Reprinted from: Organisatiewetenschap en praktijk
edited by P. Verburg, P. Ch. A. Malotaux, K. T. A. Halbertsma and J. L. Boers
H. E. Stenfert Kroese B.V., Leiden 1976

THE DESIGN OF LARGE COMPUTING SYSTEMS AS AN ORGANIZATIONAL PROBLEM*

PROF. DR. HERBERT A. SIMON

1. INTRODUCTION

Formal organizations are the principal social systems through which a modern society gets its work done. The component parts of organizations are groups of human beings who have more or less definite roles and patterns of communication. The activities they carry out can be divided into 'doing' and 'deciding', and indeed the entire organization can be thought of as a decision-making structure whose function is to guide the activities of the 'doers'. Designing an organization means designing this decision-making system.

Of course modern organizations, and particularly industrial organizations, contain machines as well as people. But the machines are generally regarded as passive components, that is, as belonging to the 'doing' rather than the 'deciding' component of the system. Hence they enter into the organization design only as part of the definition of the task environment in which the decision-making must take place. The machines have largely played the role of magnifying and multiplying the physical energy that can be applied to the organization's tasks. The information and control needed to operate the system have been supplied mostly by the human components.

Within the past generation a new kind of organization has begun to appear which contains a third type of component, the electronic digital computer. The computer is called a 'machine' (the leading professional organization of computer scientists in the United States is the Association for Computing Machinery), but it has a radically different role in organizations from the machines that preceded it. The computer is not a power-multiplier but an information processor. It does not belong to the 'doing' or operative part of the organization, but to the decision-making part. It shares, and shares increasingly, in the functions of processing information and exercising control:

* This research has been supported by Public Health Service Grant MH-07722 from the National Institute of Mental Health and by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-73-C-0074) which is monitored by the Air Force Office of Scientific Research. I am grateful to John Gaschnig for helpful comments on an earlier draft of this paper.

both control over machines and control over human activities through the decision-making process. In many industrial and business organizations today computers and automatic control components of machinery constitute a major part of the total capital equipment.

When they first appeared, computers were comparatively simple arrangements of electrical or electronic components. Although they grew rapidly in size and speed (as well as numbers), their basic architecture remained relatively stable until the middle 1960's, when so-called 'time-shared' computers began to appear. While the computers of the first two decades perform one job at a time, time-shared computers perform a number of jobs more or less simultaneously (or so it seems to the users). More recently, in the 1970's, we have begun to see even more complex computing systems that consist of whole networks of computers of the earlier kinds which are capable of intercommunicating in sophisticated ways.

These developments, the growing use of computers in human organizations and the growing complexity of computers themselves, create two new problems of organization design. The first is the problem of structuring organizations that contain both human and computer components in close interaction with each other. The second is the problem of designing the modern large computers themselves, for the most complex of these, even when standing alone and unincorporated in a human system, are beginning to look more and more like organizations, with their own internal problems of structure and coordination. In this paper, I should like to address these design problems in order to see how they resemble, and how they differ from, traditional problems of organization design. I shall place my main emphasis upon the second question, designing computers as 'free-standing' organizations, because it is the more novel of the two and is just beginning to be recognized as an important problem.

2. CENTRAL ISSUES OF ORGANIZATION DESIGN

Designing an organization involves dividing up its tasks among its components and then arranging for the coordination of the activities of those components: division of labor and coordination. Only a few words need be said about these familiar organizational features.

For reasons that have been stated by Adam Smith and many others, the division of labor usually comprises not only a parceling out of tasks, but also specialization of those who perform them. The former does not logically

imply the latter, for tasks might be divided up in some more or less random fashion, with every component receiving essentially the same mix of activities. A division of work is required, quite apart from considerations of specialization, whenever the total job to be done exceeds the capacity of a single component – that is, whenever an organization is needed at all. Specialization arises because it is usually economical to group tasks that are similar along one or more dimensions so as to reduce the time required to learn to perform the whole collection of tasks and the time required to turn from one task to another. Specialization is limited by the need to use the full capacity of components.

