DIRECT ARCHITECTURAL IMPLEMENTATION OF A REQUIREMENTS-ORIENTED COMPUTING STRUCTURE

M. S. Anastas and R. F. Vaughan
The Boeing Company
P.O. Box 3999
Seattle, Washington 98124

Many of the advances in computing technology that have a direct bearing on software productivity have been integrated into a unique requirements-oriented computing structure devoid of transfer of control constructs. Overhead is a major obstacle to such a structure. A computer architecture based on this structure is therefore derived for which combinational logic can be used to solve this overhead problem. A family of computers called Transition Machines is thereby defined which addresses many of the major problem areas affecting software productivity.

Introduction

It is ironic that when software productivity is unilaterally identified as a colossal problem, the major advances in this area have all been within the traditional software framework provided by Von Neumann machines. These advances have been in the areas of program specification, top-down design, structured programming techniques, decision diagramming, the implementation and acceptance of structured higher order design and compilation languages, and the parameterization and modeling of software design issues. But there have been no parallel major advances to the state-of-the-art in computer architecture to support these strictly software and programmer-oriented advances.

Hardware advances have all been characterized by one or more of the following descriptive adjectives: smaller, faster, more reliable or cheaper. All of these are amazing, but they don't address the cost of software, and before systems (the real products) are cheaper, software productivity must be accommodated by machine organization.

It is proposed here that a new framework for software may be essential to obtaining the desired productivity. To obtain this framework, software productivity requirements are addressed directly. Once such a framework has been defined, an architectual implementation of this framework is developed which meets these original requirements.

Software Productivity Requirements on Structure

The requirements that have been proposed for a new computation structure follow basically those identified by enthusiasts of structured programming, but the approach here is to extend these requirements and impose them at the architecture level of the machine. These extended requirements are the following:

1. a requirements-oriented structure.

2. a structure for which only the essential aspects of program control must be specified by the programmer.

3. a structure which eliminates transfers of control (GO TO, CALL, queueing requests, etc.).

4. a structure which simplifies the error-prone aspects of decision logic.

The significance of each of the requirement areas is discussed below.

Requirements-Oriented Programming

It is commonly accepted that the specification of program requirements, the transformation of these requirements into program design data, and the subsequent verification that the design meets the requirements are the major software development activities. These are therefore the areas where improvements must be made if software productivity is to be significantly increased.

One reason that these development activities require such a large percentage of the overall development process is because of the incongruities between the typical situation/response nature of program requirements and verification tests, and the procedural implementation of these requirements in the final design. Thus, there is a major translation from requirement to design which is a time-consuming, error-prone process. Once the design has been established and implemented, it no longer resembles the structure of the requirements. The gulf between requirements and design hampers program understandability and therefore documentation, maintainability, etc. Thus, it is accepted as virtually impossible to verify that a program of any magnitude actually implements its requirements precisely. A computation structure for which there was more direct traceability to requirements would certainly be associated with a tremendous increase in productivity.

Since requirements are typically of the form: "When a certain situation arises, perform an associated function," it would be nice if programs also had that form. Situations are typically describable in terms of propositions on parameters which are represented in the data base. The associated functions correspond to programs which change parameter values. Thus information contained in requirements and design data are essentially the same with the difference in

CH1516-4/79/0000-0093$00.75 © 1979 IEEE

form being primarily organizational.

Elimination of Degeneracy

In the conventional software framework the programmer translates the computation requirements into a sequence of actions to be performed by the machine. In many cases, however, there is no sequence of actions implied in the requirements, so a sequence is artifically induced by the programmer to provide concreteness since the computation structure supported by conventional languages and computer architectures does not accommodate the specification and execution of unordered statements. Thus, variant "correct" program versions can exist with entirely different structures. These are degenerate solutions to the problem described by the requirements specification. In essence the programmer has the ability (actually the responsibility) to induce his own design philosophies and personal preferences into what would hopefully have been an objective translation of a requirement that could have been directly compared with other versions. This has the effect of making the implementations less directly relevant to the requirements such that verification must be performed only at the lowest level to determine that the final result does in fact precisely meet the requirements.

Exploitation of the parallelism inherent in the computation is also precluded by arbitrary determination of execution sequence. This reduces program effectiveness in parallel processing environments. The amount of parallelism available in typical programs has been investigated by Kuck[3] indicating that the average number of possible parallel paths in a program is linearly related to the size of the program rather than related as the order of log of the size as was formerly thought to be the case[5]. Thus there could be a significant increase in reaction time for large real time programs.

