The Louisiana Computer Exposition on Distributed Systems Based
on Mini and Micro Computers, University of Southwestern Louisiana.
March, 1979. pp. 143-166.

A DISTRIBUTED OPERATING SYSTEM MACHINE

TREVOR MUDGE
University of Michigan

ABSTRACT

A design methodology suitable for the design of operating system machines is presented, which yields highly parallel systems without the designer having to worry about the details of coordination and synchronization. It is based on a basic building block called the processing element, and an interconnection discipline for developing networks of processing elements. The processing elements and the interconnection discipline are defined, and some illustrations of how to use them to construct a target system are included. Finally, deadlock and determinacy of such target systems is discussed.

A DISTRIBUTED OPERATING SYSTEM MACHINE*

1. INTRODUCTION

One of the major aims of designers of computing machinery is to develop techniques for building computer systems that combine greater throughput with lower cost. The increase in speed and complexity together with the decrease in cost of computer hardware components, typified by the present state-of-the-art microprocessor, suggests that this aim should be easily met. Unfortunately, due to the pitfalls associated with the existing, largely ad hoc, design procedures for organizing these components, this is not so.

An obvious way to improve a computer system's throughput is to design the hardware so that many of the basic computations are performed in parallel. When a computation is segmented so that parallel execution can be carried out, intermediate results must be passed between segments and the final result must be assembled before finishing the computation. This requires that the hardware, in addition to performing the actual computations, must coordinate and synchronize the transfer of intermediate and final results. In some classes of problems (e.g. the solution of partial differential equations by relaxation methods) this coordination and synchronization is highly structured, making is possible to construct so-called "applications directed" computers for their solution. (The Illiac IV computer is a good example of this class of computers, see [Ba].) However, in the general case the coordination problem becomes increasingly complex, and a situation of diminishing returns is

----

* This work was supported in part by the National Science Foundation under Grant NSF-ENG-78-5779.

reached -- linear increases in throughput resulting from parallelism require higher order increases in the amount of hardware to provide coordination. This diseconomy is also reflected in the cost of systems as well as in the difficulty of the design problem associated with such systems. As a consequence of this, current design practice is to forego much potential parallelism in favor of manageable coordination.

This paper outlines a methodology for the design of computer systems, by realizing them as a distributed network of dedicated logic building blocks which interpret the target computer's operating system. In general, using the design methodology results in systems which have a high degree of parallelism and which make wide use of cheap readily available LSI components such as single chip microcomputers and bit-slice processors. Furthermore, by basing the design methodology on a data flow [DM] approach to coordination and synchronization the above diseconomies are reduced, making more complex systems feasible.

The dataflow approach to controlling systems can be summarized as follows: In conventional computer systems a basic computation, such as the execution of an instruction, is carried out upon the receipt of a control signal from a centralized controller (consider the design of a microprogrammed computer). To make sure that the instruction executes with the correct data requires the controller to have knowledge of the data flow. In the case of parallel processing the number of possibilities for data flow patterns becomes very large, thus making the controller correspondingly complex. By allowing the data to drive instruction execution, and hence in a sense create its own control signals dynamically, this complexity, which eventually becomes overwhelming in large conventional systems, is much reduced in similar data driven ones. (For a discussion of the relationship of data flow concepts to operating systems see [De].)

For our purposes the tasks of an operating system can be split into two classes. One class allocates hardware resources such as processors, I/O devices, and memory. The other class allocates software resources such as files, processes, and message channels between processes. The design methodology is aimed at those tasks in the first class. However, we believe that using it to design a target system will simplify the subsequent design of those subsystems that handle tasks from the second class. The operation of an operating system results in a collection of basic computations having a high degree of parallelism, but whose interaction does not conform to a simple regular structure. This suggests that a regular interconnection pattern of simple modules is not appropriate. However, modularity can be retained, as can a well defined interconnection discipline, provided it does not rely on a regular interconnection pattern.

The design methodology uses as a building block a processing element (PE). This is a piece of programmed logic that contains some form of function unit(s) and can perform a fixed program of register-to-register type operations, with look-ahead. The look-ahead allows the operation of the PE to take advantage of any parallelism implicit in its program. An interconnection discipline between PEs is established that enables the designer to specify the target operating system by writing a modular program, that defines the fixed program in each PE. This can be developed as a top-down procedural characterization of the register-to-register operations within each PE -- the designer need not be concerned with explicitly specifying parallelism. Each procedure and subroutine in the program corresponds to a programmed PE. The look-ahead scheme uses a tagging technique which allows registers and function units within a PE to be reserved until their intended contents and operands are available. In this way potential parallelism among the register-to-register operations issued by the fixed program is scheduled dynamically. The function units perform their operations only upon the receipt of all of their operands, thus the dynamic scheduling is effected by the availability of data -- hence the target system is data driven. The hierarchical organization arises as a result of allowing the function units within PEs to be realized by other (possibly shared) PEs, or special purpose hardware such as, memories, tape units, disk controllers, etc. The look-ahead scheme does not distinguish between these, but treats them all as virtual function units. Because such a wide range of subsystems are regarded as function units, and because their individual execution times may be highly data dependent (consider a sequential memory for example), the dynamic scheduling of each function unit operation, resulting from the look-ahead scheme in each PE, is an efficient solution to the exploitation of potential parallelism in the target operating system. As in the intra-PE case, the flow of control in the inter-PE case is accomplished by the movement of data. This time it is between modules in the hierarchy.

