A MODEL OF PARALLEL COMPUTATION STRUCTURES

by

M. Yoeli
Dept. of Comp. Science
Technion
Haifa, Israel

J.A. Brzozowski
Dept. of Comp. Science
University of Waterloo
Waterloo, Ont., Canada

Research Report CS-76-43
October 1976

This work was done when the first author was visiting the University of Waterloo. This research was supported by the National Research Council of Canada under Grant No. A-1617.

A MODEL OF PARALLEL COMPUTATION STRUCTURES

by

M. Yoeli and J.A. Brzozowski

Abstract

We introduce a new mathematical model for asynchronous digital systems involving parallel processing. The model is based on the separation between device structure and control structure. A number of basic control modules are used as building blocks for control structures. We use the pulse signalling convention, and show how the control modules can be implemented. The model retains the ability of Petri nets to represent concurrent activities, but avoids several disadvantages of the Petri net approach to the modeling of hardware. We also introduce a mathematical framework in which the correctness of a computation structure may be verified. Finally, a number of open problems are stated.

1. Introduction

Considerable attention is presently being given to the design of asynchronous digital systems involving parallel processing. The synthesis, verification, and implementation of such systems is simplified if the system is conceived as consisting of two parts: a device structure (or "data flow structure") and control structure [BR-YO, CLA, HOW, PA-DE]. It is clearly advantageous to realize the control structure in modular form.

Petri nets have been proposed [MIS,PA-DE] as a mathematical model of asynchronous control structures involving parallelism. However, Petri nets have certain disadvantages: (1) The model unrealistically permits the accumulation of any number of tokens in a single place, implying unbounded memory. (2) In a Petri net representation, modularity is not easily recognizable. For instance, a module like an arbiter appears as a particular pattern of places and transitions. (3) Conventional Petri nets do not admit the concept of inhibiting the firing of a transition. This concept is required to represent, for example, a priority arbiter (see Section 4). This limitation led to the introduction of "inhibiting arcs" [FL-AG,AGE].

Another approach to modular asynchronous systems was introduced by Keller [KEL1]. This is based on sequential machine models, and lacks the advantages of Petri nets, where concurrent activities are more clearly represented.

In this paper we attempt to avoid the above-mentioned

disadvantages of Petri nets, but retain their basic ability of modeling parallelism. We first introduce several basic, easily implementable modules from which complex control structures may be built. We then propose a mathematical model for asynchronous parallel systems having control structures composed of these modules.

We assume that all devices operate asynchronously. Each device is given a GO signal [BR-YO] (or "ready signal" [PA-DE]) by the control structure to start its operation. Upon completion of its task, the device returns a DONE signal ("acknowledge signal").

Various signalling conventions have been discussed in the literature [KEL1, PA-DE]. We treat specifically the "pulse" signalling convention which is easily implemented and frequently used.

The model we introduce is suitable for design: the notation is quite similar to flowcharts of parallel programs; the task of verification can be precisely formulated (see Section 5); and modular implementation is direct and straightforward.

The report represents only a preliminary investigation and further work is needed. A number of open problems are listed in Section 6.

2. Introductory Examples

We begin by describing a simple problem and its implementation. Let A, B, and C be binary words of length $n \geq 1$. We denote by $\perp A$ the nonnegative integer whose base-2 representation is A. We write A+B = C to state that $\perp C$ is the sum modulo 2^n of $\perp A$ and $\perp B$. The letters A, B, etc. will be used to denote both n-bit registers and their contents.

For our purposes an (n-bit) adder is a device interconnected as shown in Fig. 1.

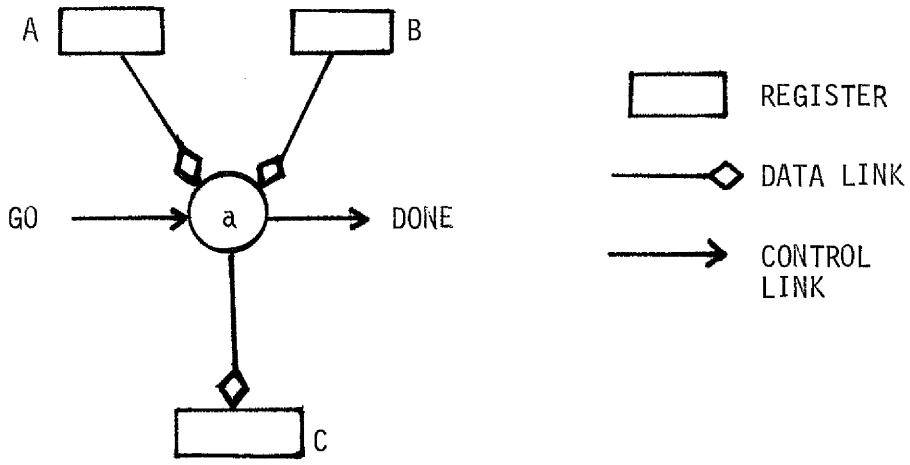
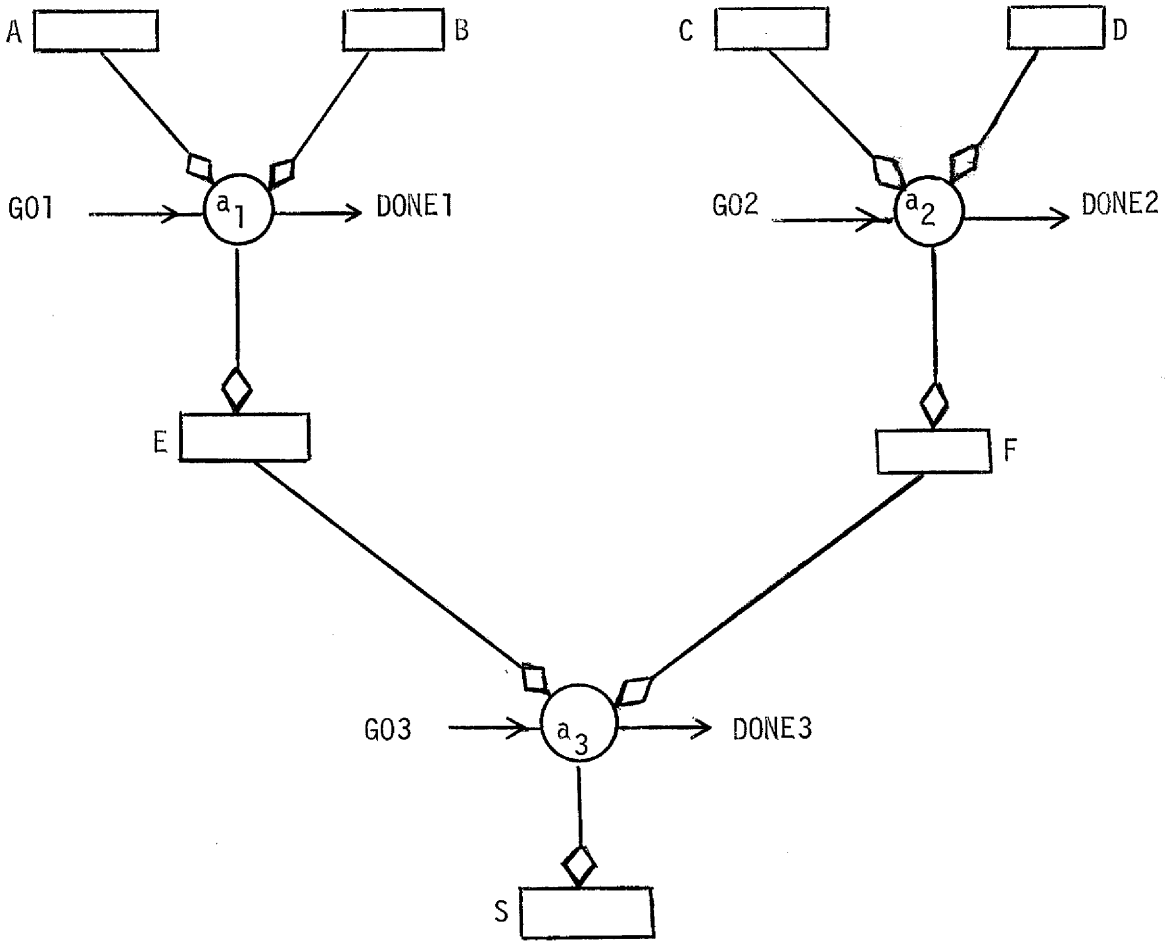


