Reprinted from

IBM

Systems Journal

Vol 25, Nos 3/4, 1986

Impact of memory systems on computer architecture and system organization

by R. E. Matick

Impact of memory systems on computer architecture and system organization

by R. E. Matick

The largest part of computer architecture, in both the central processing unit and the overall system, has been and continues to be directly influenced in one way or another by the types of memory systems available. This is readily apparent in certain areas such as I/O architecture and memory hierarchies. However, the pervasiveness of this influence throughout the entire system is not so obvious. This paper demonstrates this relationship and shows how it has affected computer architecture over the years. Two approaches are used, the first being a direct look at how specific architectures attempt to circumvent the limitations of the associated memory system. This includes such topics as the internal architecture of CPUs: memory hierarchies and virtual memory, I/O architecture, file structuring, and data base architecture. Second, a gedanken (thought) experiment is used to predict future trends. It is assumed that very large-scale integration will evolve to the point at which we can have nearly any main memory system we desire, with some reasonable constraints. The architectural changes that might take place will be seen to be precisely related to the weaknesses in current memory systems which various architectures currently attempt to circumvent.

The current picture that any observer, skilled or unskilled, sees of much of computer systems and technology is one of vast knowledge, great complexity, and often much confusion. The reason for this picture is that the various fields have grown explosively in a short time and in many directions simultaneously. There is seldom enough time to digest it all. It is the author's conviction that much of our knowledge can be reduced to simple fundamentals that capture the "essence of things." This paper is an attempt to pull together the evolution of a vast segment of computer architecture and organization into a simple framework that can be useful to the expert and nonexpert. The subject is storage, and the key to understanding all storage as well as much of computer architecture is the problem of addressing or accessing information. This is the major theme of this paper.

The word "architecture" as used in this paper often, but not exclusively, implies the "instruction set" associated with any given system. Although an attempt is made to maintain this definition here, there is an inherent difficulty in doing so because the "architecture" is not always apparent or available. The system programmer, for instance, has access to instructions that the user neither sees nor needs. Also, I/O, data base, and even system commands are often MACRO instructions that make use of more primitive instructions. Hence, the user's view of architecture can be different from the actual hardware. The term architecture is thus used in a slightly more general sense.

There are fundamentally two major areas of computer design that have been and are now the nuclei of most of computer evolution: (1) the overall processing functions of the system and (2) the memory system that serves this processor. The evolution of computer architecture and design has been the history of new ways to circumvent the limitations of previous generations in these two major areas. In order to understand this, and particularly the relationship between memory and the remainder of the system, it is expedient to start with some simple concepts as to exactly what fundamental functions are required. There are basically two types of problems that account for most computer usage. Although there are other classes, they tend either toward one or the other or toward both areas of computer design, e.g., artificial intelligence. Consider first *computational-type problems* (scientific, engineering, financial, etc.) that take some given parameters, perform some arithmetic and logic processing on them, and produce a *result.* The second class, called *data-based* problems, are those in which a large *file is* accessed for *limited information.* The information may or may not be subsequently updated (by insertions and deletions) or undergo some computational processing and then be refiled. Both types of problems were solved by human beings long before there was any notion of a computer. In fact, it was such problems that gave rise to the need for computers. Consider what is required in a general sense to deal with these two classes of problems.

In computational problems, the processor—human or machine—must take the arguments from some input source, perform the required computation, and record the result on some output source. The input arguments as well as the result must be stored somewhere—in our brains, on a sheet of paper, on a tape, disk, or other storage medium, such as main memory. When many computations are to be performed over many arguments, our brains are neither as adequate nor as reliable as other memory resources. As a bare minimum, a sheet of paper (or equivalent) is required. Thus it is already clear that computational problems require that the arguments be retrieved from some storage medium, and, after processing, that the result be recorded on some storage medium that may be the same or different from the one used for the arguments.

In such computational problems, accessing the information usually involves the following simple steps: (1) access the first argument of the first column; (2) process it with the first argument of the second column; and (3) record it in the first position of the third column. Even in cases where this horizontal accessing becomes a complex diagonal accessing, such as with matrix parameters, the accessing is still well-defined.

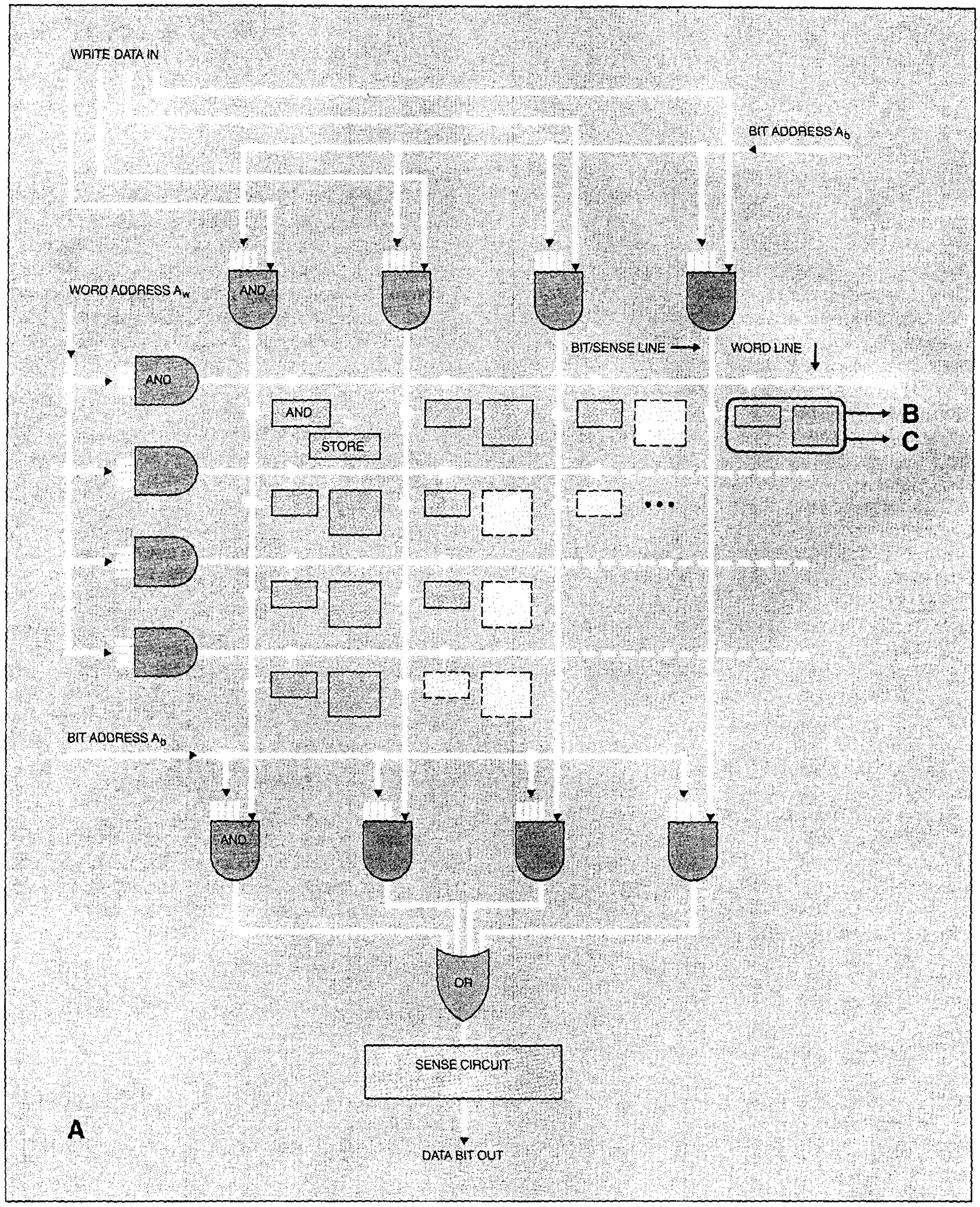
In file-accessing problems, the requirements are much the same as those just given, with one additional and severe complication. The arguments to be retrieved are not usually so readily defined, and/or the place where the result is to be stored is not well-defined. Consider an automobile insurance company file that consists of individual folders (records) organized by client names, in alphabetical order. If we wish to determine the premium for any given individual, it is necessary to access the file for the record—a simple matter in this case—but then this may be followed by an *associative* search, i.e., compares, of several lines or pages until the desired item is located. The associative search, which is slow, can be avoided by having the file well-organized by columns or *fields*, with the premium being one of the columns. It is then necessary, however, to have some index stored either in our brains or at the beginning of the file to indicate where (i.e., on which page) the column exists. An even more complex accessing problem is typically encountered when it is necessary to find and revise records on all clients with subcompact cars, and/or with premiums below a given value. If the file and records have not been organized for such accessing, the problem becomes very time-consuming.

If one wishes to enter new information into a well-organized file, the question becomes one of where to record the new information and how to index it for easy access. Thus, unlike computational problems, the locations or addresses of the arguments and results are not necessarily well-defined and present fundamental difficulties.

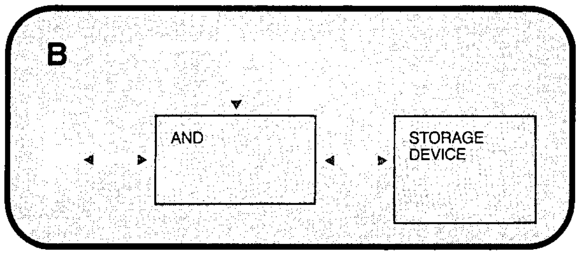
The point of this discussion was to illustrate by practical example the two major fundamental functions in all types of everyday problems, namely accessing (reading or writing) information from some storage medium, and processing this information. This paper dwells on the accessing problem to show how the practical limitations on our ability to access large amounts of information have influenced much of computer architecture.

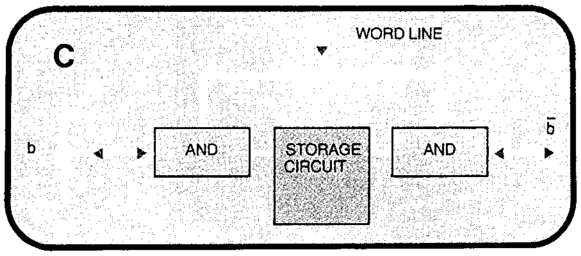
The accessing problem. The previous discussion illustrates the fact that random access to large amounts of information is one major, essential function in all data processing. Then why do we not simply produce such a memory? The reason, as always, is cost and performance. The way in which this directly affects

Figure 1 Schematic of a random access memory showing required "and" functions: (A) random access array, (B) dynamic cell, (C) static cell, 2 bit/sense lines per cell



the memory system design can be understood from a few simple fundamentals. All memory systems require a storage medium, in addition to a coincidence function for writing and a selection mechanism for reading, as discussed in Reference 1, Chapter 2. The selection mechanism for reading and writing of any storage medium fundamentally requires an AND logic function in some form. In random-access memory systems there are AND functions



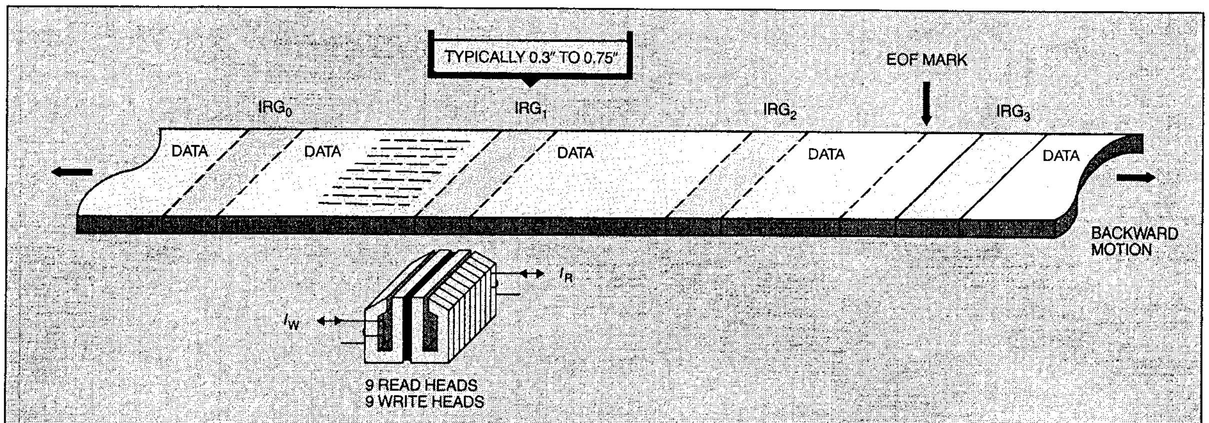


at several places. For instance, Figure 1A shows the word address being decoded via hard-wired AND decoders that select one of the physical word lines. (Note that the physical word line need not be the logical word.) There is at least one AND-type function physically connected to each storage cell, such as for the dynamic cell shown in Figure 1B and two AND functions for static cells shown in Figure 1C. There is typically one AND for each bit/sense line for writing and one for reading as shown, plus other AND-like decoders at various levels not shown, such as chip select, island select, module select, etc.

In order to reduce cost as well as increase density, secondary-storage devices remove the reading/writing AND functions from each cell as well as from the bit/sense lines and replace these many ANDs with a few that are located elsewhere and shared over many bits. Obviously, this immediately precludes the ability for truly random accessing, a severe limitation but one that is necessary to reduce cost. In addition, the random access cells, each of which is made up of several distinguishable semiconductor devices wired together on a silicon chip, are replaced by magnetized spots on a continuous magnetic medium and have no physical structure. The magnetized spots are created by placing a magnetic head, essentially a toroid with a small air gap, as close as physically possible to a suitable magnetic medium, as shown in Figure 2. The external field near the gap magnetizes the material and under proper conditions can leave a magnetized spot as small as 1/10000 inch or smaller. The AND function for writing is the AND of the write current I_W with the physical location of the medium. For reading, whenever the medium is moved at constant velocity, a sense signal I_R is induced in the sense head. A strobe pulse, derived from the coded, magnetized spots, provides the AND of the physical position and electrical signal for reading. All such media require a constant velocity between head and medium for both reading and writing. This becomes more critical as the density increases. However, magnetoresistive sensors used in some tape units produce read signals which are independent of velocity.

