taking all illegal instructions as halts: we obtain Fig. 2. This is the representation considered by Riguet.

Representation at the program level is provided by

- flow diagrams (Rutishauser's, Plan-Kalkiil)
- Algorithm schemes (A. Markov, A.A. Lyapounovland, V.I. Yanov)
- Incidence matrices (Karp)



[illegible] Instruction-by-instruction representation.



[illegible] Simplified instruction-by-instruction representation.

We use the notion of operator which can be a single instruction or a sequence without conditional branching out of the operator; the terminal control point of an operator is unique.

These schemes do not explicitly account for subroutine linkage because the terminal control point of the last operator of the routine is not unique. To make it unique one must follow it by a series of tests to identify the call point, an unrealistic representation. This can be avoided by returning to the instruction level.

To prepare for discussion of parallel processing we now introduce a specific representation.

2. <u>Program Structure</u>

As in the flow diagram approach we consider data as distinct from program. We consider all programs to be pure procedure (by use of indirect addressing, index registers, etc.). Our reason for doing so is not the relocation problem (that we do not consider) but the problems of parallel processing.

Programs are considered as made up of a finite and fixed number N of segments with a unique entry point to each. They are labeled by the integers 1 to N, label 0 being reserved for halt (or trap to supervisor). When a subroutine call is to be explicitly shown, the calling segment is split, creating a label for the return point. The subroutine entry point has of course its own label.

3. <u>Data Equivalence</u>

The program, being rigidly fixed, can be thought of as in a protected or read-only store. All other information, including the contents of processor registers, is called data. In a finite machine the data can have a finite number of states only. At any time there are likely to be large numbers of states which are equivalent as far as the execution of the job, or of the current segment, is concerned.

At halt or at end-of-segment time this equivalence has two causes: 1) only a fraction of the words in store are of interest; the others contain immaterial information. 2) when the original data is inappropriate, exception conditions may occur leading to the recording of error flags and possibly a premature halt.

In a production situation, as distinct from debugging, all these final states belong to the single garbage class.

In effect then, if the job terminates, we are only interested in the final state within an equivalence. At all previous stages we are only concerned with distinguishing those states that will lead to non-equivalent final states, while the states that will lead to endless operation are all members of the garbage class.

In some way, an equivalence can be defined for any program segment by considering it as a program in its own right.

4. Program Description

Each program segment, whenever it is called into operation has access to the current state of the data. Its effect is to map this initial state into a final state and into one of the $N > 1$ labels 0 to N. If the label producing algorithm was to lead to anything but one of these, this would be considered as label 0 with error flag. We thus consider the program perfectly debugged for the initial data not necessarily meaningful.

Each segment is defined by two single-valued functions.

$$\text{segment } i \begin{cases} D_1 = f_i(D_o) \\ L = g_i(D_o) \end{cases} \qquad i = 1, \ldots, N$$

where D_o and D_1 range over the possible data states, while L is in the set of integers 0 to N.

The description of the program by these 2N functions is felt to be more general than the description of Yanov. The function $g_i(D_o)$ partitions the states D_o into M + 1 classes. The tests of boolean variables constrained by change lists are much more restrictive, even when the maximum of 9 possibilities to which Yanov's scheme can be extended is taken into account. These are obtained, for a given operator, by combining the 3 possible change conditions of the initial value zero with the 3 similar conditions for the initial value one. The 3 conditions are no change, complementation, either. This still does not account for the fact that the effect of an operator depends on the previous operator string and the initial data.

3. Special Cases

1. $g_i(D_o)$ = constant; such a segment is called a "box" (operator) and a string of boxes is a box.

2. $g_i(D_o) = n_1$ or n_2; such a segment is a combination of a box and a two-way branch. asAshebowniniFig.g33.

3. $g_i(D_o)$ is a function only of the label of the previous segment (supplied in D_o). This is the case of the subroutine with unconditional return.

4. $f_i(D_o) = D_o$ the case of a pure branch, which can always be viewed as successive two-way branches.

Remarks: Internally generated traps (overflow, etc.) do not transcend this representation since they amount to replacing by permanent hardware the coding and execution of conditional branches.

The assumption of pure procedure programs means that the functions f_i and g_i are fixed for successive passages through the same segment and that no new segments can be created.

It is impossible to know the exact equivalence classes at any time without running the problem for various initial states but it is possible to define equivalence classes which are surely fine enough. Assume the error flag to be non program resettable, then all states leading to its setting in a segment are equivalent. Also states are equivalent if they differ only in locations known to be currently meaningless.

4. Allocation

To allow dynamic storage allocation the programmer must specify the scope of every piece of data. This amounts to specifying its release (erasure). The scopes are nested in Algol for push-down allocation but this is not a necessity.

C. Parallel Processing

1. Conditions for Parallel Operation

Given a sequential formulation of an algorithm, what is required to perform two successive segments in parallel?

Let i and i + 1 be the segments in question, with

$$(1) \qquad (1) \begin{cases} f_i(D_o) = D_1 \\ g_i(D_o) = i + 1 \quad \text{for all } D_o \text{ of interest} \\ f_{i+1}(D_1) = D_2 = f_{i+1}(f_i(D_o)) = q(D_o) \\ g_{i+1}(D_1) = g_{i+1}(f_i(D_o)) = p(D_o) = L \end{cases}$$

they amount to the single segment

$$(2) \quad D_2 = q(D_o) \qquad L = p(D_o)$$

and they are equivalent to any segment which results in

$$(3) \begin{cases} D_2 = q(D_o) \quad \text{modulo equivalence} \\ L = p(D_o) \end{cases}$$

For all D_o of interest, segments i and i+1 are said to be compatible if (3) is satisfied for any time relation of the individual instructions of the two sequences. This is the necessary and sufficient condition for their parallel implementation in our framework.

Two other concepts occur in this connection. The segments are said to commute if

$$(4) \quad f_i(f_{i+1}(D_o)) = f_{i+1}(f_i(D_o)) \text{ modulo equivalence, for all } D_o \text{ of interest.}$$

That this is necessary but not sufficient results from very simple examples such as (x)+3→(x) and (x)+5→(x) which commute but are not compatible if the additions take place in the accumulators of two different processors which fetch and store location x in two operations which are asynchronous.

Commuting segments can be made compatible by lock-out of data but this destroys the gain in time.

The segments are said to be data-disjoint if the set of locations into which segment i stores is disjoint from the set of locations accessed (for either reading or writing) by segment i+1 and vice versa, and this for all cesses of interest.

This is a sufficient but not necessaryccondition, for instance both segments could use the same location for temporary storage of a single intermediate result which is necessarily the same in both, or they could execute [illegible]→x and x+5→x by add-to-memory instructions for which lock-out is automatic.