Coordination is secured in organizations by the flow of information among the components. Some, but not all or even most, of this information takes the form of commands: decision premises generated in one component that are expected to enter into the decision processes of another component. Traditionally, the formal channels for the flow of commands have been arranged hierarchically, so that the organization, from this standpoint, can be viewed as a pyramid. However, in actual organizations, many commands are transmitted through channels other than the formal hierarchic ones. Limiting commands to the formal channels is usually called 'unity of command', and we shall have to see what role it plays in computer organizations of the kind we are considering here.

Coordinating activities consume a substantial part of the resources of any large organization. Hence, a division of work will be inefficient if it creates too heavy a demand for coordination. The central task of organization design can almost be described as balancing the advantages of dividing up the task in certain ways against the costs incurred thereby of coordinating the work of the components. Moreover, if the temporal coordination among the parts of an organization is too intricate, some components may stand idle awaiting inputs that have to be supplied by other components.

There is another way of looking at the coordination problem. The job of the 'deciding' components in an organization is to direct the activities of the 'doing' components in such a way as to accomplish the organization's tasks. The decision maker is given certain resources in the form of subordinate decision makers and 'doers'. His decisions direct the use of these resources toward carrying out the tasks for which he is responsible. When an organization is described in this way, the task of designing it becomes a task of resource allocation. How shall the subordinate decision making and action components of the organization be divided among the primary decision

makers so that the organization's resources will be used most effectively in the aggregate?

The hierarchy of authority is one device for allocating resources among decision makers. If unity of command is observed, then each decision maker knows exactly what components are available to him. Of course there are a number of other devices for resource allocation. By *scheduling*, for example, resources may be allocated to particular uses, or assigned to particular decision makers, for certain periods of time. By *pricing*, decision makers may be allocated total budgets of resources, and may be allowed to bid for specific resources until their budgets are used up. We shall have occasion to refer to resource-allocation methods in the course of our discussion.

Finally, in the course of this brief discussion of organization design, I have used terms like 'component' and 'resource', which seem somewhat cold and even inhuman when applied to the people who make up an organization. In choosing this terminology, I have deliberately abstracted from a very important aspect of organization design: concern for the motivation and morale of the participants in an organization, their susceptibility to boredom and fatigue, their opportunities for achieving personal goals and values through their work in the organization, and the potential for subgoals to form and for subgoal conflicts to arise. I omit these matters, not because I think them unimportant (although they are clearly less important for organizations whose components are computers than for human organizations), but because we will be faced with sufficient complexity even without them. The practical designer of organizations cannot, of course, afford this luxury of omission. When we design a bridge, we must be as cognizant of the properties of the materials we will use as of the laws of statics and dynamics that determine the pattern of forces in a truss.

3. DEVELOPMENT OF COMPUTER ARCHITECTURE

A history of computer hardware architecture can be relatively brief, both because the time spanned by such a history is quite short (no more than about 35 years, unless we go back to Babbage), and because computer architecture has been rather stable through a large part of that time.

The von Neumann Machine. In 1946 Burks, Goldstine and von Neumann wrote a paper, '*Preliminary Discussion of the Logical Design of an Electronic Computing Instrument*', that at the same time summed up what was then

known about the principles of computer architecture, outlined the basic design for the first stored-program computer, and was enormously influential upon the thinking of computer designers for two decades or more. The opening sentence of that paper already prescribes the basic division of labor: 'Inasmuch as the device will be a general-purpose computing machine it should contain certain main organs relating to arithmetic, memory-storage, control and connection with the human operator'. The arithmetic organ was to be the 'doing' part of the system, an active unit capable of carrying out additions and multiplications on numbers. Together with the control organ, whose task was to determine the sequence of actions to be carried out, it constituted a small, serial (one action at a time) active central processing unit. That unit would make use of information stored in a large but passive memory organ, which would contain not only data (the inputs and outputs of the arithmetic operations) but also the program itself which the control organ used to remember the correct sequence of operations. Finally, an input-output organ would give the system a means of communicating with the outside world, that is, the operator.

That design was so simple and clean, so general and effective, that there were only minor departures from it for twenty years. Because of the serial operation of the system and the centralization of program control in a single 'organ', coordination problems were minimal. It would perhaps be more accurate to say that they were 'minimal in principle', for, as we shall see, if they were absent from the hardware design, they reappeared at the software or programming level.

