Systems architecture in transition—An overview

by H. Lorin

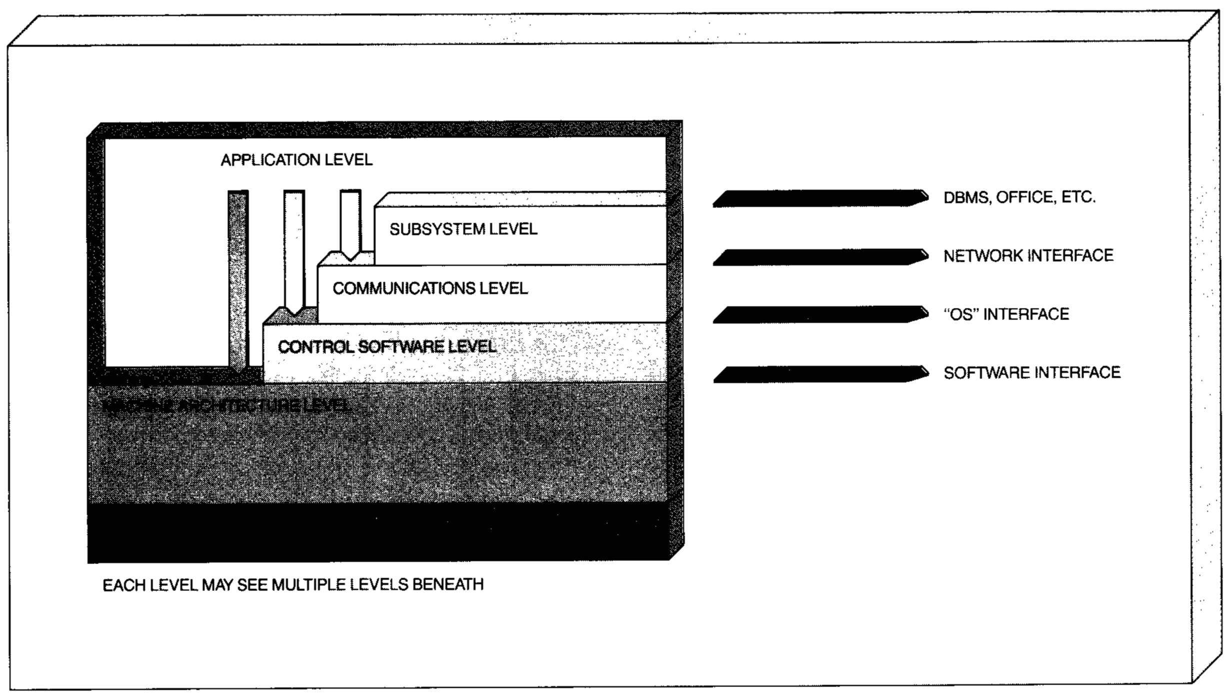
Systems architecture refers to the distribution of function and control among elements of a system. It is primarily a structural concept that includes the original meaning of the word architecture in the form of "processor architecture." This paper undertakes to describe topics of current interest in the evolution of computing structures. It discusses various unit structures that may emerge as the economics and capabilities of technology relax more and more constraints. Of particular interest is the internal structure of a central computing complex, the relation of computing elements and I/O elements, and the maturity of the I/O elements. The paper also suggests that the structures found within a single computing unit may be realized across larger elements more widely dispersed. Hardware and software issues are addressed.

In the computer industry, "architecture" was first used to mean the view of a computing system as seen by a programmer or automated code generator. Thus, the addressing scheme, register population, and instruction set are architectural ideas, whereas cache, instruction pipeline, microcode, and circuit densities are design or implementation concepts. The output of an architectural effort is a document that imposes requirements on a design. The output of a design effort is a mapping of the architecture into a technology in order to achieve stated price/performance goals for a model of the architecture. Thus, a program-compatible product line (various models each of which respond in the same way to a list of operation codes and addresses) can be defined at different price/performance levels.

Used without any adjective, the word "architecture" still has that meaning for many. However, there has always been some informality in the use of the word, and informal usage often blurs the distinction between design and architecture. In addition, we have begun to use the word with various adjectives in phrases such as "I/O architecture," "communications architecture," and "systems architecture." All of these phrases share some intent to distinguish a set of interfaces and constraints from a technology mapping. They shift the meaning of the word "architecture" away from its initial processor-oriented view of a computing system. They suggest a concern with some form of system "structure": the functional relationships between logical elements of a system or of a subsystem.

Changing uses of a word suggest changes in focus and a reordering of interests and issues. Recently we have been primarily concerned with the structure of systems—flows of data and control points of storage, and intelligence. At the same time we are broadening our notion of what a system is. We now mean something more than a single collection consisting of a processor, memory, and I/O devices. The word "system" is frequently used to mean collections of interconnected computers at various geographical distances cooperating at different levels of intensity. This paper discusses some topics of "systems architecture" in its broader meaning, although we will not exclude some concepts of "architecture" in its original meaning, topics that one might now call "processor architecture." The intent is to bring to the attention of the reader some of the considerations and points of interest relative to the evolution of computing systems and to suggest various views of the nature of that evolution.

Figure 1 System layers (n, $n-k$ hierarchy)



Elements of systems architecture

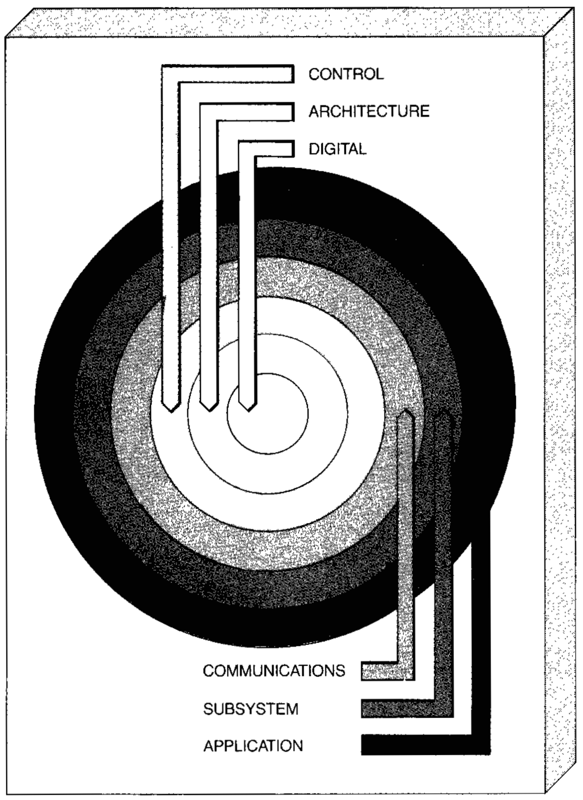
The concerns of systems architecture focus on the definition of major system elements, hardware and software, and on the flow of control and information among them. Considerations include:

1. The internal structure of a central electronic complex (CEC) involving the number of processors, whether the processors should be homogeneous or heterogeneous, whether address spaces should be shared among processors, whether there should be single or multiple sites of system control.
2. The essential structure of an I/O subsystem and its relation to the CEC. At issue is the nature of system changes implied by an increased ability to site intelligence and storage in an I/O subsystem, combined with considerably faster interconnection technology.
3. The relation among the "nodes" of a computing system connected at various distances. Within the limits of local-area networking (and beyond those limits) what degrees of interdependency, load sharing, backup, and cooperation are achievable? What views of an "aggregate" computing system should be presented to a community of users?

Another dimension of systems architecture (unfortunately blurring the edges between architecture and design) concerns the way particular functions should be delivered. A classical aspect of this dimension is the "hardware-software" interface. To what extent should the processor and systems architecture be extended to include basic functions and structures inherent in now-mature operating systems? Should certain control blocks be architecturally defined as basic data types? Should certain basic functions such as QUEUE, LIST, and SORT be included in the systems architecture? There is also a "hardware-firmware" interface. Should function taken down from traditional software be developed in various forms of microcode or "hardwired" in some way?

Figure 1 shows the traditional layers of a computing system. The hardware-firmware question asks how "fat" the underlying digital level should be. The hardware-software question asks how "fat" the microcode level combined with the digital level should be relative to the software levels. There are similar issues within the software levels, related to the distribution of functions among them. What are the functions of a basic operating system as opposed to a communications subsystem or a data base management system? Figure 1 suggests an important attri-

Figure 2 Classical layer representation (n, $n-1$ hierarchy)



bute of real systems that is often hidden by representations like Figure 2. Figure 2 suggests, for example, that Level 5 "sees" only Level 4. That is, Level 4 is the "interpreter" of the functions at Level 5. Such a system would be a perfectly encapsulated hierarchy in which the nature of any level beneath Level 4 would be of no concern to Level 5. In contemporary systems structure, however, multiple lower levels are visible to a higher-level function. Thus, an application program is constructed from calls to the services of a subsystem, but it may also make service calls to the operating system. Of course, each time a function is represented as a machine instruction, the application sees the instruction set and addressing schemes of the underlying processor architecture. The full range of compatibility, portability, and cohabitability issues of current computing systems, hardware and software, derive from the multilayered views that are available to end users, application programs, subsystems, etc.