The advent of cheap widely available LSI components has spurred research into the problem of effectively combining large numbers of these components together to construct powerful computing systems. This research falls into three categories.

The first of these categories is characterized by attempts to design tightly coupled systems using microprocessors. A good example is given in [N], where an SIMD (single instruction stream, multiple data stream) machine is proposed. It calls for a main memory, eight control units, a distribution switch, and a set of 1024 identical processing elements each with their own local memory. (Actually, the eight control units will allow up to eight SIMD programs.) The motivation for such designs is, of course, economic -- a system approaching Illiac IV's capabilities at a fraction of the cost. It is interesting to note, that in this particular example it has been suggested that the processing elements, which are microprogrammed, should have writeable control stores. Then each could be tailored to fit the process being handled by its control unit.

The second of these categories is characterized by attempts to design loosely coupled systems of microprocessors by specifying an operating system that can run on distributed hardware. A typical example from this category is the ROSCOE operating system [SF]. It is organized around an explicit message scheme to give inter-process communication, and is being implemented on five DEC LSI-11s. Presently, many software design problems need to be overcome.

The last of these categories is characterized by the development of interconnection rules for sets of microprocessors, so that any desired target system can be designed. This is the approach taken in this paper, and in [Mu]. Other work typical of this approach is the work reported in [Bi]. The design methodology of [Bi] proposes that the hardware of a computer be viewed as a set of cooperating sequential processes. The sequential processes, which are just sets of totally ordered (with respect to time) processes, are to be realized in a modular fashion by using microprocessors. The way in which they cooperate (i.e. the coordination and synchronization among them) is defined explicitly in terms of Petri nets [P], which are implemented using additional random logic.

The problem posed by the under-utilization of ever increasing hardware capability is sufficiently large that it is not clear which of these categories of research will be the most fruitful. Because of the diverse nature of computer applications it is likely that each of the three will have an appropriate context.

2. THE DESIGN METHODOLOGY

In this section an outline of the design methodology is presented. This outline includes: the definition of a PE, and the interconnection discipline.

2.1 The Definition of a PE

A PE is shown in Figure 1. The main components are:

1. A control memory - CM.
2. A program counter for the CM - PC.
3. A set of general purpose registers - GPR = $\{R_1,\ldots,R_M\}$
4. A set of function units - FU = $\{F_1,\ldots,F_N\}$
5. Interconnection logic - ICL.
6. A tag manager - TM.

The operation of the PE can be thought of as a sequence of register-transfers under the control of a fixed program contained in the CM. These register-transfers are of the form:

$$R_I \leftarrow F_J(R_{J_1},\ldots,R_{J_K}) \qquad R_I, R_J,\ldots,R_J \in \text{GPR} \cup \{\text{PC}\}$$

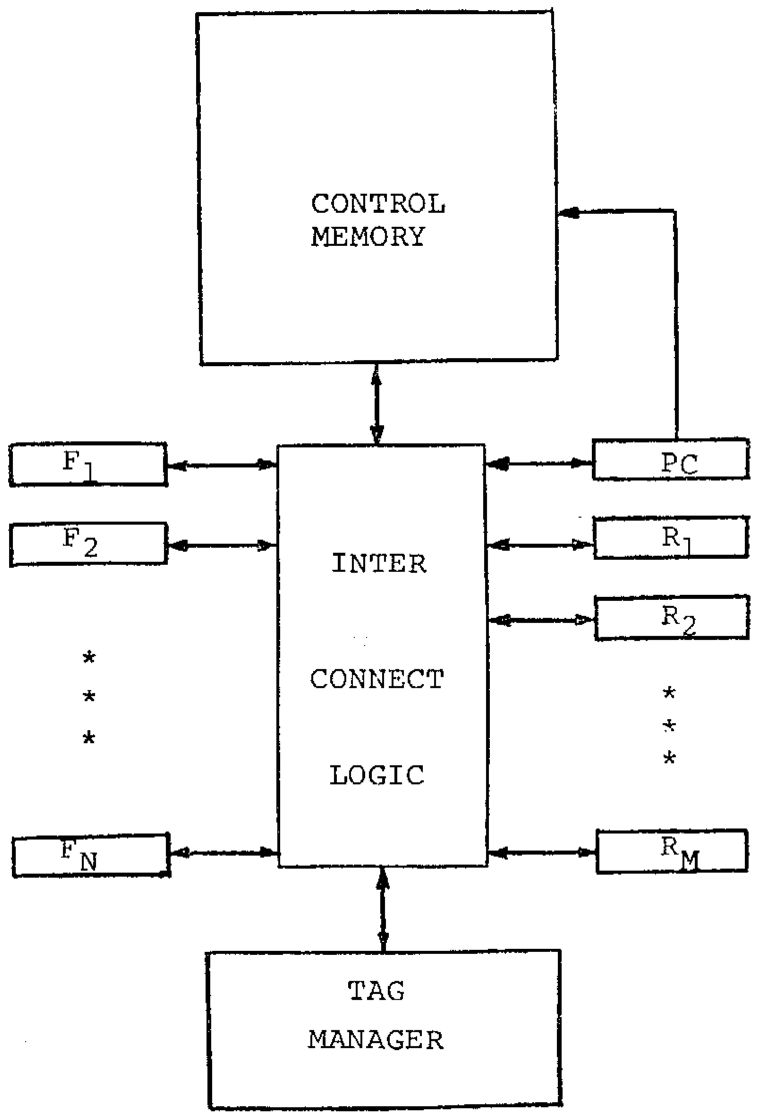


Figure 1. The Processing Element (PE).