Thus we have

$$\text{data-disjoint} \supset \text{compatible} \supset \text{commutation.}$$

At execution time, a check for compatibility could only be made by simulation-before-execution (to check effective addresses for disjunction or to check final results) taking more time than sequential execution. The responsibility thus rests squarely on the shoulders of the programmer-compiler team.

2. Public and Private Data

In a monosequence execution, the same segment can be executed numerous times (loops, subroutines) usually with different initial data (different equivalence class) at each passage. In multiprocessing, similarly, the same segment will have to be executed several times and this by different processors. It will therefore occur that the same segment is being used simultaneously by two or more processors. These executions must be compabible, - in practice, data-disjoint.

This requires:

a) pure procedure segments referencing data by indirect means.

b) the indirect means must include at least one element which is distinct for each processor, for instance an index register R associated with the physical processor, P.

It follows the intermediate results generated by two processors executing the same segment go into disjoint locations and that cross references from one processor P1 to the data generated by P2 are in general undesirable and may be impossible. To find the location of a result generated by P2 we need to know the state of R in P2 as of the time the result was stored. Even if R has not been changed by P2 it is impossible for P1 to discover whether it is physical processor P2 or P15 that is involved. Looking at all processor program counters is no help, since P2 may already have finished the segment (the assumption of completely asyhchronous time relationships))

In conclusion we must distinguish two types of data: public and private. Public data is accessible to all processors unless a temporary lock-out condition prevails. Private data is associated with specific processors and accessible only to them. This is not a "look-out" but a long-term memory protection. It can be accomplished by pointers in the processor and protected pages or segments in main memory. In case of interrupt, only pointers need to be stacked.

In main memory, access to any page or segment by any of the proposed schemes (Wilburn, Van Horn, Dennis) will involve checking that the reference is to public data not locked out or to private data owned or co-owned by this processor. Otherwise transfer to emergency procedures is in order. This should of course only result from error (except for lock-out).

Ownership of private data can be extended to more than one processor only when new ones are called to serve, i.e. at fork time.

[illegible] Forks

Program execution always starts by activating a single processor, providing it with a program counter setting (label of segment) and a pointer to an initially empty set of private data. To get more than a single execution sequence, an increase in active processors must be possible: the fork. Forks are a new type of macro instruction that can be specified in a program segment. Their function is the creation of a new set of private data and segment label for the benefit of an additional processor. If none is available, a pointer to this data is sent to the processor scheduling system, which hands out such pointers to physical processors as soon as they become available. The fork can optionally specify one of several priority levels for consideration by the scheduler. If only one job is considered, the scheduler could easily be implemented in hardware. [illegible] the scheduler must also consider the requirements of other jobs and I/O traffic. It will keep a number of queues, one per priority level, [illegible]ing the priorities by some algorithm. A processor then becomes necessary for the scheduling job. It can be obtained by interrupting any one of the active processors. (The interrupted sequence resumes immediately if a processor becomes available before the end of the scheduling activity). Alternatively, a special purpose processor can be reserved for scheduling and other supervisory tasks.

Expense for hardware to speed up the red tape involved in forks is justified. It reduces the minimum segment length for which parallelization is worthwhile: large numbers of low priority requests for short segments can then be stacked. Even if the majority will ultimately be sequentially processed, due to the limited number of processors, an over-all gain in job completion time will result. This applies only to the highest priority job in multiprogramming. It is very questionable whether parallel processing of any but the highest priority job is meaningful. The most likely situation is the presence of one job of extreme urgency and of others forming a background which prevents wastage of processing capability. In any case a lower priority job would have no incentive to fork for short segments under a reasonable scheduling algorithm.

Richards is quoted by Conway as having shown that a single queue of "state words" is not optimum. I do not know his assumptions but it seems plausible that segment length should be taken into consideration.

4. Segment Definition

Let D_o be the public data at the time processor k begins execution of segment i. Since other processors are constantly modifying public data, D_o is only defined modulo equivalence for the operations of segment i. The equivalence classes for execution of i by k can not be affected by the other processors since they are assumed to be compatible. Let P be the initial state of data private to k (within equivalence for co-owned data). We allow only one fork per segment (fragmenting segments where necessary). At termination of segment i we have at most two processors with private data P_1 and P_2 which are to continue with segments labeled j_1 and j_2. The public data has undergone a transformation into equivalence class D.

We have, modulo equivalence for segment i

$$\text{segment } i \begin{cases} D = f_i(D_o, P) \\ P_1 = h_i^1(D_o, P) \\ j_1 = g_i^1(D_o, P) \\ P_2 = h_i^2(D_o, P) \\ j_2 = g_i^2(D_o, P) \end{cases}$$

This description is symmetrical with respect to the two prongs of the fork. Asymmetry is introduced if we make the convention that processor k is to continue with P_1, j_1 while a pointer to P_2, j_2 is sent to the scheduler. If we want both requests to go to the scheduler, we can do so by construction of segment j_1.

$$\text{segment } j_1 \begin{cases} f_{j_1}(D, P_1) = D \\ h_{j_1}(D, P_1) = \emptyset \quad \text{the empty set} \\ g^1_{j_i}(D, P_1) = o \quad \text{label of halt} \\ g^2_{j_1}(D, P_1) = P_1 \\ h^2_{j_1}(D, P_1) = j_3 \quad \text{the desired continuation.} \end{cases}$$

Forkless segments are characterized by $h^2_i = \emptyset$ and $g^2_i = 0$, this amounts to the single processor situation on the cartesian product D x P (within equivalence, since other processors modifying D and co-owned parts of P are running). In case $j_1 = j_2 = o$, $h_1 = h_2 = \emptyset$ we say that the processor "quits". Instead of signifying job completion this means that all private data is released (returned to free storage unless co-owned by another processor) and the scheduler informed of the processor's availability. Only when the scheduler has no requests in store for the job and all processors working on it have quit is job completion reached (but see exception under look-ahead below).

5. Flow Diagrams

For monosequence processing, diagrams need only the following symbols: the box (operator), the junction (or confluence), the two-way branch, the subroutine return connector, the halt.

With proper interpretation only two new symbols are necessary for multiprocessing: the fork and the locked brahch. Definitions of the flow diagram symbols in terms of segments are now given:

$$\underline{\text{Box}}: \quad g^2_1 = o \qquad h^2_i = \emptyset \qquad g^1_i = \text{constant}$$

Figure 4a.

Junction : indicates equality of successor labels of two boxes; it is not a "join", no waiting is involved. Figure 4B.

Branch: $f_i\ (D, P) = D$; $h_i^1\ (D, P) = P$; $g_i^1\ (D, P) = j_1$ or j_2

$g_i^2\ (D, P) = o$; $h_i^2\ (D, P) = \emptyset$

Figure 4c.