Fig. 1 Interconnections of an adder

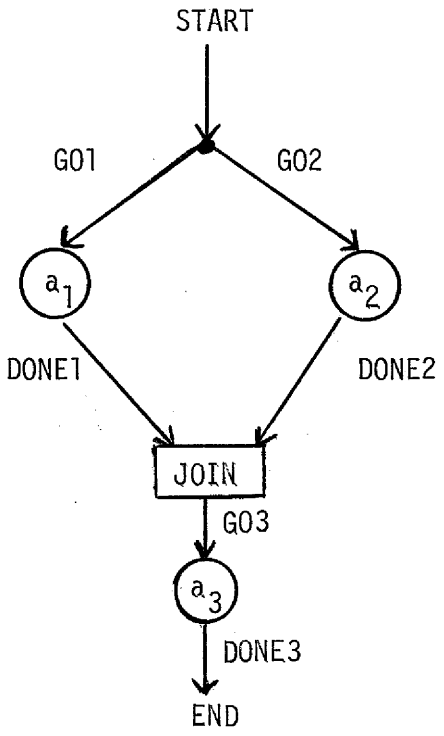
The adder may be in two states, namely OFF and ON. If it is in the OFF state and a pulse signal is received on its GO input, the adder will perform the operation C ← A+B (i.e. the contents of register C is set to A+B), and will issue a pulse signal on its DONE output. As long as the operation is in progress, the adder assumes the ON state. During

this period, the contents of registers A and B are not to be changed. If another GO pulse arrives, while the adder is in its ON state, this GO pulse is ignored. The actual GO command is the 0-1 transition of the GO pulse arriving while the adder is in its OFF state.

Such adders are easily designed. In general, we assume that all devices perform similarly, i.e. start on GO signals and produce DONE signals. If the time taken by an operation is fixed, the DONE signal may be generated by a timer.



(a)



(b)

Fig. 2 Computation structure to perform S ← A+B+C+D

(a) Device structure

(b) Control structure

Assume now that three adders are available to perform the computation S ← A+B+C+D. The expression A+B+C+D can then be evaluated as (A+B) + (C+D), where the additions A+B and C+D are performed concurrently. The corresponding set-up is shown in Fig. 2. We assume that the numbers to be added are already stored in registers A, B, C and D.

The JOIN in Fig. 2(b) is an asynchronous network whose state diagram is shown in Fig. 3(a).

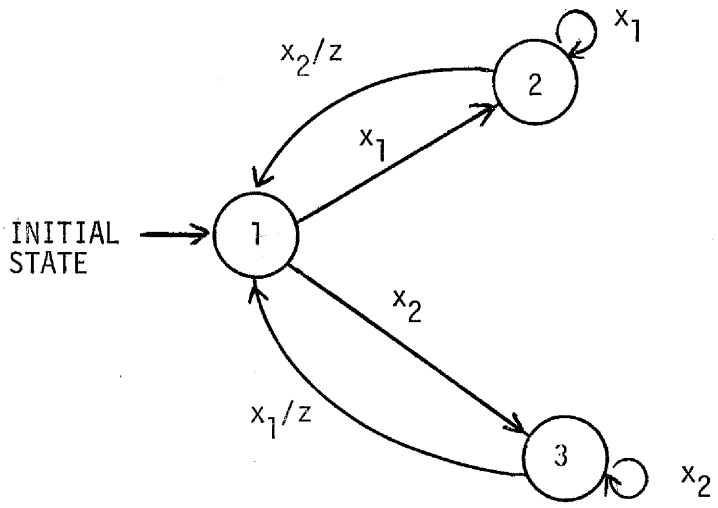


Fig. 3(a) State diagram of JOIN.

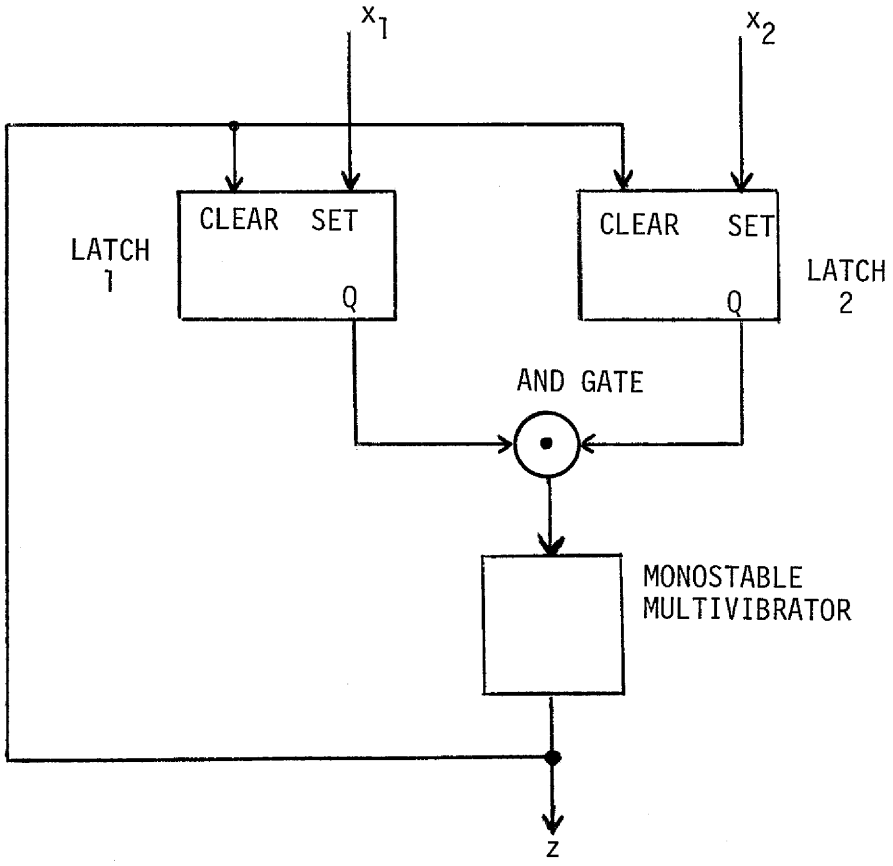


Fig. 3(b) Implementation of JOIN.