R_I is the range register and $\{R_{J_1}, \ldots, R_{J_K}\}$ are the domain registers. F_J is the function performed by function unit F_J: we shall assume that each F_J can range over a set of functions and that this set may not be the same for each F_J. A subset of the register-transfers modify the PC, so that conditional branching can be performed within the fixed program in CM. Each register-transfer has a unique index, so that the program in CM is just a sequence of these indices.

The function units have input and output buffer registers. Their use can be illustrated if we decompose the above register-transfer. If x_1 through x_K are the input buffers to F_J, and y is the output buffer, then the register transfer becomes:

$$\begin{array}{l} x_1 \leftarrow R_{J_1} \\ x_2 \leftarrow R_{J_2} \\ \quad \cdot \\ \quad \cdot \\ \quad \cdot \\ x_K \leftarrow R_{J_K} \\ R_I \leftarrow y \qquad (y = F_J(x_1,\ldots,x_K)) \end{array}$$

These basic steps in the performance of a register-transfer could be reflected in the PE by organizing its instruction format in a more horizontal fashion than that proposed above. Instead of just containing the index of a register-transfer, each instruction could be broken up into fields. The bit pattern in each field would then represent one of the basic steps in the performance of the register-transfer. These steps are also register-transfers, but of a simple data movement type -- no data transformations occur during them.

The date movements are handled by the ICL. This may be a system of one or more busses. In general, some form of arbiter must be incorporated into the ICL to avoid collisions during data movements. The standard microprogramming technique of performing the data movements, that make up the register-transfer, in a sequential order rather than attempting to initiate them all at once, makes the job of the ICL easier. (In microprogramming this is achieved by interpreting the fields of a horizontally organized microinstruction sequentially.)

So far we have presented the PE as a programmable machine that performs register-to-register operations one after another. It differs from a stored program computer only in that program and data are separated: program resides in the CM and data in the general purpose registers. In order to speed up the execution of the fixed program in the CM, advantage can be taken of the inherent parallelism present in the program. This is done automatically with the aid of the TM by a look-ahead scheme which we shall describe next.

The term look-ahead refers to the fact that an instruction can be executed out of sequence and possibly in parallel with other instructions provided the following two conditions are true. First, that there are enough function units having the required capability available. Second, that the sequence of values taken on by the general purpose registers remains unchanged. However, as we shall see next, a value may be represented by a tag that indicates where that value may be found.

Each function unit has a unique index associated with it called a tag. This tag can be used to reserve registers that are to receive a result from the function unit. The proposed look-ahead scheme examines an instruction, and if there is an available function unit capable of executing it, the contents of the domain registers it specifies are sent to the input buffer registers of that function unit using the ICL. The range register specified by the instruction is loaded with the tag associated with that function unit. The function unit is then set to go and the next instruction is examined. We term the above procedure issuing (an instruction).

The instruction examined is determined by the contents of the PC. Normally the PC is just incremented after each instruction is issued. In the case of branching where the PC is modified by a function unit, the PC is loaded with the associated tag of that unit until the new value of the PC is computed. During such periods instruction issuing halts, i.e., we shall not, in this paper, consider the possibility of look-ahead past a branch in the PE's program, although additional look-ahead could be achieved if it were known which outcome of a branch occurred most often. It also halts when an instruction is encountered that calls for a function which cannot be performed by any of the available function units. It resumes as soon as a unit becomes available.

When a function unit completes its operation it sends the result directly to those general purpose registers and function unit input buffer registers that contain its associated tag. The case of a tag being in the input buffer register of a function unit, and hence of a result being forwarded directly to the input of another function unit, arises in the following manner. Recall, that in issuing an instruction the contents of those general purpose registers specified as domain registers by the instruction are sent to the input buffer registers of a function unit. The content of some of the domain registers may be tags (indicating that those registers are awaiting results from function units), thus these tags are sent to input buffer registers. Naturally, when a function unit has a tag instead of data in any of its input buffer registers, it must wait for the data to appear before it can complete its operation.

The above look-ahead scheme is essentially that described in [T] and used in the floating point unit of the 360 model 91. The scheme is described with more precision in [K] where the question of optimal parallelism is discussed. In this last reference it is shown that the sequence of values taken on by the general purpose registers remains unchanged (if it is allowed that each tag can stand for the value of the next output of the function unit that it is associated with) as a result of using the proposed look-ahead scheme. Reference [K] also points out that when a function unit generates a result it must perform an associative search involving the contents of each general purpose register and input buffer register to determine where to send the result. This results in an unduly complex ICL. An approach which reduces this complexity is embodied in the TM. Its operation will be described next.