Now let us look at the addressing problem for secondary storage systems. For a strictly sequential system, such as one long piece of tape that can pass only back and forth across the read/write heads as shown in Figure 2, it is obvious that there cannot be any random accessing capability. If the head happens to be at the beginning of the tape and the desired information is at the end, the tape must sequentially pass over every piece of data stored on the tape. How can discrete pieces of data be written and read rather than the entire tape? Since the tape must move at constant velocity for reading and writing, there must be start/stop intervals on the tape during which time the tape accelerates and decelerates. These intervals are called Inter-Record Gaps (IRG), and each IRG contains special characters at the leading edge of the gap. When the tape is moving, special circuits in the tape controller sense these characters and stop if it is appropriate. Obviously, these gaps can be passed over with proper logic in the tape controller. The various functions that a typical tape unit can perform for addressing the medium are listed in Figure 2. There are usually no other functions available for any finer addressability. If the user wants to read or write Record No. 40 and happens to know (in his brain or built into the I/O program) that the head is positioned at the start of Record No. 30, the I/O

Figure 2 Tape accessing functions



STARTING AT IRG_1, SYSTEM CAN:

READ/WRITE FORWARD – READ OR WRITE ALL DATA BETWEEN IRG_1 AND IRG_2, STOP AT IRG_2	READ/WRITE BACKWARD – READ OR WRITE ALL DATA BETWEEN IRG_1 AND IRG_0, STOP AT IRG_0
FORWARD BLOCK – ADVANCE TO IRG_1, NO READ OR WRITE	BACKWARD BLOCK – BACKSPACE TO IRG_0
FORWARD FILE – ADVANCE TO IRG_3 (FIRST IRG AFTER EOF MARK)	REWIND – REWIND TAPE BACK TO BEGINNING

program must issue a FORWARD BLOCK command 10 times, either directly or by an equivalent loop. Then the READ or WRITE command is issued. However, if the user does not know the current position of the head, the simplest procedure is to REWIND to the beginning and FORWARD BLOCK 39 times. The user can attempt to determine the current position by reading the very next record to the CPU and can use the data to find the relative position, if possible. This may or may not be possible, and it is complex, requiring an I/O routine that can be modified to accept the current position as a parameter. In any case, this is not part of the tape unit addressability. Fundamentally, the only addressability is to a specific tape unit, when there are such multiple units, and to the next IRG or IRG after an End of File (EOF) tape mark.

Tape-like storage provides a very inexpensive, durable, convenient method for storage of massive amounts of information such as for archives. However, the sequential nature of tape greatly limits its versatility, especially as the density increases and the volume of data also increases. These addressing limitations were significantly reduced by the IBM 3850 Mass Storage System (MSS), which was introduced in 1974. The MSS incorporates the low cost and high density of tape media with the continuous motion and track addressability of disks, plus a very large total capacity.

Further improvements in addressing capability are provided by disk and drum devices. Disks and drums circumvent some but not all of the accessing limitations of tapes. We now consider disks with a movable-arm (movable-head) unit, as diagrammed in Figure 3. Basically, the tape of Figure 2 is cut into many short pieces, the pieces are placed in a circular fashion end to end on a hard disk surface, with many tape pieces in concentric circles, and the medium is rotated at constant velocity. Since each of these tracks on disk is comparable to a tape unit, there is direct hardware addressability to each unit, i.e., to each track. If multiple disks are stacked on one spindle, there are essentially more units (tracks) and hence more addressability. Stacking disks has the further advantage that in a movable-arm system with one head per disk surface, any position of the arm selects a cylinder of tracks (SEEK CYLINDER). As illustrated in Figure 3, switching from the top track (head) to any of the others in the same cylinder (SEEK HEAD) is done at electronic speed, which is essentially one system cycle time. Thus a disk, by its nature of existing in two dimensions, provides many more directly addressable units of data than a tape. This direct or random addressability is a very important parameter in all memory/storage systems and represents the unit that is directly addressed by fixed hardware, requiring only one instruction containing the unit address and no additional software. For main memory, this unit is typically a memory word or double word of 4 or 8 bytes, directly addressed by the hardware decoders from the instructions READ (address) or WRITE (address). For a simple disk system, this unit is a track, directly accessed by the hardware from the instructions SEEK CYLINDER (address): SEEK HEAD (address), where *address* is the cylinder/head number, i.e., the exact track address. Thus the directly addressable unit is much larger on disk than it is in main memory. For instance, a track—even on early System/360 disks such as the 2311—could contain up to 3694 bytes of user data. On a more sophisticated disk system that has sector addressing, this addressable unit is a sector, directly addressed by an instruction SET SECTOR (address), where address is the explicit sector number. However, before the sector can be accessed, a SEEK to the correct track is required, which is a two- or three-step operation. A sector on a modern 3380 disk is about 512 bytes long, whereas the track can hold up to 47476 bytes of user data. Except for cases of simple sequential processing, the directly addressed unit on a disk, namely, a full track or sector, is typically too long to be entirely useful. A sector of 512 bytes is nearly a half page of double-spaced, ordinary text and a track of 3694 bytes is roughly two to four pages of text. The unit desired for processing or just to be accessed is in the range of 10 to 100 characters, e.g., "What is the name of employee with serial number 1234567," "List the X Airline flights to LA on Tuesday," "Enter reservation for A. B. Jones." Thus it is clear that even for simple files, finer addressing is required at some level. The methods chosen for implementing fine addressing on disks are reflected back into the entire I/O architecture, from hardware to software access methods, and even to main memory organization itself. The reason for this pervasiveness is that any finer addressing at the disk level requires the use of Store Addressing Information (SAI) and I/O software subroutines executed by the channel in conjunction with the disk unit, to perform search operations. SAI is discussed in Reference 1, Chapter 7. The track can be logically divided into fixed- or variable-length records, with each record preceded by some Store Addressing Information. The SAI is simply a well-defined address stored on the disk in an area designated by gaps that contain special codes. Logic circuits in the controller sense these gap codes and know when the stored addressing information is to start. In this manner, a given address can be located by a search command. These search operations consist of repeated read and compare functions performed on the stored addressing information until a match occurs or the end of the search is reached. The SAI can be various gaps, address markers, record count, or ID keys, as shown

Figure 3 Schematic of typical movable-arm disk unit

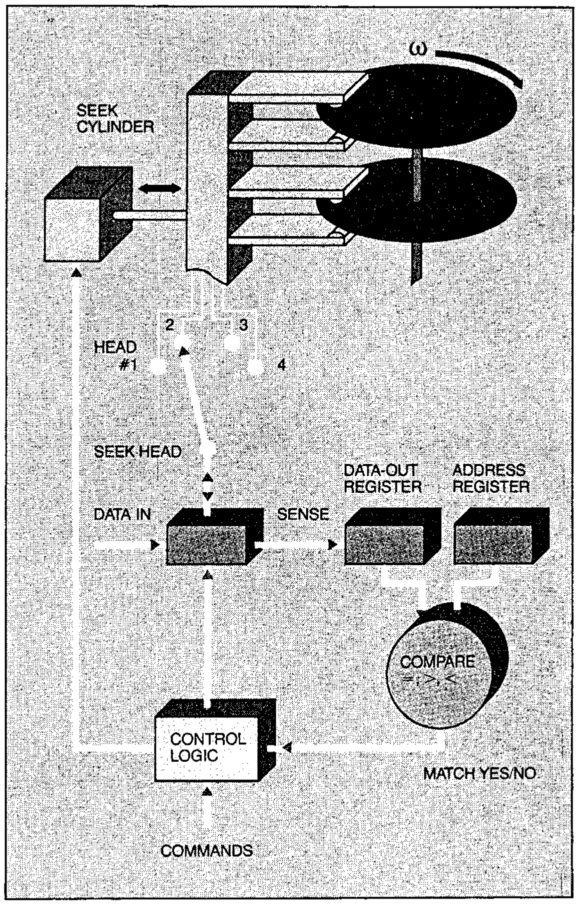
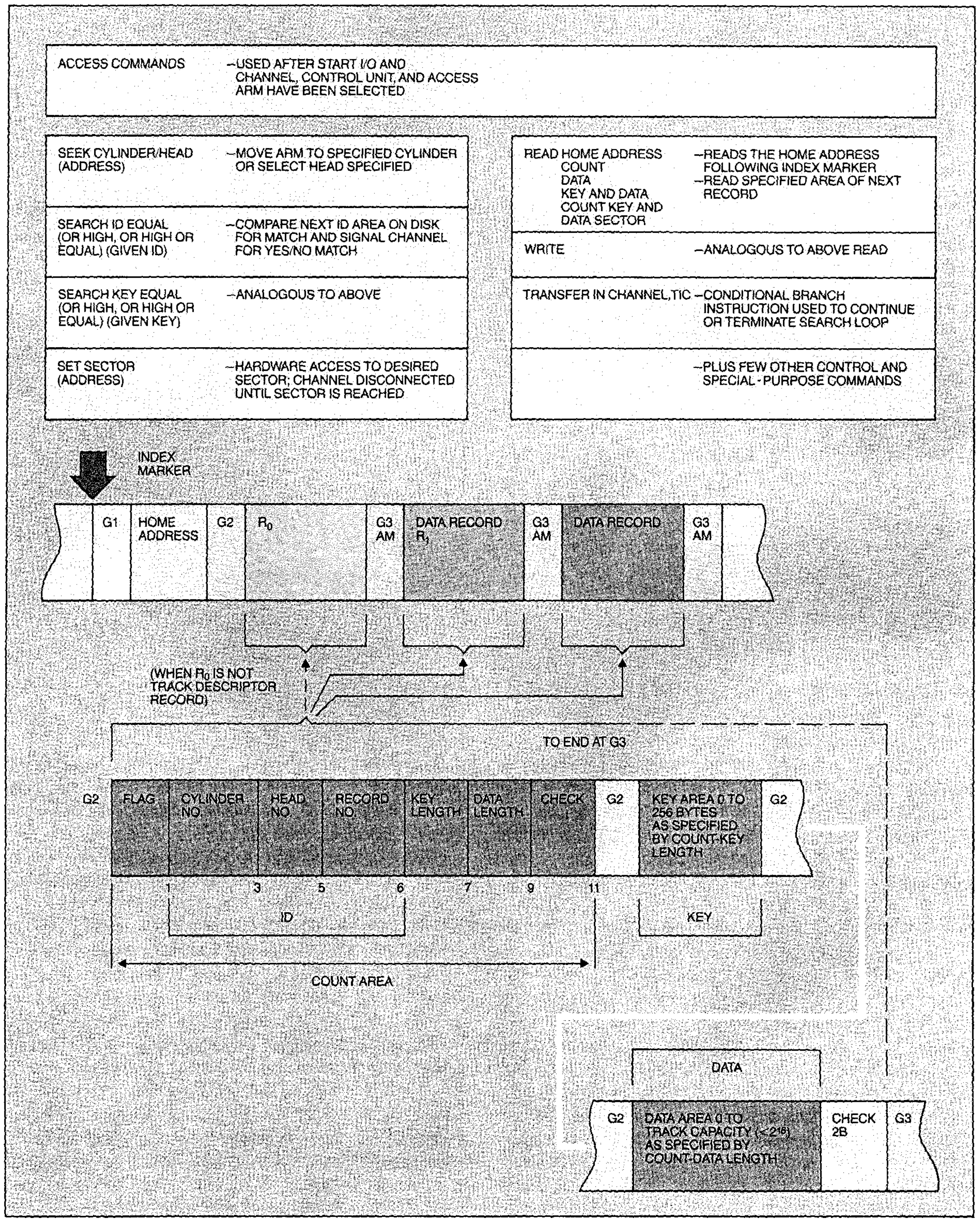


Figure 4 Essential accessing features of IBM System/360/370 disk systems showing various stored addressing information



ACCESS COMMANDS	–USED AFTER START I/O AND CHANNEL, CONTROL UNIT, AND ACCESS ARM HAVE BEEN SELECTED
SEEK CYLINDER/HEAD (ADDRESS)	–MOVE ARM TO SPECIFIED CYLINDER OR SELECT HEAD SPECIFIED
SEARCH ID EQUAL (OR HIGH, OR HIGH OR EQUAL) (GIVEN ID)	–COMPARE NEXT ID AREA ON DISK FOR MATCH AND SIGNAL CHANNEL FOR YES/NO MATCH
SEARCH KEY EQUAL (OR HIGH, OR HIGH OR EQUAL) (GIVEN KEY)	–ANALOGOUS TO ABOVE
SET SECTOR (ADDRESS)	–HARDWARE ACCESS TO DESIRED SECTOR; CHANNEL DISCONNECTED UNTIL SECTOR IS REACHED
READ HOME ADDRESS COUNT DATA KEY AND DATA COUNT KEY AND DATA SECTOR	–READS THE HOME ADDRESS FOLLOWING INDEX MARKER –READ SPECIFIED AREA OF NEXT RECORD
WRITE	–ANALOGOUS TO ABOVE READ
TRANSFER IN CHANNEL,TIC	–CONDITIONAL BRANCH INSTRUCTION USED TO CONTINUE OR TERMINATE SEARCH LOOP
	–PLUS FEW OTHER CONTROL AND SPECIAL-PURPOSE COMMANDS

in Figure 4, or other forms. If this fine addressing is either not provided or not used, the unit word must be transferred to main memory, so that the byte addressing and logical capability of the CPU can be used. The System/360 and System/370 I/O architecture provides a programmable track format for variable-length records. The SAI for finding records consists basically of two areas: (1) the ID part of the COUNT area and (2) the KEY area, as shown in Figure 4, along with the fundamental accessing commands of such a disk. Either one or both of these can be used in various ways, the trade-off being the more SAI, the less available user data space.

Only the direct addressing commands, SEEK and SET SECTOR, are carried out entirely by the control units. The channel is not involved, and, in fact, it can be servicing other attached disks. Any finer addressing requires search commands, which tie up the channel for the entire length of the track. It should be apparent that a memory system with random access to a small unit inherently requires considerably more hardware and thus is likely to be expensive. Cost reduction is achieved by removing the cell structure and sharing the read/write transducers. The further the cost is reduced, the larger the unit of direct addressability. Fine addressing then requires Stored Addressing Information, which consumes available data space, and slow read-and-compare searching techniques. These limitations on accessibility affect not only the process of finding the information, but also its read-write rate. The sharing of transducers coupled with high density influences the design toward serial data paths, which have been the standard over the years.

One final and serious access limitation of disks is that in order to achieve high speed and high density, only one head of a cylinder can be read or written at a time. Multiple heads or tracks of one cylinder cannot easily be used in parallel, because the mechanical and electrical tolerances require separate mechanical track-following for each head via a feedback servo mechanism. In a similar fashion, it is not feasible to distribute the data across several disk drives, because the disks are not in spatial rotational synchronism. (A later section on a *gedanken—thought*—experiment shows how this might be alleviated with a simple synchronous-to-asynchronous conversion buffer.) With today's technology, it is not possible to distribute data across multiple tracks in any form so as to improve bandwidth.