The inherent relationships between program segments is also obscured such that later program maintenance is more likely to induce errors. Thus, by requiring only the essential aspects of program control, the requirement translation process would be simplified, many of the previously arbitrary decisions on sequence made by the programmer would be eliminated, and exploitation of parallelism would be supported.

Eliminating Direct Transfers of Control

The direct transfer of control has been identified as the major source of programmer coding errors[4,7]. Many of the developments in software engineering have been directed to the elimination of these structures from programs. The result has enhanced software productivity, but since these structures are still supported to varying degrees in compilers, and unilaterally at the machine level, many errors can still be attributed to them. Conventional executive requests to execute a specified program are also GO TO's, the only difference being that they are implemented at a high level. There seems, however, to have been no concerted effort to eliminate these.

The executive request problem is an interesting one in that a program could be completely correct but not be executed under the right conditions because it was incorrectly requested. Thus, there is an intricate coupling between programs. There is also the possibility of a totally unsuspected program requesting the execution of a program for a completely inappropriate situation. Before a program can be completely verified to meet its requirements, every set of conditions under which it can be requested must be known, and therefore every requesting program must be verified along with it.

Simplification of Decision Logic

The elimination of GO TO's is significant from an error-proneness point of view, but decision logic structures are very major offenders also.[7] Decision diagramming has been used to address some of the error proneness of these logic structures. But they are a monitoring and evaluation tool not implemented as a part of the program structure in the design and thus their use constitutes a divergence (additional effort) from the central development path.

The typical decision logic constructs involve a conditional transfer of control which therefore allows circumvention of GO TO-less dogmas at the detailed implementation level. They also have the feature of treating program activation conditions in an indentured manner without global awareness. A particular test for $a>b$, for example, may only be executed if $a<c$, $d>e$....But this total situation may not be readily apparent to the programmer writing/reviewing the code. Therefore, very complex meshes of logic may be implemented. This makes it very difficult to provide assurance that the specific conditions of execution are not precluded from ever being realized because of a decision higher in the structure.

Computation Structure

Figure 1 shows a program whose structure is propounded. It addresses many of the software productivity concerns described above. It is characterized by a totally event driven structure with only assignment statements even for the implementation of decision logic. This is facilitated by using logical operators to assign the status of data base conditions that are determined dynamically during the execution of programs. As a result, there is a complete elimination of direct transfer of control structures. A very requirements-oriented control structure has resulted, for which only the essential aspects of program control must be indicated, and these obtain global significance. Parallelism is easily exploited in that multiple sets of satisfied activation conditions can result in multiple tasks being simultaneously executed.

In this structure each program is divided into two components: 1) a complete specification of the globally defined conditions required to enable the program (WHEN a list of conditions is met), and a

```
DECLARATIONS
  BOOLEAN  INITIALLY ("1"), FACTORIAL.READY ("0")
           SUM.READY ("0"), CONTINUE.SUM ("0"),
           CONTINUE.FACTORIAL("0"), COMPUTE.RATIO ("0")
  INTEGER  COUNT, FACTORIAL, SUM
  REAL     ANSWER
PGM1:  WHEN (INITIALLY) DO
         BEGIN:
           COUNT : = 0
           FACTORIAL : = 1
           SUM : = 0
         END
       THEN
         SET (FACTORIAL.READY, SUM.READY)
         RESET (INITIALLY)
PGM2:  WHEN (SUM.READY, FACTORIAL.READY) DO
         BEGIN:
           COUNT : = COUNT + 1
           CONTINUE.SUM : = COUNT ≤ 20
           CONTINUE.FACTORIAL : = COUNT ≤ 20
           COMPUTE .RATIO : = COUNT > 20
         END
       THEN
         RESET (SUM.READY, FACTORIAL.READY)
PGM3:  WHEN (CONTINUE.FACTORIAL) DO
         BEGIN:
           FACTORIAL : = FACTORIAL * COUNT
         END
       THEN
         SET (FACTORIAL.READY)
         RESET (CONTINUE.FACTORIAL)
PGM4:  WHEN (CONTINUE.SUM) DO
         BEGIN:
           SUM : = SUM + COUNT
         END
       THEN
         SET (SUM.READY)
         RESET (CONTINUE.SUM)
PGM5:  WHEN (COMPUTE.RATIO) DO
         BEGIN:
           ANSWER : = SUM/FACTORIAL
         END
       THEN
         RESET (COMPUTE.RATIO)
```

FIGURE 1: REQUIREMENTS-ORIENTED PROGRAM STRUCTURE

specification of the conditions to be updated on completion of the program (THEN a list of condition revisions); and 2) a set of data transformation statements. Table I shows the complete dichotomy of control and data transformations. There are obvious similarities to the formal program specification technique, described by Parnas[6]. This is the computation specification that is proposed to be implemented directly in a machine architecture.