In this state diagram, x_1 and x_2 represent pulses on input lines 1 and 2, respectively, and z represents an output pulse. Furthermore, we specify that the simultaneous arrival of both x_1 and x_2 is to have the same effect as x_1 followed by x_2, or x_2 followed by x_1. A straightforward method of implementing a JOIN is shown in Fig. 3(b).

The control structure of Fig. 2 may be modeled by the control net shown in Fig. 4. The control net is a mathematically precise concept (see Section 3) that is useful for analysis, synthesis and verification of control structures. It is quite close to our concept of control structure, but is more independent of the actual implementation and signalling conventions. In this example, the control net is used as an aid to the analysis of the computation structure of Fig. 2.

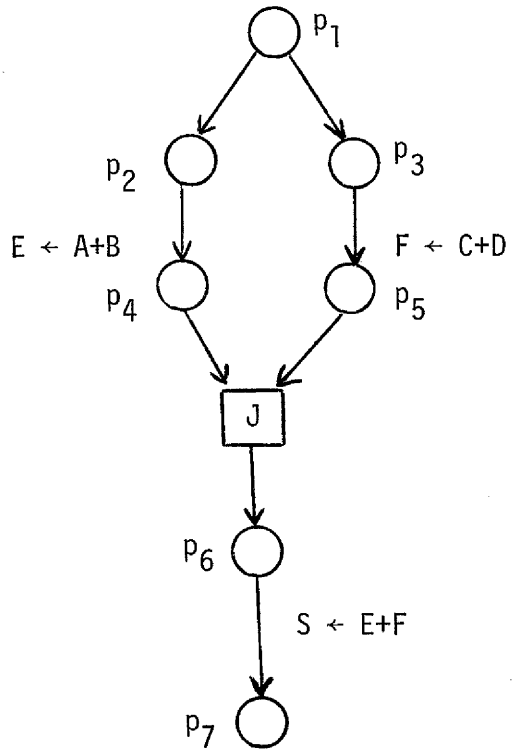


Fig. 4 Control net representing control structure of Fig. 2.

The circular nodes in Fig. 4 are called places. A place may be marked by putting a token inside the circle. In distinction from Petri nets a place may not contain more than one token. The square node labeled J represent the JOIN of Fig. 2(b). The various stages of the computation are modeled by corresponding token distributions. For example, the marking of place p_1 represent the appearance of a START pulse and the markings of places p_2, p_3, and p_6 represent the ON-states of adders a_1, a_2, and a_3 respectively.

Precise firing rules for control nets, defining all possible transitions between token distributions, will be set up later. Presently we illustrate the application of such firing rules in an informal way. A typical sequence of token distributions, together with their interpretation, is shown in the following table.

TABLE 1

p_1	A START pulse arrives
p_2, p_3	Adders a_1 and a_2 receive GO pulses and assume their ON states.
p_2, p_5	Adder a_2 has completed its operation and issued the DONE pulse. Consequently the JOIN is in state 3. Adder a_1 is still ON.
p_4, p_5	Adder a_1 has completed its operation and issued the DONE pulse. The JOIN is now ready to return to state 1.
p_6	The JOIN has issued an output pulse; thus, a_3 has received a GO pulse and is ON.
p_7	The computation is completed and an END pulse appears.

For our next example assume that the above computation S ← A+B+C+D is to be performed by means of two adders a_1 and a_2 only. We may still evaluate A+B and C+D concurrently, but we have to use, say, a_1 again to evaluate S. The corresponding device structure is shown in Fig. 5. The devices marked I transfer input to output (I denotes the identity operator).

Clearly the device structure must be the first step of the design. Next we can construct a corresponding control net. This is similar to the use of flow charts in programming.

Two control nets corresponding to Fig. 5 are shown in Fig. 6. In both nets the adder a_1 is used twice to perform the operation S ← A+B. This is indicated in Fig. 6 by labeling the expression S ← A+B accordingly.

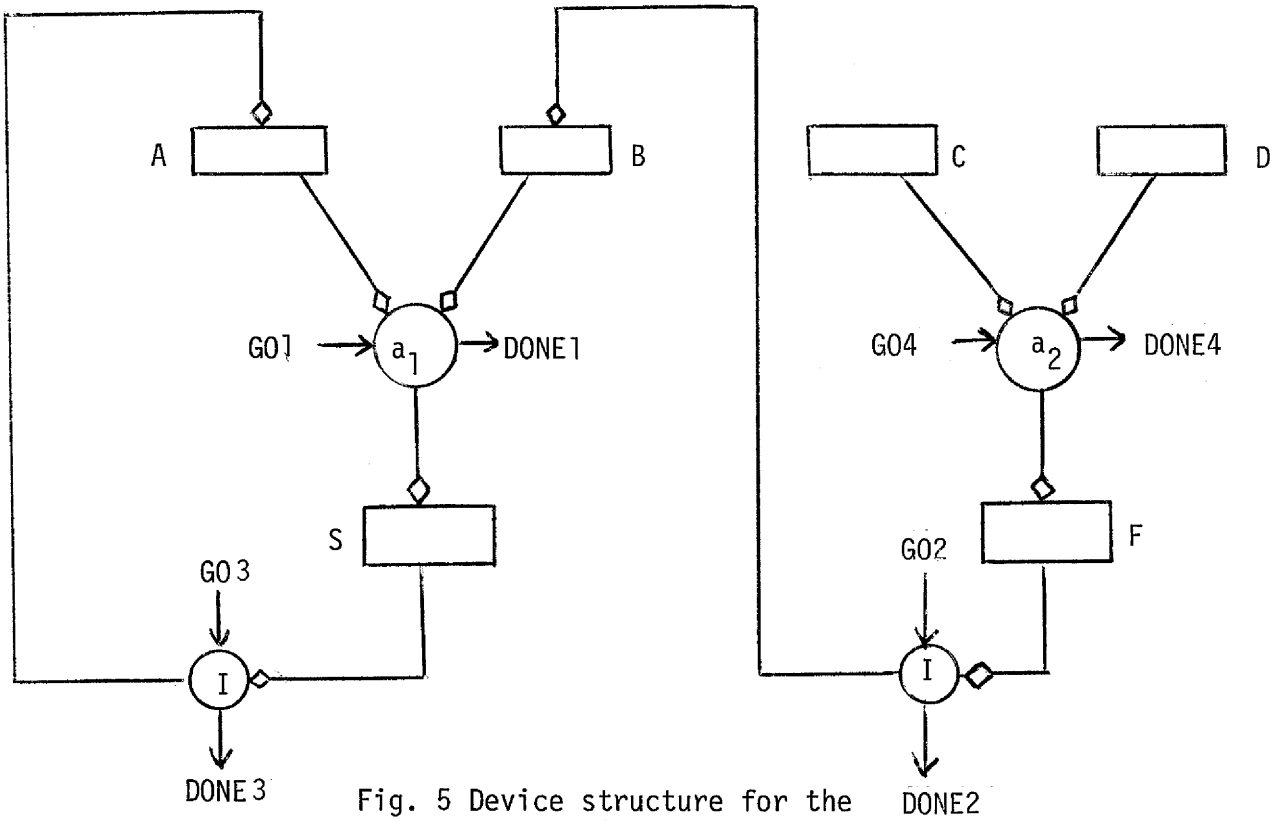


Fig. 5 Device structure for the computation S ← A+B+C+D, using two adders.