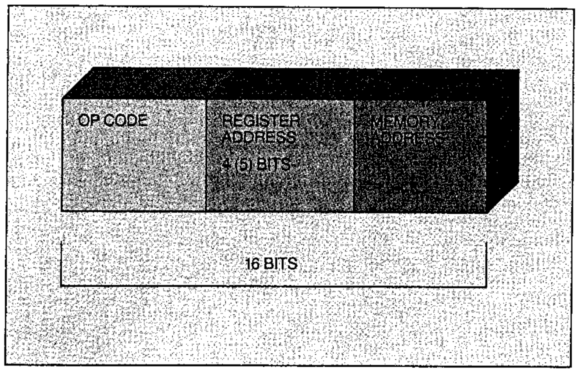
Thus we deduce that a desirable memory system should have the ability to randomly access a large number of relatively small data units. Some associative compare capabilities are also desirable. The exact nature of the associative functions is not entirely clear yet, but they will be defined as we proceed.

Practical limitations of memory have enormously affected the central processor.

Central processing unit architecture

Practical limitations of memory have enormously affected the architecture of the central processor. As indicated in the introduction, the processor must access the information from the storage medium before any processing can take place, and it then must record the result on a storage medium before continuing. A fundamental requirement of an ideal storage system operating with an ideal processor under worst-case conditions is that it should have random access to about four entries for every full cycle required to process the instructions. These accesses are as follows: one for the next instruction, two for the two arguments to be processed, and one for a result to be stored, if required. For a one-cycle processor, the storage system must either have four random access ports or be capable of four full cycles for each processor cycle. Both of these are extremely difficult to provide. All random access main memories, past and present, have only one random access port, and they have at most one access per processor cycle. From earliest times, the speed difference has been compensated for by the use of registers external to the memory array and electrically closer to the processor. Also, such registers are capable of being accessed and manipulated at a level as fine as the bit level. Almost all computers have some special registers such as an accumulator and a multiplier-quotient register. In principle, such registers are unnecessary because they are typically loaded and/or unloaded to memory. However, it is difficult, expensive, and redundant to make every word in memory have a variable bit addressability or even variable word-length addressability, as well as sufficiently fast access

Figure 5 Typical 16-bit instruction with separate memory and register address fields



to the processor. Hence an intermediate staging register is used, due to the limitations of memory.

In addition to these more specialized types of registers, it has been found that faster access to memory can be obtained by using a small register stack that is typically loaded and unloaded from memory. This is usually referred to as the general-purpose-register stack. This stack is an extension of main memory to make it look more like the ideal. Because many arguments and parameters are reused many times, they are stored in this smaller and much faster local memory. In fact, typical register stacks can often access three, four, or more entries simultaneously for precisely the reasons just given, namely the simultaneous reading of arguments and storing results. One difficulty is that the address of an argument in this small stack is quite different from that of its equivalent memory address, which necessitates a new means for accessing. Unfortunately, this requirement affects the architecture of the instruction set, regardless of what solution is used. In the most common cases, the register addresses are just included within the instruction, thereby taking substantial code space. For instance, for a 16-bit instruction which addresses 16 (4 address bits) or 32 registers (5 address bits), only 12 or 11 bits respectively are available for the operation code and memory address. (See Figure 5.) A reduced address length reduces the amount of memory that can be accessed, and an insufficient address length is the major reason for the use of numerous addressing modes in some architectures.

A significant consequence of including the register address in the instruction code space is that it has severely limited the number of registers that can be used. Early computers, such as the IBM 650 and1401, had no such general-purpose registers but rather a few special registers such as an accumulator, distributor, or multiplier/quotient register. The IBM 704 had three index registers for base-index addressing but no general-purpose registers. In these systems, the main memory operated at the same speed as the CPU, hence memory itself served as the register stack. In fact, it was common first to pick a memory technology and then design the CPU to match the memory speed. However, as technology improved and computers become more useful, increased computational power was needed. This pushed the designs toward faster processors and larger main memory. Thus processor speed increased faster than main memory. (See Reference 1, Chapter 1.) The value of having small local registers whose speed matches that of the processor became quickly evident. However, the number of such registers has not changed much since the time when the advantages were first recognized. For instance, the IBM 7094 had a stack of seven *index registers*, occupying three bits of the instruction code. IBM System/360 and System/370 architecture has 16 general-purpose registers (plus 16 special-purpose, system-control registers), and currently the maximum number on most machines, IBM and others, is 32 registers. The latter has most often been accompanied by an increase in instruction length.[2] Although more registers are desirable, the compromises offset the advantages. The compromise solution to this problem has been to introduce a cache memory into the system that is an extension of main memory. A cache is capable of working at the speed of the processor but with considerable additional complexity.

In any case, even though the general-purpose registers have helped circumvent some of the bandwidth limitations of main memory, they have introduced a whole new set of problems and proposed architectural solutions. The two most serious problems are the small number of general-purpose registers and the need to have the address residing within the instruction. A small number of registers requires a significant amount of overhead. For example, two different tasks or procedures (subroutines) that require all of the registers must have the same register addresses within their instructions. Hence, if it is necessary to switch problem state or to branch, part or all of the register stack must be saved in main memory and the new state must be loaded from main memory to the registers. This can be very time-consuming, particularly when state changes occur frequently. Rapid state changes occur with multiprogrammed/multitasking systems, which are quite common. The same is true with structured programming languages that may have many nested subroutines, each with several parameters, e.g., Pascal and the C language. This problem is becoming more serious as processor performance increases, especially in microsystems and minisystems. As a result, many methods have been explored for augmenting the register stack and saving changes of state while attempting to minimize the impact on the instruction code. We briefly consider several methods dealing mainly with the underlying problems related to memory accessing as previously described.

Many methods have been explored for augmenting the register stack and saving changes of state.

The BELLMAC-32 microprocessor[3] provides hardware-implemented instructions for managing a stack of 16 registers for a highly structured language such as C, in which procedure calls or returns occur on average roughly once in every 20 instructions executed.[4] The number of registers used and their allocation for procedure calls and returns are handled by the hardware, which typically makes these processes faster. Without such facilities, the operating system or user would have to manage the registers, using the available more general instructions. However, in addition to the added hardware complexity, several of the registers have specific use for managing the register stack, and thus they are unavailable for holding variables. Fundamentally the only architectural advantage is ease of use; but this is obtained at the expense of fewer usable registers.

The 801 architecture[2] uses five operation code bits to provide the user with 32 directly accessible registers and an efficient compiler to make use of them. The additional registers considerably aid both the compiler and the user programs. The resulting reduced instruction code space for memory addressing and other functions drives the architecture toward one instruction length of 32 bits, rather than mixed 16- and 32-bit instructions. This has significant impact on the cache and memory bandwidth requirements.[1,5]

The Bell Laboratories C Machine[4] increases the register stack to typically 1024 registers, but, rather than consuming 10 instruction-code bits for addressing these registers, the C Machine introduces a new addressing mode. The stack address is calculated during predecode time and is stored with the instruction in an I-buffer. In essence, the effective instruction length is made larger only inside the CPU, but at the expense of considerable logic complexity and stack pointer maintenance. Four new instructions are also required to manipulate the stack because it obviously must be loaded and unloaded properly for maximum performance. Unfortunately, such a large stack is useful mainly for highly structured languages in which there are many nested subroutines, each with several parameters. The larger stack can hold most of these subroutines as required by any one given user or task. Whenever a task-switch occurs, the stack must be made available to the new user/task, and the old stack must be saved in main memory. A larger stack takes longer to save and subsequently restore. For systems with frequent task-switches, the penalty can be dramatic, especially because memory bandwidth is usually the critical performance-limiting parameter.

The Carnegie-Mellon TM architecture[6] attempts to reduce compiler complexity while maintaining the efficiency of high-level language object programs. This is achieved mainly by the use of a value cache, which is fundamentally similar to other caches, except that it stores common subexpressions that are likely to be reused and hence do not have to be reevaluated each time they are invoked. The advantages of this value cache depend on significant occurrences and identification of common subexpressions in the work load. In addition, a push-down evaluation stack for data words and a control stack are introduced. The net effect is the introduction of additional fast local memory with combined implicit and explicit addressing.

Members of the University of California at Berkeley have designed a reduced-instruction-set computer (RISC)[7] that makes use of a larger physical register stack but a smaller, fixed, logical stack and renames the register window on a task switch or procedure call. This can often prevent the need to save the stack on a subroutine jump as well as a reloading upon return. In essence, the RISC has made the stack larger without directly consuming instruction code space, but it requires other hardware for proper addressing and manipulation.

The IBM System/38 incorporates a stack of 16 general-purpose registers but simplifies task switching by the use of a task dispatcher built directly in microcode.

These are but a few of the numerous methods that have been attempted as a way to overcome the fundamental memory-accessing problem. All methods tend to fall into one or more of the following three categories: (1) add small, fast, local cache-like storage; (2) add more registers and consume the instruction code space; and (3) add more registers but avoid impacting the instruction code space by including complex hardware/software bookkeeping capability. However, it should be clear that all solutions are compromises and the fundamental issue remains that of multiple accesses to a large memory at processor speed with adequate addressing capability.

The fundamental issue remains that of multiple accesses to a large memory at processor speed with adequate addressing capability.

Address space. The limitation imposed by memory has had a profound effect on another aspect of CPU architecture which, over the years, has spilled over to influence the entire system architecture and organization. This problem is centered around the limited size of main memory and the resulting limited address space provided to the user. In early computers the maximum size of the memory address was quite small, less than that of a PC today, because early memories were quite expensive. Because there were few addressable words, the addresses were totally contained within the instructions, and the address bits consumed only a small part of the instruction code space. Fifteen to eighteen address bits were adequate for early large computers, from the IBM 704 through the Univac 1108. As memory technology improved, larger main memory capacity became available; consequently, more address bits were needed. In fact, the need for main memory is open-ended, being subject to a Parkinson-type law that might be termed the Law of Expanding Storage: Problems expand to fill the storage allowed for their completion.[1] To accommodate the larger address for larger memory without significantly increasing the instruction length, *base addressing* became common. The base or starting address of any memory reference is stored in a special-purpose register or in a particular general-purpose register, and the instruction code contains only the displacement or offset from that base. When the address length exceeds the displacement length, a new base is loaded into the base register. Thus a very large address space can be accommodated with a relatively small displacement field in the instruction. This also provides a simple means for relocating programs. Thus a program with fixed addresses can be placed anywhere in memory by adding the base to all addresses. Some systems use an index register as well as a base; fundamentally, they serve the same purpose. Note that this base addressing has a direct and dramatic effect on the instruction set format as well as machine organization. All instructions using base addressing must indicate a base register (either directly or implicitly) and contain a displacement. If an index register is also used, it must be addressed directly or indirectly. The processor now must add the base plus displacement (plus index, if used) before a memory reference is possible. This last function, known as address generation, was not necessary in early computers, but it is quite common today. Base addressing has introduced problems of its own, especially in computers having small instruction lengths, that is, 8 to 16 bits. There are typically not enough bits to specify the operation, the base register, and displacement. Also, in such systems, there are not enough registers to keep track of instruction locations, data locations, and various subroutine branch addresses. Hence, many such systems have numerous and often quite complex addressing modes. This is the case for such systems as the DEC-VAX, Intel 8086 and 8088 microprocessors (the latter used in the original IBM PC), the Motorola 68000, and many others. These complex addressing modes are fundamentally unnecessary but are required because of practical constraints.

Virtual memory and memory hierarchy

Virtual memory hierarchies, introduced commercially in the early 1970s, were aimed at solving several problems. One was to avoid the severe main memory fragmentation that occurred in a multitasking system. Programs with data were variable in size, yet each had to occupy its own physically contiguous memory addresses. Hence, as programs of different sizes were completed, new programs of different sizes were brought in that did not necessarily match the vacant slots, leaving large portions of memory unused. The use of a fixed unit of transfer—typically a page of 4K bytes—coupled with demand paging were introduced to solve this problem of fragmentation. There is a perfect fit between the area cleared and the size of the incoming entry. (Note that fragmentation now occurs within the virtual address space.) A second problem solved by virtual memory is the severe limitation of address space given to the user. In early systems, such as the IBM 704, the programmer had a maximum of 32K words or 15 address bits. In a nonvirtual memory, the maximum number of address bits must equal the actual number of physically addressable words or bytes of main memory. In a virtual memory, the user can be given an architecture-specified number of address bits, e.g., 24 or 32, regardless of the actual size of main memory. This eliminates the need to overlay programs and greatly simplifies the programmer's task. Over the years, continued refinement of virtual memory implementations has produced systems that appear to have the very large storage capacity of disks, but with an average speed about equal to that of main memory (or cache in appropriate systems).

However, all these features have a cost. Memory management and virtual memory hierarchies (including cache) have had more significant and far-reaching consequences for system architecture and organization than any other concept. Hardware architecture has had to include special instructions for virtual memory management, depending on the system. For instance, there are instructions for loading a real address, and several instructions for purging selectable portions or the entire Translation Lookaside Buffer (TLB). In some systems with caches, there are similar purging instructions for the cache and its directory.

In software architecture, the concepts of pages and segments as well as the use of segment and page tables for translation are so thoroughly entrenched that they are impossible to change. Early virtual memory designers decided to use tables for translation because of the ease of implementation and the simple, general technique provided for the sharing of physical memory. Thus two different users' virtual addresses can reference the same data page by having the same address pointer in different page tables. It was also argued that the use of software rather than strictly hardware to do the translation would not lock the architecture and, therefore, would be both flexible and changeable. Ironically, the latter prophecy has turned out to be quite the opposite, but for reasons that were unforeseen. Once segments, pages, and the associated tables were introduced, they were found to be convenient for performing many other functions for controlling and managing the system. Hence tables are inherent in the implementation of operating systems, from the bottom layers up through the top, and any attempt to change at any lower level would ripple up to the top. With large operating systems such as MVS and VM, the cost of change would be prohibitive. In retrospect, there would be more flexibility in a hardware implementation. With the ever-increasing cost and complexity of system programming and the decreasing cost with improved performance of VLSI, it is becoming more attractive in many cases to implement functions in hardware rather than software. It should be kept clearly in mind that the term *software* really means the use of *general-purpose* hardware rather than special-purpose hardware.

Virtual memory hierarchies have influenced the entire memory-I/O subsystem architecture.

Another irony of the use of translation tables is that because they consume considerable main memory space, a large multiprogrammed system typically limits the size of the virtual address allotted to each user. For instance, on VM/370 the architected virtual space is 24 address bits or 16M bytes. However, users are typically allotted 1 or 2M bytes, all of which is virtual space and not physical memory. The full virtual space can be used only at special times or with special permission. If all users were given the full virtual space in any multiprogrammed system, all of main memory might be consumed by the translation tables. One solution is to page the page tables dynamically just like other data. This is actually done in MVS and new versions of VM. However, additional means for translating to the page tables are required. This results in the consumption of more system resources, although there is a net gain.