Processor architecture, of course, is directly relevant to the hardware-software issues. Currently we express preferences between Reduced Instruction Set Computers (RISC), Complex Instruction Set Computers (CISC), and High-Level Language Architecture (HLL). In this area there is sometimes more passion than crisp definition. A full understanding of the various trade-offs in the processor architecture area involves an appreciation not only for architecture, but for design methodologies, underlying technologies, and compiler methodologies as well as for the costs of hardware and software production in different technology intervals. A "good architecture" (processor architecture) should be easily and efficiently represented in the technology, and should provide for simple, efficient compilation. But there may be other considerations whose importance becomes greater over time. A serious problem is that we lack a universally accepted metric either for a "good architecture" (although many have been proposed) or for the "complexity" of a compiler, and we seem certainly unready to deal with issues about the quantitative impact an architecture should have on the cost of producing and maintaining code, etc.

The remaining sections discuss issues of the central electronic complex structure, I/O structure, and hardware-software-firmware, with the intent of suggesting what the current trends are, what the limits on current trends may be, and what mature computer systems may look like in the future. Nowhere is a statement of the direction or intent of a particular vendor intended.

Unit structure

I use the word "unit" for lack of another word to suggest what we formerly called a "computer system." The word "node" is sometimes used for this, but Systems Network Architecture (SNA) purists use that word with a somewhat different meaning. A fundamental problem is that the characteristics of this "unit" are quite variable. Some try to define it as that collection of resources which falls under the control of a single operating system. But this definition runs into the phenomenon of processors within a single frame that have individual operating systems. In some years we may think of all of the computing power at a single establishment (a site of business activity for an enterprise) as effectively a single computational node, regardless of its physical disbursement around the establishment and regardless of the number of "local operating systems."

Let the word "unit" represent the aggregation of processor(s), memory, and I/O devices that we have been used to thinking of as a "computer system." The introductory section suggested that we have some choices to make about the number and type of processors in this unit and the relationships among them. There is a wide range of opinion, prototypes, experiments, and literature about the virtues and uses of various unit structures. The measurements of quality involve issues of raw performance, availability, configurability, and evolvability.

Figure 3 shows a few possibilities for the internal structure of a computing unit. Part A shows a symmetrical tightly coupled multiprocessor with the following essential features:

1. All processing elements are identical in architecture and design. (They are model-identical.)
2. All processing elements have full addressability over the addressing range of the system. They get service from memory at effectively the same rate.
3. There is a single operating system responsible for the resources that are "global" across the context of the system. In a common version of this structure, any processor may execute the algorithms of the single operating system upon shared system status data and resources.

Part B shows a system in which the processor set is not homogeneous. There is a group of "general-purpose processors," but there is also a group of functionally specialized processors that are used for such things as I/O operations, communications management, system scheduling, and vector manipulation. Certain parts of the operating system and subsystems, as well as certain application programs, or parts of application programs, may be executed on the specialized processors.

A system like the IBM 3090 lies somewhere between these models. It is not exactly symmetrical, but its asymmetry is limited to I/O and vector processing. And that itself raises an observation that there is another dimension to structure. We may think of the general-purpose processors and the special-purpose processors as full cooperating peers, or we may recognize that some are more equal than others and see the structure as hierarchical. The I/O processors are in some sense subservient to the general-purpose processors or (and more interestingly) the other way around. Both systems shown in Parts A and B of Figure 3 have an important property of memory-shared systems. That property is sometimes called

Figure 3 Computing unit structures. (A) symmetrical multiprocessor, (B) functional multiprocessor, (C) multicomputer

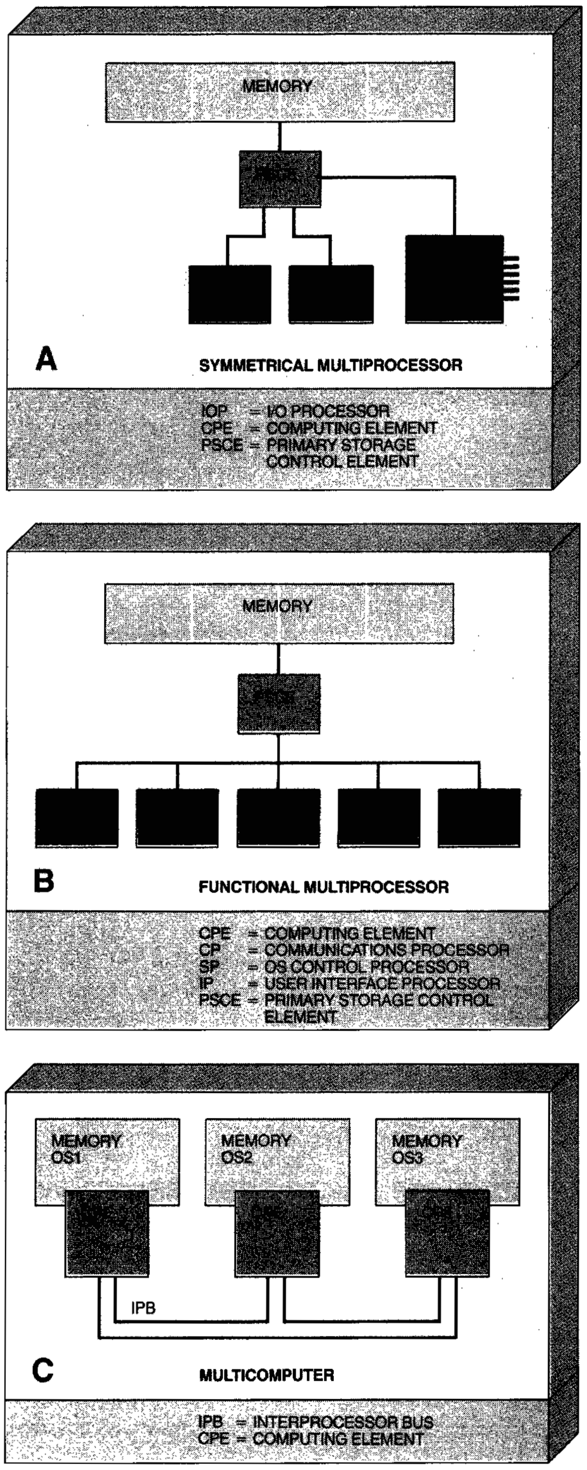
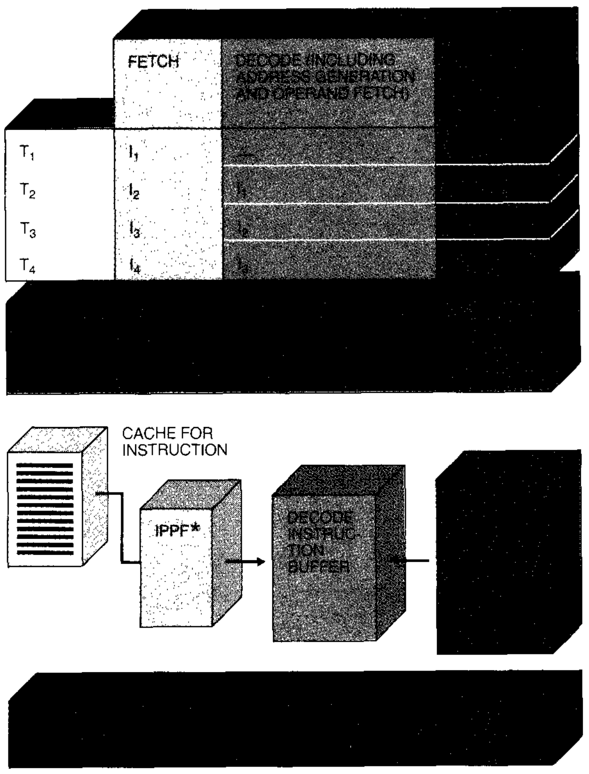

MEMORY
A
SYMMETRICAL MULTIPROCESSOR
IOP = I/O PROCESSOR
CPE = COMPUTING ELEMENT
PSCE = PRIMARY STORAGE CONTROL ELEMENT
MEMORY
B
FUNCTIONAL MULTIPROCESSOR
CPE = COMPUTING ELEMENT
CP = COMMUNICATIONS PROCESSOR
SP = OS CONTROL PROCESSOR
IP = USER INTERFACE PROCESSOR
PSCE = PRIMARY STORAGE CONTROL ELEMENT
MEMORY OS1
MEMORY OS2
MEMORY OS3
IPB
C
MULTICOMPUTER
IPB = INTERPROCESSOR BUS
CPE = COMPUTING ELEMENT

Figure 4 Pipeline structure



"space-time coherence." In a space-time-coherent system, it is possible to represent the status of a complete system at a particular time, and it is also possible to shift resources to processes and processors with great efficiency (relative to systems that do not have this property). The property comes from the shared memory that enables the single operating system to manage and inspect all resources within the context of a single system. As we will later see, this property of "space-time" coherence is what is particularly lacking in "distributed systems."