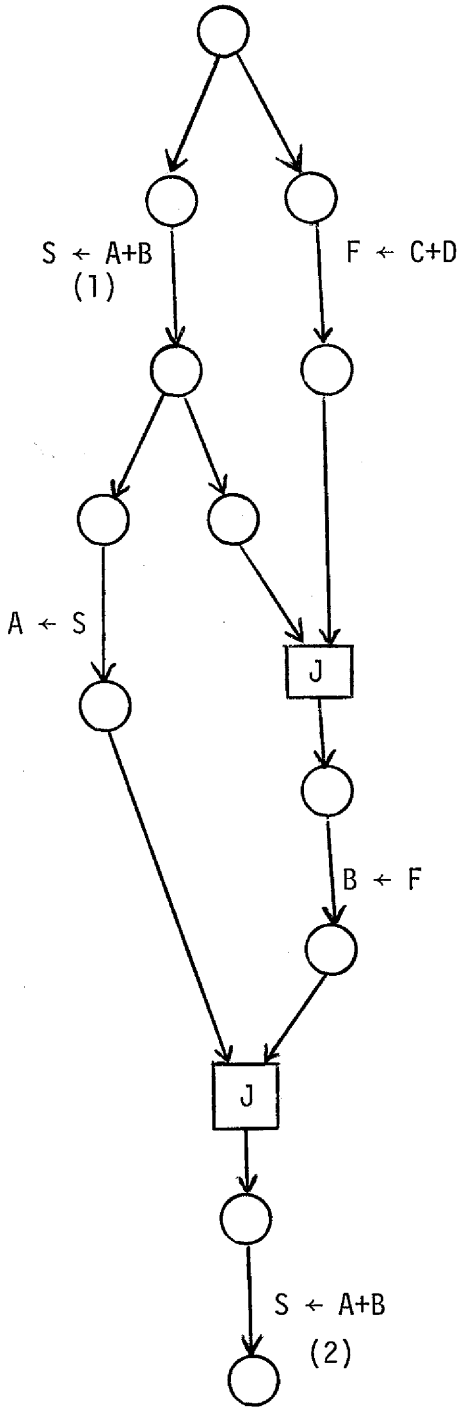


Fig. 6(a)

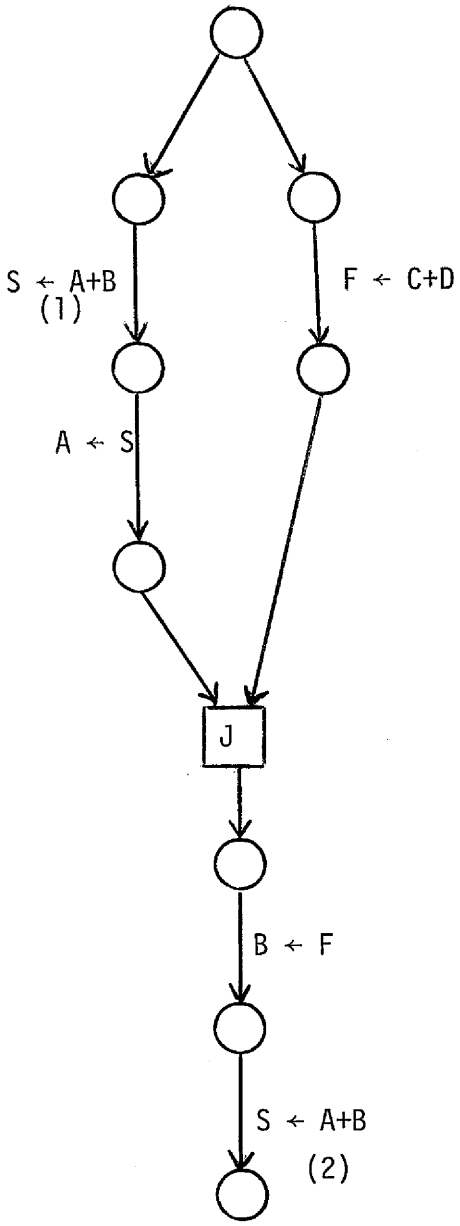


Fig. 6(b)

Fig. 6 Two versions of control net corresponding to Fig. 5.

In general, if an adder (or some other operational device) is to be used twice, an interface device called a call module [KEL1,ST-OR-CL] is required. The external connections of such a module as well as its state diagram are shown in Fig. 7, where we assume that a GOi pulse may appear only if the device is in its initial state (1), and GO1 and GO2 cannot appear simultaneously. The concept of the "two-port" call module, shown in Fig. 7, is easily extended to "multi-port" call modules, enabling the multiple use of any operational device. We choose to associate the call module with the device structure, rather than the control structure. Alternatively, the concept of control net could be extended to include call modules.

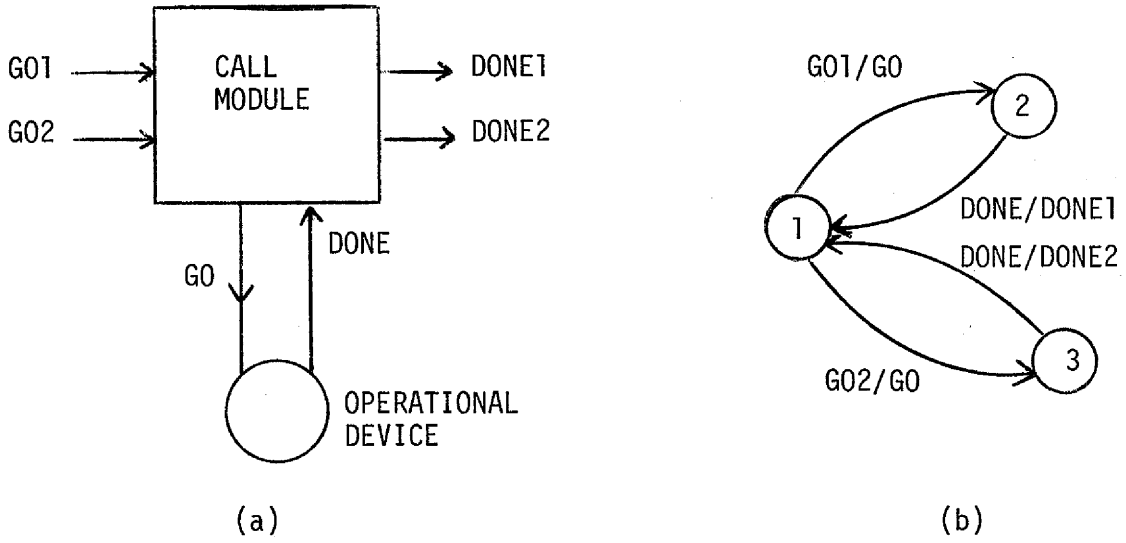


Fig. 7 Call Module (a) External connections
(b) State diagram

The control structures corresponding to the control nets shown above are easily derived. We leave this to the reader.