Virtual memory hierarchies have influenced the entire memory-I/O subsystem architecture. Data on paging disks must be treated in units of pages, which do not always match the inherent track or record length available on disks. Page faults in main memory of a high-performance processor, even though the miss ratio is less than one percent, still occur at a rate so high that supplying sufficient I/O to keep the CPU busy requires special consideration and design. In some systems, the I/O is also accessed with virtual addresses. This requires additional translation facilities and supervisory software.

Another important part of the memory hierarchy that is not usually seen by the application programmer, but is of significant concern to the system programmer, is the cache. The main purpose of the cache is to give the memory hierarchy the appearance of operating at the CPU cycle time. In a multiprocessor configuration, such as the IBM 3090 system, when one processor's cache experiences an access miss, the data may be in the cache of another processor. Searching for the data may require cross-interrogation, control, and possibly data transfer. This is known as the "cache coherency problem," and it affects the architecture, machine organization, and operating system.

Thus it is apparent that controlling and managing the virtual memory hierarchy is a pervasive task requiring substantial operating-system resources, as well as hardware and architectural assistance. If a very large, fast memory could be implemented to replace the memory hierarchy, substantial simplification in hardware and software could result.

I/O architecture and file access methods

The term *I/O architecture* is used here as it was originally used in early computers, namely, to refer to the input and output of relatively simple data—financial, scientific, or any other—for which the location or address is reasonably well known and for which no complex addressing is needed. For practical reasons, data base and complex file addressing make use of secondary I/O storage, but they have some additional addressing requirements that are discussed in the next section.

Fundamentally, means are required to get information from our brains, piece of paper, or other source, into the computer memory system. With the computers of around 1950, the user could and did take a deck of cards containing program and data to the machine room, place them in a card reader, and push the start button. The cards were read one at a time and went into main memory. Whenever the data constituted a large file, such as payroll or inventory, the files were maintained on punched cards and called *unit records*. These files were processed in a batch mode, which required a large number of transactions to be accumulated over some time period, perhaps a day, before being processed. Cards were read into memory, one at a time, and processed sequentially in the order of entry. If the file had to be processed in a different order, sorting was performed, often off-line using a mechanical card sorter.

The first commercial magnetic storage drum was used as main memory.

The introduction of commercial magnetic tape in 1951 made unit-record processing cheaper, faster, and more reliable, with larger capacity. However, the procedure was logically the same. The main difference was that cards went to tape, as an intermediate stage, primarily for buffering the speed difference between main memory and cards. Tape could also hold a queue of programs to be executed sequentially, for better system efficiency. A similar but reversed process occurred for output. The cards could then be separately printed off-line.

This speed differential between memory and input/output devices, as well as a lack of good interface to main memory, has been the main factor influencing I/O architecture. I/O devices such as cards, keyboards/terminals, tapes, disks, etc. are typically several orders of magnitude slower than main memory, both in access time and data rate. It is uneconomical to idle the entire system while doing I/O. A better solution is to overlap I/O functions with other tasks. The manner in which this is done has profound impact on the instruction set and overall system organization. Thus, a brief look at the evolutionary path of I/O architecture is instructive. We will dwell mainly on tapes and disks, but the same problems and solutions apply to all I/O.

The first commercial magnetic storage drum was used as main memory on the IBM 650, which was first delivered in 1954. Magnetic drums were quickly superseded by magnetic cores for main memory, and since that time, drums have been used as secondary storage devices. Drums are analogous to disks, although they have shorter seek time (one head per track), smaller rotational delay (typically four times higher rotational speed), higher cost, and lower storage capacity per unit area (limited surface area). Therefore, they are not discussed separately.

Early tape systems ran strictly under the control of the CPU. The CPU instruction set contained specific instructions for operating and controlling the tape. In fact, the instructions were very much like those for accessing main memory, and consisted of an operation code and the address of the tape unit. As detailed more fully in the introduction and illustrated in Figure 2, a READ instruction started the tape moving from wherever it happened to be until the next start/stop gap was reached. During the time the tape was moving, everything between the Inter-Record Gap (IRG), or End Of File gap (EOF), or any other start/stop gaps on tape was transferred to or from main memory, as specified by the operation code. The user could store the information desired in any order between these start/stop gaps, but any finer addressing had to be accomplished by using the CPU functions via main memory. Such was the case with computers from middle-1950s to early-1960s vintage, e.g., from the IBM 704 (1955) to the IBM 1401 (1960). If the data between gaps were long and the required information was short, the system was inefficient in two respects: (1) the CPU was busy throughout the entire tape operation; and (2) for nonsequential processing, main memory could be consumed by unnecessary data. Both of these conditions are undesirable.

The introduction of the IBM 777 Tape Record Coordinator in 1956 on the IBM 705II[8] and the subsequent data channels on the IBM 7090 (1960) and on the 7094 ushered in a radical change in both computer architecture and machine organization.[9] These units were the precursors of the System/360 and System/370 channel architecture. In the 7090/7094 systems, the CPU instructions that operated the tape, as well as other I/O, were taken out of the CPU and given to a smaller, special-purpose processor called a *data channel.* These instructions, called *commands*, were still stored in main memory, but they were fetched, decoded, and executed by the channel in conjunction with the tape unit. The processor started the I/O by the use of a few new instructions, which meant that there had been an architectural change. On the 7090 series systems these new instructions addressed up to eight channels and any of ten tape units on each channel. The instructions also gave the memory address of the first I/O command. With that, the CPU could overlap multiple I/O instructions to many channels and tapes for better efficiency. Also, the data channel could select any user-specified number of words between two start/stop gaps to be transferred to or from main memory. All tape information was organized in groups of six 6-bit characters, or 36 bits. Thus, for nonsequential processing, if the user knew which words of the record were desired, the unnecessary data transfers and consumption of main memory could be avoided. If the user did not know the location of the desired words, the entire data stream between the start/stop gaps had to be transferred, which presented no problem if purely sequential processing was being performed. In either case, any finer addressing required the CPU.

In a system with a data channel, the processor essentially needs only one type of instruction for any and all I/O operations. The instruction operation code must specify only that an I/O operation should commence, and the address field need specify only the I/O unit and where the I/O subroutine is located in memory. This architecture quickly evolved to the System/360 channel architecture, with only such practical changes as control signals. One significant architectural change in the System/360 was that the starting address of the I/O subroutine was not stored in the CPU instruction, but rather in a fixed architecture-specified memory location, called the Channel Address Word. This word had to be loaded with the correct I/O subroutine-starting address before the start-I/O instruction was issued. Now, as new I/O devices with different instruction sets (commands) evolve, only the channel and device controller require changes; the CPU architecture remains unaffected. However, the machine organization with respect to I/O would have to undergo significant changes, due mainly to the practical limitations of tapes and disks. This is discussed later in this section.

Disk architecture. Disk systems, due to technical difficulties, became available slightly later than tapes. Thus, their I/O architecture benefited from the tape learning process. Even though early disks were accessed by CPU instructions in much the same manner as tape or main memory (i.e., operation code and address), nevertheless the actual accessing was executed by the disk controller. Thus the CPU could be processing other data already in main memory, and, although the disk instructions were part of the CPU architecture, the CPU did not execute them.

A significant problem with large, fast computers is that of providing sufficient I/O bandwidth to keep the CPU busy.

As discussed earlier and shown in Figures 3 and 4, disks have a greater random accessing capability than tapes. As a result, they were originally considered as an extension of main memory and were organized in very much the same way. For instance, the IBM 1405 disk system (1960), one version of which was marketed as the very popular RAMAC 1401, divided the entire disk storage into fixed words called records with 200 characters per record. Each disk surface contained 100 tracks of 10 records per track. A maximum of 50 disks or 100 surfaces provided 100000 records, or a total of 20 million characters. Each word had an indelible, well-defined address which ranged from 00000 to 99999, stored just prior to the data area. Even though the record on a given track had to be addressed by the Search (Read and Compare) technique previously described, the architected form of addressing was more like main memory addressing. (The first IBM disk system, the IBM 350 RAMAC, which was announced in 1956, was identical but had 50000 records of only 100 characters each.) A single record could be accessed with a single instruction READ/WRITE (Address) and did not require SEEK commands (move arm to track address) followed by SEARCH, TIC (Transfer In Channel to repeat the search until found), then READ/WRITE as in System/360 disk architecture. The addresses and records had fixed physical locations on the disk, and dedicated hardware performed the repetitive read-and-compare operations until the address was located. In essence, the disk was an extension of main memory but with a much slower access time and longer word length. As with all disks, the entire data record of 200 characters was serially read/written. The characters were assembled into main memory words in the disk controller and transferred as such to main memory.

In these early systems, even though the disk controller executed the I/O instructions, these instructions were still part of the CPU architecture. Every new disk system could conceivably require changes in the CPU architecture, which had the effect of discouraging the introduction of new devices. The concept of I/O via a data channel, as used for tapes, was suitable for disks as well as any I/O and became part of the System/360 I/O architecture in 1964.

A significant problem with large, fast computers is that of providing sufficient I/O bandwidth to keep the CPU busy. The System/360 I/O architecture allows up to 256 channels, each with up to 256 devices attached. Practical constraints greatly limited the number of such multiple units that could be attached. However, the evolutionary path, still prevalent today, is that each succeeding generation of computers has increasing numbers of channels and I/O devices, especially disks. Even with all the significant advances in disk technology, the I/O speed has remained quite slow relative to the ever-improving CPU and main memory cycle times. The ideal situation is to match the I/O bandwidth exactly with the CPU rate of processing the data. Thus multiple I/O devices are necessary and can easily be accommodated. A given I/O device can provide only a read or write to main memory once every several hundred or thousand CPU cycles, depending on the model. Because the CPU does not require a main memory access on every cycle, especially if there is a cache, there are many free memory cycles to support many I/O devices. When the CPU and channel require memory access on the same cycle (e.g., during a cache miss and reload or other event), a priority protocol must be used. In some cases, the channel is locked out. If the I/O devices ever become very much faster, many fewer will be required to keep the CPU busy.

Evolution of I/O architecture. The I/O on early systems consisted of simple data, tables, sequential numbers, or records. For commercial data processing, sequential records were processed in a batch mode, where each record was processed in the order of appearance on tape, e.g., a payroll file. Technological advances made both the CPU and I/O faster, denser, cheaper, and with greater capacity. This fueled the demand for even more capability. Larger, more complex files and emerging time-sharing interactive systems all required more random processing capability for larger amounts of data. The limited capacity and high cost of main memory required these files to be stored on higher-capacity, lower-cost secondary-storage devices. In order to achieve high capacity at low cost, secondary-storage systems have had to use continuous media with external read/write transducers and share a few of these over many bits of storage. This is synonymous with I/O device requirements, so that I/O devices have become secondary storage, supplementing main memory. The result has been storage that has relatively slow access time, slow data rate (serial), and a very large, directly addressable unit word (track or sector) compared to main memory.

Larger, more complex files and emerging time-sharing interactive systems all required more random processing capability for larger amounts of data.

To circumvent these limitations and make secondary storage appear more like main memory, both in performance and ease of use, I/O—particularly DASD—has evolved in the following three major areas over the years:

- Hardware
- System organization via virtual memory architecture
- Software utilities in the form of file access methods

In the following sections, hardware and file access methods are discussed, with particular emphasis on DASD.

Hardware. The evolutionary thrust of DASD architecture and system organization has been to make relatively inexpensive, high-capacity storage appear increasingly like main memory in terms of its accessing capability. As discussed previously, early disks were considered as extensions of main memory and were addressed very much like main memory. Thus, each well-defined portion of disk space had a fixed, indelible address. However, the unit of addressability was much larger than typical memory words.

As density increased, the desire for low cost still required the sharing of read/write transducers, i.e., one head per track. The unit of direct addressability, namely a track, became very large in number of bits. This is undesirable, and, in fact, the optimal unit of addressability is closer to a memory word size. Also, files stored on disks had quite variable data areas. Thus, in order both to reduce the size of the addressable unit and to provide a variable record length for more efficient storage, the System/360 variable-track format was introduced and is still widely used. Locating a record on a track requires search and compare operations, which are carried out in conjunction with the channel. Thus, the channel is busy (captured) for up to one disk rotation time.

As the track density continued to increase with time, a record became an increasingly smaller part of the total track length. Having the channel busy for the search over a large number of records in a full track (worst case) is an inefficient method of using the channels. One approach to increasing efficiency is to divide a track into sectors and let the disk controller sense the sector (rotational) position. In this way, the channel can be released for other I/O until the desired sector is reached. Although this provides finer addressing capability, the access time can still encounter a serious delay. For example, if there are many active disk units tied to the same channel, when one desired sector is reached, the channel may be busy with another device and at least one additional full rotational delay is encountered before a second attempt can be made to connect to the channel. One solution is to connect a second channel to the disk unit; this option was made available at the time of introduction of the System/360 DASD.

Another solution is to move some of the logic function out of the channel and into the storage controller and provide buffering, where necessary, between the disk and channel. This has been introduced recently in System/370 3380 DASD/3880 Storage Control, which uses a DASD-cache to hold active data for fast access.[10]

Improvements in read/write head cost and performance permitted other significant improvements in disk accessing capability.

It is interesting to note that in 1979, when the IBM 3370 DASD was introduced, it had two features. First, although there was still only one head per track, two access mechanisms were included, each moving half the heads, so that seek delays to different halves of the disk could be overlapped. Also, the variable-track format was replaced by fixed-block data areas. Each 512-byte block of data had a unique, sequential block identification number that could be used to access the block. A command needed only to specify the block ID number or range of ID numbers desired, and the storage control converted this to the correct physical address. The channel was released until the block was reached. This architecture is very reminiscent of the original RAMAC accessing, and it resembles main memory addressing more than traditional DASD accessing.