One feature of the von Neumann machine that has already been mentioned deserves special emphasis. Since program was stored in the same memory as data, that memory had to be capable of handling general symbols, and not simply symbols interpreted as numbers. Moreover, the control unit had to be capable of manipulating program instructions, and also of interpreting their meanings. Thus, the machine was not merely an enlarged desk calculator, but an extremely general symbol manipulating or information processing system. As we know today, it was general enough to handle information represented in natural language as well as information represented in the language of mathematics. The intimate interaction of people with computers in the organizations of today and of the future rests on this generality.

The general architecture of the von Neumann machine was dictated partly by the fact that it minimized problems of coordination (which would have been difficult or impossible to solve at a time when the most elementary concepts of computing were just being born), and partly by the higher costs of

active as compared with passive components. It was far cheaper to store large quantities of information in a passive memory and bring them into the central processor whenever they were to be acted upon, than it would have been to provide a memory where operations on data could be performed *in situ*. In fact, the designers contemplated the possibility of using a whole hierarchy of memories, for reasons of economy: 'We are therefore forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible'.

Thus the notion of hierarchy intruded itself into the design, as it does into the design of all complex systems. In fact, there were several other hierarchies contemplated besides the hierarchy of memories. The entire machine, as we have seen, was divided into a number of principal organs. Each organ, in turn, had its parts and subparts. The memory, for example, was divided into units called 'words', and the words, in turn, into bits. This hierarchy resembles the hierarchies that emerge from successive divisions of work; it was not in any sense a hierarchy of authority. The central control organ, for example, did not transmit its instructions to particular memory locations through intermediate agents, but acted on those locations directly.

Time Sharing. This is the way that things stood until a number of years of experience had been acquired in using machines of this type in a variety of organizational contexts. Then it was gradually discovered that the larger system, consisting of a von Neumann machine and its human user, was not always a well-balanced organization. There was no problem when the system was used to perform very large numerical computations, for then the interaction between computer and user was infrequent. Because of the machine's enormous speed, problems arose when more frequent interaction was necessary – for example, in 'debugging' programs, or when using the system as an information-retrieval device requiring only relatively simple computations for each retrieval request.

In jobs requiring frequent response of user to computer, like those mentioned above, the discrepancy between their processing speeds caused the expensive computer to be idle a large part of the time. Observation of this inefficiency led to the notion that one computer should be teamed up with an appropriate number of human users, so that while one user was thinking or responding, the computer could be processing the job of another. This is the basic idea that underlies multiprocessors, or time-sharing computers.

A similar discrepancy in processing speeds arose between the central computer and its input-output devices: card readers and line printers. Initially

this problem was met by disconnecting the central processor from these input and output devices. The card reader produced a tape while the computer was working on another job, and the tape could then be transfered physically to the central processor. Similarly, the processor produced an output tape that could be dismounted and then mounted on a separate device for input to the line printer. Later (around 1960), these procedures led to the idea of enlarging the computing system to include a separate input-output processor, with provision for automatic transfer between it and the central processor. This innovation represents an early form of parallel processing, a topic that will be discussed in the next section. Notice that these solutions create a buffer between the peripheral devices and the central processor, and that it is this buffer that permits them to be desynchronized. It was not at once realized, in the early systems involving tape transfers, that such a buffer, although it may increase the efficiency and throughput of the central processor, may also (and frequently did) create substantial job delays because of the time jobs spend in queues between the three stages of processing. This was a truth known to industrial engineers and job-shop schedulers that had to be reinvented, unfortunately, by computer designers and programmers.

Time sharing does not, in principle, require major alterations in computer hardware. Its basic requisite is some means for allocating the computing resources among a number of users. This can be accomplished, for example, by assigning to each user, in turn, a small slice of the central processor's time. If the user does not finish his task in this time slice, then some means must be provided for 'saving' his program, data, and exact point of program interruption until he receives his next time allocation. Since rapid-access memory is limited in size, it is usually impossible to save a user's job in central memory while other users' jobs are being processed; instead, it is 'swapped' into a slower secondary memory until it is needed again for active processing, at which time it is swapped in again. Thus, in practice, a time-sharing computer requires: (1) means for interrupting a job, retaining all its data and program, and (2) means for swapping in and out, with great rapidity, the contents (or part of the contents) of high-speed memory. Memory sharing is as important a feature of a multiprocessor as is process sharing.