For our third example let * denote some associative operation on the set of n-bit words and assume that we wish to perform the computation

$$R \leftarrow \underbrace{A*A*\ldots*A}_{\text{k appearances of A}} .$$

where $k \geq 2$. The corresponding device structure is shown in Fig. 8 (the control connections are omitted). The device indicated by * performs the operation D ← B*C. The initial value of (the counter)K

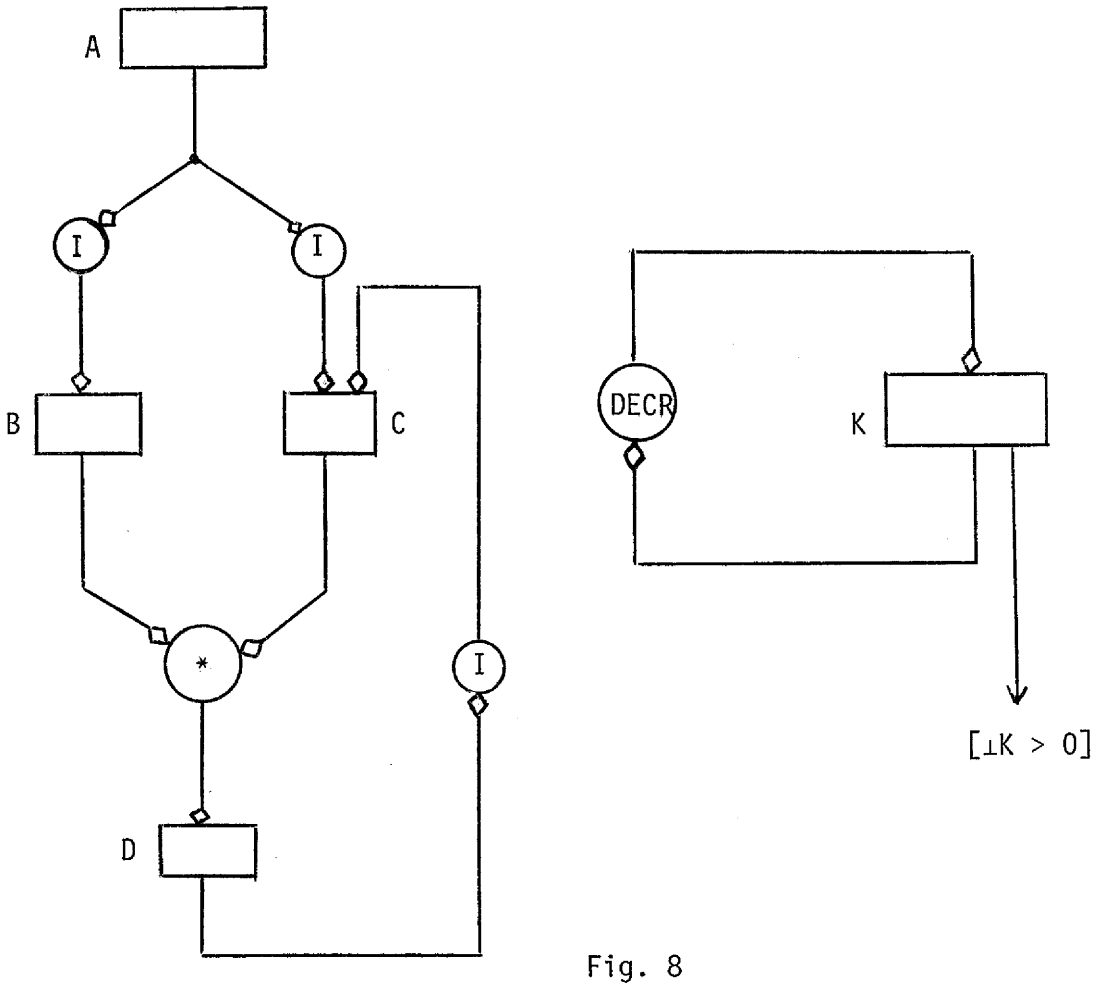


Fig. 8

is $\perp K = k-1$. The function DECR is defined as follows:

$$\underline{\text{if}}\ \perp K > 0\ \ \underline{\text{then}}\ \perp(\text{DECR } K) = (\perp K) - 1,$$

$$\underline{\text{else}}\ \text{DECR } K = K\ .$$

A control net for the present example is shown in Fig. 9. The square node labeled U is a UNION. Its implementation is simply an OR gate. The square node labeled D is a DECIDER. A decider may be implemented by a decoder, having a level input c and a pulse input x. Its two outputs are combinational, namely $T=c\cdot x$ and $F=c'\cdot x$, where $\cdot$ denotes AND. The control structure is now easily derived from the control net of Fig. 9.

In our first example we started with a computation structure (Fig. 2) composed of a device structure and a specific control structure using pulses for signalling. The corresponding control net (Fig. 4) is a mathematical abstraction of the control structure. Another level of abstraction involves the concept of a control schema. The control schema corresponding to Fig. 4 is shown in Fig. 10, where σ_1, σ_2, σ_3 are symbols representing aribtrary operations. Figure 4 becomes a particular "interpretation" of Fig. 10. Thus control schemata correspond to program schemata. Frequently, one can derive many features of a control structure from its corresponding schema. In the next section we give a precise definition of control schemata.