Locked Branch: $g_i^1\ (D, P) = j_1$ or j_2

$g_i^2\ (D, P) = o$ $h_i^2\ (D, P) = \emptyset$

Figure 4d.

The locked branch differs from the combination of a box and a two-way branch in that the elements of data relevant for f_i, h_i^1, g_i^1 are locked to all other processors during execution of the locked branch. In most cases a single add-to-memory operation is sufficient (with a copy of the same left in the accumulator), lock-out can then be automatic by virtue of the memory access system. In more complex cases the page or segment containing the relevant items has to be marked as locked (not as private) before execution begins and is unlocked at completion. If it is found locked requests are repeated until access is granted. This is feasible because locked operations will only amount to a few instructions and the number of processors fighting for the data is limited by the number of physical processors.

Lock-out of co-owned private data by one of the owners to all others is likewise required for all elements of data relevant to a locked branch.

Return connector $f_i\ (D, P) = D$; $h_i^1\ (D, P) = P$;

$g_i^1\ (D, P) = \phi(P)$; $h_i^2 = \emptyset$; $g_i^2 = o$

Figure 4e.

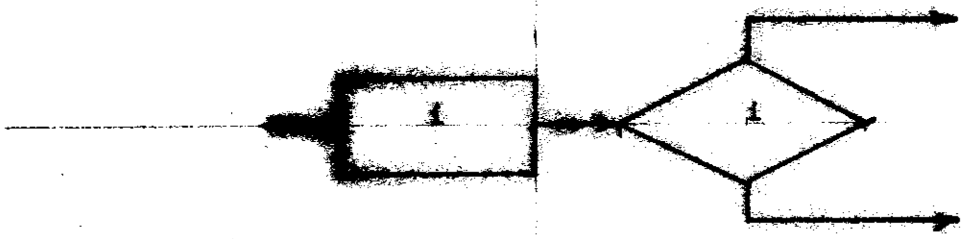
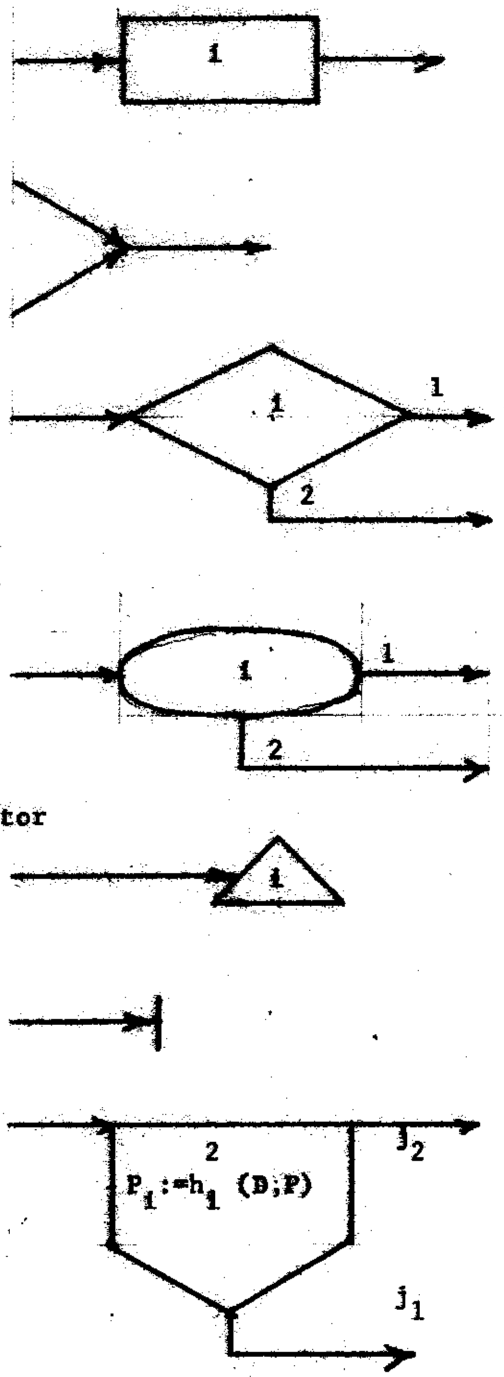


Fig. 3 Flow diagram for a segment



a) box

b) junction

c) branch

d) locked branch

e) return connector

f) exit

g) fork

Fig. 4 Block diagram symbols for specialized segments.

[illegible] return address is saved as private data as are parameters and [illegible] results, to permit simultaneous multiple execution of the [illegible]. With nesting and recursion large amounts of private data [illegible] generated temporarily.

Halt or quit: $g_i^1 = g_i^2 = o; \quad h_i^1 = h_i^2 = \emptyset \quad f_i\ (D, P) = P$

Figure 4f.

Fork:

$$f_i(D, P) = D$$
$$h_i^1\ (D, P) = P$$
$$g_1^1\ (D, P) = \text{constant} = j_1$$
$$g_i^2\ (D, P) = \text{constant} = j_2$$

Figure 4g.

[illegible] essential feature of a fork is the creation of an initial set of [illegible] data h_i^2 (D, P) and a label j_2 for the new processor (this is [illegible] to Conway's state-word), with a request for scheduler assign[illegible] of a physical processor.

[illegible] Basic Example

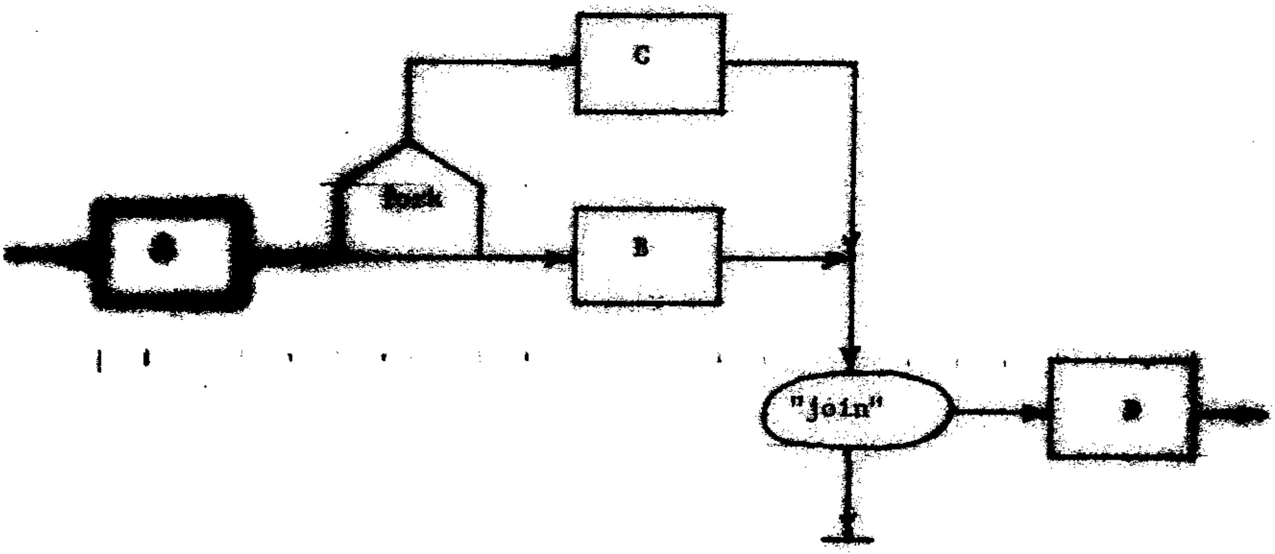
A basic example is illustrated by Figure 5. Let A, B, C, and D be [illegible] segments and let the executions required from B and C be compatible. [illegible] public variable T: = 2, before the fork. The locked branch is an [illegible] memory of -1 to T; the quit exit is taken unless the result is zero, [illegible] case continuation is with D. This is the usual "join" operation, [illegible] the locked branch is a more powerful concept.