No sharp line can be drawn between a classical von Neumann machine, in which users sign up for successive blocks of time and a full time-sharing system. The first step in the transition is to assign to users permanent storage, or files, in secondary memory. When the user has consumed his time allocation, he stores his program and data in his file until he again has access to the central processor. Responsibility for transfering files rests on the user. This

step requires no hardware alterations. Only a single user is connected directly to the machine at a given time, and he is usually assigned a time slice measured in minutes.

The second step toward time sharing is to modify hardware so that a number of users can be connected with the computer at the same time. Of course only one of the users can employ the central processor at a given moment, but a buffer memory can be provided so that the system will accept messages from an inactive user, and place him in the queue for slices of processing time. When the slices are relatively brief, say a few seconds, such a system is usually called a 'fast batch' processor. If thirty users are connected to the system and each uses his full slice each time his turn comes, then the cycle time might be of the order of a minute or two.

At the third step, the slices become very thin (milliseconds or even microseconds in duration). Then a full task will not usually be completed in a single slice, and provision, requiring some hardware modification, must be made for halting a program midway in such a way that it can be restarted without loss of information. Provision must also be made for automatic swapping in and out of high-speed memory.

Now all of these systems resemble a familiar form of industrial organization – a job shop. It is a peculiar job shop only in the respect that there is a single machine on which all jobs must be scheduled. Nevertheless, all of the familiar theories and algorithms of job-shop scheduling may be applied to determine how resources should be allocated among users: in particular, to determine how wide the time slice should be for optimal performance (Simon, 1966b). The regrettable fact has already been mentioned that the early designers of time-sharing systems were unfamiliar with job-shop scheduling principles, and much of this theory had to be reinvented. Recognition that scheduling a time-sharing computer was identical with the job-shop problem could have saved that extra effort.

Notice that in talking about the steps from the classical computer to the time-sharing computer, it has not been possible to separate hardware design completely from software design. Many changes in computer hardware were, and are, first introduced as changes in the programs that control the allocation of the machine's software. Only when these controls have to be exercised frequently and rapidly does it become advantageous or necessary to incorporate them in the hardware design.

Parallel Processing. Even though the time-sharing computer looks to its users like a parallel processing system, for many users can be linked with it at the

same time, it still can be implemented, essentially, with a serial machine. The next, and mainly recent, development in computer hardware has been the introduction of genuine parallelism – several active components operating simultaneously. As was mentioned earlier, parallelism was first introduced on a limited scale, and for specialized components, in order to overcome particular problems of imbalance of capacity. We have seen that time sharing was a response to the imbalance of capacity between computer and user. Similarly, programs that have large amounts of output generally under-utilize central processing capacity, for output devices are slow compared with internal processing components. As we have seen, this problem can be solved by associating a small processor with the output device, capable of handling its needs, and installing a buffer between central processor and output so that output data can be handed over, in bulk, to the output processor. Then the central processor can be occupied with other tasks while output is taking place, and more than one output device can be associated with a single central processor.

As early as 1959, specialized parallelism was also introduced within the central processor to permit it to operate more efficiently. Machines were designed that, while performing one operation, could 'look ahead' at the next operations to be performed and could secure and position the operands for those operations in anticipation of the availability of the organelle that would perform the operation.

The term 'multiprocessor' is usually used for a computer that has more than one general-purpose central processor. The term 'parallel processing' is usually reserved for computers having a very large number of central processors, each of which performs, simultaneously with the others, some particular part of a total task. Very few parallel processors have been built.

One may ask what is gained by building a parallel processor. Of course the total information processing capacity of a system will be multiplied, approximately, by the number of its processors. But the same effect could be achieved by arranging the equivalent number of independent serial processors side by side, and assigning different jobs to each.