NAME	CONTROL INFORMATION	DATA TRANSFORMATION
PGM 1:	WHEN (INITIALLY) THEN SET (FACTORIAL.READY, SUM.READY) RESET (INITIALLY)	BEGIN: COUNT: = 0 FACTORIAL: = 1 SUM: = 0 END
PGM 2:	WHEN (SUM.READY, FACTORIAL.READY) THEN RESET (SUM.READY, FACTORIAL.READY) SET VARIABLY (CONTINUE.SUM, CONTINUE.FACTORIAL, COMPUTE.RATIO)	BEGIN: COUNT: = COUNT + 1 CONTINUE.SUM: = COUNT ≤ 20 CONTINUE.FACTORIAL: = COUNT ≤ 20 COMPUTE.RATIO: = COUNT > 20 END
PGM 3:	WHEN (CONTINUE.FACTORIAL) THEN SET (FACTORIAL.READY) RESET (CONTINUE.FACTORIAL)	BEGIN: FACTORIAL: = FACTORIAL*COUNT END
PGM 4:	WHEN (CONTINUE.SUM) THEN SET (SUM.READY) RESET (CONTINUE.SUM)	BEGIN: SUM: = SUM + COUNT END
PGM 5:	WHEN (COMPUTE.RATIO) THEN RESET (COMPUTE.RATIO)	BEGIN: ANSWER: = SUM/FACTORIAL END

TABLE I: CONTROL/DATA TRANSFORMATION DICHOTOMY

An abstract model of computation which was used as a mathematical basis for a model of computer architecture suitable for direct execution of syntax as shown in Table I has previously been identified by Keller as "Named Transition Systems."[2] A named transition system is a triple $(Q, \rightarrow, \Sigma)$. The components correspond respectively to the set of all possible system states $(q_1, q_2, q_3 \ldots)$, a set of all transitions between states $(\rightarrow_1, \rightarrow_2, \rightarrow_3, \ldots)$, and a set of names $(\sigma_1, \sigma_2, \sigma_3, \ldots)$ associated with groups of individually programmed transitions between states. Since there is a one-to-one correspondence between the indices on sigma and the names themselves, the indices will be used to indicate the names: i implies σ_i, and $I = \{i\}$ implies Σ. The index $i \in I$ is associated with a group of system transitions described by the statement:

$$\text{when } R_i(\xi) \text{ do } \xi' = \psi_i(\xi)$$

The symbols in this statement are defined as follows:

- i = the index of the group of transitions whose common feature is that they all result in the data transformation indicated by the function ψ_i.
- ξ = the set of all data items in the system.
- $R_i(\xi)$ = the predicates on the data set, ξ which are essential to defining the appropriateness of transitioning according to whichever member of the group i is associated with performing the data transformation $\psi_i(\xi)$.
- $\psi_i(\xi)$ = the programmed functional data transformation associated with the group index i which operates on the data set, ξ and results in a revised data set ξ'.

Keller seems to have envisioned this model as only a formalism to be used in the specification and verification of parallel programs, but in the remainder of this paper it is used as a computation structure on which to base a computer architecture.

Derivation of Transition Machine Architecture

A family of computer architectures (referred to throughout this paper as "Transition Machines") has been derived which directly implements the previously described computation structure. Figure 2 is a paradigm showing the general characteristics of this architecture; it is comprised of two major components as was the computation structure described above. The first component maintains status indications for all the relevant data base conditions in the system. It also contains indicators associated with each subsystem (subsystem is used interchangeably with application program task throughout the rest of this paper) specifying the subset of global data base conditions required to activate the specific subsystem and indicators specifying the modification to the conditions implied on completion of the subsystem. The second is a computation com-

ponent which executes the code associated with the data transformation aspect of each subsystem.