9. [illegible]ware Control of Data Ownership

For each page of segment a set of data control bits have to be [illegible] for instance as follows. With four physical processors, we use ten [illegible] shown in Fig. 6.* The interpretation of the data control bits is [illegible] the following example.

1.	Free storage:	000000 1111
2.	Public data:	101111 1111
3.	Public data locked out by processor 3:	101111 0010
4.	Private data of yet unscheduled processor	010000 1111
5.	Private data owned by processor 3:	000010 1111
6.	Private data co-owned by processor 3 and one or more yet unscheduled processors:	010010 1111
7.	Same as 6, locked out by processor 3:	010010 0010
8.	Private data co-owned by processors 1 and 3:	001010 1111
9.	The same locked out by processor 3:	001010 0010

* The lock-out bits can be replaced by bits of a binary number spedifying the locking processor with zero meaning no lokk-out.



[redacted] [redacted]ic example.

D	S	P1	P2	P3	P4
		L1	L2	L3	L4

[redacted] [redacted]ol bits for data ownership.

Separate bits for private ownership and lock-out are necessary, [illegible]ise [illegible] on what pattern to restore at the end of the [illegible] branch is lost. With a six bit set, if the locking processor stores [illegible] bits [illegible] locking, restoration of the initial pattern may want to [illegible] changes in the ownership status of the segment and there is no reason [illegible] them wait. A 7→9 transition occurs when the scheduler assigns [illegible] processor one with this segment on its initial private data list. [illegible] transition occurs if Proc. one releases the segment or quits. In [illegible] the lock-out bits are then immaterial though processor 3 has no way [illegible]ing, if he was not sole owner (Sole ownership cannot be changed [illegible] the owners; see below)

[illegible] [illegible]iation of Data Ownership

Programs begin with a single processor and all public data. The [illegible] processor can create data private to itself as copies or [illegible]ns of public data, it can operate on both kinds.

At the first fork, it specifies what initial private data the new [illegible]sor will have (there may be none). To this effect

a) It wreates data which becomes private solely to the new processor.

b) It declares some of its own private data to be common to the [illegible] processor and to itself.

[illegible] "allocation" activity (for lack of a better word) we call a "fork [illegible]t."

Both processors can produce new forks with the same rules so that [illegible] structure expands.

The scope of elements of data is delimited on one side by public, [illegible]te and fork declaration, and on the other side by the following erasing [illegible]ions.

Erasure of public data (return to free storage) can be effected by [illegible] processor. That this should not cause conflicts is part of the compati[illegible] requirement.

Release of private data owned solely by the processor will cause its [illegible]. A processor can not always know, without a locked branch on the data [illegible] bits, whether its ownership is sole. Once it has made the data common to another processor the future course is no longer under its control. But if ownership is sole at any time this fact can not be changed without the processor's consent.

Release of private data which is common to other processsors (which may still be unscheduled) only lowers the corresponding data control bit. When the last-but-one processor releases we are back to sole ownership and finally, when the last processor releases, to erasure.

When a processor quits it automatically releases all its private data.

[illegible]. Programmers Description of Data Control, Forks, Locked Branches

We modify Algol, to eliminate the block concept; declarations and releases are explicit and at the whim of the programmer (not nested), begin and end serve only for statement parentheses.

Examples:

```
    declare public real X;  integer array NAME (1:N*M)
```

and later

```
    release X
```

while the array stays on.

Similarly

```
    declare private ŷ
```

and later

```
    release ŷ
```

or perhaps

```
    quit
```

We use here ^ when we want to underscore the fact that a variable is private, for exposition purposes.

At a [illegible] we can specify

```
  fork  <declaration part> <assignment part>
        <priority part> to <designational expression>
```

for instance

```
  fork declare private Î common ŷ; Î: = Ĵ + K -1;
  priority 5    to lab;
```

[illegible] is public, $\hat{J}$ is known to the current processor but will not be known to the new one, $\hat{I}$ will be known to the new one only, $\hat{Y}$ will be known to both.

The priority level is for the scheduler, it could be an arithmetic expression. When scheduled the new processor will begin at label "lab," at that time the values of $\hat{y}$ and $\hat{J}$ may have changed, that of $\hat{I}$ not.

At a locked branch

```
        begin  lock T;  T: = T-1; if T > o quit else go to lab end
```

If the fork in the basic example is part of a subroutine which can be called by several processors, the variable T must be declared private to the entering processor who then extends its ownership by a common declaration at the fork. This shows the necessity of distinguishing between privacy and locking.

[illegible] Example of Matrix Multiplication

We presume the following declarations of public data:

```
        integer N, real array A(1:N,  1:N),  B(1:N,  1:N),   C(1:N,  1:N)
```