There are several circumstances under which a parallel system may be more advantageous than an equivalent amount of computing power in separate processors. Again, balance of processing capacity in the entire system is the key to most of them. If, for example, an expensive output device is under-utilized by a single processor, it may be desirable to link two processors to that device. If a number of different users are linked to the system, then employing a sufficient number of central processors to give an adequate

total computing capacity will avoid the need to duplicate users' memory files. Again, if different users were assigned to independent machines, certain programs used by several persons would have to be duplicated unless their processors were linked to a common memory containing those programs. A multiprocessor might be designed to take advantage of one or more of these potential economies.

The price that must be paid for multiprocessing and parallel computing is to provide for the coordination that is required among the several processing units. To take a silly example, suppose a column of figures were to be added, and that the task of adding up the first half of the column was assigned to Processor A, and the second half to Processor B, which also had the responsibility for producing the final result. Then provision must be made for transfering the subtotal from A to B. If one processor finished its work before the other, then it would be idle until the transfer could be made. Furthermore, B would have to be able to ascertain when A had completed its work so that the correct result, and not a partial sum, would be transfered.

Parallel processing, when it involves intimate computational interaction among the components, imposes new information requirements to permit a component to determine the status of the computations being made by other components, and introduces also new possibilities that processing capacity will be idle from time to time. These problems are less severe when the linkages are weak, that is, when relatively small amounts of information have to be transmitted from one component to another, and when delicate timing considerations are not involved. (Data buffers between components, mentioned earlier, are one means for reducing idle capacity by removing synchronization requirements.)

All of these organizational problems created by parallel computation are essentially identical with the problems encountered in human organizations where different units have interdependent tasks to perform. Imbalances in capacity among components lead to designs in which one component or another is duplicated. As the interdependence among components increases, correspondingly sophisticated means must be introduced to secure coordinated action from them.

4. SOFTWARE DESIGN AS AN ORGANIZATION PROBLEM

Organization design is involved in the construction of computer software (that is, the programs that determine the actual behavior of the computer)

as well as the underlying hardware. In fact, by means of programming, one hardware configuration can often be made to simulate a quite different one. Since the structures of large programs are even more complex than hardware structures, they introduce correspondingly complex problems of design. The basic considerations again relate mainly to specialization among program components, and the coordination of those components.

Hierarchy. Computers, like all other complex systems produced by man or nature, are structured in layers. Clusters of the components in each layer constitute the basic components of the next layer above. Bell and Newell, in their work on *Computer Structures* (1971), have identified a number of such levels in computer architecture, one of them being the program level. The discussion here will be limited to hierarchy within the program level.

The bottom part of the programing hierarchy is a hierarchy of computer languages. In the hardware of any computing system there is embedded a set of primitive processes that can be executed by the system. The computer is programed by means of a series of instructions that cause the execution of these primitive processes. The formalisms that define the vocabulary and syntax of such programs constitute the machine language, the lowest level of the programming hierarchy. Machine language is the only language that the computer (hardware) can understand, but it is extremely difficult for humans to understand. Hence, one or more additional languages are provided for every computing system, together with translators (interpreters or compilers) that permit programs in those languages to be translated into machine language for execution by the computer. The translater is itself a program stored in and executed by the computer.

These so-called higher-level languages (for example, FORTRAN, ALGOL, LISP, SNOBOL) are user-oriented languages in that they are aimed at making the user's programming task easier. Each instruction in the higher-level language is more or less meaningful in terms of the things the user is trying to do, and is usually translated into a whole program of instructions in machine language. Thus the layering of languages seals off from the user the detail of structure of the machine he is programming, and allows him to work in terms of larger chunks that abstract from that detail. In fact, the same FORTRAN or LISP program will run on many different computers, always producing the same result although the computers have quite different machine languages. The user need have only an abstract picture of the hardware, a picture of an aggregated machine whose particulars are both unknown and unimportant to him.