Part C of Figure 3 shows a system without shared address space and with consequent multiple copies of operating systems with associated subpools of resources. Processes running on one processor communicate on a message basis with processes running on another. Processors may be homogeneous or heterogeneous. This system is a form of "distributed" system (although it may exist within a single physical frame) that lacks the property of space-time coherence. This means that it is very difficult to get an accurate picture of the status of all resources at a single point in time. From any point of enquiry it would be necessary to poll other processors to collect status information. While that was being done, the status of previously polled processors would be changing. Thus it would be difficult to balance loads across such a system and difficult to do global performance analysis. The property of space-time coherence could be afforded such a system if the interconnection bus were fast relative to the processors themselves and/or if a single point of systems control were established. Perhaps one shared control memory with or without an associated control processor, for example, could bring a distributed system closer to the characteristics of a memory-shared single-operating-system design.

These three suggested unit structures represent a somewhat compressed view of the variations possible within a single unit. However, the view gives us enough of a sample to focus on the following three major questions:

1. What should the "granularity" of the processors be?
2. What is a proper split of work in certain heterogeneous structures?
3. What are the proper relationships between the processors and memory (memories)?

Of course, the measures of goodness are performance, price/performance, availability, and configurability, and there are examples and counterexamples of preferability under different workloads and assumptions of technology, etc. If this were not true, there would by now not be so many contending approaches.

Granularity. Granularity refers to the speed and capacity of an individual processing element. A small-grained system achieves speed with a large population of slow or simple processors. One hundred or one thousand or one million processors might combine together to deliver aggregate service to a workload. A large-grained system has a smaller number of processors, each of which is relatively fast. The ultimate large-grained system is the uniprocessor.

The question of granularity can be seen as an issue in how to use circuits. Is it "better" to create a large population of small processors or to develop systems with a small population of more capable processors? "More capable" may mean more powerful instruction sets, more elaborate instruction pipelines, specialized execution units (e.g., vector). (Figure 4 shows the basic concept of pipelining.)

There is a systems trade-off between the complexity and power of a single processor and the population of processors. An aspect of this trade-off is the use of technology or the use of design to achieve speed in a larger processor. For example, the internal structure of the IBM 3033 computer system is quite complex, involving somewhat sophisticated pipelining and tracking of branch histories. The internal structure of an IBM 3084 system is quite simple by comparison, while the IBM 3090 system reintroduces some of the more sophisticated design of the IBM 3033. One can view this sequence, in an abstract way, as an issue in granularity. To build a faster system, should one stress more processors or speed them up? If one speeds up processors, should one use circuits for elaborate designs or try to simplify designs in a faster technology?

In a broader context, we are faced with systems structures that may now include 500 or so processors and theoretical systems structures having in excess of a million computing "elements." What role will such small-grained systems play in the development of computing? Can one expect that the mainstream commercial computing systems of the future will consist of large populations of small-grained machines as part of their essential structure? Will speed be achieved by massive parallelism on small units of work? The constraints on the efficiency of large populations of processors are (1) memory interference between processors, resulting in delays at the memory interface for data and instruction reference, (2) the overhead of processor-to-processor communications, (3) the overhead of creating processes relative to the expected duration of a process, and (4) the creation or discovery of proper workload structures to achieve populations of concurrent tasks or subtasks.

Given these constraints and the state of the art in overcoming the fourth constraint, one may reasonably project that

1. Mainstream commercial systems will have small populations of large-grained processors for the foreseeable future. Tightly coupled multiprocessors with populations of 2 or 4 processors will be characteristic, and speed will be designed by balances of more elaborate processor design and faster technology.
2. The small-grained systems will develop along two lines: as stand-alone specialized processors and as functional enhancements to general-purpose large-grained processors. Tremendous computing power is needed for speech recognition and for artificial intelligence function to support natural-language interfaces. A large population of small processors is a frequently encountered model of computing in such applications. There are machines, for example, with up to 65000 processors that seem directed to these areas. Since interface functions will likely occur at workstation levels in the future, it may be that network-connected personal computers will be the first of the breed of highly parallel machines.

Small-grained functional components may be added to systems in a more integrated manner to support vector operations for example, or data base functions that might profit from massive parallel search capability. An advantage of the small-grain approach to "supercomputers" is the potential configurability and flexibility. It may be that there is no such thing as a general-purpose supercomputer. Some scientific problems are processor-intensive, some are memory-intensive, and still others are I/O-intensive. In addition, there is wide variation in algorithmic structure, control flow, and data flow from one application to another. Small-grain approaches may permit supercomputers specialized according to application to be built out of different configurations of processor populations with specialized data and control flow relations among them.

The structure of large-population, small-grained processor systems has received much attention over time. Various models of peer and hierarchic structures have been promulgated. One model of general interest, shown in Figure 5, is the "cube." In a cube organization, we consider the system to consist of an n-dimensional cube with processors or process/memory units at the vertices. A cube organization in n dimensions can have $N = 2^n$ processors and $n * 2^{n-1}$ interconnection paths. Thus, a three-dimensional system can have 2^3 (8) processors and 12 interconnections. This organization is one of a class of network topologies that addresses the problem of interconnecting large populations of elements while avoiding an unwieldy interconnection network whose paths increase as the square of connected elements. In general, network topologies for large population systems attempt to achieve something in the area of $N \log N$ interconnection pathways.

Heterogeneity. Multiple-processor systems need not be homogeneous either in function or in underlying architecture and design. Across the context of a unit of computing there are naturally heterogeneous ele-

Figure 5 The cube

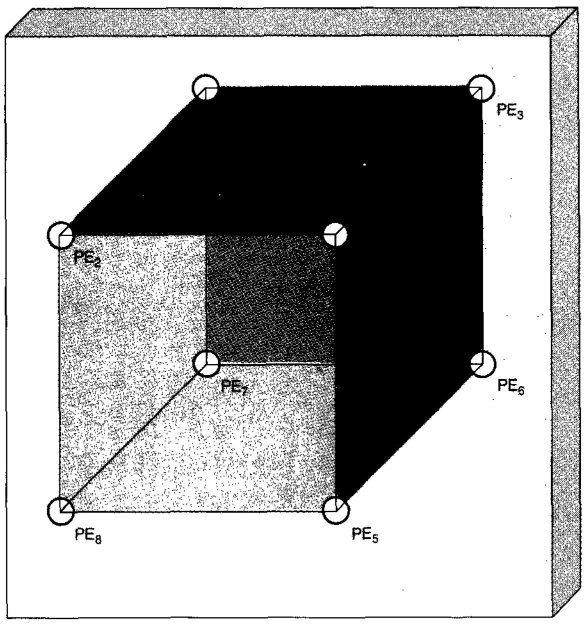
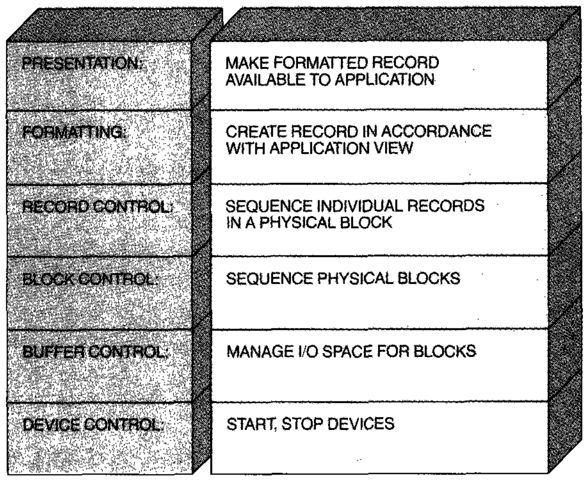


Figure 6 Layers of I/O function


PRESENTATION:
MAKE FORMATTED RECORD AVAILABLE TO APPLICATION
FORMATTING:
CREATE RECORD IN ACCORDANCE WITH APPLICATION VIEW
RECORD CONTROL:
SEQUENCE INDIVIDUAL RECORDS IN A PHYSICAL BLOCK
BLOCK CONTROL:
SEQUENCE PHYSICAL BLOCKS
BUFFER CONTROL:
MANAGE I/O SPACE FOR BLOCKS
DEVICE CONTROL:
START, STOP DEVICES

ments—processors, controllers, devices, etc. Even within the processor population one may have I/O processors, computational processors, scheduling processors, compiling processors, etc. The central problem of heterogeneity is how to divide function among the elements.

Consider I/O operations. In Figure 6 we show some basic I/O functions. Each line represents a potential split point. We can bundle all of these functions onto a single processor, or we can group them across a computational element (CPE) and I/O processor (IOP) in various ways. We might even dedicate an element to each of the layers. Part A of Figure 7 shows the CPE executing all function except the lowest level of device support. Part B shows the CPE and I/O engine as cooperating peers. In such a structure, the CPE might enqueue I/O requests in a shared memory area (or send messages to the I/O engine). The I/O engine inspects the queues at its own discretion and undertakes work when it finds requests. When an I/O event has completed, the I/O engine enqueues completion packets that the CPE discovers at times that are convenient for it. Thus, I/O ceases to be a disruptive and interruptive phenomenon for the CPE. This condition is especially important in pipelined or cache-oriented CPEs, where longer intervals of predictable processing have a large payback.

Memory sharing. Memory sharing is a topic discussed in terms of performance, security, or availability. As regards availability, high-availability systems have been developed using both approaches so that the issue does not seem critical in this regard. From the security standpoint, other architectural features, such as the structure of the operating system and the basic addressing scheme, seem more significant. So the question of memory sharing seems primarily related to performance in the presence of different types of workloads.