The TM contains a two-dimensional matrix of bit values. The rows correspond to the tags and the columns correspond to the general purpose registers and the function unit input buffer registers. If there is a tag for function unit x in register y, then there is a one bit in the matrix at the intersection of the row assigned to tag x and the column assigned to register y. This keeps track of the tags, so that when a function unit generates a result it knows where to send it by checking the columns having one bits in the row corresponding to the function unit's tag. This avoids the associative search. It also facilitates one other tricky point associated with tag movements that arises when a tag in a general purpose register is overwritten with another tag. In such a case, the bit corresponding to the first tag is set to zero by clearing the matrix column corresponding to the general purpose register, then the bit corresponding to the second tag, that overwrites the first tag, is set to one in the matrix. Since the rows and columns of the matrix are indexed, all these operations can be performed efficiently. The matrix is updated whenever a new instruction is issued, as, in general, this results in moving tags to input buffer registers and the overwriting of tags in general purpose registers. It is also updated whenever a function unit generates a result: after the bits in the row corresponding to function unit's tag have been used to determine where to send the result the row is cleared.

Figure 2 illustrates the look-ahead scheme in a PE having three function units F, G and H, and 6 general purpose registers. The bit matrix in the TM is shown also, and the presence of a one bit in it is shown by a cross. Four snap-shots of the PE are shown as the three instructions at the top of the figure are issued. Notice how in the third snap-shot the tag for function unit F (namely t_F) has been overwritten by that for H (namely t_H)

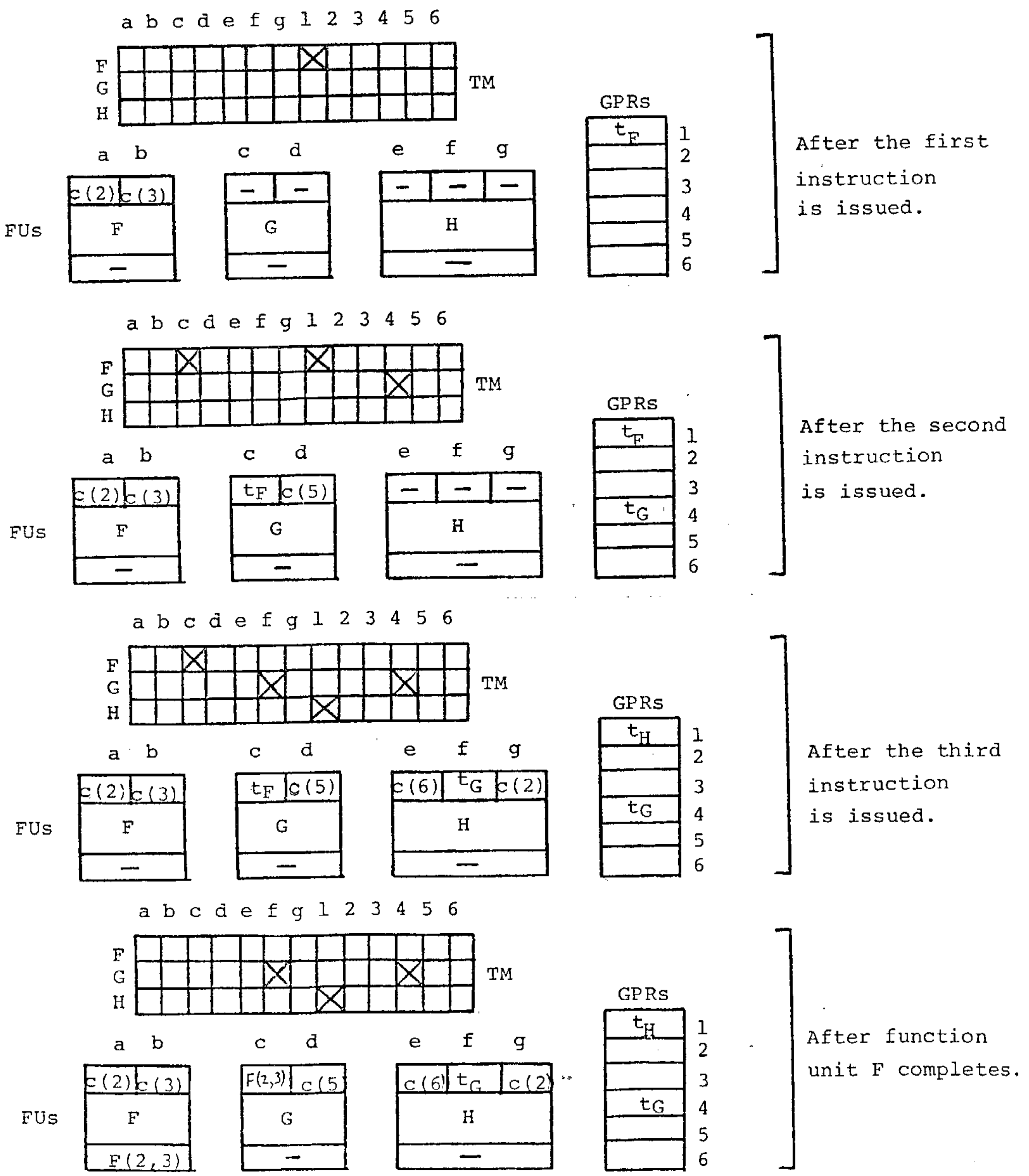
The TM matrix in the above scheme becomes unwieldy for PEs with large numbers of general purpose registers and function units. In these cases an alternative scheme may be used. Its operation will be described next.