The significance of this kind of layering can be illustrated by a homely example. The driver of a car does not have to know anything about the detail of its mechanical structure. What he needs to know about a brake or an accelerator is the functional relation between his manipulation of brake or accelerator pedal and the response of the car. Provided that the response function does not change, it does not matter to him whether the device operates mechanically, hydraulically, electrically, or by extrasensory perception. Consequently, to the driver the car is represented by a few aggregate parameters that correspond to the ways in which he interacts with it in order to control it. (This layering principle plays an important role also in natural systems; as is explained in my essay on *The Sciences of the Artificial.*)

The hierarchic structure of human organizations can be understood in the same way. The hierarchy of formal authority not only provides reliable channels through which higher level executives can communicate; it also makes the organization sufficiently simple (in its abstracted, aggregated representation) to be understandable to its members. The detail of structure and operation of each organization unit does not need to be known, but only some broad facts about its functions – its inputs and outputs. Each member needs to know more detail about those parts of the organization with which he interacts frequently, and less detail about those parts with which he interacts frequently.

Structured Programming. The notion that hierarchy can be used to seal off the detail of one part of a system from another, and to enable the entire system to be treated in abstract, aggregated form, has been exploited not only in the construction of higher-level programming languages but also in the organization of complex programs themselves. The set of ideas for doing this is now often called 'structured programming', but it has a history that begins several decades before that name was coined for it. (The ideas of structured programming were mainly introduced in the 1950's in connection with the design of so-called list-processing languages. An early account of what would now be called structured programming will be found in Newell, et al., 1964, pp. 103-116.)

The key concept, perhaps, is the idea of a closed subroutine. If we know about a segment of computer code that, for any positive number we input to it, it will output the square root of that number, then we can be indifferent to the way in which it goes about calculating that square root, and we certainly do not need to know any of the intermediate numbers it calculates along the way to the solution. For user purposes, the segment of code is fully defined

by specifying its output as a function of its input. In this sense it is just like a segment of machine language code produced by a compiler as its interpretation of an instruction in a higher-level language. However, we do not need to define a new language in order to make use of the square root program. What we need in the user's language is the capability for (1) assigning a name to any segment of instructions so that we can refer to it, and (2) 'calling' that segment by name, providing it with appropriate input arguments, and causing it to be executed. In a language having such capabilities the segment, after naming, becomes a closed subroutine. It can now be used in essentially the same way as any other instruction in that programming language. Equally important, the using routines do not need to know its internal structure, but only what relation holds between its inputs and outputs.

Closed subroutines may be used recursively. That is, new closed subroutines may be defined that use previously defined subroutines (or even themselves) as component instructions. In this way, a hierarchically organized program can be constructed with an arbitrary number of levels of subroutines, the subroutines at each level calling subroutines at one or more levels below.

Each subroutine has two faces: an outward and upward looking face, and an inward and downward looking face. The upward-looking face consists of its name and input-output specifications; the downward-looking face consists of the instructions of which it is composed, including the names (but not the contents) of the subroutines it employs. Again, the analog to an organizational hierarchy is very close, and the reasons for structuring the program in this manner very similar.

In the most disciplined versions of structured programming, all communications between routines must take place through their specified inputs and outputs. This organization provides extreme simplicity and comprehensibility of programs, but it has its costs. Just as excessive restriction to formal lines of communication may unduly congest the coordinative capabilities of human organizations, so this limit upon program communication may be too constrictive in its effects. In the next section, another form of programming will be discussed that removes some of these difficulties while retaining most of the advantages of closed subroutines.

The 'Blackboard' Concept. Consider a computer program that is constructed as a hierarchy of subroutines. The information that is produced when these routines are executed can be classified, from the standpoint of any particular subprogram, as (1) information generated by the subprogram for its own

internal use during execution, (2) input and output information for communication with the routine that called it, and (3) information that might be of general use to other routines. In its strictest form, a subroutine hierarchy does not, as we have seen, produce or use information belonging to the third category. This restriction may be both inconvenient and unnecessary.

Let us take as an example a computer program for playing chess. In the course of exploring one variation from the current position, the program may discover characteristics of the position (e.g., that a particular piece can be moved to attack two others simultaneously) which would be of use if known while exploring other variations. One way in which this information could be communicated would be to assign some specified part of memory as a public 'blackboard', and to transfer the information to that location (Simon, 1966a). Of course, the matter is not quite as simple as this. For the scheme to be useful, at least two conditions must be satisfied: (1) every subroutine that is to be able to extract information from the blackboard must know of its existence and location; and (2) a common set of conventions, amounting to a common language, must be established to encode all information written on the blackboard, and all routines using the board must know those language conventions.