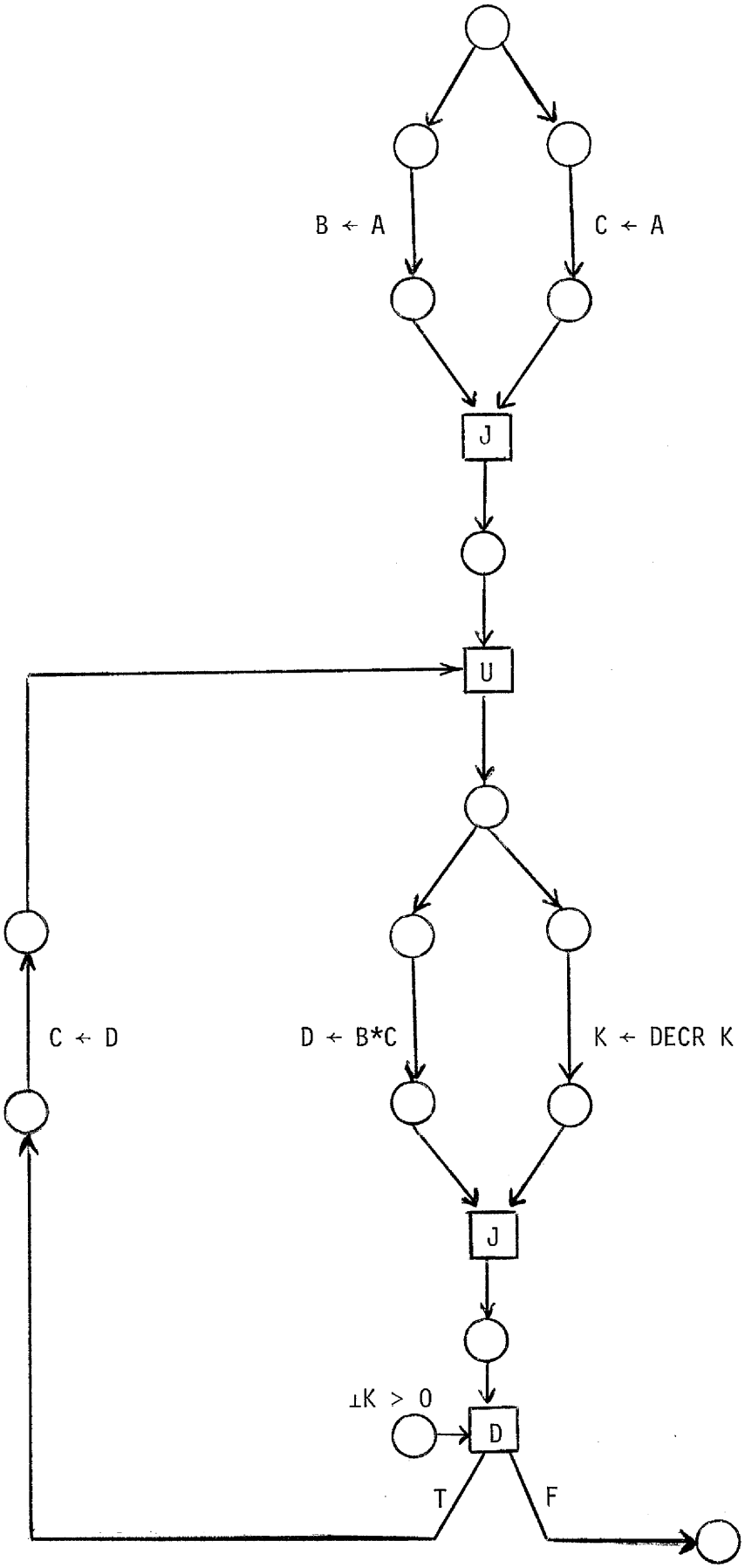


Fig. 9

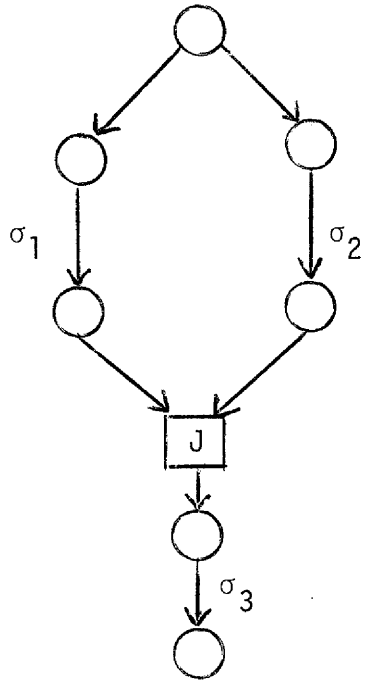


Fig. 10

3. Simple Control Schemata and Nets

In this section we develop a mathematical formulation of the concepts of control schemata and control nets. We begin with a class of schemata and nets that we call simple. All the examples of Section 2 belong to this class. In Section 4 these concepts are generalized.

A simple control schema (SCS) is a finite, directed, labeled graph satisfying the following conditions:

1) The nodes are partitioned into places (shown as circles) and "square" nodes. Square nodes are of three types, namely: a) UNIONS b) CHOICES and c) JOINS. The type of a square node is indicated by means of labels U, C and J, respectively.

2) The indegree of a place is either 0 or 1.

3) Square nodes are connected to places only.

4) UNIONS and JOINS have indegree ≥ 2 and outdegree 1.

5) CHOICES have indegree 1 and outdegree ≥ 2.

6) If there exists an edge from node p to node q, then p is an input node of q, and q an output node of p. If a place p has outdegree > 1, then all its output nodes are places. Such a place p is called a fork.

7) If p is a place with outdegree 1, q a place with indegree 1, and e is an edge from p to q, then e is an operation edge. Let Σ be a finite alphabet. Every operation edge is labeled by a letter of Σ.

An example of an SCS has been given in Fig. 10. The motivation for having FORK, JOIN, and UNION nodes should be clear from Section 2. The new node type, CHOICE, represents either (a) a module that randomly selects one of its outputs, e.g. for use in arbitration

networks or (b) an abstraction of the DECIDER node. We will return to DECIDERS later in this section.

One could implement a two-input CHOICE node by exploiting the critical race in a latch [BR-YO]. However, this may not be an acceptable solution if the "glitch phenomenon" is present [CH-MO]. Therefore this problem requires further study.

Let S be an SCS and P the set of its places. A state or marking of S is any subset Q of P. The marking Q of S is indicated by placing tokens in all places belonging to Q. With any SCS S we associate a transition relation ($\rightarrow$) on its set of states. For $m,m' \subseteq P$, we have $m \rightarrow m'$ iff m' is obtained from m by the application of one of the following firing rules: Note: By "marking" a place p we mean putting a token on p, if it had no token previously, and doing nothing otherwise.

F1) If e is an operation edge from place p to place q, and p is marked (i.e. $p \in m$), remove the token from p and mark q. The new marking is therefore: $m' = (m-\{p\}) \cup \{q\}$.

F2) If the place p is a fork and it is marked, remove the token from p and mark all output nodes of p.

F3) If U is a UNION, and at least one input place of U is marked, remove the tokens from all input places of U and mark the output place of U.

F4) If C is a CHOICE, and its input place is marked, remove the token from its input place and mark any one of its output places.

F5) If J is a JOIN, and all its input places are marked, remove the tokens from all its input places and mark its output place.

In the firing rules introduced above the presence of a token in a place represents either the presence of a pulse or the "set" state of a latch. For example, tokens at the input places of a UNION represent pulses. On the other hand, a token at an input place of a JOIN represents the set state of a latch. (See Fig. 3(b).) The reader should verify that the firing rules are consistent with the implementation of the various modules and operational devices. For instance F1) corresponds to the following. If an operational device is in its ON state, it will eventually complete its task. It then issues a DONE signal and returns to its OFF state.

Before defining control nets we introduce formally an additional type of square node, namely DECIDER shown as a square labeled D. The indegree of a DECIDER is two, and its outdegree is two. Its two input nodes are places, one of them distinguished as its condition place and the other as its signal place. The condition place has indegree 0. The two output nodes are places, one labeled T and the other F.

The firing rule for a DECIDER is:

F6) If D is a DECIDER, and its signal place is marked, remove the token from the signal place. Mark the output place labeled T, if the condition place is marked; otherwise mark the output labeled F.

In an "uninterpreted" schema, the condition place of a

DECIDER is omitted; what remains is equivalent to a CHOICE node, as mentioned before.

A simple control net (SCN) N consists of

1) A finite directed graph whose nodes are places, UNIONS, CHOICES, JOINS and DECIDERS.

2) Firing rules F1) through F6).

3) A domain $M \subseteq \{0,1\}^r$, where r is a fixed positive integer. Intuitively, r is the total number of binary register cells appearing in the device structure.

4) A set G of unary operations on M (i.e. $g:M \to M$, for every $g \in G$), together with a mapping $e \mapsto g_e$ of the operation edges of N into G. Thus g_e represents the transformation of the domain caused by operation corresponding to e.

5) A set H of predicates on M together with a mapping $p \to h_p$ of all condition places of N into H. This implies that a condition place p is marked iff $h_p(d)$ is true, where $d \in M$ is the present state of the domain.

To illustrate these concepts, refer to Fig. 2. The domain M consists of registers A through F and S, all of length n. Thus $r = 7n$. The mapping associated with a_1 performs $E \leftarrow A+B$ and leaves registers other than E unchanged. See Fig. 9 for an illustration of predicate mapping.