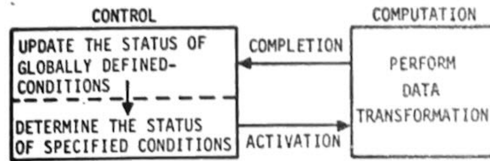

FIGURE 2: COMPONENTS OF TRANSITION MACHINE

The operation of these two components is as follows. The control component first determines an eligible subsystem by examining the current status of the global conditions specified by each requirements indicator associated with the subsystem. The eligible subsystem is identified to the computation component which then executes the specified sequential arithmetic operations associated with the subsystem and returns status indications specifying the conditions that have been modified dynamically by the subsystem. The control component updates the global condition indications associated with having completed the subsystem. The cycle is then repeated until the system runs to completion.

The dynamic status update is a requirement imposed by having no data base operations performed directly by the control component as part of the eligibility determination. The control component must therefore separately maintain the status of relevant conditions current in the data base. This is accommodated by incorporating an update to the global condition indications on completion of each subsystem.

The discussion that follows outlines a mathematical derivation of the constructs essential to the Transition Machine architecture from the named transition system model. This derivation proceeds from an intuitive statement, given here without proof, that any state $q \in Q$ can be represented by a binary status vector $S = (s_1(\xi), s_2(\xi)\ldots)$ such that:

$$q = q' \text{ iff } s_j = s'_j \text{ for all } j \in J$$

Where each element $s_j \in S$ is a binary status indication associated with a proposition on the data set ξ. For example, s_j iff $a<b$. Also included are logistic operations such as s_k iff "c available". J is defined as the set of all indices on S.

Eligibility Determination

An "enabling predicate", $R_i(\xi)$ can be represented by disjunctive sets of indices $(K_i^1, k_i^2, k_i^3, \ldots)$, where the sets of indices $K_1^\ell = (k_1, k_2, \ldots)$, are subsets of J and are defined such that:

$$R_i \text{ iff } s_k \text{ for all } k \in K_i^\ell \text{ for some } \ell \in L_i.$$

L_i is the range $\{\ell\}$ of disjoint sets of indices K_i^ℓ on the system status vector which are individually sufficient to satisfy the enabling predicate $R_i(\xi)$. Thus, an equivalent to the evaluation of the enabling predicate $R_i(\xi)$ can be effected by a complex logical operation on S. The determination of which elements of S to involve in this evaluation are determined by the set of indices, K_i^ℓ:

$$R_i = \bigvee_{\ell \in L_i} \left(\bigwedge_{k \in K_i^\ell} s_k \right)$$

The logical operators are defined as follows:

$$\bigvee_{n \in N} x_n \equiv x_1 V x_2 V x_3 V \ldots x_n \ldots \text{ for all } n \in N.$$

$$\bigwedge_{n \in N} x_n \equiv x_1 \wedge x_2 \wedge x_3 \wedge \ldots x_n \wedge \ldots \text{ for all } n \in N.$$

$V \equiv$ logical "or" operation

$\wedge \equiv$ logical "and" operation

All that is actually required however is an implementation of each of the evaluations which imply R_i which we will define as R_i^ℓ. To this end, we define:

$$R_i^\ell \equiv \bigwedge_{k \in K_i^\ell} s_k$$

The superscript on K_i^ℓ and R_i^ℓ can be eliminated by redefining the subscript as a running index, i' for which there is a one-to-one correspondence between values of i' and unique pairs of values (i, ℓ). The distinction between i' and i will be ignored, with the understanding that there may be multiple disjunctive transitions of the same name which are associated with unique indices. Thus the disjunctive expressions in $R_i(\xi)$ are handled separately but are still associated with the same data transformation $\psi_i(\xi)$. The range, I on the subscript i, defined by $i \in I$ can be used to restrict the range, J on the status vector, since only the indications in S that pertain to one or more enabling predicates have implementation significance.

$$J = \bigcup_{i \in I} K_i$$

This will result in a minimum length vector S which will be sufficient to support an implementation of the enabling predicates. And it becomes irrelevant whether our initial intuitive statement is true to its fullest extent or not.