Each function unit maintains a linked list of names of registers where its result is to be sent. This avoids the associative search again, but runs into trouble when a tag is overwritten because some function unit will then contain an incorrect destination in its linked list. In such a case the following action can be taken: First, before the old tag is overwritten use it to address its associated function unit. Second, in this function unit delete from the linked list the name of the register which contains the tag to be overwritten (this register has identified itself in the first step). Finally, overwrite

Partial contents of the CM

```
1 ← F(2,3)
4 ← G(1,5)
1 ← H(6,4,2)
---
```

Read C(x) as "contents of register x", & t_x as "tag for function unit x"



a b c d e f g 1 2 3 4 5 6
F
G
H
TM
GPRs
t_F
1 2 3 4 5 6
FUs
c(2) c(3)
F
G
H
After the first instruction is issued.
tF c(5)
t_G
After the second instruction is issued.
t_H
c(6) tG c(2)
After the third instruction is issued.
F(2,3) c(5
F(2,3)
After function unit F completes.

Figure 2. An Illustration of the Look-ahead Scheme.

the tag.

2.2 The Interconnection Discipline

In this subsection we shall discuss how networks of PEs can be interconnected to form the operating system of a target machine.

Figure 3 shows the interconnection of two PEs. From the point of view of PE_1, PE_2 looks like a function unit, and vice versa. PE_1 invokes the use of PE_2 by issuing an instruction of the form $R_I \leftarrow F(R_J)$. As soon as a, the input buffer register for F, is loaded with data (not a tag), the data is transferred to y, the output register of function unit G in PE_2. To PE_2 it appears that function unit G has completed an operation and the data in y is sent to all the registers containing the tag associated with G. Eventually PE_2 places data into x, the input buffer register for G. This can be regarded as the result returned after PE_2 has operated on the data that was initially sent to y. This result is sent to b, the output register for F in PE_1. Thus, PE_1 believes that it has used a function unit F, unaware that F has been decomposed into a series of register-transfers that are performed by PE_2. Since the processing performed by PE_2 may take a widely varying amount of time (i.e. its execution time may be very data dependent), the look-ahead scheme becomes important if the operation of F is not to block the issuance of instructions in those cases when F takes a long time. Notice it is the wide variation in the execution of F that makes the efficient use of the hardware difficult. If F took a fixed time, albeit long, other events could be set to run concurrently. However, if F is done rapidly on some occasions, but takes much longer on others, a dynamic method of scheduling events (such as the look-ahead procedure) is the only efficient solution.