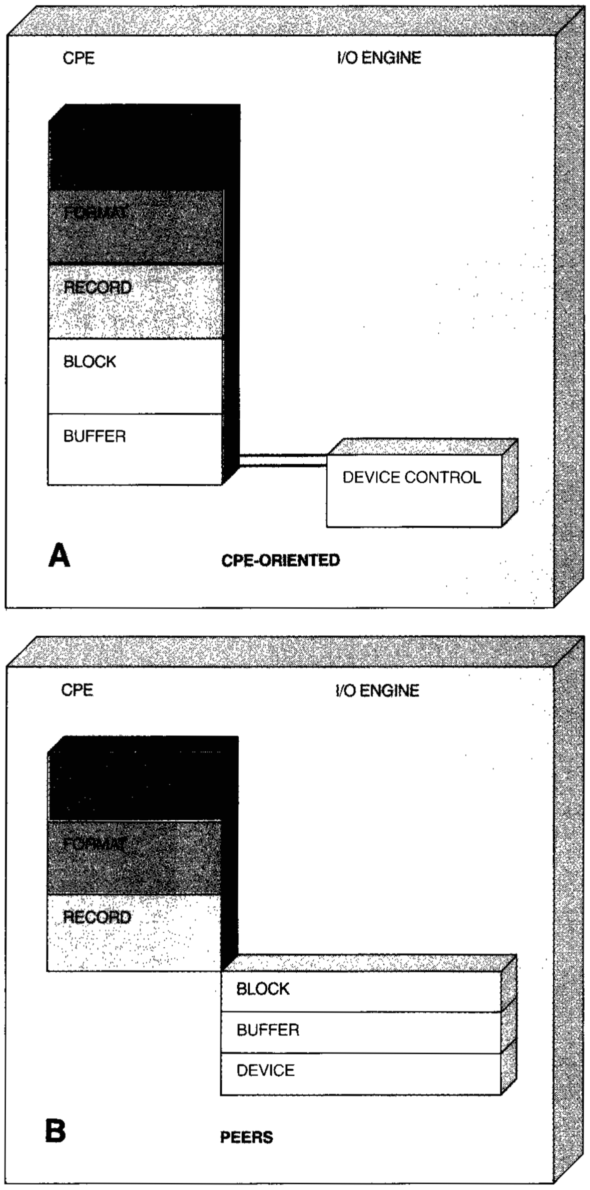
Much depends upon the nature of the workload and the degree of desired cooperation between processors. With multijobbing, multitasking, or other contexts where the interaction between processes is fairly "loose," nonmemory-shared systems may be quite effective. Where processes do not cooperate, or where they need only synchronize one another from time to time, the message burden is not severe. Algorithms for "explicit" synchronization exist for both memory-shared and nonmemory-shared systems. (A process wishing to acquire a resource for change must request a LOCK on the resource for the interval in which it is in the "critical section." The critical section is that part of the code where lack of exclusive control over a resource may bring about an improper result.) The support of LOCK synchronization without some point of shared memory is not necessarily more burdensome than with shared memory.

There are some who feel that when concurrency and close cooperation are to be achieved within the structure of a single computational program, memory sharing is the preferred approach. This approach is preferred because the burden of sending messages across addressing boundaries may become a performance bottleneck. A problem with memory-shared systems, of course, is the memory interference problem. Processors making references to shared memory (and to the interconnection network that associates the processor elements with memory elements) get in one another's way. An area receiving important attention in small-grained memory-shared systems is the topology of the interconnection network between memory and processors. The design of memory-interconnect topologies that minimize pathway count as well as minimizing interference delays is a fast-developing area. It is necessary to achieve the performance of full crossbar networks with considerably fewer pathways in the network. We saw how the "cube" approaches this problem. However, it is necessary to go beyond this approach and to do such things as recognizing intersecting and duplicated references, and reducing the number of loads and stores, etc. Some developers design intelligence into the memory-interconnect network with more circuits in this part of the system than in individual processors or populations.

It was mentioned previously that high-availability systems may or may not share memory. In the IBM family of computers, high-availability versions of the Series/1 may be built without memory sharing, whereas the System/88 is a memory-shared system. A high-availability system, unlike a parallel processor, tends to minimize interdependency among processors since increased interdependency reduces availability. (The probability that all processors will be down is lower in an eight-processor unit than in a two-processor system. However, so is the probability that all processors will be up.)

A third concern sometimes mentioned in connection with memory sharing or nonsharing is security. It is necessary to show that barriers can be built between the virtual address spaces of operating processes. Given a certain address space, a process cannot forge access to another address space. Given a portion of an address space, a process cannot acquire access to an unauthorized portion. Physical isolation of processes within the context of their processors has been promulgated as one way of achieving such isolation. This isolation would surely be the case in a network where each unit had a special class of data, and the model is sometimes proposed for processors within a frame. However, emerging security requirements

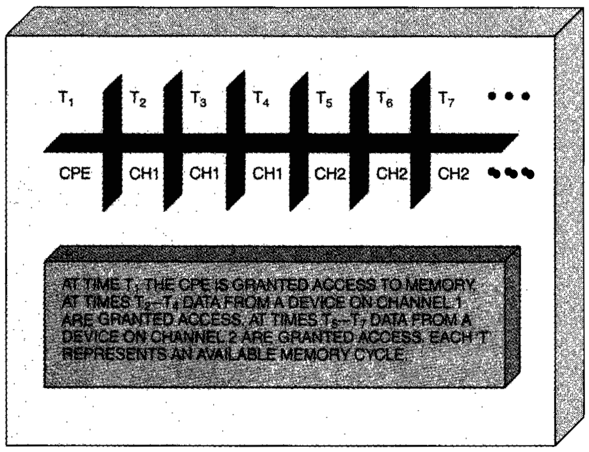
Figure 7 **Relations between CPE and I/O function. (A) CPE-oriented, (B) peers**



may require that the isolation of virtual address spaces be achieved in memory-shared models which have data of differing sensitivity within a single unit context. This scheme suggests that the security issue is one of addressability control and not of physical sharing.

What links all of the structures that we have discussed—homogeneous, heterogeneous, with or with-

Figure 8 Memory interference, CPE and I/O operations


T_1
T_2
T_3
T_4
T_5
T_6
T_7
CPE
CH1
CH1
CH1
CH2
CH2
CH2
AT TIME T_1 THE CPE IS GRANTED ACCESS TO MEMORY. AT TIMES T_2–T_4 DATA FROM A DEVICE ON CHANNEL 1 ARE GRANTED ACCESS. AT TIMES T_5–T_7 DATA FROM A DEVICE ON CHANNEL 2 ARE GRANTED ACCESS. EACH T REPRESENTS AN AVAILABLE MEMORY CYCLE.

out memory sharing—is the need for internal buses that are fast enough to take the load of interprocessor interaction or processor-to-memory interaction without becoming a bottleneck. Fifteen years ago the objection to the multicomputer organization was the time it would take for one processor to communicate with another processor relative to the time it would take to execute instructions itself. As internal buses became fast, this issue relaxed and the Multiple Instruction Multiple Data (MIMD) machine became mainstream design. In a later section we will suggest how extensions of this phenomenon of relaxation of communications as a system constraint may impact system structure at higher conceptual levels where we are concerned with networks.