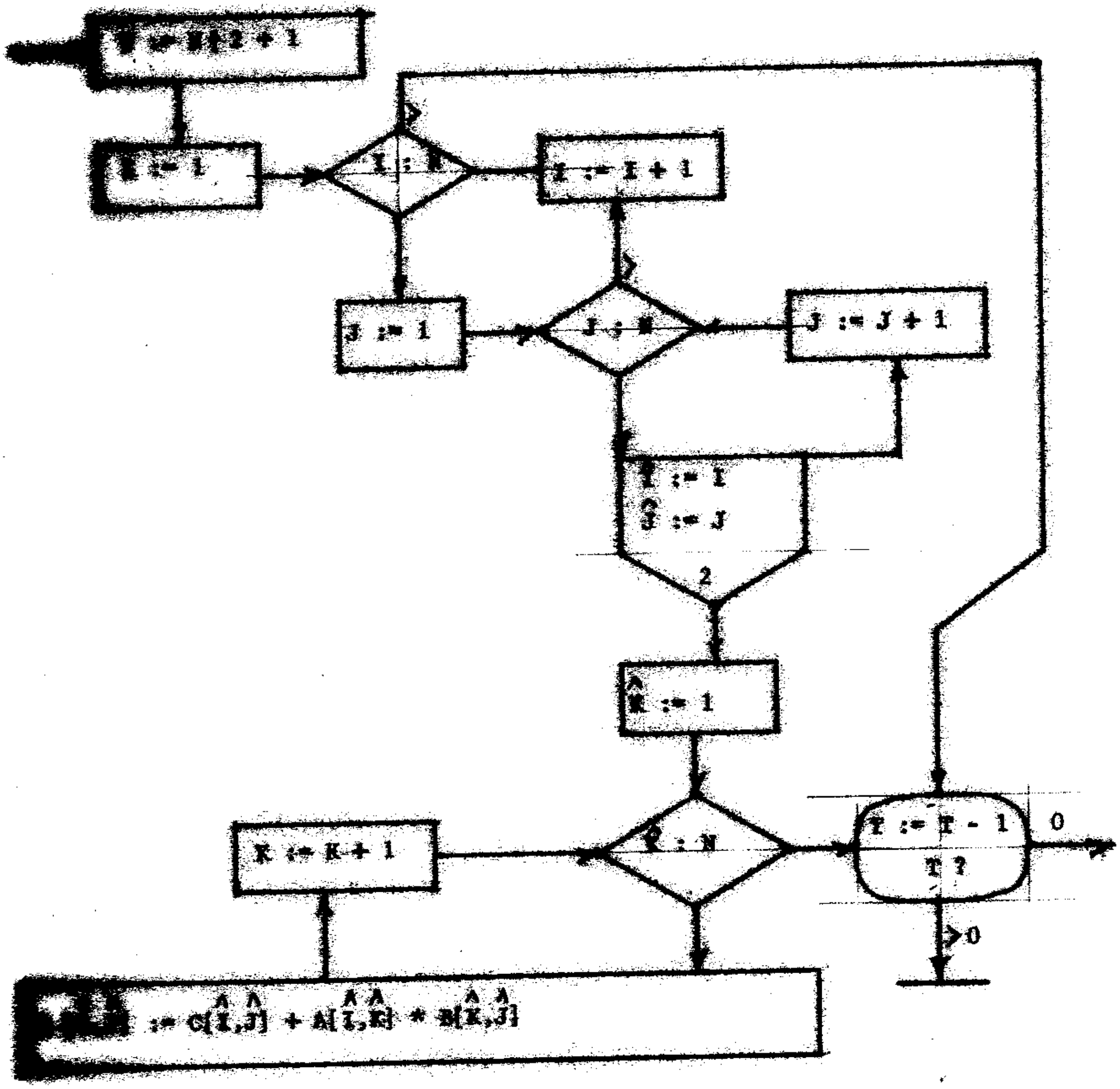
A possible program is the following which is diagramed in Fig. 7.

```
declare  public  integer  T, I, J; T:=$N^2$ + 1
for I: = 1 step 1 until N do
        for J :=1 step 1 until B do
fork declare private integer  $\hat{I}$, $\hat{J}$; $\hat{I}$: = I; $\hat{J}$: = J;
priority 2 to term;
test:  begin lock T; T: = T-1;
if T> o quit else go to next end
term:  declare private $\hat{K}$; C($\hat{I}$, $\hat{J}$): = o;
for $\hat{K}$: = 1 step 1 until N do

        C ($\hat{I}$, $\hat{J}$): = C($\hat{I}$, $\hat{J}$) + A($\hat{I}$, $\hat{K}$) * B($\hat{K}$, $\hat{J}$);

release $\hat{K}$, $\hat{I}$, $\hat{J}$;
go to test;
next: - - - -
```

Note that we can just as well have the initial processor quit unconditionally after issuing the N^2 forks (set initially T := N^2).



[illegible] 7 Flow diagram for matrix multiplication.

How many storage positions for $\hat{I},\hat{J}$ must be availabld? For k<< N^2+1 processors, if the scheduler queue system builds up to a maximum of N^2-k we need k+q stores which could be close to N^2. To economize this storage one has to test queue length (a public read-only variable) before each fork. If q is too large the processor queues itself with appropriate priority.

For instance

```
if Q > 10 fork declare common all, priority 2 to next; quit
next: . . . .
```

The "declare common all" saves all private data (if any-none is involved in the example) for the re-emergence of the processor. It is then up to the scheduler.

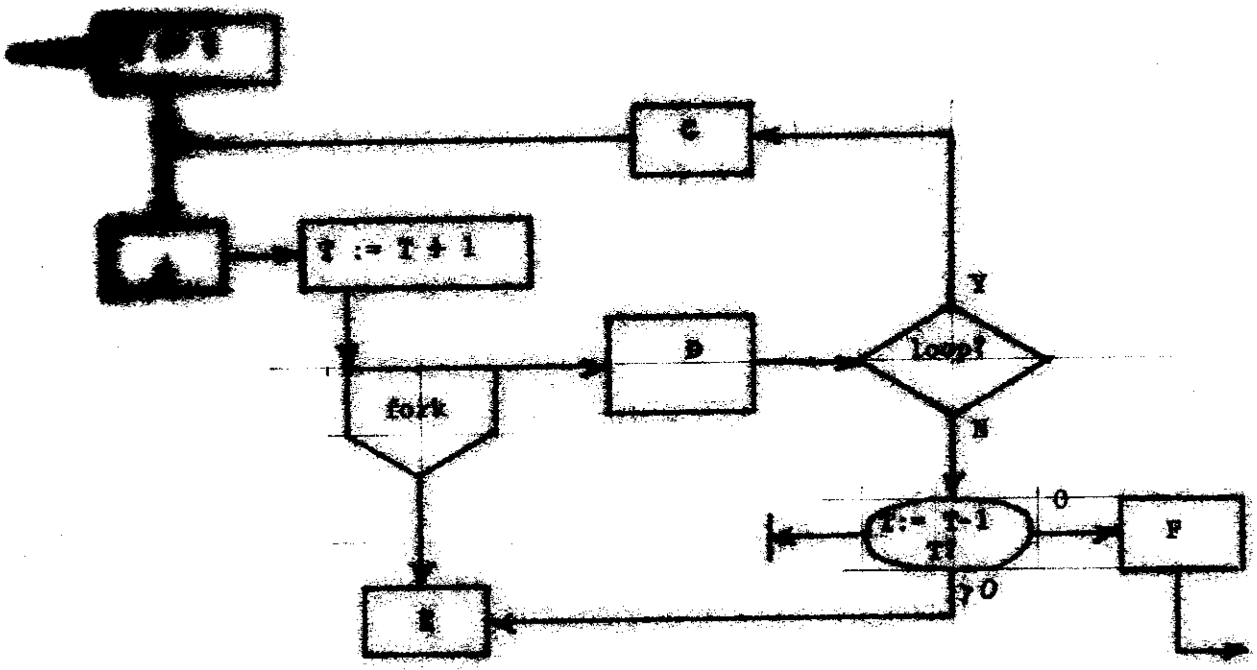
[illegible]. Loops of the "While" Type

Loops of the "while" type differ from the previous case in that the loop variable can not be set at entry. Instead we start at one and index it each time the range of the loop is entered as illustrated in Fig. 8. Fig. 9 shows a case where two side computations are done for each traversal of the range. If B is traversed n times, so are C, D, G and I. After loop exit, H and J will be traversed once when all n executions of G and I, respectively, have been accomplished. Box K is executed once upon completion of both H and J.