It should not be thought that the blackboard scheme is equivalent to allowing arbitrary flows of control among subroutines. The blackboard does not disturb the 'unity of command' of the subroutine hierarchy, for it leaves to each subroutine the decision of what information, if any, it will take from the board and what use it will make of it.

The Hearsay II system, a speech understanding system developed by Erman, Lesser and Reddy, provides an excellent illustration of the use of a blackboard (see Erman and Lesser, 1975). Constructing a computer program that will recognize and interpret human speech has proved to be a very difficult task, which has been only partially accomplished up to the present time. Because speech sounds are highly variable, unlike the phenomic and alphabetic schemes by which they are usually represented, success in interpreting them depends on using a number of different sources of information: the sound stream itself (encoded in some way for the computer), information about the phonemes that occur in English, information about the strings of phonemes that correspond to actual English words, information about the syntactically admissible sequences of words, and information about what sentences would make sense in the topic under discussion.

The designers of Hearsay II propose to make use of all of these sorts of information by organizing their program around a set of 'knowledge sources'.

Each knowledge source (KS) corresponds to one of the kinds of information mentioned above. Its task is to use that information to form hypotheses about the translation of appropriate segments of the speech stream. For example, the phonemic KS would hypothesize that certain segments correspond to certain phonemes, the lexical KS that certain larger segments correspond to certain words, and so on. In general, each KS would generate its hypotheses from the information provided by the KS at the next level below: the phonemic KS directly from the sound stream, the lexical KS from the phonemic KS, the syntactic KS from the lexical KS, and the semantic KS from the syntactic KS, although levels may be skipped or combined under certain circumstances. The hierarchy of levels is again an abstracting scheme. KS's at the higher levels operate with far more global information about the sound stream than do the KS's at the lower levels.

The blackboard in Hearsay II provides the necessary communication among the knowledge sources. When a KS forms a hypothesis, it communicates that hypothesis to the blackboard, along with information as to how much evidence substantiates it. When a KS uses information from the blackboard to draw inferences at its own level, it leaves behind information about the inferential connections it has drawn among hypotheses. All information on the blackboard is written in a single common language that all of the KS's are capable of interpreting.

Summary: Software Architecture. The design of computer programs is as much an organization problem as is the design of computer hardware. The principal design consideration in organizing software is to keep the demands for coordination within reasonable bounds. One very effective way for accomplishing this is to organize programs as hierarchies of closed subroutines, and to limit the transmission of control information to hierarchical lines. A rather different device, with which experiments are now being made, is to supplement the subroutine structure with a public pool of information, the blackboard, to which all routines can contribute and which all can read. These two approaches are complementary rather than exclusive.

5. COMPUTER NETWORKS

Let us return briefly to the topic of computer hardware. There is a good deal of current interest in the possibilities of connecting numbers of computers by communication links into large networks. One such network, the ARPA net,

which connects some thirty computers all over the United States, is already in existence. The network permits users at one location to run programs that are stored at other locations, to communicate with users at other locations, to transfer programs and data from one location to another: in short, to treat the network as though it were a single enormous computer.

From the earlier discussion of multiprocessors and parallel processing it should be clear that no sharp line can be drawn between a single computer and a system of computers communicating through a network. Hence, the network idea really represents the next stage in assembling large and complex computer organizations, and we now have systems that consist of a number of computers linked together in such a network, each of those computers being, in turn, a multiprocessor containing several central processors as well as input devices, output devices, and other specialized components. The nature of the linkage, however, is somewhat different at each level.

At the multiprocessor level, the system is generally designed so that the user need not be aware that he is connected to anything but a single computer. The allocation of resources – the decision, for example, as to which processor will handle which jobs – is generally handled centrally and automatically, although there may be provisions whereby a user can request particular facilities such as tape drives and printers to be temporarily assigned to his use by the central system. In any event, the multiprocessor is characterized by a great deal of central control and automatic management.