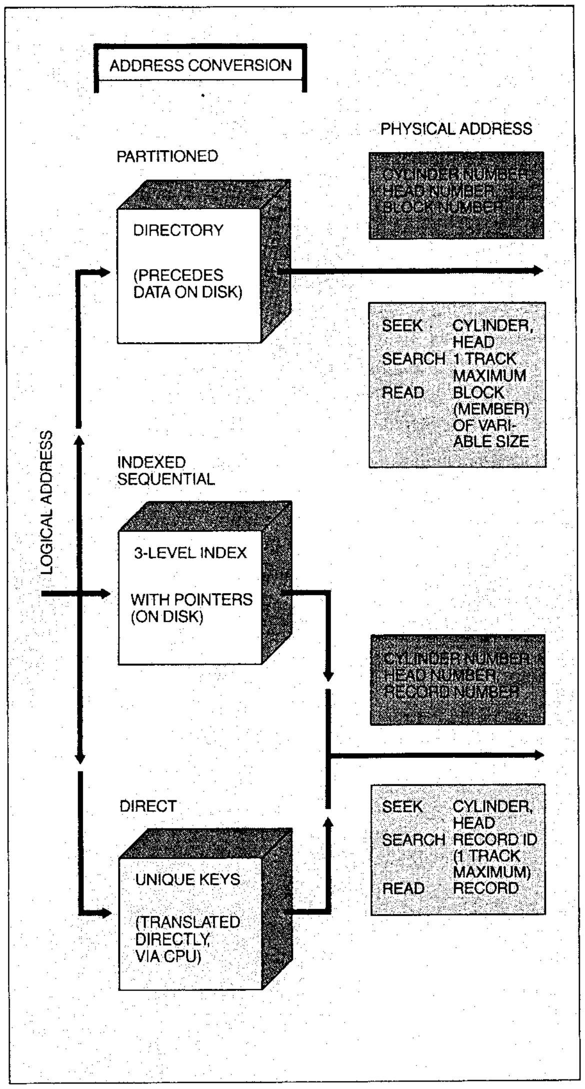
The improvements in read/write head cost and performance, brought about in part by thin-film heads as well as VLSI for high-density, high-speed, low-cost logic, have permitted other significant improvements in disk accessing capability. These factors have permitted a major departure from traditional one-head-per-track, one-access-arm-per-spindle design. The recent 3380 DASD units have two access mechanisms per spindle, two spindles, and two separate control units per disk unit. Also, the channels connect to the disk units via a different interface, namely an IBM 3880 Storage Controller, which has two separate *storage directors*. The essence of all these interfaces is to provide multiple paths between multiple CPUs, channels, and disk units. The paths can be dynamically selected to avoid access delays due to channel misses as well as to allow additional independent access paths to the data on a given spindle, without tying up the CPU (System/370 Extended Architecture). The 3880 DASD cache, previously mentioned, helps in this respect. All these factors contribute significantly to reducing the I/O queue time in a multiprogrammed system[11] and are becoming essential as the central processors continue to improve in instruction execution rate.

Software utilities and access methods. Files of either relatively simple or more complex types (data base) often reside on many cylinders of disk space, and can even appear on multiple disk drives. Once the head/arm has been moved to the correct cylinder, the nature of disk accessing requires at least half a rotation on average or sometimes a full revolution of the disk, regardless of the organization of the data. If the entire cylinder must be searched, one SEEK maximum (slow) and a maximum sequential search through every track in the cylinder are required. With standard System/370 architecture, this searching ties up the channel for the entire time of search; the channel cannot be released even momentarily. A search that requires multiple cylinders is even slower, because arm movement is very slow. A chained search through multiple cylinders would tie up the channel again for the entire time. Even though the channel can be released during the SEEK time between cylinders, when the next cylinder has been selected, the channel may at the same time be busy doing other I/O, thus prolonging the current search. This issue is an overall system queue optimization problem. Hence, lengthy searches are undesirable for two reasons: First, they require a long time, due to the sequential nature of the disk; and second, they tie up the channel, thereby preventing overlapping of other I/O through this channel during the search. Note that on disk systems that provide sector addressing, the channel is not needed during the search for the sector address, because this is done by the disk control unit. For such cases, a sector is equivalent in terms of addressability to a track on a nonsectored system.

The conclusion, for the addressing of files on disk systems, is that the optimal organization and structure are those that provide the correct track (or sector) address in the least time, with the least amount of Stored Addressing Information (SAI). This is exactly what all access methods attempt to do. However, there are typically some additional considerations and constraints. For example, some files are very simple and may be accessed only in a very simple, sequential manner. This gives rise to the *sequential* and *partitioned* organizations and access methods. The former requires virtually no additional Stored Addressing Information, whereas the latter requires a small directory. As would be expected, a file with a more complex organization and accessing requirements needs more SAI and/or more time to find the track and record address. This is typical of *indexed sequential* and *direct* access methods. The basic difference between the latter two is the amount and complexity of the SAI needed to convert the record symbolic name or number to the proper cylinder, head, record address (CHR number). Once this address is obtained, there is no difference between the indexed sequential and direct access methods and only a minor difference between them and the *partitioned* access method. This is illustrated in Figure 6. After the address conversion is accomplished, the partitioned method typically reads/writes a block or a long record, whereas the indexed sequential or direct organizations read/write a smaller record. In the partitioned method, if finer addressing to a smaller unit is required, the entire block must be transferred to main memory (i.e., unnecessary transfers and memory consumption) and processed via the CPU.

The indexed sequential method uses the most additional SAI and takes longer to convert the name to record address. The direct access method uses a simple one-to-one correspondence to convert a symbolic name to a cylinder, head, and record number, but it requires considerable involvement of the user in defining the unique keys with minimal wasted space. All these access methods require some form of associative searching to obtain the desired address. However, because such functions are not available in hardware, these access methods make use of a simple and ancient indexing technique that is described next.

Figure 6 Comparison of logical to physical address conversion of the three major access methods

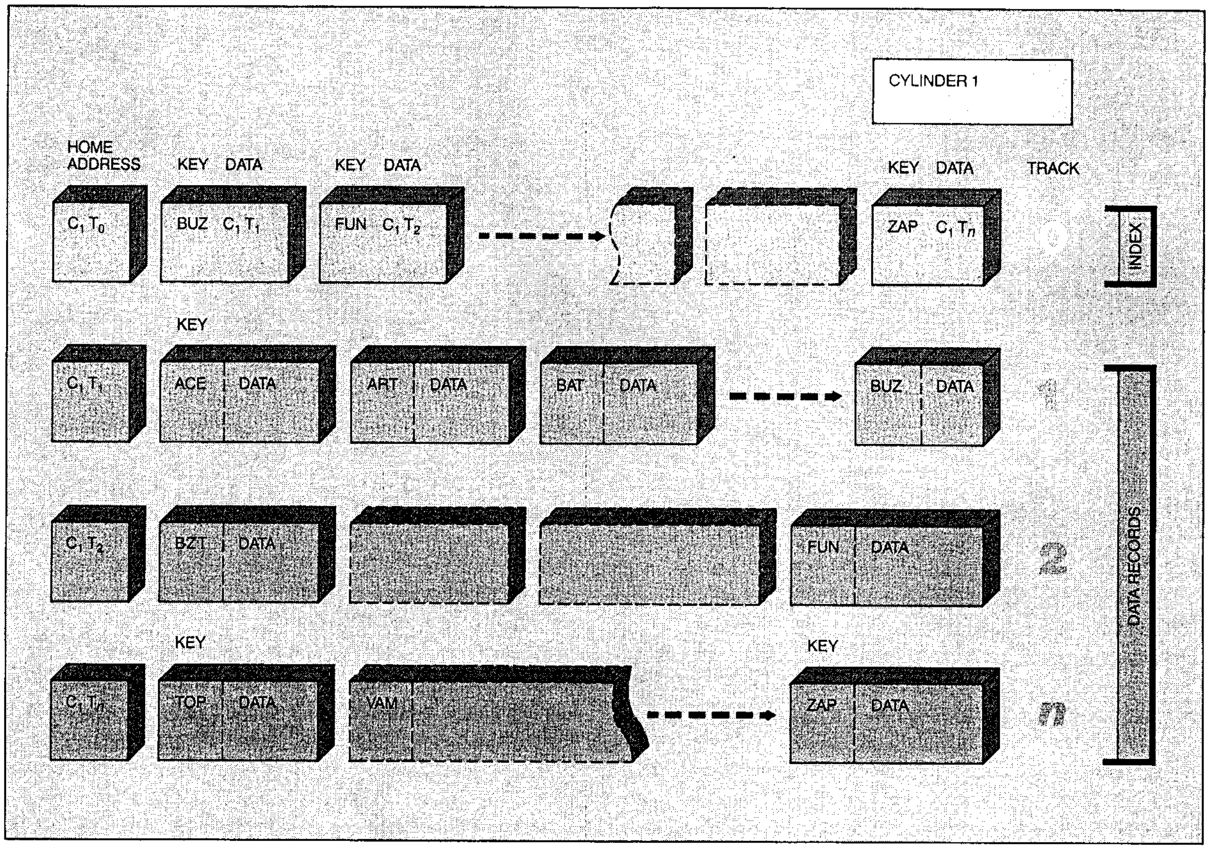


Volume search technique. Whenever a large file is to be accessed, it is desirable to get to the correct track address as quickly as possible, so as to minimize the search time. If the records are of variable length or the names—either alphabetic or numeric—used to reference them do not provide unique key addresses, a software indexing technique is commonly used in various access algorithms to minimize the time required to search for the physical address. This is referred to here as *volume search technique.* It is quite powerful, yet simple and easily understood. Fundamentally, this technique is needed because of the lack of a fully associative accessing capability in the storage system. Associative software is provided in lieu of associative hardware. Consider a file composed of variable-length records that requires $n - 1$ tracks of one cylinder for storage. The index is stored on the first track, so that *n* tracks in total are required. The records have three-letter symbolic names that serve as keys for read and compare (search) addressing, and they have a key-data format, as shown in Figure 7. The records are stored in alphabetic key sequence along the tracks, starting with the first name ACE on track 1 and continuing through to record ZAP on track *n*, all on cylinder 1. The index, stored on track 0, consists of records of fixed length having a key for addressing followed by data. There is one such record for each track of data, the key being the name of the record on that track with the highest alphabetic value. The associated data constitute the home address of the track containing this data record. Thus the first record of the index has Key = BUZ and data of C1 T1 (cylinder 1, track 1), where BUZ is the largest name on track 1. The second record of the index points to C1 T2, which has FUN as the largest name, and so on. If we wish to access record VAM, a search is made through the index, looking for a key that is greater than or equal to VAM, i.e., SEARCH KEY > or = VAM. This match occurs on the last record of the index. The data portion is read, giving a pointer to C1 T*n*. Now a search is made of track *n* for KEY = VAM. When obtained, the data can be read/written.

Obviously this technique can be generalized to cases where the file and/or index requires multiple cylinders, multiple disk units, etc. Whenever the file requires multiple cylinders, a second level of index becomes desirable to point to the correct track for the start of the record index, so as to avoid long searches. When the index itself requires several tracks, another level of index becomes desirable for similar reasons. This technique is used in partitioned and indexed sequential organizations.

Sequential access method. In the simplest, purest sequential organization, records are stored in tandem

Figure 7 Example of index and record organization with keys for simple volume search



fashion, starting at one track of a cylinder and continuing through as many tracks as necessary. Only the cylinder, head, and record number for the start of the file are available for addressing and no other SAI is provided. Obviously, because of this, any finer addressing requires a transfer to memory and use of CPU logic. Of course, a user may provide additional keys, while still using a sequential organization. These keys would be used for an occasional random search. Such things are done, and one hardly ever finds an access method used in its pure sense. If the sequential organization requires frequent random accessing to small records, a reorganization to index or direct access becomes desirable.

Partitioned access method. Suppose a sequential file, LIBSUB, is composed of two library subroutines, SubA and SubB, stored sequentially in that order. Further, suppose only one of these is typically used at any one time by an application program. If SubB is needed, it is inefficient and wasteful to read all of SubA and then SubB to main memory. A better way is to give each member a sequential record address (cylinder, head, and record number), which requires a count area (SAI) for each member. Also, a directory is needed at the start of the file to relate symbolic names to the physical address, CHR number. The simplest type of directory consists of two separate records, each with a key and data. The keys are SubA and SubB, and data are Cylinder = x, Head = y, and Record Number = z, and Cylinder = x, Head = y, and Record Number = $z + 1$, respectively. Now an access to SubB first requires a sequential search through the directory records on their keys. A match on SubB and immediate read of its data give the CHR number of the desired data record, SubB. If the record is stored on the same track, it can be read within the same revolution. If there are additional members to the file, they can be treated in an analogous fashion. If the number of members becomes too large, the additional SAI for each record in the directory and even for each member may consume substantial storage space. It may be desirable to *block* the directory records and member records to reduce the SAI. If blocking is used for the directory, the records within the block should be organized so as to allow the volume search technique previously described to be used, namely, Search On Key > or = Given Name. The IBM partitioned access method has certain rules about how this is done, but the fundamental purpose of the access method is still the same.

Indexed sequential access method. The indexed sequential access method is basically a generalization of the partitioned organization, wherein the members become individual smaller records of typically 256 bytes or fewer. The directory becomes a two- or three-level index structure organized and accessed

Batch processing is done more efficiently with a sequential type of organization, whereas on-line processing requires a more random type of access.

by a generalized volume search technique described previously. The conversion of the symbolic name to physical CHR number via the index may take two, three, or more track searches. In large files, this can become an even more lengthy search, which makes this a relatively slow process. Additions and deletions of records and indices can also become a complex process. However, it provides a rather versatile access method that can handle a wide variety of files, and the indexing is transparent to the user if an IBM utility such as ISAM is used.

Direct access method. In its pure form, all records of a direct organization contain a unique key that is defined by the user. This key may convert to the correct CHR number directly, or it may require a small transformation via the CPU. In either case, the conversion is fast, and access time is minimized. However, the problem of key definition, record insertions, and deletions can be troublesome and time-consuming.

Another access method, Virtual Storage Access Method (VSAM), evolved from the access methods just discussed. In many applications it is desirable to be able to process a data base sometimes in batch mode and sometimes in on-line mode. Batch processing is done more efficiently with a sequential type of organization, whereas on-line requires a more random type of access, as provided by either indexed sequential or direct organization. The essence of VSAM, with respect to data accessing capabilities, is that it allows either or both, at the discretion of the user. The 1973 version of VSAM provided the functional equivalent of a sequential and an indexed sequential organization on DASD. In 1975, a new version of VSAM added the functional equivalent of direct organization. An enhanced version, Virtual Storage Extended VSAM, provided improvement in performance, usability, and functions but basically maintained the same accessing capability. There are other features unique to VSAM, such as device independence and ease of inserting new records.

Faster access to memory can be obtained by using a small register stack that is loaded and unloaded from memory.

Thus we see that the fundamental differences in all access methods are (1) the amount and type of addressing information, (2) where it is stored, (3) how it is organized, accessed, and used, (4) the size of the unit (record) addressed for transfer to main memory, and (5) the method for adding and deleting records. These are all related either directly or indirectly to the addressing (accessing) capability of secondary-storage devices. Most, if not all, of this file and I/O architecture would disappear if the proper type of memory could be built at a reasonable cost and effort. Unfortunately, this is not possible today, although it may become possible in the future, as discussed later in this paper.