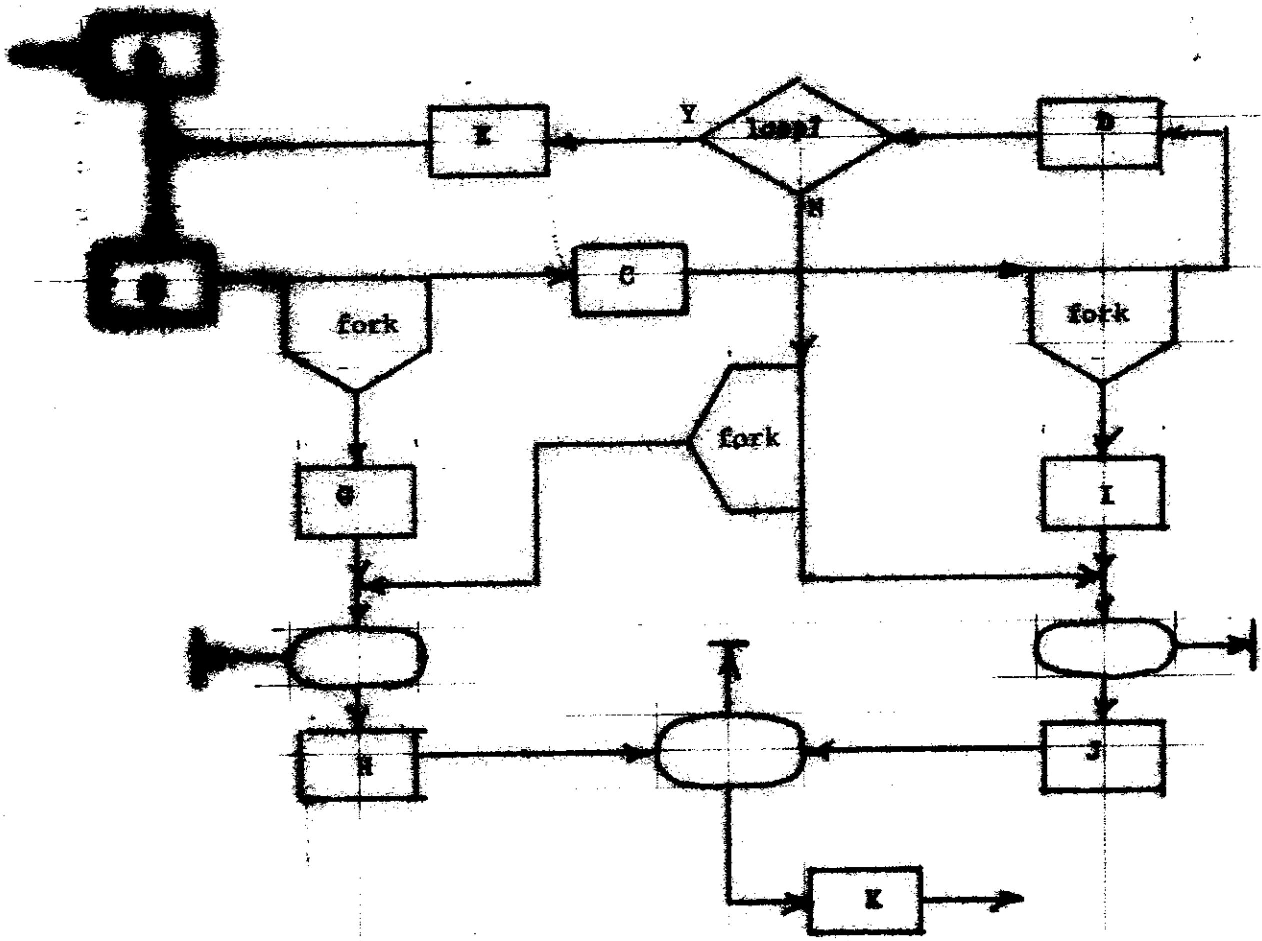
[illegible]. Use of Locked Branches for Scope Termination

It it is important to release storage space of some large public array as soon as possible, we can use locked branches as shown in Fig. 10. A possible coding is

```
declare public M;   T:=2;
a: - - - - ;
fork declare common all, priority 2 to C;
```



[illegible] A loop of the "while" type.



[illegible] Another "while" loop.

```
b: - - - -; begin lock T; T:=T-1;
        d:if T=0 go to $l_1$ else $l_2$ end
           $l_1$: release M;  $l_2$: - - - -
c: - - - -; begin lock T; T:=T-1;
        e:if T = 0 go to $l_3$ else $l_4$ end
           $l_3$: release M; $l_4$: - - - -
```

[illegible] is private no locked branch is necessary, we simply release M at [illegible] and the data control bits take care of the rest.

[illegible] **Publication, Grabbing and Lock-Out of Data**

A processor can publish any of its private data, this being equivalent [illegible] declaration of a public copy before release; this can be abbreviated [illegible] X.

A processor can grab public data by declaring a private copy of it [illegible] releasing (= erasing) the public data. Abbreviate grab X.

By first grabbing and later publishing data elements we obtain lock-[illegible] in the ordinary sense, independently of the locked brahch concept. It [illegible] believed that such lock-outs are rarely justifiable in programming since [illegible] tend to nullify the speed advantages of parallel processing.

[illegible] **[illegible]ng-in of Subprograms**

A segment of the box type (defined previously) has the property [illegible] for each entrance of a processor there will be one and only one exit [illegible] corresponding processor. To analyze complex flow diagrams it helps to [illegible] subprograms with a single entry and exit which have the same property. [illegible] condition is that the number of quits always balances the number of [illegible] before emergence of the processor or of one of its descendants at [illegible] exit. Just as for segments, these boxes can,in general, be subject to [illegible] simultaneous executions - namely if a new processor enters before [illegible] previous one exits.