At the network level, in its ARPA implementation, the system looks more like a federation than a centralized organization. Each user enters the network through one of the component processors, and must take explicit action to attach himself to (and later detach himself from) the other processor with which he wishes to be connected. Once the attachment has been accomplished, he is treated more or less identically with every other user of the system to which he has attached. Control and allocation of resources remain completely decentralized: the host system may decide whether or not it wishes to accept a request for attachment and what facilities it will grant to the applicant. Of course, a network could be organized in quite different ways from the ARPA net implementation. What that implementation illustrates is that quite conventional organizational problems – in this case, the assignment of authority for resource allocation – arise within any such network system.

The ARPA net also illustrates another important point about the structure of a hierarchic system. Coordination problems in hierarchies are generally kept within bounds by the fact that the rate of interaction *between* components tends to be one or more orders of magnitude less than the rate of inter-

action *within* components (that is, between components at the next lower level of the hierarchy). (The significance of this gradient in interaction rate for the dynamic stability of the system is discussed in Simon, 1969, Chapter 4). In the ARPA net, instructions to attach or detach constitute only a tiny fraction of the events with which a component processor deals. If we think of such events as changes in the system structure (user moves from one subsystem to another), then, because these events are rare, the system structure is highly stable and each of the subsystems can operate 'almost always' as though it had a fixed structure and were isolated from the others.

In the design of computer networks, there are a number of hardware problems at the microlevel, involving the detailed coordination of the message-sending and message-receiving processes, that have no direct counterpart (at least, at the same level of precision) in human organizations. All of the larger issues of network organization, however, are quite familiar to anyone who has been concerned with problems of human organization.

6. THE ORGANIZATION OF MAN-COMPUTER SYSTEMS

Two conclusions emerge rather clearly from the foregoing discussion. First, contemporary and emerging large computer systems are indeed complex organizations, and designing them raises all of the questions that are familiar to us from the design of human organizations. Second, as computers come to play a larger and larger role in business, governmental and educational organizations, it becomes important to develop the art of building large composite systems in which men and computers live side by side in very close symbiotic interaction.

I will not add to an already lengthy discussion by trying to set forth the principles of design for such organizations, for I am not sure that I know what those principles are. It does not appear, however, that they will be very different from the principles that govern the design of effective human organizations of the traditional sort. In particular, it would seem that hierarchy (of communication and resource allocation as well as formal authority) will continue to play a central role in the structures, and that many of the design problems can profitably be viewed as problems of devising a system for resource allocation.

Computing systems have the advantage, for analytic purposes, over human systems that we can define rather precisely how they are put together and how they operate. Hence, as we begin to develop explicit theories of organization

for application to computer systems, we can expect organization theory to take on a definiteness and exactness that it has not always had in the past. At the very least, terms like 'centralization', 'authority', 'coordination', which we have sometimes tended to use in rather vague senses, are likely to acquire more precise meanings. Hence, we can expect a beneficial flow of ideas and insights from the new field of computer architecture back to more traditional areas of organization design. This valuable flow (which can be a two-way flow) will only take place, of course, if the researchers and designers in each of these fields of investigation keep a watchful and interested eye upon developments in the other.

7. REFERENCES

Bell, C. G., and A. Newell, *Computer Structures: Readings and Examples*. New York: MacGraw-Hill, 1971.

Colodny, R. (ed.), *Mind and Cosmos*. Pittsburgh, Pa.: University of Pittsburgh Press, 1966.

Erman, L. D., and V. R. Lesser, *A multi-level organization for problem solving using many, diverse, cooperating sources of knowledge*. Unpublished manuscript, Department of Computer Science, Carnegie-Mellon University, March, 1975.

Newell, A., et al., *Information-Processing Language-V Manual*. Englewood-Cliffs, N. J.: Prentice-Hall, 2nd ed., 1964.

Simon, H. A., Scientific discovery and the psychology of problem solving, in *Mind and Cosmos*, Chapter 3, 1966a, pp. 22-40.

Reflections on time sharing from a user's point of view, *Computer Science Research Review*, 1966b, 43-51. Computer Science Department, Carnegie-Mellon University.

The Sciences of The Artificial. Cambridge, Mass.: M.I.T. Press, 1969.