For convenience a matrix of logical elements, r_{ij}, can be defined to replace the sets K_i, with elements defined as follows:

$$r_{ij} \text{ iff } j \in K_i$$

Each row in this matrix is associated with a disjunctive enabling predicate and each column is associated with a specific binary status indicator in S. The expression for an eligibility indicator E_i can now be defined as an operation on each element of S as follows:

$$E_i = \bigwedge_{j \varepsilon J} \overline{r}_{ij} V s_j ,$$

where $\overline{x}$ defines the logical complement of the binary indicator x. This equation defines a logical "dot product" which relates a vector of enabling predicate status indications to a binary status vector of global conditions. The analogy with inner products is more obvious from the equivalent expression:

$$\overline{E}_i = \bigvee_{j \varepsilon J} r_{ij} \wedge \overline{s}_j$$

The matrix is an indication of which conditions pertain to which enabling predicates.

A method has now been described for implementing the WHEN $R_i(\xi)$ aspect of transition systems by generating the E vector from the logical dot product of the R matrix and the S vector

Data Transformation Activation

What remains to be shown is how to implement the DO $\xi' = \psi_i(\xi)$ component. The data transformation aspect can be supported by the following more conventional computer structures:

1. All data items in ξ that are modified by the function ψ_i in generating ξ' can be specified by a set of "write" pointers to the data items.

2. All data items in ξ that are read as inputs during the data transformation ψ_i can be specified by a set of "read" pointers.

3. The starting address of the program code representing the function ψ_i can be specified by an "execute" pointer.

By providing these constructs to a conventional processing unit, the data transformation $\psi_i(\xi)$ can be implemented as the execution of a sequential program.

System Status Update

It should be noted here that on completion of the transformation $\xi' = \psi_i(\xi)$ there is an implied simultaneous update to the system status vector S (ξ). This is required to incorporate the implications to the data base into the status indicators in S (ξ). We therefore define a function G_i as follows:

$$S' = G_i\ (S(\xi),\ \xi')$$

To further define this function, $G_i\ (S,\xi')$ it is noted that for any data transformation $\psi_i(\xi)$, the function $G_i\ (S,\ \xi')$ has one of four possible effects on each element s_j of the status vector:

1. s_j is set true always

2. s_j is set false always

3. s_j remains unchanged always

4. s_j is determined dynamically during the data transformation.

By defining the following two control vectors, T_i and F_i and a variable update vector V_i in the same space as S, $G_i\ (S,\xi')$ can be reduced to a simple logical expression.

$T_i = (t_1, t_2, t_3 \ldots t_j \ldots t_J)$ where t_j is true if and only if the indicator is to be set true or left unchanged.

$F_i = (f_1, f_2, f_3 \ldots f_j \ldots f_J)$ where f_j is true if and only if the indicator is to be set false or left unchanged.

$V_i(\xi') = (v_1, v_2, v_3 \ldots v_j \ldots v_J)$ where v_j is set appropriately (true or false) for the indicators s_j that are determined dynamically by ψ_i, and v_j are "don't care" for s_j not determined dynamically.

With the above sense assignments the update vector function $G_i\ (S,\xi')$ can be implemented as follows:

$$s_{jnew} = (t_j \wedge \overline{f}_j)\ V\ (\overline{f}_j \wedge v_j)\ V\ (t_j \wedge s_{jold})$$

Table II shows the truth table associated with this update function. There are many sense assignments to the update vectors which would have resulted in different update functions. This particular definition has the advantage of performing, in addition to the required update, a masking operation which prevents inappropriate variable condition updates from being introduced through the vector, $V(\xi')$.

T_j	F_j	S_j NEW	IMPLIED MODIFICATION TO S_j
0	0	V_j	SET VARIABLY TRUE OR FALSE
0	1	0	SET FALSE
1	0	1	SET TRUE
1	1	S_j OLD	UNCHANGED

TABLE II: STATUS UPDATE SENSE ASSIGNMENTS

Machine Implementation

To implement the control component shown in Figure 2 as a device, hereafter referred to as the System Controller, a set of data constructs has been defined on which logic operations can be performed to implement efficiently the required control functions. These constructs include the following:

S = A single global system status vector register which contains one binary status indicator for each global condition that is relevant to the eligibility of at least one subsystem.

R = A relevance vector for each subsystem which contains one binary status indicator for each global data base condition in the system, indicating whether or not the condition is required to enable the associated subsystem. (The set of global data base conditions is defined as the union of all conditions required to enable the individual subsystems which comprise the system.) These vectors are arranged as rows in a matrix which is maintained in a memory module.