The box concept can be generalized to the single entry multiple exit case. The condition is then that for every entry there will be an emergence at one and only one of the exits. Again all forks must be balanced by quits before the emergence. Different boxes can call the same subroutine, provided the subroutine is itself a box (no"side-effects").

15. Execution Trees

To check a program as to compatibility of all segments that can be executed in parallel, it is helpful tofind all commutation conditions. These can be found by imagining execution with a single processor with a decision at each fork and each quit, diagrammed as a direct tree graph.

At a fork the number of sequences ready to run increases by unity. This number is shown by the exit degree of the corresponding tkee node. At a quit the number decreases by one. Therefore successive tree nodes differ by ± 1 in their exit degree. The root (entry to program) has exit degree 1 while tips of the leaves (exit degree 0) are reached when all sequences have quit. Each arc is a transformation of the data, or operator. Writing the string of operations for every path from root to a tip and requiring these strings to be equivalent gives the commutation conditions. The tree is in general dependent on the initial data, so that a family of trees (forest) has to be considered.

Boxing-in allows to go through this procedure in steps, beginning with the trees for the innermost, simplest boxes. At the next level, the commutation conditions having been checked, these boxes are equivalent to an operator.

16. Look-ahead

Definitions of look-ahead programs:

Definition 1: The final exit can be reached with some processing still occuring.

Definition 2: The program can proceed to completion of its task without having to wait for execution of all the sequences it initiated.

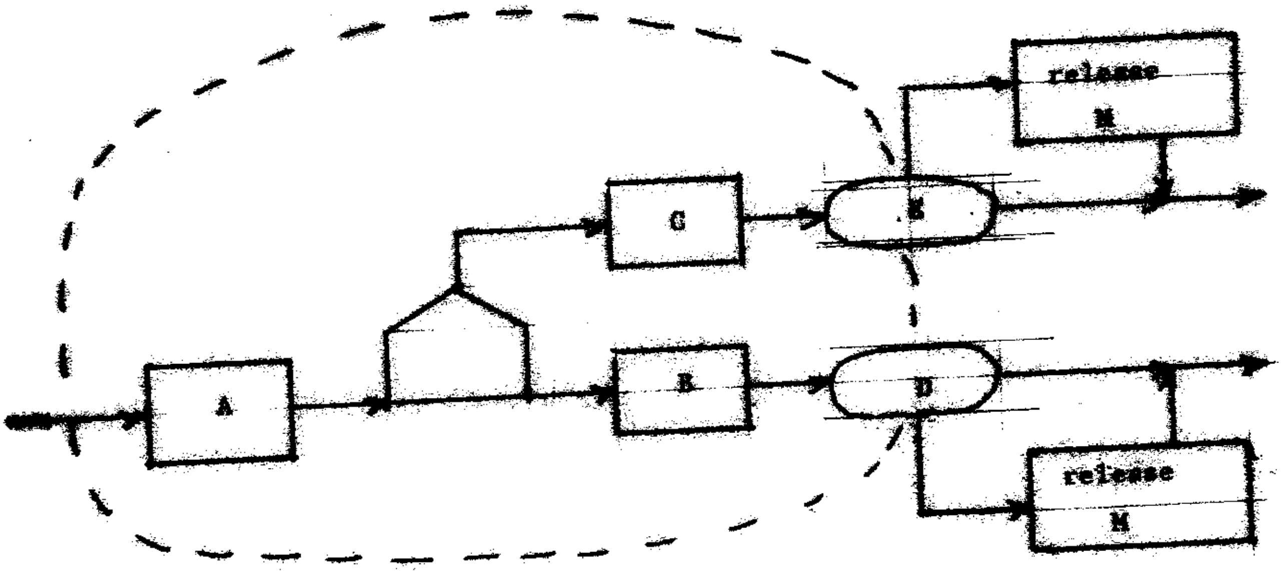


Fig. 10 Use of a locked branch for scope termination.

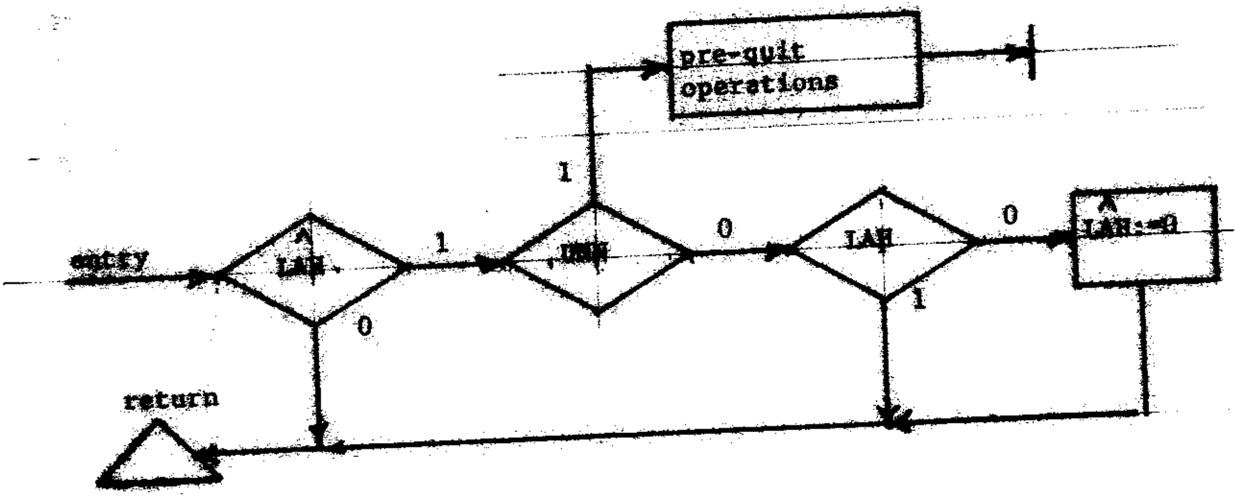


Fig. 11 Subroutine for look ahead control.

Definition 3: The entire program cannot be considered a single box.

A look-ahead sequence is one which may turn out unnecessary; it is initiated by a fork and proceeds in "look-ahead status". At a later time another processor will discover whether it has become a necessary or unnecessary sequence. The fork is taken as soon as the data is available, necessity is established or disproved at a later time.

The major problem is to communicate the change in status to the look-ahead processor (and its deseendants) despite the impossibility of direct processor to processor communications.

The suggested solution uses boolean variables and a hardware feature. A private look-ahead bit $\widehat{\mathrm{LAH}}$ is associated with every processor (in its "state word") and is normally zero. At a look-ahead fork this bit is set to 1 and a pointer is set to two bits LAH and UNN declared common to both sides of the fork. The subroutine in Fig. 11 is then prepared, partly in hardware form. At the fork we set

$$\mathrm{LAH}: = \widehat{\mathrm{LAH}}: = 1 \quad \mathrm{UNN}: = 0$$

When necessity is established, set LAH: = 0, when it is disproved set UNN: = 1.