Note that 4) above admits the identity operator as a valid operator. We can view this identity operation as a delay.

4. General Control Schemata and Nets

We now introduce general control schemata and nets which extend the concepts of simple control schemata and nets introduced earlier. These generalizations will enable us to properly model more complex systems, e.g. computation structures which contain priority arbiters.

General control schemata are defined similarly to simple control schemata, except that a new type of square node, namely TRANSFORM is introduced. As will become evident, a JOIN node is a special case of a TRANSFORM node. A TRANSFORM node will be labeled T. All the input and output nodes of a TRANSFORM are places. A TRANSFORM has $k \geq 1$ input places, numbered 1 to k, and $n \geq 1$ output places. Each output place is labeled by a ternary k-tuple $t \in \{0, \frac{1}{2}, 1\}^k$, where at least one component of t is 1. We say that the binary k-tuple $b \in \{0,1\}^k$ matches the ternary k-tuple $t \in \{0, \frac{1}{2}, 1\}^k$ iff b can be obtained from t by changing the $\frac{1}{2}$ - entries of t only. The following firing rule applies to a TRANSFORM:

F7) Let b be the binary k-tuple which represents the marking of the input places. Mark each output place whose ternary label is matched by b. Let the i-th input place be marked. Remove its token iff there exists an output place with ternary label t, such that t is matched by b, and $t_i=1$.

A simple example of a TRANSFORM T is the case k=2, n=1, and t=11, where t is the ternary output label. Clearly, T will perform in the same way as a 2-input JOIN. Similarly, the case k > 2, n=1, t=11..1 corresponds to a k-input JOIN. Thus a JOIN is a special case of a TRANSFORM.

Another example of a TRANSFORM is shown in Fig. 11.

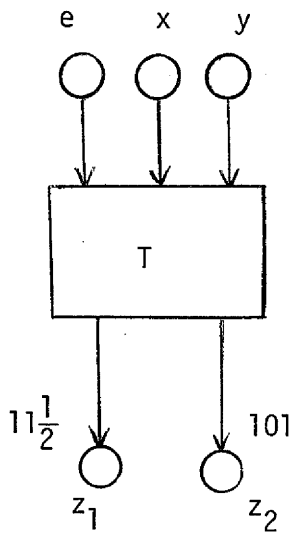


Fig. 11 Example of TRANSFORM

This TRANSFORM performs priority arbitration on the inputs x and y. The place e is the enabling input. If e is not marked, no output will occur. If e and x are both marked, the TRANSFORM will mark z_1, independently of the state of y, and remove the tokens from e and x. The state of y is not affected. The marking of z_1 may be interpreted as "x is to be served". If, however, e and y are marked, but x is not, then z_2 will become marked (indicating that "y is to be served") and the tokens of e and y will be removed.

The method used in Fig. 3(b) to implement the JOIN may be extended to realize other TRANSFORM nodes. However, such realizations may involve timing problems. It is still an open question whether all such problems may be eliminated by the insertion of suitable delay elements. This problem is of particular interest in connection with

the proper design of priority arbiters, such as the module specified by Fig. 11.

Although a variety of arbitration network designs have been proposed in the literature [MIS,O-K-N-S,PLU,SE-JU], most of these papers do not give precise specifications of these networks nor do they provide proofs of their correctness. Furthermore, in most cases the signalling convention differs from the one used in this paper.

5. Behavior of Control Nets

In this section we make precise the notions of computation performed by a computation structure, and of verification of the design. Furthermore, we define various properties of control nets that give insight to its behavior. We return to these properties in Section 6. These concepts are quite similar to those appearing in parallel program theory [ASH,KEL2].

Let N be a control net, P the set of all its places and C the set of its condition places. A total state of N is an ordered pair (t,d), where $t \subseteq P-C$ is the token state and $d \in M$ is the data state. Note that the marking $c_d \subseteq C$ of condition places is determined by d as follows:

$$c_d = \{p \in C | h_p(d) \text{ holds}\}.$$

We now introduce the net transition relation $\rightarrow$ on the set of total states of N. If t' is obtained from t by applying one of the firing rules F2) - F7), the data-state is not affected and we have

$$(t,d) \rightarrow (t',d) .$$

Assume now that t' is obtained from t by applying firing rule F1), and that e is the corresponding operation edge. Then

$$(t,d) \rightarrow (t',d'),$$

where $d' = g_e(d)$.

Let N be a control net, having a single START place p_S and a single HALT place p_H. By a computation sequence of N we mean

a finite sequence α of total states,

$$q_0 = (t_0,d_0),\ldots,q_k = (t_k,d_k)$$

such that $q_i \rightarrow q_{i+1}$ $(0 \leq i < k)$, $t_0 = \{p_s\}$, $p_H \in t_k$ and $p_H \notin t_i$ for $i < k$. The data state d_0 is the initial value for α, and d_k is the outcome.

We call a sequence α residue-free (cf[HE-YO]) iff $t_k = \{p_H\}$. N is residue-free, iff every computation sequence of N is residue-free.

By a deadlock sequence of N we mean a finite sequence α of total states,

$$q_0 = (t_0,d_0),\ldots,q_k = (t_k,d_k)$$

such that $q_i \rightarrow q_{i+1}$ $(0 \leq i < k)$, $t_0 = \{p_s\}$, $p_H \notin t_i$ for $0 \leq i \leq k$, and no total state q exists such that $q_k \rightarrow q$. The control net N is deadlock-free, iff no deadlock sequence of N exists.

Let now $q = (t,d)$ and $q' = (t',d')$ be total states of N, such that $q \rightarrow q'$. If $p \in t-t'$, then the place p is unstable in the total state q. A non-terminating sequence of N is an infinite sequence α of total states,

$$q_0 = (t_0,d_0),\ldots,q_i = (t_i,d_i),\ldots$$

such that $q_i \rightarrow q_{i+1}$ $(i \geq 0)$, $t_0 = \{p_s\}$, $p_H \notin t_i$ $(i \geq 0)$, and furthermore α has the following finite-delay property [KA-MI,KEL2]:

If p is unstable in state q_i, $(i \geq 0)$, then p is stable in some state q_j, where $j > i$. It is reasonable to assume that the finite-delay property holds in actual computation structures.

The control net N is properly terminating, iff N is deadlock-free

and no non-terminating sequences of N exist.

Usually only a part M' of the domain M is of interest after a computation has been completed. Two data states of M that agree on M' can then be considered equivalent. In general, let E be an equivalence relation on the domain M of N. The net N is E-determinate (cf. [KEL2]) with respect to $d_0 \in M$ iff the outcomes of all computation sequences with initial value d_0 are E-equivalent. The net N is E-determinate iff N is E-determinate with respect to every $d_0 \in M$.

The computation γ to be performed by a control net N may be viewed as follows. We are given two equivalence relations E_1 and E_2 on M, and a function

$$\gamma:(M/E_1) \rightarrow (M/E_2),$$

where M/E is the set of equivalence classes of M under E. The net N performs the computation γ iff every computation sequence α of N with initial value d_0 and outcome d_k satisfies the condition

$$\gamma(E_1[d_0]) = E_2[d_k],$$

where $E_1[d_0]$ denotes the E_1-equivalence class of d_0. By verification of N we mean a formal proof that N indeed performs the computation γ. The verification of a control net is, of course, closely related to the verification (proof of correctness) of parallel programs. The model proposed in this paper provides a suitable framework for the verification of control nets, which ensures the correctness of the corresponding parallel computation structure.