E = An eligibility vector register which contains one binary status indicator for each subsystem, representing the eligibility or non-eligibility of the associated subsystem.

Figure 3 shows the dimensional relationships of these constructs. The R vectors are arranged in a matrix form as shown and the E vector can then be generated by forming the logical "dot" product of the R matrix and the S vector:

$$E = R \cdot S$$

It is this basic matrix operation that the System Controller performs to determine the eligibility of subsystems. The design details of this and other operations are described in detail elsewhere.[1]

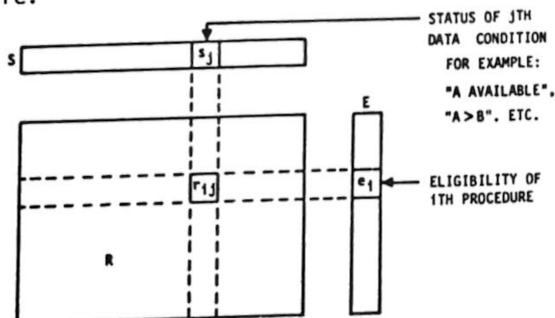

FIGURE 3: DIMENSIONAL RELATIONSHIPS OF S, E, AND R

To incorporate the update to the S vector register on completion of a subsystem, the following additional constructs are defined:

T = A "true" vector for each subsystem which contains one binary status indicator for each global condition in the system, indicating the global conditions that are to be set true or left unchanged on completion of the subsystem. These vectors are arranged as rows in a matrix which is maintained in a memory module.

F = A "false" vector for each subsystem which contains one binary status indicator for each global condition in the system, indicating whether the global conditions are to be set false or left unchanged on completion of the subsystem. These vectors are also maintained as a matrix in a memory module.

V = A single variable (dynamically updated) vector register returned by the computation component (processor) on completion of a subsystem. This register contains one binary status indicator for each global data base condition indicating the updates determined dynamically during the execution of the subsystem. (Note that due to the sense of the definition of T, F, and V, the range of conditions that can be modified variably are limited, such that any unauthorized updates will be masked out by the T and F vector. This is as shown in Table II.)

For software reliability and typical security concerns of access permission, there are three additional constructs included in the System Controller. These are the "read", "write", and "execute" pointer arrays which allow/restrict the computation component to perform the functions $\xi' = \psi_i(\xi)$ as described previously.

The Transition Machine's general implementation of these constructs is shown in Figure 4, applicable to a single System Controller and computation component. The computation device is identified as a processor, this is in a more or less traditional sense. In the extreme, the processor capabilities will be greatly restricted, but this can be effected in a conventional microprogrammed implementation. The processor to memory interface is completely conventional.

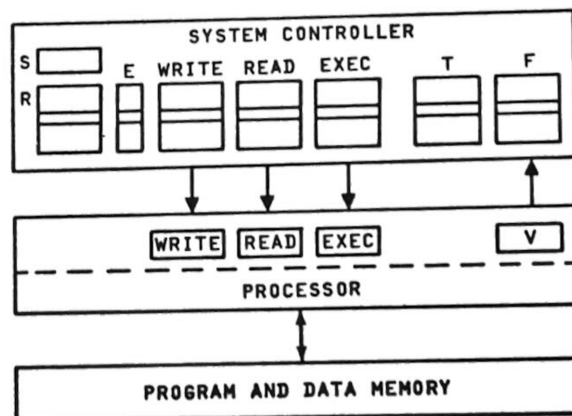

FIGURE 4: TRANSITION MACHINE ORGANIZATION

The general operation of these machines is shown in the flow diagrams of Figures 5 and 6. These figures respectively illustrate the processor and System Controller interface logic implemented in microcode.

Special Design Considerations

The previously defined control and data constructs that are maintained in the System Controller will expand in size with the software system implemented. To maintain flexibility, the System Controller must therefore be designed to support a range of system sizes. This can be accommodated in a number of ways, for example, the constructs can be stored in large blocks of conventional memory dedicated to the System Controller, or the System Controller can be designed from modular components which would allow an incremental expansion of the device. An alternate approach, one applicable to