Input/output

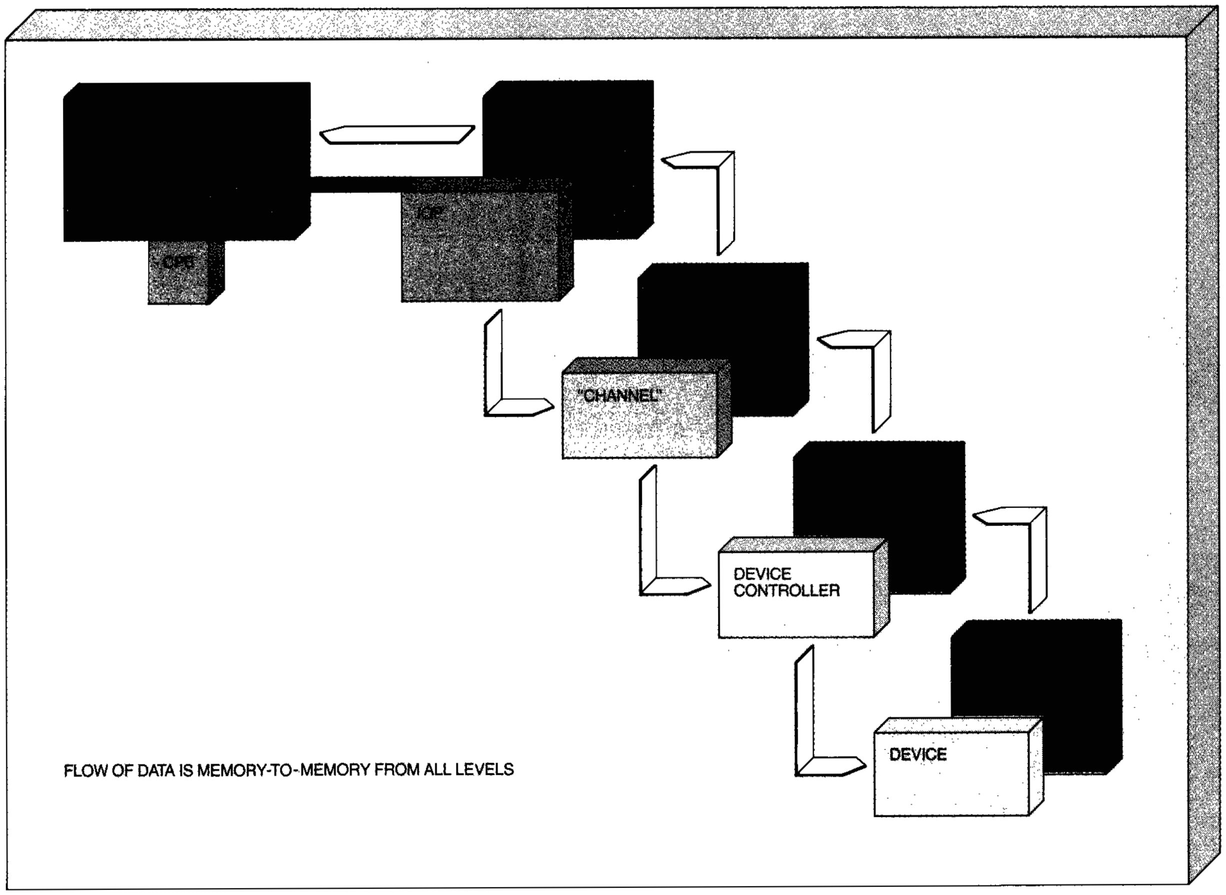
The second aspect of systems architecture mentioned in the introductory section is the relation of computational processors to I/O function. I/O operations have always been a problem because I/O devices characteristically operate at speeds that are orders of magnitude slower than the speeds of processors or memories. Therefore, much attention must be given to maintaining an adequately rapid data flow to and from the computing elements.

The problem has been partially solved by providing parallel pathways into the memory. By reading and writing multiple devices on multiple channels simultaneously, a maximum aggregate data rate can potentially be achieved. This maximum rate is the rate at which the memory, given the requirement for also serving the processors, can respond to I/O transfer. One problem with this approach is illustrated in Figure 8. The line chart shows instances of granting memory access to a processor and to elements of the I/O subsystem. Because of memory interference between the I/O subsystem and processor, the speed of the processor is effectively reduced. Some units have attempted to use independent memory banks to alleviate this problem. The hope is that the processor will work in one memory bank while I/O function is flowing into or out of another. In a complex multiuser environment, however, it is essentially impossible for an operating system to avoid processor-I/O memory contention. It is possible that some form of specialized data caching may be useful to address this problem.

The ultimate goal of I/O design is to make data transfer appear to the processor as if it were occurring at memory speeds rather than at device speeds. One method of achieving this concept is deep buffering throughout the I/O elements. Large memories may be placed at different stages of the I/O flow, and I/O operations move as memory-to-memory transfers through these stages. Figure 9 shows this concept. With sufficient intelligence associated with the stages it may be possible to build macro analogs of processor caches in order to significantly increase the data rate. An associated architectural notion might be to extend the concept of uniform addressability (a processor uses the same address whether an element is in cache or primary memory) throughout the I/O buffers.

In a previous section, I discussed the split of work between the computational elements and I/O elements of a structure within a single frame. Important advances in computing come from the scaling up of structures because of the relaxation of interconnect constraints. Structures within a "processor," such as an instruction preparation unit fronting a family of execution units, are replicated by the structure of the Job Entry Subsystem (JES) with a Support Processor scheduling Main Processors. (The phenomenon of transferring structures from one level of system abstraction to another goes in both directions. We see, for example, the idea of "loose coupling," originally associated with channel interconnection, now occurring on a single board.) Let us look again at the structure in Figure 6, but this time at a higher level of system abstraction so that it looks as depicted in Figure 10. Here we show much larger functional granules, more similar in size to major software components. Figure 11 shows some possible splits of

Figure 9 Deeply buffered I/O function



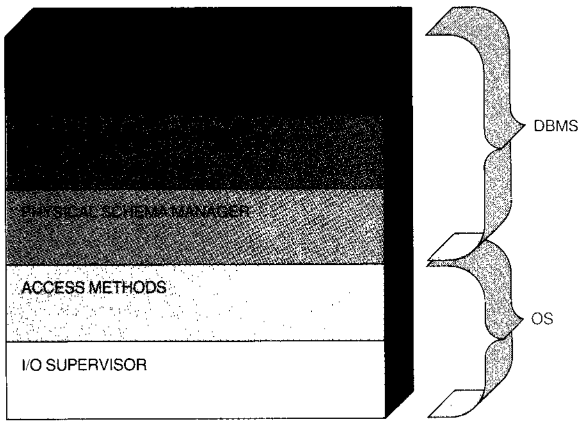
work between the processor component and the I/O component of a computing unit when we cross frame boundaries.

In Part C of Figure 11, perhaps most interesting of the three shown, we have the "Data Engine" as the heart of the system (from the view of function and control), and the "processor" as only an application host. This concept is that of the "Data Node." In its most extreme form, the Data Node:

1. Is the repository of all enterprise-level data.
2. Is the single point of change for any data element.
3. Runs no application code, but responds to "scripts" from application processors requesting changes to data.
4. Imposes a security screen between its data and application requests.
5. Provides data locking and synchronization.
6. Provides translation from different data models to a uniform physical view.
7. Forms and transmits application-specific views to applications as they request them (but not for the purpose of update).
8. Applies "integrity" rules to data so that no change request from an application can cause an operation that violates the integrity rules.

The search for an efficient instrument to perform these functions (or some of them) has been underway for over a decade. We search for an engine capable of fast search, excellent manipulation of directory and index structures, good recovery, etc. The search has led to a number of quite different architectures and structures for the Data Node. One candidate, for example, is in fact a small-grained processor population capable of massive parallel searching. Another approach is the enhancement of a general-

Figure 10 Software I/O layering



purpose processor architecture so that it performs certain functions more efficiently. The search for the optimum data engine encounters the same problem as the earlier search for a perfect sorting engine. So much of the function requires the power of a general-purpose architecture that only marginal contributions to system throughput can be made by optimization of unique elements. What seems to be required is an ability to configure instances of the general-purpose processor into an effective data engine. A way to do this is to provide architectural enhancement (much as one might do for vector manipulation) in a general-purpose processor and to refine elements of the I/O pathway. In this way, systems could be configured from populations of "application elements" and "data elements," where the data elements were those with the enhanced architecture and refined I/O elements.

This structure can be upscaled to a networking concept. Figure 12 shows an "establishment"-level system where application units and data units are configured around a local-area ring. Such a structure would have the advantages of isolating data from application malfunction and of providing multiple data repositories accessible from a family of application hosts. In such a structure as Figure 12 we begin to lose some conviction about the distinction between a single computer unit and a network. With the distribution of intelligence and function into the I/O subsystem, and with each application element talking to a data element, our concept of a single system is considerably extended. The availability of a good local-area interconnect facility allows us to scale some structures common to a single frame upward to an establishment (a single site of business for an enterprise).

Systems and system structures

The last section ended with the observation that fast, reliable, and inexpensive interconnection technology

Figure 11 System structures. (A) channel style, (B) peer style, (C) data engine style