6. Open Problems

As we have mentioned before, this work is incomplete. In this section we list several open problems.

A) Hardware Implementation of TRANSFORMS

We had suggested that the implementation of Fig. 3(b) for the JOIN can be extended to the design of TRANSFORMS. However, this straightforward extension has a critical race under certain multiple-input changes. It appears that such races can be eliminated by inserting delays in appropriate lines. However, formal methods of verification have yet to be developed.

B) Verification of Control Structures

Any control structure composed of the basic modules described here can be viewed as a large asynchronous network, with the various DONE and GO signals as inputs and outputs, respectively. The inputs to this network are not arbitrary, but are contrained by the outputs. Assuming that the basic modules are well designed, what conditions must be placed on the control schema in order to guarantee proper behavior of the overall network? (A similar problem is discussed in [KEL1].)

C) Proof of Correctness of Computation Structures

We have already formulated in Section 5 the problem of verifying whether a given computation structure performs the desired computation. Efficient techniques of proving correctness need to be developed. For some related problems see [ASH] ane [KEL2].

D) Relationship between Schemata and Nets

Certain properties of a control net can be derived by analyzing

the corresponding schema. Further insight is needed in this connection.

E) A Hierarchy of Control Structures

Consider the "hierarchy" shown in Fig. 12, where D,U,F,J,T, and C represent DECIDER, UNION, FORK, JOIN, general TRANSFORM and CHOICE, respectively. Intuitively, the control structures become more "powerful" as various modules are added, as indicated in Fig. 12. This concept of "power" needs to be made precise. For related topics see [PE-BR].

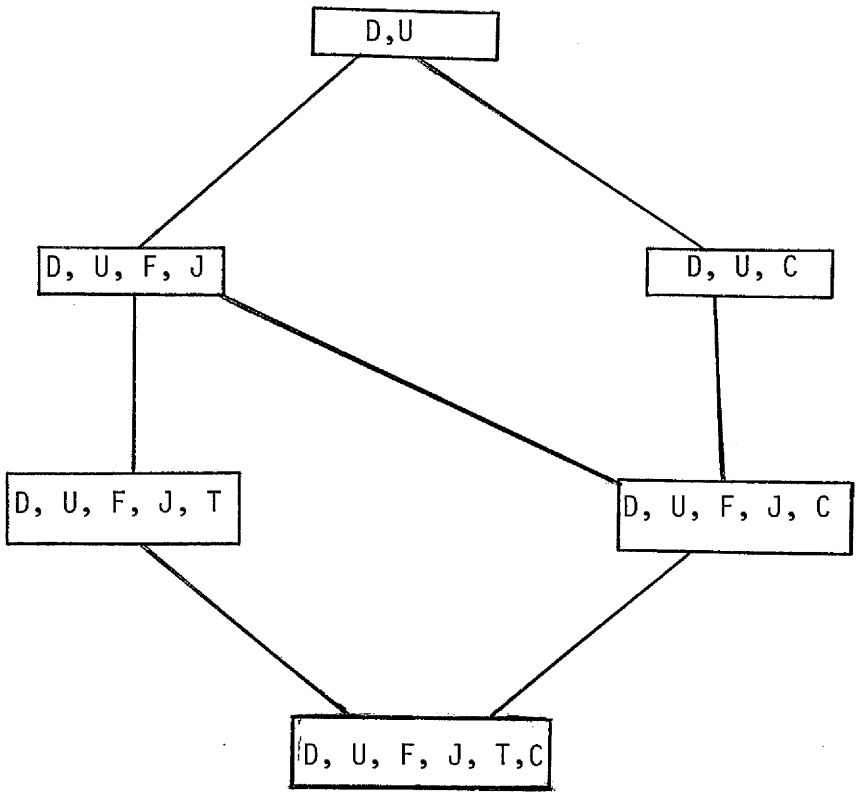


Fig. 12 Control structure hierarchy.

References

[AGE] Agerwala, T. "A Complete Model for Representing the Coordination of Asynchronous Processes". Hopkins Computer Research Report #32, Computer Science Program, Johns Hopkins University, Baltimore, Maryland, July 1974.

[ASH] Ashcroft, E.A., "Proving Assertions about Parallel Programs". J. Computer and System Sciences, Vol.10, 1975, pp.110-135.

[BR-YO] Brzozowski, J.A., and M. Yoeli, Digital Networks. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1976.

[CH-MO] Chaney, T.J., and C.E. Molnar, "Anomalous Behavior of Synchronizer and Arbiter Circuits". IEEE Trans. Computers, Vol.C-22, 1973, pp.421-422.

[CLA] Clark, W.A., "Macromodular Computer Systems". 1967 Spring Comput. Conf., AFIPS Conf. Proc., Vol.30, 1967. Washington, D.C.: Thompson, pp.335-336.

[FL-AG] Flynn, M. and T. Agerwala, "Comments on Capabilities, Limitations and 'Correctness' of Petri Nets". Computer Architecture News, Vol.2, Dec. 1973.

[HE-YO] Herzog, O. and M. Yoeli, "Control Nets for Asynchronous Systems, Part I". Technical Report #74, Dept. of Computer Science, TECHNION, Haifa, 1976.

[HOW] Howard, B.V., "Parallel Computation Schemata and their Hardware Implementation". Digital Processes, Vol.1, 1975, pp.183-206.

[KA-MI] Karp, R.M., and R.E. Miller, "Parallel Program Schemata". J. Computer and System Sciences, Vol.3, 1969, pp.147-195.

[KEL1] Keller, R.M., "Towards a Theory of Universal Speed-Independent Modules". IEEE Trans. Computers, Vol.C-23, 1974, pp.21-33.

[KEL2] Keller, R.M., "Formal Verification of Parallel Programs". Communications ACM, Vol.19, 1976, pp.371-384.

[MIS] Misunas, D., "Petri Nets and Speed Independent Design". Communications ACM, Vol.16, 1973, pp.474-479.

[O-K-N-S] Ohmori, K., N. Koike, K. Nezu and S. Suzuki, "MICS-A Multi-Microprocessor System". Information Processing 1974, Proc. IFIP Congress 1974, J.L. Rosenfeld, ed., Amsterdam: North-Holland, pp.98-102.

[PA-DE] Patil, S.S., and J.B. Dennis, "The Description and Realization of Digital Systems". Revue Française d'Automatique, Informatique et de Recherche Opérationelle, Feb. 1973, pp.55-69.

[PLU] Plummer, W.W., "Asynchronous Arbiters". IEEE Trans. Computers, Vol.C-21, 1972, pp.37-42.

[SE-JU] Sechovsky, H. and S. Jura, "Asynchronous Speed-Independent Arbiter in a Form of a Hardware Control Module". 1976 National Computer Conf., AFIPS Conf. Proc., Montvale, N.J.: AFIPS Press, pp.777-782.

[ST-OR-CL] Stocki, M.J., S.M. Ornstein, and W.A. Clark, "Logical Design of Macromodules". 1967 Spring Joint Comput. Conf., AFIPS Conf. Proc., Vol.30, 1967. Washington, D.C.: Thompson, pp.357-364.