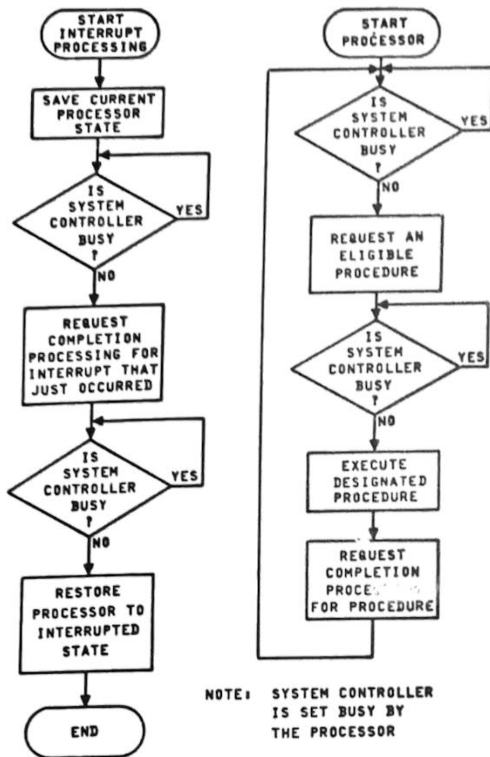

FIGURE 5: MICROPROGRAMMED PROCESSOR INTERFACE LOGIC

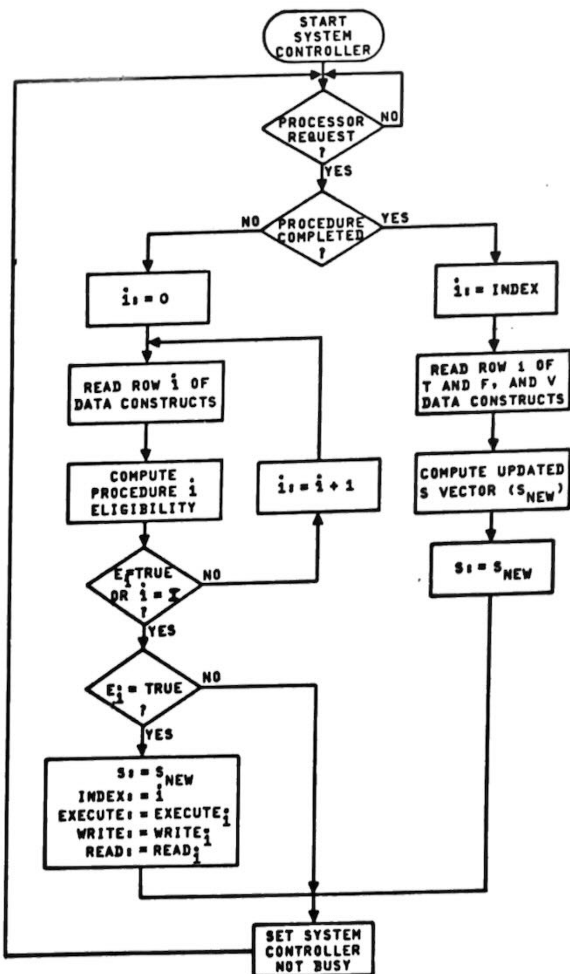

FIGURE 6: MICROPROGRAMMED SYSTEM CONTROLLER LOGIC

large systems, would be to implement an operating system which could dynamically overlay the constructs of a fixed size System Controller or multiple System Controllers. These and other approaches are currently under investigation. There seems to be no significant developmental problem to support the required flexibility.

Processor interrupts can be accommodated by having an interrupt subsystem identified in the System Controller. Only the status update aspects will be effectual; the associated relevance vector will preclude internal activation.[8] The microprogram associated with interrupt processing is shown in Figure 5.

The details of the operation of the machine are dependent upon the specific objectives of the implementation. Three specific objectives would be:

1. To implement a System Controller to effect the feasibility of exploiting the inherent software productivity advantages even in a conventional mainframe computer.

2. To implement highly efficient multiprocessor control. Two additional System Controller constructs are required and a semaphore implementation of the System Controller/processor interface. Detail design considerations for parallel systems are discussed in references 1, 8, and 9. Figure 7 illustrates a general parallel Transition Machine configuration.