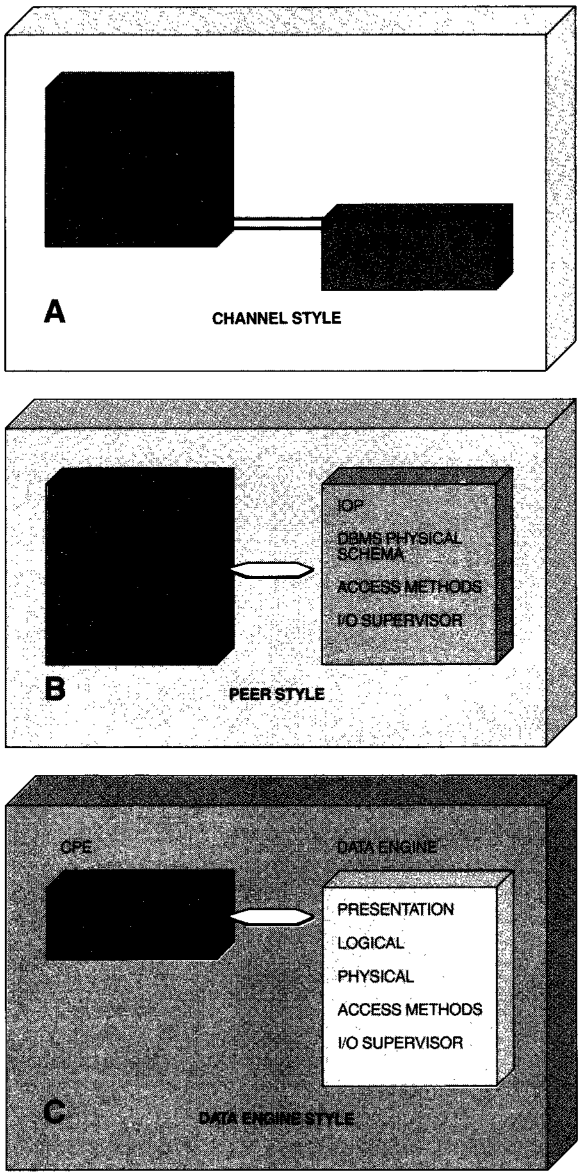
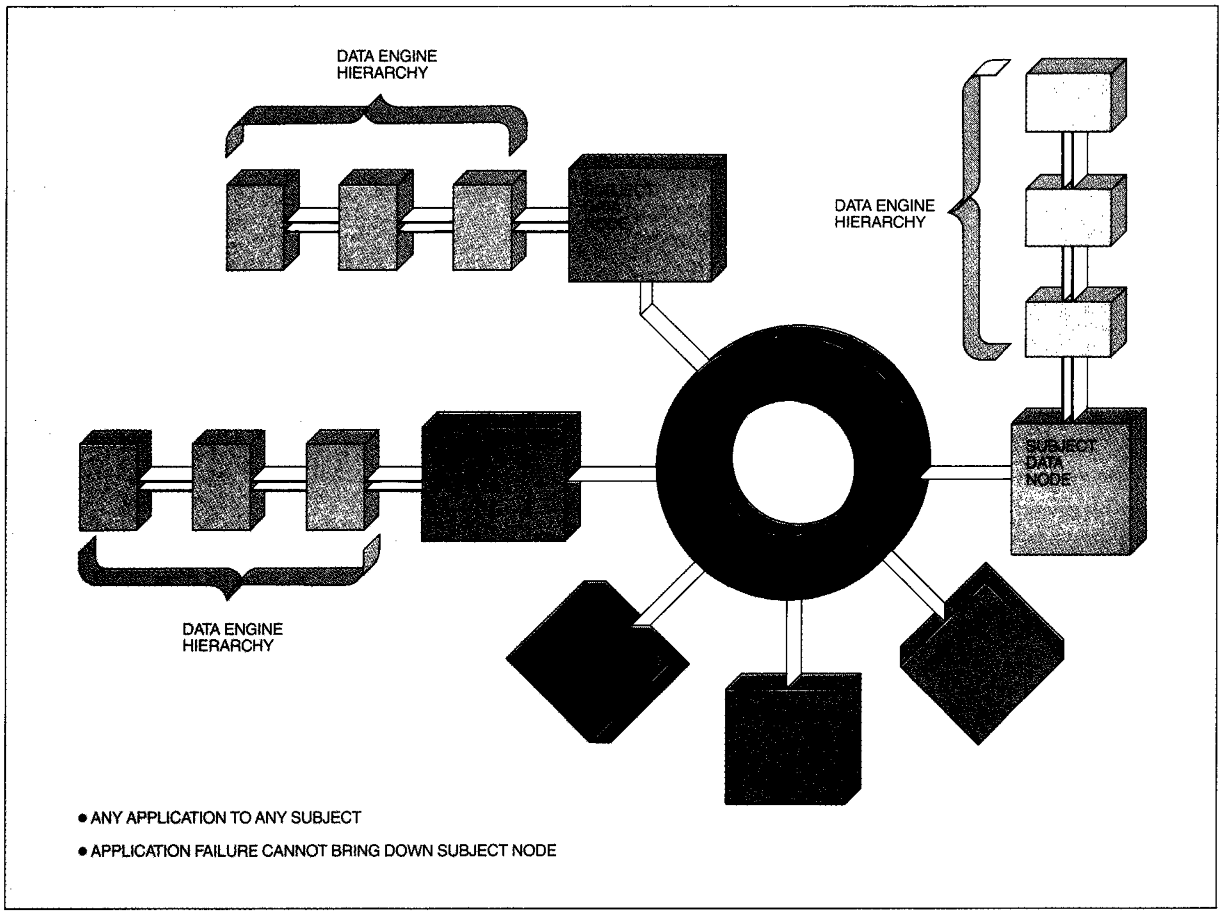


Figure 12 Application and data nodes


DATA ENGINE HIERARCHY
DATA ENGINE HIERARCHY
SUBJECT DATA NODE
DATA ENGINE HIERARCHY
• ANY APPLICATION TO ANY SUBJECT
• APPLICATION FAILURE CANNOT BRING DOWN SUBJECT NODE

forms the basis for thinking of aggregations of computing units as a single system. The variety of structures possible within a frame become possible over a wider geographical area. In effect, a population of units such as IBM 3090s, System/38s, IBM 43XXs, and varieties of IBM Personal Computers may combine in variations of data and control flows with the increasing sense that they are a single system.

One important impact of the emergence of larger system contexts may be on the structure of software. We may be forced to reconsider our ideas of what constitutes a "single" operating system. This reconsideration must have end-user and programmer aspects as well as operational aspects. Is it desirable that all end users have a single view of a system through standard end-user interfaces? Is it desirable that all application programmers have a single view of a system? Is it desirable that there be a single and unified view of operational control? Or is it better to allow multiple views that are functionally mapped into one another at deeper levels of the software structure? As an example, should we strive to present a single image of query and relational data base or should we allow multiple images that can communicate with one another across various "bridges?" This question can be best understood in terms of the levels of a system where homogeneity may exist:

1. *Implementation and design.* Each unit of the interconnected system is the same model of the same architecture running identical software at all software levels.
2. *Architecture.* Each unit of the system is architecturally identical but may differ in model. In such a system we may well find different software

environments because the operating system for the largest system cannot fit on the smallest, etc.

3. *Operating system.* The same operating system interfaces exist for operations and application development despite the fact that there are different architectures and/or models underneath.
4. *Subsystem.* The same set of interfaces and functions exists for application developers and some end users (those who deal directly with subsystem interfaces rather than working through an interface defined by an application program), although there may be different operating systems underneath.

Given the existence of multiple architectures and models, the real choices seem to be whether to establish homogeneity at the operating system level or at the subsystem level. Those who believe in a single operating system solution (represented commonly by the UNIX® community) believe that it is easiest to replicate an operating system among various architectures. With such proliferation of the operating system all higher-level software layers can be made available with little effort. Those who believe in subsystem-level homogeneity claim that the replication of subsystems across various operating systems is the best first step toward achieving system coherence. The difficulty with operating system portability is the high degree of machine-specific function that is in any case associated with the operating system level. Therefore, it may be best to duplicate subsystems or interfaces to subsystems on top of already existing mature operating systems.

The underlying phenomenon that motivates the need for more uniform software interfaces and for more function sharing across units on a network is the anticipation of dramatic improvement in interconnect facility. One provocative speculation comes from the convergence of I/O and local-area network technologies. Traditionally we have been motivated by orders of magnitude of difference in cost, reliability, and speed of I/O systems and any networking facility. Software structures have responded by separating I/O and communications function quite rigorously. But if it becomes possible, within the context of local-area networks at first, to approximate the speed of local I/O functions, we must consider once again our ideas of sameness and difference. In what way does a processor channel interacting with a highly intelligent I/O controller differ from an application processor interacting with a file server, when identical technology is involved? If a brand-new operating system were to be developed, would the split of work between its I/O and communications portions be as history has developed it? Would the split of function between operating system and subsystems be as it is today? In reaction to the changing of the underlying technology we may expect significant new developments in software structure. But along what path? Shall we integrate new function into older operating systems, or should we begin to "hide" these older systems, to think of them as only components of a single operating system that contains all software elements and maintain them only so long as we maintain the architectures on which they run? These questions are just now coming to our attention and it is difficult, given the constraints of resource and history, to predict the long-term evolutionary trends.