Data base architecture

File access methods are exclusively concerned with file organization for fast, efficient disk accessing. Data base systems in common usage today often make use of techniques borrowed from file access methods, but they provide other services as well. The architecture of data base systems attempts to give an overall structure to data so as to achieve three major goals:

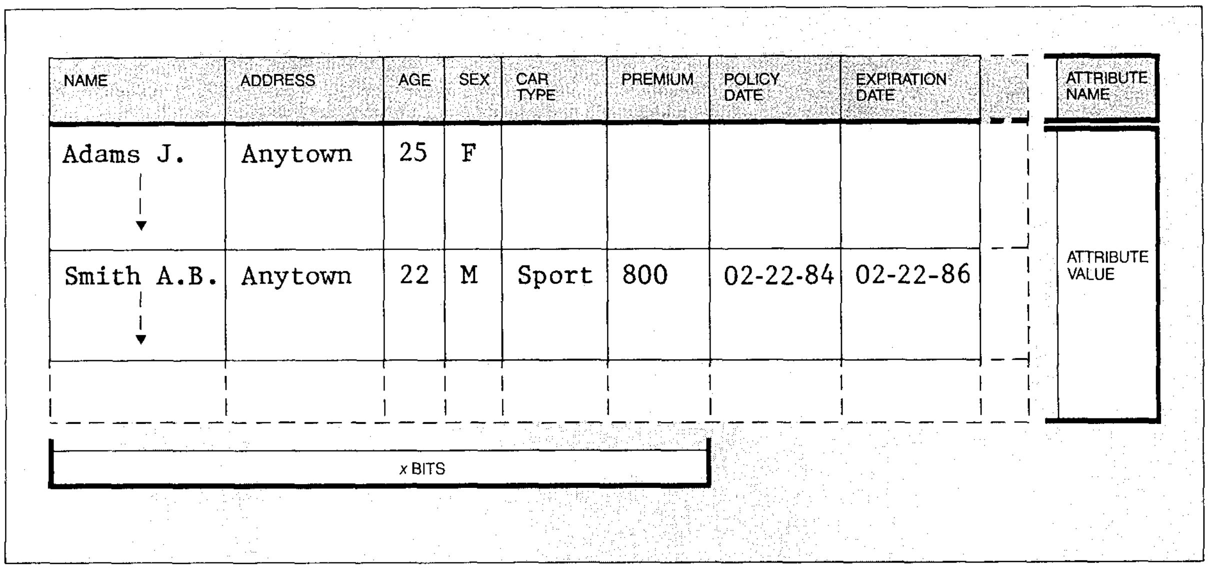
- Provide a compact, efficient file structure with fast disk access.
- Minimize the amount of unnecessary data transfers from disk to memory.
- Provide means for fast, efficient associative-type processing, typically on a general-purpose CPU and memory. (Data base machines are discussed later.)

Data base and complex files require larger storage than main memory can usually provide. Hence, secondary storage is used for such applications—tapes in early systems and disks with tape-like archives in more recent times. A major simplification of system design and improved performance could be obtained if the memory hierarchy could be eliminated and replaced by one very large main memory. This would be particularly true for computational problems, where the addresses of the arguments and results are relatively well known. Although data base problems would also be considerably simplified, they still have some other fundamental accessing requirements that are not solved simply by larger memory. As outlined in the introduction, data base access usually requires some logical operations on the stored data before the address of the desired information can be determined. This requires that a large amount of data be searched against a given criterion (i.e., equal or greater) to find the fields that satisfy the criterion.

The fundamental issue in all file and data base operations is finding or generating the address of the desired item or items.

The smallest unit that might be used to compare against in any file is typically a byte. Making a random access memory that can access any arbitrary byte in a very large file is expensive, but it is, nevertheless, possible. However, that is only one small part of the problem. The essential problem is that theoretically, for a general file, there are an infinite number of *addresses* that can be used to access the data. These addresses are often referred to as *access paths,* because the usual method of implementation uses several levels of indices and lengthy paths through them for finding the address CHR number. This large number of addresses comes about from the fact that the address of the desired information quite often depends on the existence or nonexistence of certain bytes or combinations of bytes in an arbitrary length of bytes (i.e., the record). Also, this combination of bytes, to be used either as the address or to compare against, changes dynamically for different user requests. For instance, consider storing and accessing in main memory the automobile insurance file described earlier in this paper. Assume

Figure 8 Insurance file organized as two-dimensional table



that each client's record can be stored as one logical word in main memory and that the names are externally converted into the binary word address for accessing. We ignore the problem of obtaining a contiguous, unique address. Each item in the record is stored as a field of given length, as shown in Figure 8. If we wish to access the record for, say, A. B. Smith, the name is converted to a binary address (ASCII or EBCDIC to binary) either in our brains, by the keyboard, or by a subroutine in the processor. This address provides a direct hardware path to the desired record, and we can print or process any of the fields of the record.

This is, unfortunately, not the only type of access to a file. Suppose we wish to retrieve records for all clients who are male. There is no address available for such a direct access. A simple solution would be to store the records as just described, but catenate the male/female as one bit of the address—for example, the higher-order bit. This would separate the records into two logical portions, one for males and one for females. Now suppose we wish to access all clients who are male and age 25 or under. There are two problems: (1) There is no direct address for these records; and (2) There will probably be multiple records to be accessed, because the address is not unique.

In theory, we can conceive of a memory that has multiple access paths in the form of multiple address ports, one for every possible combination of fields of the record that might be used for accessing. This is very impractical, because the possible number of such addresses can be very large—approaching infinity—and typical memories have but one memory address port. Furthermore, in the multiport address case, it would be desirable—although not essential—to have as many data in/out ports as there are possible multiple records satisfying the address criterion. In general, this is equal to all the records in memory, or one data port per logical word in our case. Memories typically have one data port. Hence, the user is forced into simulating a multiport memory with a single-port memory. It should be immediately obvious that this requires many accesses to main memory and is therefore slow.

How is this actually done? In the simplest case, the user first stores the correct bit string for SEX AGE into a general-purpose register, as required for the search. Next, each record (word, in this case) of memory is accessed for those particular fields, which are compared against the desired bit string in the general-purpose register, using the arithmetic/logic unit of the processor. Note that the program must either know which fields are the correct ones or perform a sequential search. All words that fulfill the search criteria are stored in a separate array or are written out to disk. Each word in the file must be processed sequentially in this manner, which can take quite a long time. The problem is compounded by the fact that the original file is typically much more complex to start with and will be stored on disk or some other secondary storage medium. The complexity is further compounded by the fact that the required data are often distributed over several separate files, and these can be organized with different access methods. There can be large amounts of redundant information and/or information not pertinent to the task at hand. Some of this information can only be judged as irrelevant by testing against the search criteria. Other data may be known to be irrelevant but are buried within the record on disk. In such cases, the entire record including irrelevant fields must be transferred to main memory, because there is no fine addressing capability on disk.

In the example of locating records for male clients of age 25 or less, the file was loaded into main memory and searched word by word or field by field. For the more complex files, we can do the same thing, but with a high cost. First, enormous amounts of irrelevant data (undesired fields) are transferred from, say, a disk to main memory, which consumes a precious system resource, the memory bandwidth. Second, the files are often larger than main memory and, therefore, must be brought in piecemeal. Third, the search through every record again consumes memory bandwidth as well as processor cycles. Also, the records do not normally fit nicely into logical words, so additional processing is required to find the address of relevant data. Fourth, the total data needed for the search criteria are often contained in two or more files. This requires a search on the first file, creation of a new file that satisfies the first search, and using the new file either to access the second file directly and compare (simple case) or to cross-search all entries of the second file with each entry of the new file (complex case).

In the example, suppose the client's SEX is indicated in the first file and AGE in the second. Assuming both files are accessed in terms of client NAME, we search the first file for all clients who are MALE and create a new file of matched cases. Because there is only one possible entry in the second file for each client name in the new file, the client names of the new file are used as direct addresses (entry points) to the second file for a comparison on AGE. Matches are flagged or written. A much more complex search (akin to relational JOIN operation) is required if the files are accessed with different entry addresses. Suppose the second file is accessed by PREMIUM instead of by NAME. The first record of the new file must then be compared with every record of the second, the next record of the new file likewise must be compared with every record of the second file, and so on through a very lengthy process.

Theoretically, many of these problems could be eliminated by a high-speed, large-capacity, suitably multiported memory, with associative compare capability and high-speed and high-bandwidth I/O to keep it adequately supplied. However, such a system is not practical today in terms of cost, complexity, and speed. The memory arrays would be extremely unwieldy and slow because of the circuit loading and low density of storage cells. The packaging and wiring would be complex and large in size, thus further reducing the speed. Suitable I/O devices are very expensive and of low capacity.

Software systems attempt to make a general-purpose system behave as a very large, high-function associative processing system.

As a result of these limitations of the memory system, other means have been used as well as proposed to achieve the same result. These means have taken two main directions, one providing hardware-assist features in various forms and the other using the general-purpose hardware already available, combined with system subroutines and algorithms—software.

Data base hardware. Over the years, there have been many attempts to provide data base oriented machines—in fact, too many to discuss separately. The general approaches have been (1) to provide a separate small, fast associative array processor that handles all the complex compare and flag functions, rather than the CPU, and (2) to provide small associative preprocessors at the disks that preprocess the data, thereby reducing both the amount of unnecessary data transferred and the final amount of CPU processing.

Nearly all such machines have been experimental, though there have been a few commercial machines. The reasons for this are that there are two bottlenecks in data base processing: (1) the complex associative compare addressing requirements; and (2) the large disk I/O bandwidth requirement. Boral and DeWitt[12] offer an excellent discussion of these issues with respect to many different hardware approaches. All data base hardware approaches have focused on the first bottleneck and have thus been limited by the I/O bandwidth requirement. In addition, it has never been clear that the proposed additional hardware and the resulting complexity are really any better than using a high-speed, general-purpose CPU, which is a simpler system.

Data base software. Large data base software systems that have found widespread usage have, from earliest times, been used with general-purpose processors and standard memories. In essence, software systems attempt to make a general-purpose system behave as a very large, high-function associative processing system, and do so with reasonable interactive response time. To achieve this, they must strive to achieve all three goals indicated at the beginning of this section. The extent to which these goals are achieved varies among the common data base systems—GIS, IMS, CODASYL—and results from the conflicting and often contradictory demands.

Early data base systems provided multiple address ports (access paths) into the data on disk by the use of various indices and pointers. These access paths were contingent upon the method of organizing the data—hierarchical tree or network—as well as assumptions as to the types of associative searches that were to be performed. These access paths could become quite long and intricately interwoven. The problem is that if it is desirable to have fast access to a particular field of a random record residing on disk, this field should be indexed in a multilevel indexing scheme analogous to the keys in the volume search technique previously discussed. However, every field of the data can, in principle, be used for the associative search. Hence, for a large data base, not only can the index consume large volumes of data space, but the search through various levels of the index can be slow. Thus only certain essential fields are typically indexed. If one were to attempt a search that was not inherent in the access paths, this might be either impossible or extremely difficult and slow. The speed with which an index can process a query is highly dependent on the amount of clustering of indices, whenever a paged, virtual-storage hierarchy is used. The indexed keys are usually stored in sorted sequence. If a sequence of records, contiguous by key, is processed, one reference to the disk will transfer many usable records to main memory.

The data organization should allow completely general associative searching capability to any fields or combination of fields in the data.

On the other hand, if the sequence is contiguous not by key but by some nonindexed field, a page transfer may be required for each record, a severe performance penalty.

In complex indexing structures, changes in the indices could inadvertently remove some pointer and hence eliminate some access paths. The essence of typical data base access methods is to provide a minimal indexing scheme that satisfies the majority of search operations. If an occasional unusual search is performed, the occasional penalty is tolerable. All these problems, obviously, are related to the limited accessing capability inherent in the storage system. Ideally, the organization of the access paths should be independent of the data; that is, the organization should allow completely general associative searching capability to any fields or combination of fields in the data. The relational data base systems IBM DB2 and SQL/DS attempt to be less restrictive and provide data independence. The essential idea is to store data in relatively simple relational record form with as few predefined access paths as possible.[13,14] Whenever a complex function, such as join, is required, its access paths are created on the fly and maintained until they are no longer required. The concept is simple in principle, but an efficient, fast implementation is essential and quite complex.[13]

In summary, data base hardware and software architectures are attempts to approximate an associative searching system using ordinary, general-purpose hardware in order to circumvent the need for a memory system with inherent associative capability.

This approach is driven by the practical constraints imposed by today's technologies.

Gedanken experiment: Toward the future

The essential ideas developed thus far have been the following: (1) The major limitation on computing systems has been the lack of an ideal memory; (2) The ideal memory system is complex, expensive, and probably unattainable; (3) Actual memory hierarchies attempt to give some appearance of an ideal system at low cost, but with severe performance limitations and great complexity.

Even given unlimited resources, it is not clear that the ideal or near-ideal memory system is attainable in reality. However, we can image what it might look like and how this might change current systems. Such a thought experiment would not only bring current systems into proper perspective but perhaps would also point the direction for future goals. We will now perform such a *gedanken* or *thought* experiment.

Assume that the density and cost of integrated circuit technology has progressed to a point that allows the construction of a very large, fast, complex main memory that is very cheap. We also assume a few other innovations. One is that the main memory has five primary ports that can support simultaneously three (4 or 5) reads and two (1 or 0, respectively) writes, with a cycle time equal to that of any CPU we wish to build. We assume totally independent, asynchronous secondary ports that can either read or write to the memory. These secondary ports can be serial ports, and they are implemented at the chip level.[15] Thus, with proper organization, there can be many asynchronous secondary ports. The implication of these assumptions is that we can have a main memory of extremely large size (say 4 gigabytes) that allows 32-bit addresses. The storage array is assumed to have certain other special features. For example, a separate portion of the logical words have fully associative capability to do the following comparisons: equal, not equal, high, and low. The associative part does not have the multiple-primary-port feature but rather only the usual one primary port. However, this part as well as the entire memory has the asynchronous-secondary-port feature. The remaining words do not have this associative feature but have the multiport feature on the primary port, as shown in Figure 9. The associative part consists of a number of 4K-byte (virtual) pages that are used to do associative searching (that is, they are content-addressable). Only part of the memory is associative because that is all that is required, and the additional complexity with its cost and performance degradation is important, even in this *gedanken* experiment. (The exact number is an overall system optimization issue that is not easily specified independently of processor versus I/O speed.) The memory is logically organized as words of 128 bits. These 128 bits in all words of the associative portion are compared in one cycle with the 128 bits in a *compare-data register*, through the *compare mask*. The compare mask provides a finer specification of which bits are to be compared and used for associative addressing where the data item fields can have any bit length. Each word has two *match flags* associated with it. All flags can be compared on one cycle, with the bits set in the *compare flags* register, through the *compare flag mask* register. These flags, in combination with the associative pages, the nonassociative pages, and multiport features, provide an extremely fast, efficient, and simple way to do complex associative (i.e., content-addressable) searching of files, which is the essence of all data base operations.