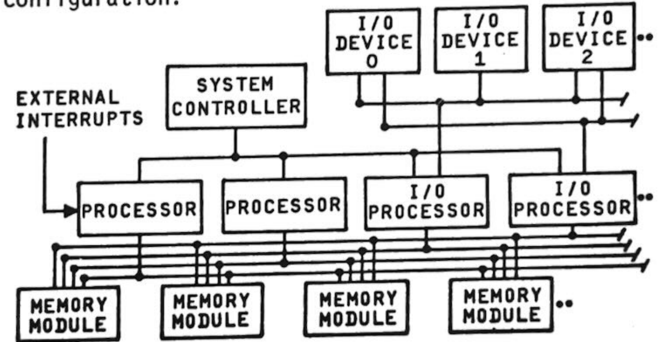

FIGURE 7: PARALLEL TRANSITION MACHINE

3. To implement multilevel secure ADP systems. Additional design considerations are involved in such implementations. Some of these considerations are treated in reference 10.

The advantages associated with the first implementation objective have already been discussed as a part of the derivation of the architecture. These advantages will apply also to the remaining two objectives which are the exploitation of inherent parallelism applying particularly to multiprocessor implementations, and verification advantages applying in particular to the certification requirements of secure systems.

Conclusions

Significant advances in hardware technology have greatly expanded the areas of application for

computers. Computer hardware has become faster, smaller and cheaper than ever thought possible. Software development costs on the other hand have been steadily increasing. Attempts have been made to reverse this trend with the development of structured software engineering techniques such as top down design, decision diagramming, and so on, but these attempts have been only partially successful. The cost of software development is still increasing.

To solve this problem a major re-evaluation of the computational requirements of computing systems has been needed. From such identified requirements a computation structure has been derived and from this computation structure a machine architecture identified which has been developed to implement the structure. This top down design approach has culminated in the Transition Machine architecture described above. The resulting architecture eliminates the difference in form of the program requirements and implementation. This characteristic facilitates software productivity and also the development of multilevel secure systems and efficient multiprocessors.

The results described here have indicated the tremendous potential of addressing software productivity by a unique computer architecture. This potential extends into the areas of ADP security and the coordination of many microprocessors in multiprocessing configurations. In conclusion, however, it is appropriate to address some of the problems yet to be solved. They can be summarized as follows:

1. However attractive from an analytical viewpoint, the software structure is nontraditional, and the disadvantages of this must not be underestimated. This problem could be ameliorated by the development of translators for accepted structured compilers, but some of the advantages of the structure would be lost by this approach. Ultimately a unique software development approach should be accepted.

2. A unique set of software development support tools will be required by the new structure. In addition to compilers, there is much potential for automated static program analysis.

3. Operating systems will be required which can page the control constructs in System Controller memories. This is basically the same problem as the current paging of task control block data, but the differences in form make this an area where extensive investigations must be performed.

The software productivity problem is one of the major problems whose persistence across the complete range of current software development practices is legend. The complete solution would seen to require not just a new software development methodology but a new computation structure supported by hardware. It is therefore incumbent that there be a breaking away from conventional computer architectures.

References

1. Anastas, M. S. and Vaughan, R. F., "Parallel Transition Machines", Proceedings of 1979 International Conference on Parallel Processing, IEEE, Michigan (August 1979)

2. Keller, R. M., "Formal Verification of Parallel Programs", Communications of the ACM 19, 7 (July 1976)

3. Kuck, D. J., "A Survey of Parallel Machine Organization and Programming", ACM Computing Surveys 9, 1 (March 1977)

4. Ledgard, H. F. and Marcotty, M., "A Genealogy of Control Structures", Communications of the ACM 18, 11 (November 1975)

5. Minsky, M., "Form and Content in Computer Science", ACM Turing Lecture, Journal of the ACM 17, 2 (February 1970)

6. Parnas, D. L., "A Technique for Software Module Specification with Examples", Communications of the ACM 15, 5 (May 1972)

7. Tanenbaum, A. S., "Implications of Structured Programming for Machine Architecture", Communications of the ACM 21, 3 (March 1978)

8. Vaughan, R. F. and Anastas, M. S., "Microprocessor Based Transition Machines", Proceedings of COMPCON '79 Fall, IEEE, W.D.C. (September 1979)

9. Vaughan, R. F. and Anastas, M. S., "Limiting Multiprocessor Performance Analysis", Proceedings of 1979 International Conference on Parallel Processing, IEEE, Michigan (August 1979)

10. Vaughan, R. F. and Anastas, M. S., "Preliminary Analyses to Obtain an Expanded Model and Preferred Implementation of Verifiably Secure ADP Systems", Boeing document D180-25090-1 (February 1979)