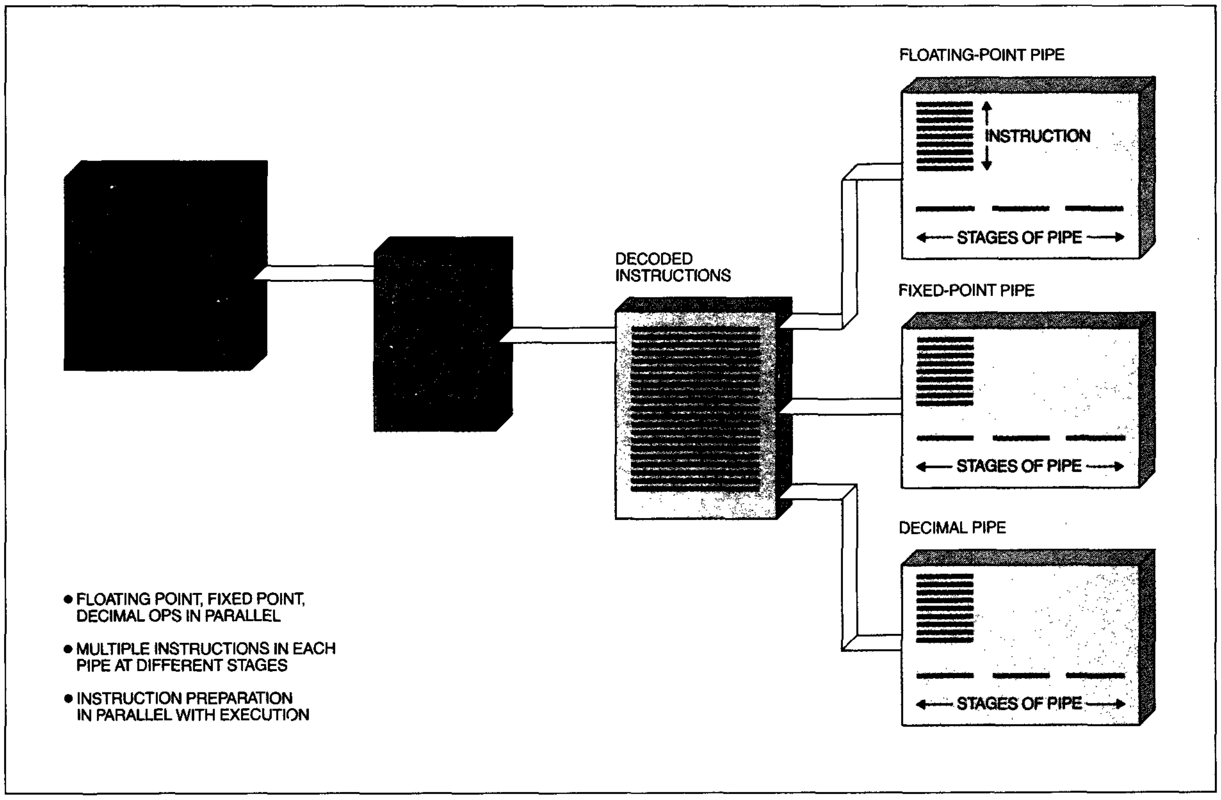
An alternative to the control flow concept is the data flow concept.

What seems reasonably sure is that communications and computer elements will continue to merge with and interpenetrate each other so that in time the structures and architectures now possible in a single frame will become possible at long geographical distances.

Data flow machines

Up to this point I have assumed no fundamental change in the nature of a processor. Computing systems have always been "control flow" machines. In a control flow machine the sequence of instruction execution is controlled by a sequence counter which points to the next instruction to be performed. An instruction is fetched from the memory location indicated by the sequence counter and executed. The sequence counter is augmented, the next instruction fetched, etc. Conditional branches replace the contents of the sequence counter with the starting location of a new sequence. The fact that some look-ahead or pipeline machines may do all this in a rather elaborate manner does not change the basic

Figure 13 Highly overlapped uniprocessor

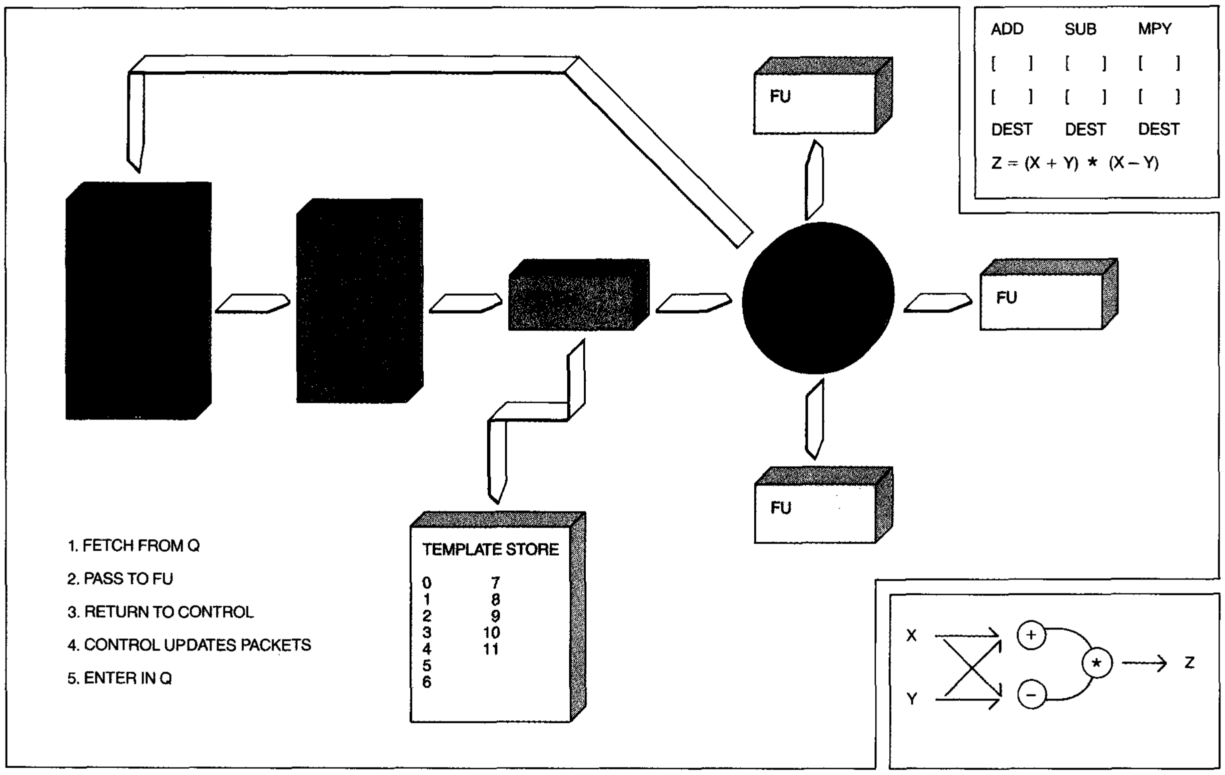

FLOATING-POINT PIPE
INSTRUCTION
STAGES OF PIPE
DECODED INSTRUCTIONS
FIXED-POINT PIPE
STAGES OF PIPE
DECIMAL PIPE
STAGES OF PIPE
• FLOATING POINT, FIXED POINT, DECIMAL OPS IN PARALLEL
• MULTIPLE INSTRUCTIONS IN EACH PIPE AT DIFFERENT STAGES
• INSTRUCTION PREPARATION IN PARALLEL WITH EXECUTION

concept. The machine depicted in Figure 13, which can execute a number of instructions in parallel, is still a control flow machine.

An alternative to the control flow concept is the "data flow" concept. We can think of a data flow machine as a modification to a deeply pipelined machine like that of Figure 13. In a control flow machine, instruction fetch, decode, address formation, and operand fetch functions are performed in an Instruction Preprocessor Function (IPPF) or I-Box that operates upon instructions as determined by its sequence counter. In a data flow machine, the sequencing mechanism of the IPPF is replaced by a concept that says an instruction is executed when its operands are available. It is a sequence-counterless machine.

Such a machine is shown in Figure 14. There is a control unit fronting a family of functional units arranged around a network. A functional unit performs when it receives a packet containing a function designator and data values on which to apply the function. When the function is complete, it sends a result packet back to the IPPF. (In this version of a data flow machine I assume that all finished operations are returned to the control unit. This restriction is by no means necessary to such a system. Some structures move packets between functional units.) The IPPF maintains a queue of instruction packets that are waiting for results from functional units. Whenever a result packet arrives at the IPPF, it fills in the operands of a waiting packet. It may then release this packet to the functional unit network. The operands required for a function to be executed are gathered from two sources: memory and returned packets. Thus, an instruction may have an operand from memory and another operand which is the result of a previous instruction. When both operands are available, the instruction is released for execution at a functional unit. In this way instructions are executed at the rate at which their operands become available. Long lists of nondependent instructions may be executed with a high degree of parallelism,

Figure 14 I-counterless SIMD data flow concept



and dependent instructions are "fired" as soon as their operands arrive. In this way instructions may be executed out of "sequence," and maximum use of functional units may be achieved.

The concept can be scaled up, and frequently is, to network structures containing large functional units, each of which may itself contain queues and local ordering. There is interest in data flow in both the supercomputer and the network model areas. It is not at all clear at what rate, or if, this concept will impact mainstream commercial processors, and a good guess is that these products will maintain their control flow characteristics for the foreseeable future.

Cellular machines

As logic becomes denser and denser and cheaper and cheaper, the basic building block of a system may itself be a very small-grained special-purpose processor. In such a system each processor is connected to a neighbor, and each processor performs a unitary function on a unit of data. Basic functions are performed by predefined network topology. Thus, a high-level function such as sort or search or Sine would be implemented by a set of cellular units connected in a network topology that maps the data flow of the function. A computational unit is itself a collection of thousands and thousands of little special-purpose processors that are organized into functional units. Such a system could be massively parallel.

An interesting implication of this concept concerns the hardware-software-firmware trade-offs of a system. As more logic is available in this form and as we learn to build a larger set of functional units using cellular structures, can we reduce the amount of software that a system requires? Can, for example, basic operating system functions such as queue management, list searching, dispatching, and ordering be built into systolic arrays? There is accumulating evidence that, over time, the answer will be positive, and a good deal of the function we associate with software may be represented in hardware, not in microcode, as we currently assume, but in preplanned cellular arrays.

The "level of an architecture"

For many years there has been continuing argument about the relative efficiency of "interpretation" versus "compilation" of programs. From the early days of computing there have been advocates of direct execution by interpreters in lieu of compilation. Program translators and program interpreters arrived in the computing culture at about the same time (late 1950s).