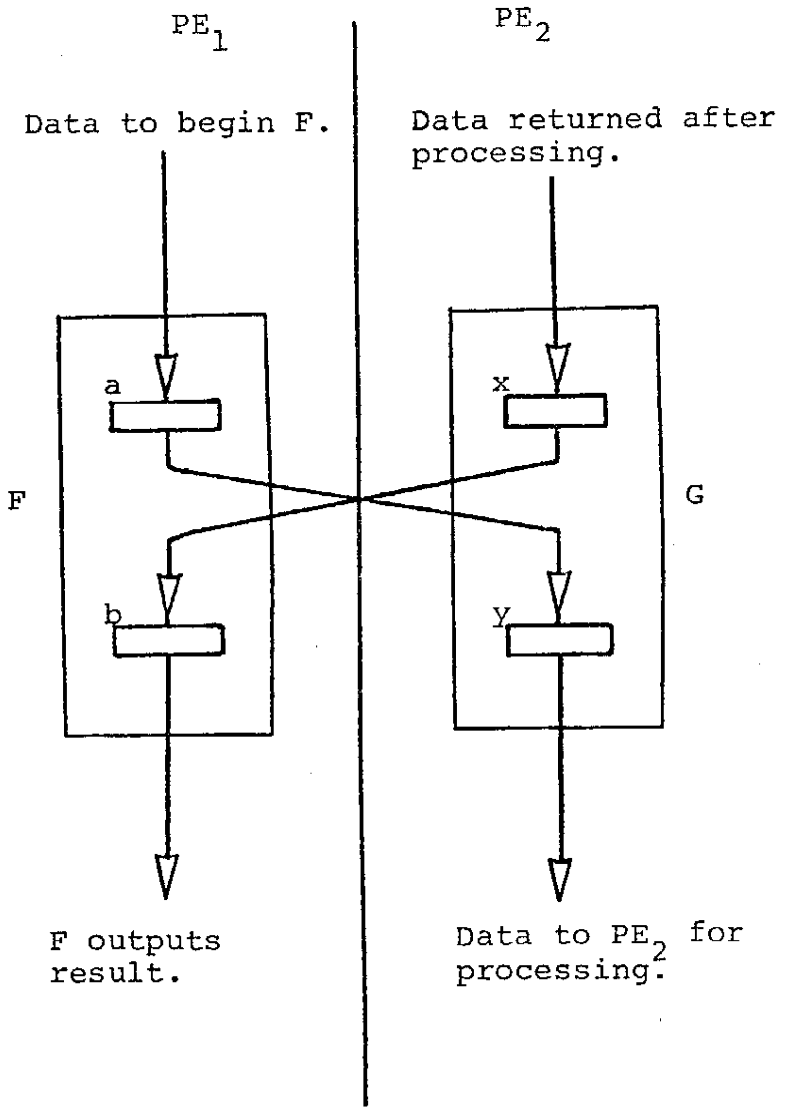

PE_1
PE_2
Data to begin F.
Data returned after processing.
a
x
F
G
b
y
F outputs result.
Data to PE_2 for processing.

Figure 3. Interconnecting Two PEs.

It is important to draw attention to the fact that data, not tags, are transmitted between PEs. Tags never leave the PE in which they were created. This modularization makes the look-ahead scheme practical. If tags could traverse PE boundaries a single tag management system would be needed for the whole system. The complexity of tag management does not grow linearly with the size of the domain in which it manages tags, but at a much faster rate. Therefore, by structuring the design methodology to yield a modular target system we minimize the problem of tag management, but retain the advantages of the look-ahead scheme.

In Figure 3 the interconnection of PE_1 and PE_2 does not bias the control flow in favor of making PE_1 the controller and PE_2 the controlled (although we have chosen to view things this way in the previous discussion), or vice versa. This interconnection discipline results in a relationship between the two parts that is analogous to that between two coroutines. By suitably programming PE_1 and PE_2 it is possible to make PE_1 control PE_2 or vice versa. In both cases, a relationship would exist that is analogous to that between two blocks in a structured program in which one is nested inside the other. However, in general, the analogy is incomplete in that the nested PE may not reinitialize its registers each time it is invoked by the other PE. Two PEs can control a third, provided they control it through separate ports. In this case the analogy of a shared subroutine comes to mind. Once again the analogy is incomplete; this time because parameters are passed through separate registers to the shared PE. The fact that the register-transfers that occur in a PE are issued sequentially as instructions from the CM means that it is a straightforward task to ensure that mutual exclusion occurs between the attempts of the two sharing PEs to get control of the PE they share: potential conflict is automatically taken care of by the fact that the separate requests can be granted by separate instructions in the shared PE, and since instruction issuing is seqential it is easy to arrange to resolve the requests.

The illustration in Figure 3 shows a situation which one argument is sent to PE_2 from PE_1 and one result is returned. In general many arguments could be sent and many results returned. This could be accomplished by time multiplexing the interchange of data or by giving F multiple input buffer registers and output registers. One case of special interest is when one PE is required to control another and the controlled PE does not require an argument and does not return a result. In such a case the controlled PE can be initiated with the transfer of a reserved bit pattern from the controller and it can signal its termination by returning a reserved bit pattern. In its simplest form this is equivalent to a request/acknowledge signaling protocol (in such cases only one bit needs to be sent and received). However, in more complex situations it may be desirable to interchange a set of status bits and argument values. Notice now, that what was initially depicted as a function unit (F in Figure 3) has been generalized to a procedure in the programming language sense.

In designing an operating system machine, as a network of PEs, a variety of types of communication are necessary. Below are listed two of the more common types with a description of how they would fit into the unifying framework of the interconnection discipline.

1. Reading and writing into a memory: Figure 4 illustrates how interconnections look for two PEs, PE_2 and PE_3, that read from different read ports (RP1 and RP2) of a memory system managed by PE_1.

2. An interrupt: Figure 5 illustrates how the interconnection looks for one PE, PE_1, that can interrupt another PE_2.

The functional description of the three PEs is shown below. For further comments see Figure 5.

$b \leftarrow RP1(a)$

b(d)	a(c)
read @ x	return "word in memory @ x"

$y \leftarrow Read(x)$

y(v)	x(u)
word in memory @ x	request to "read @ x" (x contains an address)