Even with such a large main memory, not all the information that will ever be required or processed by the CPU can be contained within it.

Even with such a large main memory, not all the information that will ever be required or processed by the CPU can be contained within it. Thus, some other storage system will be needed. We might postulate that these are other similar memory systems that are switched to as needed, or they are some other low-cost archival storage. We assume some disk-like archival secondary storage so as to be realistic. However, the relatively slow data rate of such secondary storage is always a bottleneck. Thus, we further assume that words are stored across parallel disks and that the asynchronous operation of these

Figure 9 Schematic of one possible ideal memory system

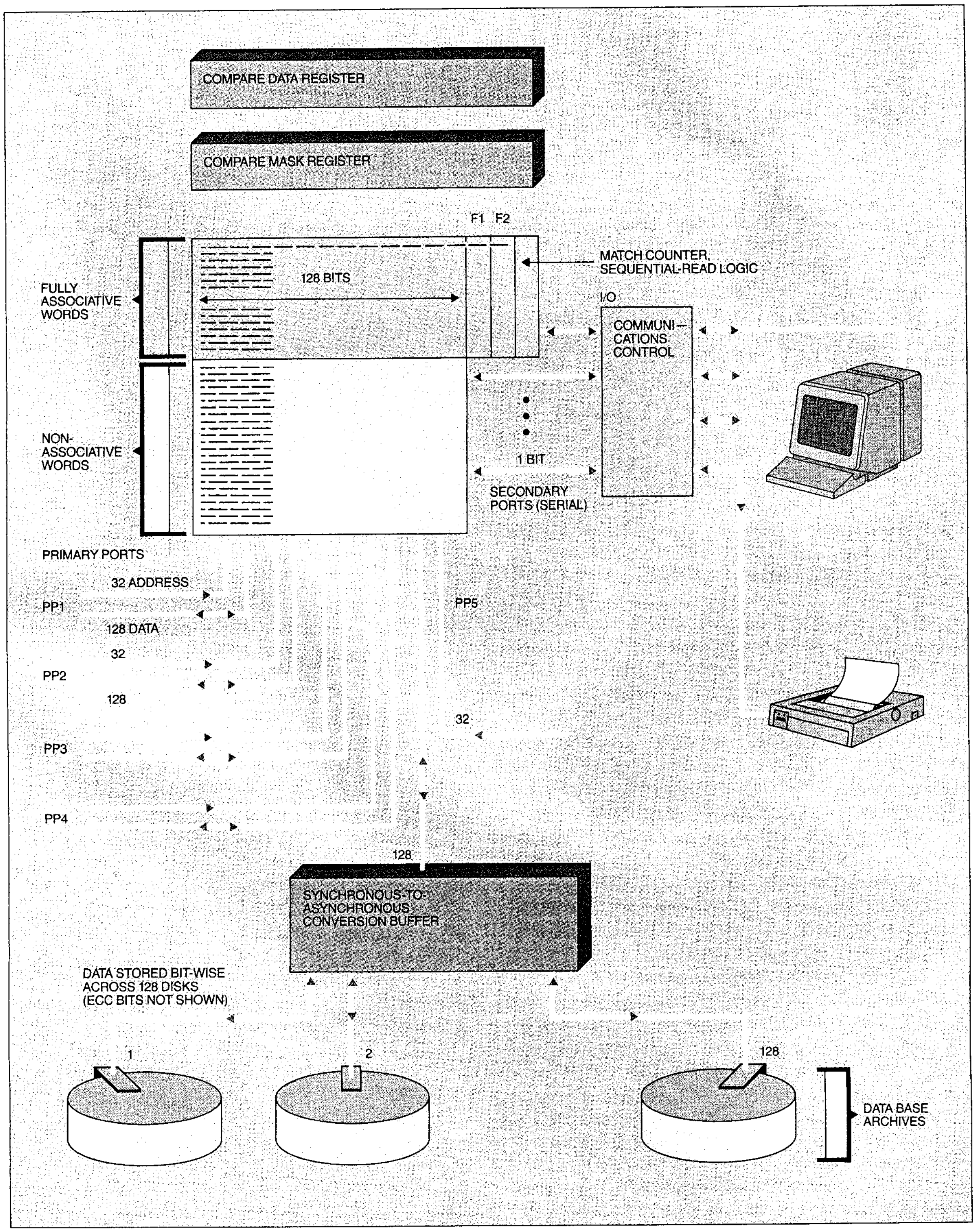
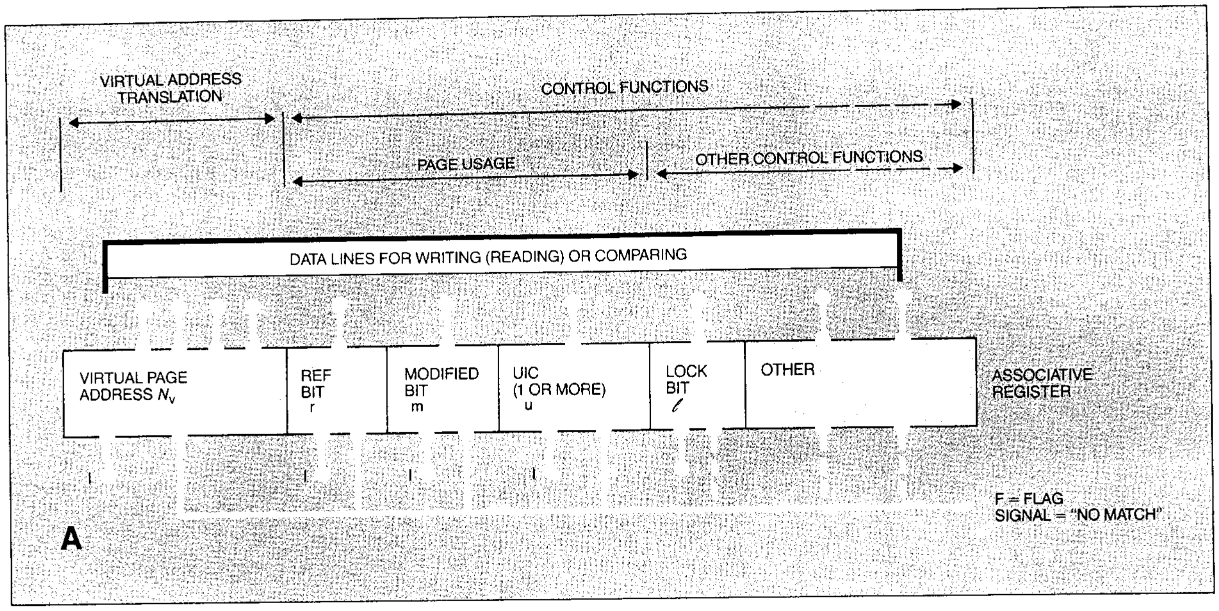


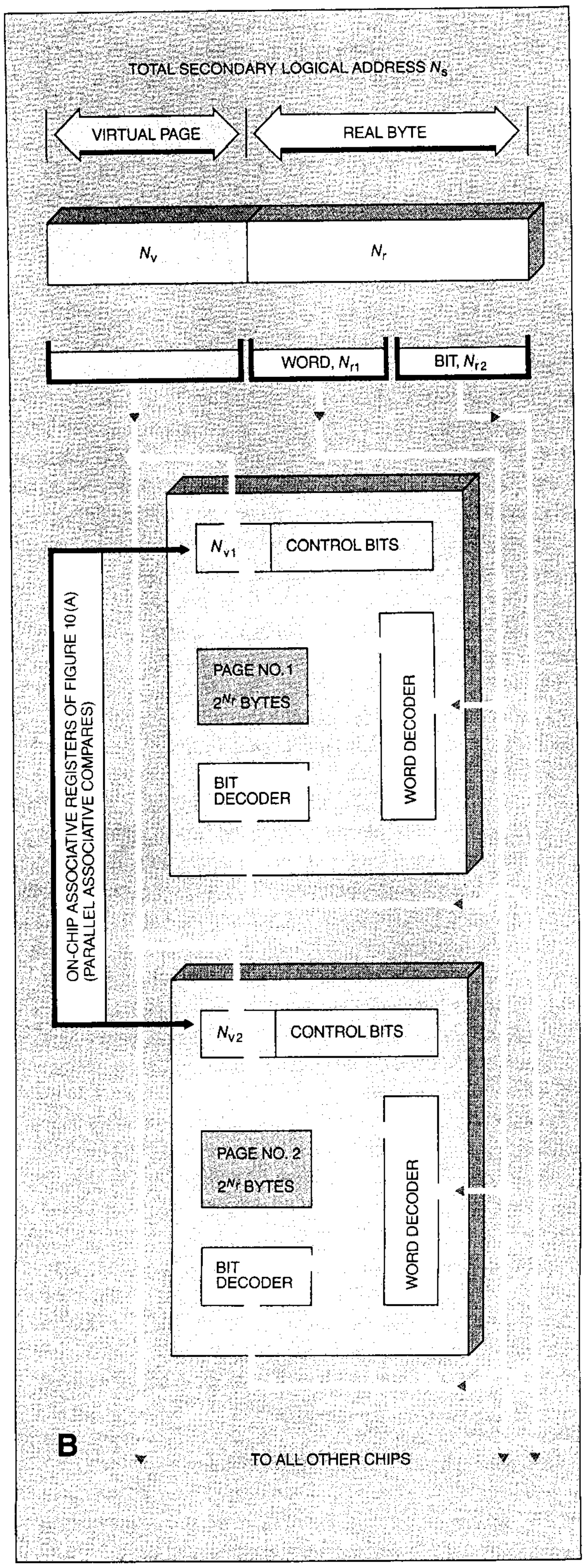
Figure 10 Schematic of hybrid memory: (A) organization of associative address register with control bits; (B) memory organization showing address translation using on-chip associative address registers



parallel disks is buffered by a two-port memory, which is very much like main memory but with fewer ports.[11,16] This solves the I/O bandwidth problem encountered in current data base machines.

Thus, our overall memory system is organized as shown in Figure 9. All input and output to the system are communicated by way of the asynchronous secondary ports in main memory, both for interactive terminals and for batch jobs. In other words, the disk Simultaneous Peripheral Operations On-Line (SPOOLING) buffers used today for such purposes are replaced by these asynchronous secondary ports. Such ports, which are partially available today, can have extremely high serial data rates and can be run independently of the primary ports that are servicing the CPU. The instruction length is increased to 128 bits, which allows 16 bits for operation code plus three fields of 32 bits each for three operands, plus 16 additional bits that can be used for the USER ID. Four of the primary ports are used for normal processing, typically with one to fetch the next instruction, two ports for two arguments read from memory, and one port for a result stored into memory. The memory is capable of all these accesses on one cycle. In cases where an instruction requiring multiple execution cycles is followed by single-cycle instructions, it is possible to end up with two (or more) results to be stored. In such a case, an instruction fetch may not be needed; this port can then be used to store two at a time simultaneously to memory. The fifth port connects to the on-line archival storage as shown.

With such a system, there is no need for the general-purpose register stack, because all needed parameters are available directly from main memory as fast as they would be from a register stack. Furthermore, there is no need for instructions and resulting wasted cycles to load or unload registers, as well as no need for special separate register addressing within the instructions. Even though the memory is quite large, eventually we will be faced with the old problem of fragmentation of real memory space, unless some form of relocation is inherent. Thus a virtual memory is used, but with a different form than that commonly in use today. The full virtual address consists of a 16-bit user ID (obtained from a register or the additional 16 bits in operation code), which is catenated to the 32-bit memory address. This virtual address is translated to the real physical address, but the mechanism for doing this translation is contained directly within the memory. This organization is known as *hybrid memory*[1] and is shown in Figure 10. It eliminates the need for page and segment tables and provides a method for complete general sharing



of common or private data at the page or segment level.[17] Any number of users can privately share pages with a given virtual address, while the same virtual address can be used to refer to a different page in another shared or private address space. This sharing is obtained by the use of two additional virtual address bits in the Associative Address Registers (AAR). These two bits are used in a special manner, with a very small hardware-implemented *shared-segment directory*.

In essence, the effective system virtual address is 50 bits, which provides many benefits. Whenever a new chunk of virtual space, either a segment, several segments, or part of a segment, must be allocated (GETMAIN or related function in System/370), there is no need to search the tables for that length of contiguous table entries. Nor is there any need to compact the address near the starting value, because tables are not used for translation. Any contiguous, random, systematic and noncontiguous, or nonsystematic and noncontiguous address is acceptable to the hybrid translation hardware. This provides greater freedom in allocating any new portion of virtual space. It is only necessary to ensure that the identical virtual address is not used more than once in the same user's given address space.

With this enormous virtual address space, any fragmentation of the virtual space for a given user at a given time is of no consequence. The result is that there are no enormous translation tables to consume main memory. Furthermore, the waste of main memory due to unusable table space that results from virtual address fragmentation is eliminated, thereby providing substantial savings. Thus all subroutines, system procedures, and so forth can be referred to symbolically, and there is no need to translate symbolic names to new, internal virtual addresses at link time, as is done today. The IBM System/38 uses a similar symbolic addressing, but that is implemented with very different and more conventional hardware. Also, there is no need for address generation, which removes one bothersome pipeline stage of the CPU and gives a faster execution rate. The base register is replaced by the symbolic user ID name as the higher-order 16 bits of the virtual address.

Input from terminals or any slow I/O device is communicated via the asynchronous secondary ports as shown. Virtual addresses for these ports are provided by the user (e.g., file name) or assigned by a supervisor. Physical pages in a memory module are assigned to these devices on a demand basis. The virtual address, which consists of the user ID plus the user-assigned file name, is stored in the AAR. Subsequent pages are given sequential virtual page numbers by the controlling system and do not require searching page tables for a contiguous set of available entries. Nonsequential page addresses or incrementing on higher-order bits are not different from sequential addresses, as far as the hardware address translation is concerned. The input information goes serially to the asynchronous secondary port of the assigned physical pages, with the correct virtual address stored in the AAR for each page. A control bit can be maintained in the AAR to indicate that the data are in the secondary port. When a page is full, either the page buffer can be transferred immediately to the array, or we can wait until a reference is made to the page. The control bit will show that the page is in the page buffer and it is first written to the array before a normal access. If desired, all I/O can be done via virtual addresses, using the AAR with the control bit just mentioned. In any case, the secondary ports replace the disk-spool buffer of today. Once the input is complete, the user issues a system command that specifies the operation, e.g., RUN, plus the starting virtual address. The system queue holds this information until the job is scheduled to run. If desired or if necessary, programs do not need to have contiguous sequential virtual addresses, because there are no translation tables. Noncontiguous addresses across page boundaries in a program can be easily handled with a simple GOTO (virtual address). It is necessary only that the virtual address has not already been used in this user's particular address space. A simple single-cycle check of its usage for a page resident in main memory is easily done by the operation *read virtual access* = (*given address*), i.e., a content search on the virtual addresses in the Associative Address Registers.