The subroutine could be run automatically, as long as $\widehat{\mathrm{LAH}}$=1, by the scheduler before assignment of a physical processor and then by that processor every k instructions.

As long as $\widehat{\mathrm{LAH}}$=1 all descendants of the processor are initiated in look-ahead status, with common ownership of LAH and UNN.

The subroutine action for UNN=1 can be a jump to the end of the look-ahead sequence.

As an example consider the "while" loop implementation in Fig. 12 where 1, 2, 3, . . ., 7 designate boxed subroutines. Box 3 is a parallel branch inside [illegible] is necessary only in some revolutions, and for which the data are available before test of predicate P decides its necessity. The method also accounts for the case that on the n^{th} revolution an unnecessary look-ahead sequence started on a previous revolution may still be running.

Note that the arrays indexed with ϵ are always of small size, regardless of ϵ, because the lower ϵ-values are progressively released. The value of ϵ could be incremented modulo a sufficiently large integer if very large revolution numbers are expected.

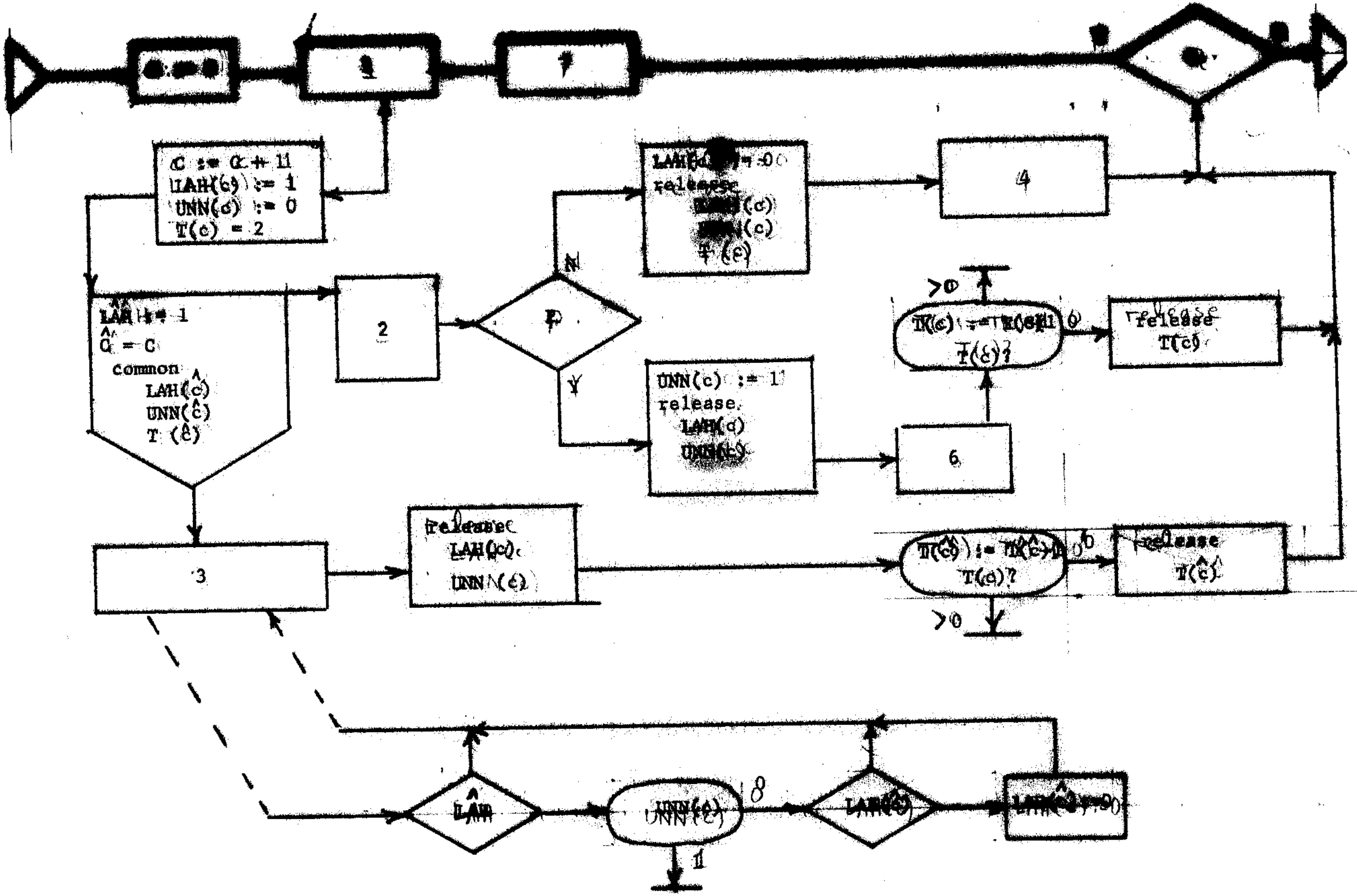


Fig. 12 An example of look-ahead.

[illegible] baker's take-a-check system).

A simpler method is available

if P=1 wait for termination of box 3 at the "join"

if P=0 do not begin box 6 without hhecking (by a common boolean variable) that box 3 has been stopped.

[illegible] way we prevent restart of the loop with old look-ahead sequences [illegible] running. It is felt that the complications of the first approach are [illegible]ile time-wise, expecially if the look-ahead-status checking subroutine [illegible] be programmed and therefore is called at longer intervals.

[illegible] Sharing Processing Capability

Some operations which can be accomplished much more rapidly by [illegible] hardware than by subroutines do not occur quite often enough for [illegible]ion of this hardware in every processor. It seems worthwhile to [illegible] an appropriate number of these units, one for operation A, 2 for [illegible] frequent operation B, etc. for, say, 10 processors. Call is by [illegible]odes": if all special units are busy, transfer to the subroutine is [illegible]ic.

If the time difference (subroutine-hardware) is considerable, one [illegible] queue requests up to a maximum length after which the subroutine is [illegible].

[illegible] Applications

An obvious class of applications for this type of multi-processing [illegible] integration of ordinary differential equation systems. Speed gains by [illegible] proportional to the order of the system (up to the number of physical [illegible]sors) are possible and parallelism is easy to specify. Such problems [illegible] occur under time pressure conditions (missile tracking). The superiority [illegible] present approach compared to DDA's or a [illegible]-type system is the [illegible]ance of compilation techniques with changes in problem size. The [illegible]uted machines must fit the problem to a fixed equipment complement [illegible] hand-tailored manner.