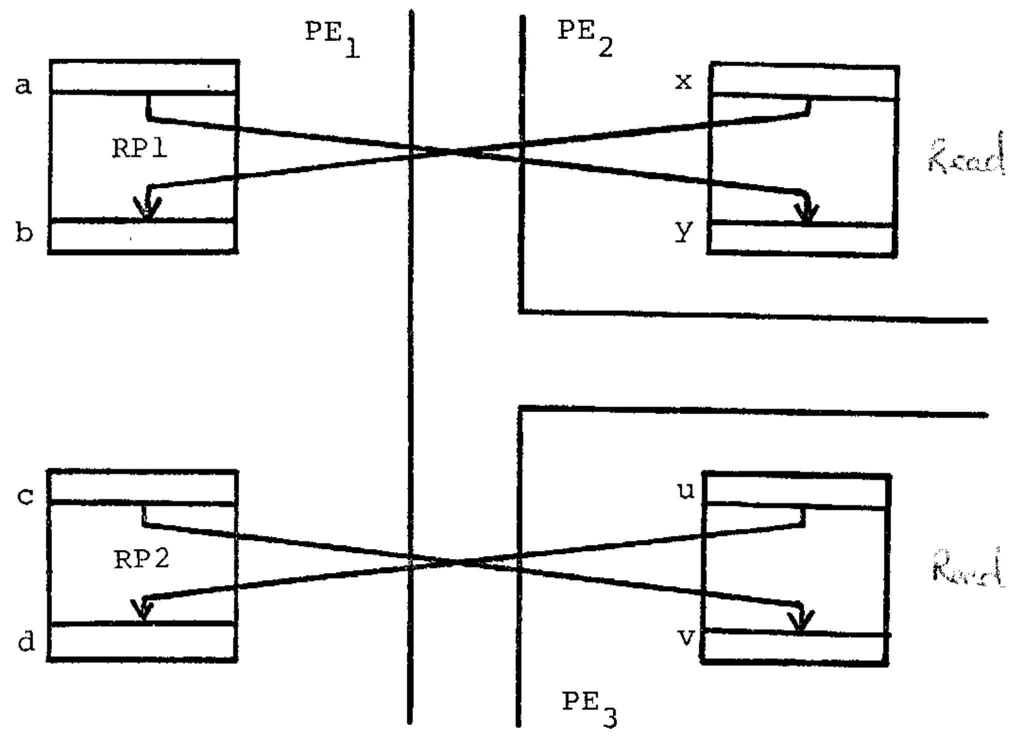


Figure 4. An Interconnection for a Memory System.

A final observation is that memories, tape units, disk controllers, etc. can all be regarded as PEs if they are interfaced to other PEs in the manner of Figure 3.

2.3 Comments on the PEs and Networks of PEs

Systems designed using PEs and the above interconnection

PE_1 interrupts PE_2 then does a bulk transfer of a fixed amount of data. A functional description of the PEs is shown below. Note that, in general, the time for a function to be performed is unbounded: timing is implicitly linked with the flow of data.

PE_1

$b \leftarrow F(a)$

b	a
rx "start tx"	~~request~~ tx ~~to~~ "interrupt"
rx "enable interrupt"	~~request~~ ~~to~~ "bulk tx"

PE_2

$y \leftarrow G(x)$

y	x
rx "interrupt"	~~request~~ tx ~~to~~ "enable interrupt"
bulk ~~tx~~ rx	request to "start tx"

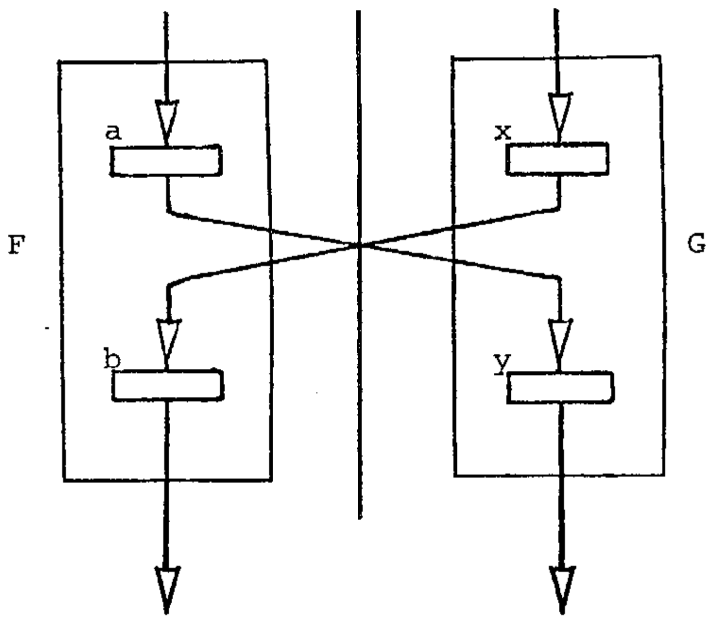


Figure 5. An Interconnection for an Interrupt.

discipline will evolve in a top down fashion to produce a hierarchically organized system. The design of each PE will be tailored to fit its specific subtask and this will be defined by its fixed program. The overall system can be designed by writing a structured program, whose components stand in close analogy with those found in the discipline of structured programming (as we noted in the previous subsection). Each block and subroutine in the program can be translated into a PE. Elements of both the control structure and data structure, in such systems, reside in each PE.

Although the program will be written as a procedural program, the implicit parallelism present in the program will be realized in the operation of the hardware, as a result of the look-ahead procedure in each PE. Furthermore, it can be observed that a register-transfer in such systems is data driven in the following sense: a register-transfer is enabled by the reception of tags into its domain registers it is then fired, so to speak, upon the reception of all of the operands into its domain registers. Thus systems resulting from using the design methodology may be regarded as data flow systems.