The user output can be stored starting at any convenient virtual address in the user's virtual address space, for example, starting at the very top and working downward.

Our system will be multitasked so that a reference to off-line data causes the equivalent of today's page fault and subsequent task switch. However, the replaced page can be the entire file or any large or small part of it, as necessary. The secondary ports in main memory make this feasible and very fast.

Complex data base operations. In the *gedanken* experimental system, files can be organized as variable-length records and stored in their raw form with neither indices nor any predefined access paths; i.e., they are data-independent. This is precisely the situation that all data base access methods strive to achieve but never quite attain. Each file can have its own structure. It is necessary only that all records of a given file have the identical form and fields, and that the programmers know or have access to a descriptor of this structure for each file. Whenever a large file requires a complex search, the associative portion of the memory is used. The file is transferred in from disk via the secondary port in a convenient unit, such as one page at a time, and it is processed a page at a time. After the first page is entered, it can be associatively processed quite rapidly before the second page is transferred. The virtual address and page-valid bits of the AAR of the memory provide the means to selectively enable any desired page of the associative memory or any other portion of memory.

To see how this system operates, consider again the insurance file example previously discussed, for which it is desired to find the names of all clients who satisfy the criteria SEX = M, AGE = or < 25, PREMIUM = or < 1000. The ideal situation occurs if the file is organized as one contiguous record per client, containing all pertinent information for that client, as that shown previously in Figure 8. It is assumed that we know or have access to the identity of each field. The compare mask in Figure 9 is set up to logically compare only the fields SEX =, AGE = or <, Premium = or <, and the compare data have appropriately aligned data M, 25, 1000. In one cycle, all words of page 1 are interrogated and matches are indicated by flags F1 set at each matched word. These flags are equipped with hardware to supply sequential-priority enable signals to the corresponding flagged words for read/write and a count register indicating the number of matched words. Thus, if there are three matched words and three F1 flags valid, a sequential read operation will read the first matched word on the first read cycle and decrement the counter by one, the second matched word on the second sequential read cycle, and third word on the third cycle, until the count is zero. Each F1 flag of the read word is reset automatically to invalid after each readout. These words can be placed in a new file in the nonassociative part of memory and written out to I/O. Subsequent pages of the file are treated in the same manner until the file is fully processed.

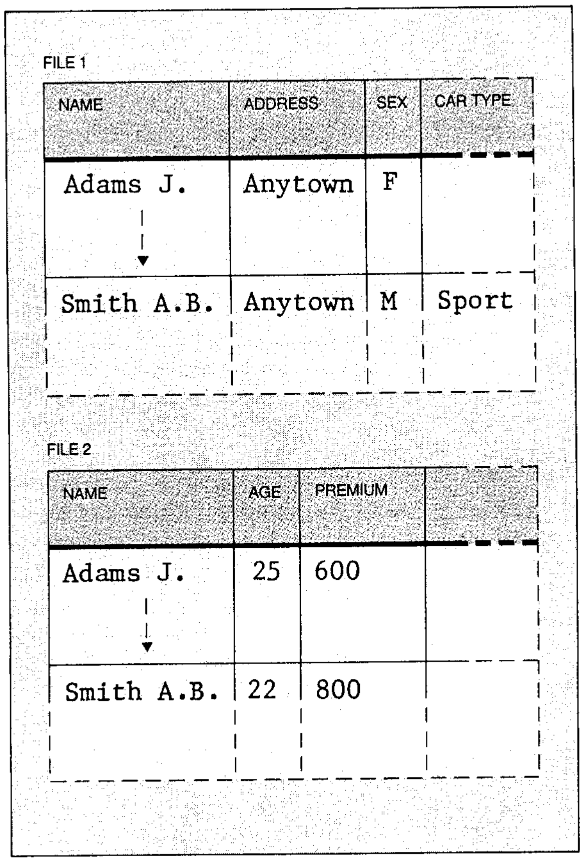
In some cases the full length of the record may exceed the 128-bit word length of memory, i.e., $x = 128$ in

Figure 8. The associative search must then be done in parts. This capability is provided by the two flags, each of which can be masked or used as part of the compare field. Suppose in the above example that the search criterion also included POLICY DATE < 1985 and that this field starts at bit number 129 of the record in Figure 8; i.e., it is too long to fit within the 128-bit word on the first transfer. There are a number of ways to handle this, but a simple way is to transfer the remainder of the record into a second page and treat this case similarly to the two-file example given later in this paper.

Unfortunately, files are never static, but rather have new fields and search requirements added continuously. Thus, typically, the desired information is contained in two or more files and in the worst case requires a many-to-many type of search, as follows. Suppose the insurance file is really two separate files, organized as shown in Figure 11. Neither file by itself contains sufficient information to perform the total access. Hence, the processing is as follows (although this is not necessarily the best way).

Page 1 of file 1 is transferred to memory. An associative search on SEX = M sets flags F1 on all matching words. A single sequential read of this page produces the record of the first male. The name field is extracted and placed in the COMPARE DATA register. In the meantime, page 1 of file 2 has been loaded. The data and mask for PREMIUM = or < 1000, and AGE = or < 25 are added to the COMPARE DATA register as well as corresponding mask bits. An associative search on page 1 of file 2 is done on one cycle using these data, yielding some match flags, F1 in this page. These matches are stored in a nonassociative part of memory, using the sequential read operation. The second matched name of page 1 file 1 is read and the above process repeated on page 1 file 2. The process is repeated for matched word 3 of page 1 file 1, and so on. By this time, other pages of file 1 and/or 2 will have been loaded. Assume page 2 file 2 is loaded. The three matches of page 1 file 1 are reused to process this new page. Either the three previous matches in page 1 file 1 can be reaccessed by an associative search, or the previous matches can be saved. The matches are saved by the use of flags F2. Whenever a sequential read of a matched word occurs, the corresponding F1 is reset invalid, but the corresponding flag F2 is set valid. A sequential read via flags F2 can be done, thereby resetting F2 but setting the corresponding F1 to valid again. Hence multiple excursions through this page can be made without having to save the COMPARE DATA and MASK

Figure 11 Insurance file with data organized in two separate files



and redoing the search. After all pages of file 2 have been searched, the sequence must be repeated with all subsequent pages of file 1 matched against all pages of file 2. This is the worst case of a many-to-many search. A many-to-one or one-to-many search would be faster as well as simpler.

The above examples implicitly assumed that the files all had a 2D table-like structure with all fields present. This can often lead to a large number of sparsely populated fields and wasted storage space. Compaction can be obtained by storing only valid fields and storing each data item as a fixed-length pair ATTRIBUTE NAME: ATTRIBUTE VALUE. In this case, the associative search can specify both the NAME and VALUE as part of the match criteria and proceed essentially as previously described.

Many important details have been excluded purposely in order to focus on the key issues, namely, that sophisticated, special-purpose memory systems can significantly simplify the searching procedure. The question of much interest is, just how feasible is such a system? The synchronous-to-asynchronous buffer for spreading data across disks in parallel is partially available today with adequate speeds, as well as the asynchronous secondary port feature of memory.[15] The hybrid virtual memory, although not fully designed, is within our reach. No memories are available today with the multiport or associative feature described here. These features of main memory require significant innovation in technology and device design to become practical. New structures are required, wherein the functions are designed directly in the device, i.e., directly in the silicon, rather than in a circuit containing many interconnected devices. This requires innovations, as well as new outlooks on the part of technologists and device and circuit designers.

Even though there are many details of the above system that require considerable expertise to understand completely, it should be apparent that in principle, a substantial part of system and I/O architecture complexity, virtual memory organization, and operating system complexity is due to the practical constraints imposed by the memory technologies.

Concluding remarks

The past 25 years have seen enormous strides in both technology and system organization to provide sophisticated, high-performance computing systems. We have seen that a major factor in this evolution of computing systems has been and will continue to be practical constraints imposed on our ability to access information. The ideal large-capacity, high-speed, low-cost memory system closely coupled to the processor has not been feasible. Hence, practical systems have used many techniques to give the appearance of an ideal memory system. This has been achieved through numerous design trade-offs at many levels of the system, often resulting in new constraints as well as affording new options. The next 25 years may very well bring a gradual change in emphasis and use of VLSI to provide better accessing capability built directly in hardware in addition to continued improvements in density and performance. In order to achieve this hardware accessing capability, technology will have to be able to build functions directly in semiconductor materials, rather than building distinguishable standard devices that are wired together—for example, an associative device rather than an associative cell, which is typically built from a dozen or so transistors. This would also provide the third dimension for stacking functions, rather than the two-dimensional form of standard chips.

In the area of disks, we can expect to see not only continued improvements in density and capacity, but most likely the smallest directly addressable unit will become smaller than a sector, approaching a memory word size. Also, the complex logic functions performed by the channels will migrate more and more to the control unit to give the channels more free time to service other I/O that will become more critical as processing speed increases. In addition, the operation of disks in parallel will likely become commonplace and provide the enormous I/O bandwidths that will be necessary for future systems.

The most difficult challenge ahead is the fusing of the system requirements within the device and technology designs to achieve practical, high-performance, high-function structures. The difficulty lies in the fact that these areas of systems, circuits, devices, and technology tend by necessity to be fields of isolated expertise. Bringing these areas together is a technical, managerial, and human problem. Achieving this is a formidable but exciting challenge.

Cited references

1. R. Matick, *Computer Storage Systems and Technology*, John Wiley & Sons, Inc., New York (1977).
2. G. Radin, "The 801 minicomputer," *IBM Journal of Research and Development* **27**, No. 3, 237–246 (1983).
3. A. Berenbaum, M. Condry, and P. Lu, "The operating system and language support features of the BELLMAC-32 microprocessor," *ACM Proceedings of Symposium on Architectural Support for Programming Languages and Operating Systems*, March 1–3, 1982, Palo Alto, CA, pp. 30–38.
4. D. Ditzel and H. R. McLellan, "Register allocation for free; The C Machine stack cache," *ACM Proceedings of Symposium on Architectural Support for Programming Languages and Operating Systems*, March 1–3, 1982, Palo Alto, CA, pp. 48–53.
5. R. E. Matick and D. T. Ling, "Architecture implications in the design of microprocessors," *IBM Systems Journal* **23**, No. 3, 264–280 (1984).
6. S. P. Harbison, "An architectural alternative to optimizing compilers" (CMU TM Architecture), *ACM Proceedings of Symposium on Architectural Support for Programming Languages and Operating Systems*, March 1–3, 1982, Palo Alto, CA, pp. 57–65.
7. D. Patterson and C. Sequin, "A VLSI RISC," *Computer* **15**, 8–21 (September 1982).
8. L. D. Stevens, "The evolution of magnetic storage," *IBM Journal of Research and Development* **25**, No. 5, 663–675 (1981).
9. C. J. Bashe, L. R. Johnson, J. H. Palmer, and E. W. Pugh, *IBM's Early Computers*, The MIT Press, Cambridge, MA (1986).

10. C. P. Grossman, "Cache-DASD storage design for improving system performance," *IBM Systems Journal* **24**, Nos. 3/4, 316–334 (1985); *IBM 3880 Storage Control Record Cache RPQ #8B0035*, GA32-0086-0, IBM Corporation; available through IBM branch offices.
11. M. Y. Kim, "Synchronized disk interleaving," *IEEE Transactions on Computers* **35**, No. 11, 978–988 (November 1986).
12. H. Boral and D. J. DeWitt, "Database machines: An idea whose time has passed? A critique of the future of database machines," *Database Machines*, Springer-Verlag, Berlin (1983).
13. M. Astrahan, M. Blasgen, D. Chamberlin, K. Eswaran, J. Gray, P. Griffiths, W. King, R. Lorie, P. McJones, J. Mehl, G. Putzolu, I. Traiger, B. Wade, and V. Watson, "System R: Relational approach to database management," *ACM Transactions on Database Systems* **1**, No. 2, 97–137 (June 1976).
14. M. W. Blasgen and K. P. Eswaran, "Storage and access in relational data bases," *IBM Systems Journal* **16**, No. 4, 363–377 (1977).
15. R. Matick, D. T. Ling, S. Gupta, and F. Dill, "All points addressable raster display memory," *IBM Journal of Research and Development* **28**, No. 4, 379–392 (1984).
16. M. Y. Kim and R. E. Matick, "Synchronous-to-asynchronous conversion buffer and applications," *IBM Technical Disclosure Bulletin* **29**, No. 5 (October 1986).
17. R. E. Matick, "Method for general sharing of data in hybrid memory organization," *IBM Technical Disclosure Bulletin* **25**, No. 5, 2606–2620 (October 1982).

Richard E. Matick *IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.* Dr. Matick received his B.S., M.S., and Ph.D. degrees in electrical engineering from Carnegie-Mellon University in 1955, 1956, and 1958, respectively. He joined IBM in October 1958 and worked in the areas of thin magnetic films, memories, and ferroelectrics. As manager of the magnetic film memory group from 1962 to 1964, he received an Outstanding Invention Award for the invention and development of the thick-film read-only memory. He spent a half year at IBM Hursley, England, developing this memory for System/360 applications. Dr. Matick joined the technical staff of the IBM Director of Research in 1965 and remained until 1972, serving in various positions, including responsibility for Research Division plans and the post of Technical Assistant to the Director of Research. He took a sabbatical in 1972 to teach at the University of Colorado and at IBM in Boulder, Colorado. Dr. Matick spent the summer of 1973 teaching and doing research at Stanford University. He is currently working in the areas of VLSI functional memory chip and microprocessor design. In April 1986, he received an Outstanding Innovation Award as co-inventor of display RAM, a new memory chip that is being used in the high-resolution display announced with the IBM RT PC and that is being used increasingly in bit-buffered displays. Dr. Matick is the author of several books and book chapters on computers, computer memories, and transmission lines. He has also written numerous papers on magnetic devices and memories, semiconductor circuits, memory, and logic, as well as on virtual memory chips and systems. He holds numerous patents and patent publications. Dr. Matick is also a member of the IEEE and Eta Kappa Nu.

Reprint Order No. G321-5276.