The current form of this continuing conversation is RISC versus HLL architectures. The RISC advocates believe that it is most efficient to provide a processor with a minimum instruction set, easily mappable into a design, and to depend upon compiler design elegance to provide efficient programs. The compilers, RISC advocates believe, can be extremely efficient because they do not need to spend much time involved in the assessment of "special cases" due to the complex instruction set. Compiler optimization can address issues of global program structure and optimization without concern about recognizing large sets of odd conditions. RISC advocates point out that the CISC is inefficient because it "special-cases" a compiler to distraction, it uses enormous amounts of circuitry for instructions that will not be used (25 percent of instructions account for 95 percent of code), and it consequently adds complexity to design as well as cost to implementation.

Essentially the view of the RISC advocate is that given a certain population of circuits, it is best to use those circuits to support efficient designs that will allow rapid execution of a minimal instruction. Rather than consume large amounts of microcode or logic to support many instructions, it is better to use control code and circuits for caches and pipelines that can help achieve concurrency and achieve an instruction execution rate close to one instruction per machine cycle.

The HLL advocates contend that an architecture has responsibilities to close the "semantic gap" between programming languages, to encourage certain forms of programming that are known to be efficient, and to recognize and prevent many types of run-time errors that compilers cannot recognize and prevent. The overall success of an architecture, they claim, lies not in the instruction execution rate, but in the reduction of the cost of creating and debugging programs. The legitimate use of circuit count and microcode space is to support higher-level architectural abstractions such as tagged data, object-oriented capability-based addressing, generic instruction sets, etc. Good higher-level architectures will ease the burdens of compilation and will significantly improve the economics of the programming process. An architecture should be extended in the direction of the forms and structures of higher-level languages and should include direct support of a number of abstract programming concepts.

There is no real metric for complexity of compilation.

What are we to make of these two positions? Both groups claim they will increase the ease of compilation, but that is the only point at which even divergences converge. The RISC group seems oriented toward price/performance and efficient use of circuits; the HLL group seems oriented toward the use of the architecture to minimize the expense of developing and maintaining programs and to the achievement of security.

How are we to approach these viewpoints? One problem is that there is no real metric for the complexity of compilation, so that the argument of which approach leads to better compilers cannot be resolved. RISC advocates point with pride at how much simpler it is to compile with the addressing conventions of the register file model as opposed to richly varied addressing structures. But on the other hand, while claiming ease of compilation, they point with pride at the sophistication necessary to compile good programs in a pipelined register file machine. Whether code rearrangement for pipelines is more or less difficult than determining which addressing form should be used is not an easily resolvable point.

An important part of the RISC argument is the assumption that higher-level functions will be implemented in microcode. This current assumption is reasonable, but by no means (in view of the last section, for example) a permanent assumption.

Surely the RISC advocates are right in their claim that too many instructions are unnecessary and wasteful. They may be a little less right in specifically insisting that all instructions execute in one cycle (or two cycles for cached memory references), because they are forcing themselves to rely on subroutine multiply and divide whose performance will be pipeline-critical. The argument against allowing more flexible addressing (two or three addresses of any type can be used in any order with any operation code) seems most uncertain and is closest to saying that good compilation can overcome a deficient architecture.

So we have a scapegoat in the middle (the CISC) with the RISC and HLL advocates on either side and some difficulty in defining the turf, along with a good deal of passion. (Certain architects from some universities will not go to meetings that architects from other universities are attending.) In any case we are in a period where RISC is having its day and where, for that reason, traditional concerns about what is a good architecture in the original meaning of the word are being revisited at a time when it is not clear how important this issue really is to system structure. Some claim that dramatic improvements in price/performance are a potential of RISC architectures, others claim that forthcoming technology will show that System/370-style CISC can achieve the same levels of price/performance. The HLL people are resting in the background and hinting that price/performance is not the measure to be used at all. The proper measure is the degree to which the architecture encourages proper use of critical software engineering methodology in application development.

The argument is interesting mostly because of the interest it focuses on what we think is important, on the relations between architecture and design and architecture and software, as well as on our wonderful ability as an industry to make great progress without actually ever really defining our terms. Everyone's view of the world is partial.

Concluding observation

We have discussed a number of ideas relative to transition in system structure. Perhaps the central and underlying themes are twofold:

1. There is going to be a significant restructuring of concepts and merging of ideas that we have traditionally thought to be distinct and separate.
2. Wide variation in system structure is going to be enabled by changing technology.

But in particular, for this coming time frame,

1. There will be dramatic improvement in local interconnection that will allow the scaling-up of structures to the establishment level. The Single System Image will begin to form for units interconnected at a site of business.
2. I/O structures will become much more elaborate and will involve a great deal more intelligence and storage.
3. Communication and I/O elements will begin to interpenetrate each other.
4. System structures will begin to include specialized communications, applications, and data server units.
5. Highly parallel machines will not become mainstream systems. Neither will data flow machines.
6. We will continue to argue about issues that are more fun than critical.

The underlying technologies seem to be telling us we can scale up our concepts and truly take large system views. The constraints on interunit interconnection will be relaxed, and aggregations of systems at significant distances can be used and managed as coherent structures. All of our dreams are coming true, and all of the problems of dreams coming true are being visited upon us.

UNIX is a trademark of AT&T Bell Laboratories.

General references

A complete bibliography in this area would be overwhelming. Following is a list of references, recent and classical, that will enrich the reader's understanding of this area and that are easily (for the most part) accessible. IBM sources are not listed.

C. G. Bell et al., "Encore continuum: A complete distributed workstation multiprocessor computer environment," *Proceedings of the AFIPS 1985 National Computer Conference* (1985), pp. 147–155.

Communications of the ACM (Special Issue on Computer Architecture) **21,** No. 1 (1978).

P. H. Enslow, Jr., *Multiprocessing and Parallel Processing*, John Wiley & Sons, Inc., New York (1974).

P. H. Enslow, Jr., "Multiprocessor organization—A survey," *1977 Computer Surveys of the Association of Computing Machinery* (1977), pp. 103–121.

T.-Y. Feng and W. Young, "Parallel control algorithm for reduced omega-omega networks," *Proceedings of the AFIPS 1985 National Computer Conference* (1985), pp. 167–174.

D. W. Hockney and C. R. Jesshope, *Parallel Computing*, Adam Hilger & Company, London (1981).

A. G. Kamlayashi, "A database machine based on data distribution approach," *Proceedings of the AFIPS 1984 National Computer Conference* (1984), pp. 613–628.

N. Matelan, "Flex 32/Multicomputer," *Proceedings of the AFIPS 1985 National Computer Conference* (1985), pp. 139–145.

H. Lorin, *Aspects of Distributed Processing*, John Wiley & Sons, Inc., New York (1980).

H. Lorin, *Parallelism in Hardware and Software: Real and Apparent Concurrency*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1972).

L. H. Magnuson, "OpenNet: A network architecture for communicating different operating systems," *Proceedings of the AFIPS 1985 National Computer Conference* (1985), pp. 619–625.

L. Polk, "Dataflow architecture for knowledge representation," *Proceedings of the AFIPS 1985 National Computer Conference* (1985), pp. 287–297.

C. V. Ramamoorthy and H. F. Li, "Pipeline architecture," *Computer Surveys of the Association for Computing Machinery*, pp. 61–102.

J. Rattner, "Concurrent processing—A new idea in scientific computing," *Proceedings of the AFIPS 1985 National Computer Conference* (1985), pp. 157–166.

A. Sekino et al., "The DCS—A new approach to multisystem data-sharing," *Proceedings of the AFIPS 1984 National Computer Conference* (1984), pp. 59–68.

G. Serlin, "New microprocessor based computer architectures," *Proceedings of the AFIPS 1984 National Computer Conference* (1984), pp. 123–130.

C.-L. Wa et al., "Prototype of Star architecture," *Proceedings of the AFIPS 1985 National Computer Conference* (1985), pp. 191–201.

Harold Lorin *IBM Corporate Technical Institutes, 500 Columbus Avenue, Thornwood, New York 10594.* Mr. Lorin joined IBM in 1965 and is currently Consulting Faculty Member at the IBM Systems Research Institute and Senior Professor of Computing Science at Hofstra University. He has previously served on the senior staff of the Service Bureau Corporation and has held a variety of professional and management positions with the Sperry Rand Corporation, the Systems Development Corporation, and the United States Air Force. He is the author or coauthor of a number of books and articles on various aspects of computing. The books include *Parallelism in Hardware and Software, Sorting and Sorting Systems, Aspects of Distributed Processing, Operating Systems, Economics of Information Processing,* and *Introduction to Computer Architecture and Organization.* Mr. Lorin is a frequent participant at professional development seminars. He has been a speaker in programs at the Massachusetts Institute of Technology Center for Information Systems Research (CISR), the New York University Graduate Center for Computer Applications and Information Systems (CAIS), the Diebold Research Program, the Association for Systems Management, and other organizations. He has organized IBM Corporate Symposia at the IBM Systems Research Institute in key information processing issues. He is a frequent contributor to special IBM institutes and seminars and is on the advisory board of *Abacus Magazine.*

Reprint Order No. G321-5275.