Two major problems associated with the coordination of processes in computer hardware are deadlock and non-determinacy. At the hardware level a process is taken to be a register-transfer or collection of register-transfers. The resources of such processes are their registers, buses and functional blocks. A set of processes is defined to be deadlocked [Ha] when no process can proceed without acquiring a resource already held by another process within that set. Necessary conditions for deadlock are: resources must not be sharable or preemptable, resources must be retained while a process is acquiring further resources, and there must be a circularity in the resource requirements of the processes. A set of processes is defined to be non-determinate if the computation performed by those processes is not uniquely determined by their initial state. Necessary conditions for determinacy are: the sequence of values taken on by the registers in the set of processes should depend only on their initial values. The concept of deadlock and non-determinacy are discussed further in [Mi] and [Di].

Based on the above conditions we make two conjectures:

1. Systems designed using the preliminary ideas presented above are deadlock-free, if the control graph of the system is circuit-free.

2. These systems are also determinate.

We define the control graph of a system to be a directed graph that indicates the control relationship between the set of interconnected PEs that make up the system. Each PE is represented as a node in the graph. Given two PEs A and B, if A controls B, then an arc is drawn from the node for A to that for B. If A and B have co-control of one another (the coroutine analogy), then the nodes for A and B are joined by an edge with arrowheads at each end. A circuit in the control graph implies a circularity in the resource requirements of the processes. This satisfies the necessary conditions for deadlock (whether a circuit-free control graph implies no circularity in the resource requirements remains to be proved).

The determinacy of each PE follows from the discussion in [K], whether this remains true for sets of PEs, interconnected according to the above discipline, will require proof based on a type of structuring theorem.

It should be noted that guaranteeing that processes at the hardware level are free of deadlock and non-determinacy does not preclude a programmer, who writes programs for the system, from writing programs that define higher level processes that deadlock and are non-determinate.

Finally, for implementing PEs using LSI components there are two choices; either a conventional microprocessor architecture may be chosen, such as the Intel 8080, or a bit slice architecture may be chosen, such as the AMD 2900 series. The first choice will almost certainly make it necessary to give up intra PE look-ahead. Also, pin limitations will make target systems bus limited, and be the overriding factor in limiting the speed of the system, The second choice would be a better one. A bit slice architecture is not bus limited, would allow intra-PE look-ahead to be easily implemented, and be expandable to fit whatever word size the target system required. In the case of the AMD 2900 series in particular, bipolar speeds would be available. This would allow data transfers between PEs in the order of 100nS.

3. CONCLUSION

A design methodology suitable for the design of operating system machines has been presented, which yields highly parallel systems without the designer having to worry about the details of coordination and synchronization. Since the appearance of [Mu], a similar application of the look-ahead algorithm of [T] has been suggested to speed-up instruction execution in multifunction processors (see [Hi]), by placing the look-ahead at the microprogram level. This can be regarded as a firmware implementation of the type of control found in the floating point unit of the 360 model 91.

4. REFERENCES

[Ba] Barnes, G., et al., "The ILLIAC IV Computer," IEEE TC, C-17, No. 8, pp. 746-757, Aug. 68.

[Bi] Bisiani, R., and F. Tisato, "A Multi-microprocessor System as a Set of Cooperating Processes," Proc. Euromicro Workshop, eds. R. Hartenstein and R. Zaks, Jun. 75.

[De] Denning, P.J., "Operating Systems Principles for Data Flow Networks", Computer Magazine, pp. 86-96, Jul. 78.

[DM] Dennis, J.B., and D. P. Misunas, A Preliminary Architecture for a Basic Data-Flow Processor, Comp. Structures Group Memo 102, Proj. MAC, MIT, Aug. 74.

[Di] Dijkstra, E.W., "Co-operating Sequential Processes,"Programming Languages, F. Genuys Ed., Academic Press, New York, 68.

[Ha] Habermann, A.N., "Synchronization of Communication Processes," CACM, Vol. 15, No. 3, pp. 171-176, Mar. 72.

[Hi] Higbie, L.C., "Overlapped Operations with Microprogramming", IEEE TC, C-27, No. 3, pp. 270-275, Mar. 78.

[K] Keller, R.M., "Look-Ahead Processors," Computing Surveys, Vol. 7, No. 4, pp. 177-195, Dec. 75.

[Mi] Miller, R.E., "A Comparison of Some Theoretical Models of Parallel Computation," IEEE TC, C-22, No. 8, pp. 710-717, Aug. 73.

[Mu] Mudge, T. ,"Research Initiation into a Design Methodology for High Performance Digital Computers", Proposal for research supported by the National Science Foundation under Grant NSF-ENG-78-5779, Dec. 77.

[N] Nutt, G.J., "Microprocessor Implementation of a Parallel Processor," Proc. 4th Annual Symp. on Computer Architecture, Mar. 77.

[P] Petri, C.A., Communication with Automata, Suppl. 1 to Tech. Rept. RADC-TR-65-377, Vol. 1, Rome Air Development Center, New York, 66.

[SF] Solomon, M.H., and R. A. Finkel, "A Multi-microcomputer Operating System", Proc. 2nd Rocky Mountain Symp. on Microcomputers, pp. 291-310, Aug. 78.

[T] Tomasulo, R.M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Jour. R&D, Vol. 11, No. 1, pp. 25-33, Jan. 67