2

DESIGNING A COMPUTER:

THE ECLECTIC INFORMATION PROCESSING SYSTEM

JAMES H. HAYNES

CEP REPORT, VOLUME 4 No. 1

JANUARY 1971

The Computer Evolution Project
Applied Sciences
The University of California at
Santa Cruz, California 95060

DESIGNING A COMPUTER:

The Eclectic Information Processing System

Abstract

The outstanding manifestation of the third computer generation has been the continuing crisis in software development. This crisis has been a natural result of the nature and historical development of computers. The thought processes of computer users have changed rather drastically since the beginning, while the hardware has changed rather little. The gap which now exists between computer users and their machines has been bridged with compilers and with operating systems; but these special programs still have to be created by someone. Software design and construction remains a major undertaking. Our proposal for getting the problem under control is twofold: software should be written in conceptually-powerful high-level languages, and machines should be designed to execute programs written in the languages to be used. This paper explores some implications for system design of this philosophy. Many of the ideas presented can be attributed to Iliffe and his work with the Rice University and ICL Basic Language machines. The other main source of ideas has been the great Barton tradition, beginning with the B-5000 of Burroughs and continuing through the work of Dent, Hauck, Cleary, and McKeeman.

This work was supported in part by National Aeronautics and Space Administration under grant NGR05-061-005 and by National Science Foundation Grant GJ-150.

I

How We Got Into This Mess

"Machines should work; people should think." - IBM

The builders of ENIAC were experienced desk-calculator users in search of a much faster way to solve a particular kind of numerical problem. It is not surprising that their machine turned out to be an enlarged electronic version of its mechanical counterpart. The electronic computer became much more generally useful with the advent of such improvements as easily-alterable program storage, program self-modification, subroutine libraries, and assembly programs. Hand-in-hand with these developments programming became something of an arts-and-crafts activity, requiring of its practitioners above all else cleverness. The person having a problem to solve but no previous computer experience was more and more easily intimidated.

Probably the most significant development in all the history of computing was the creation and marketing of FORTRAN. For one thing, FORTRAN made computing power readily available to a class of users who had been repelled by computers before; and in sheer numbers of users this class far outweighed all of the professional programmers. More fundamentally, FORTRAN gave to users a conceptual machine which accepted statements in a language not far removed from their own thinking, just as ENIAC had accepted a language that was familiar to its users. The effects of FORTRAN were to create a huge new market for computing power and to bring intellectual forces to bear on the problems and opportunities of user-oriented languages. Thus the commercial success of FORTRAN put at the disposal of language

developers economic muscles which they had never enjoyed before.

There remained a great many applications to which FORTRAN was unsuited, including the writing of the FORTRAN translator itself. This was natural enough at first, because language translation was complex and mysterious, and because some of the benefits of programming in user-oriented languages were not yet visible. We may now attempt to list the features desired in programming languages.

1. Ease of learning the language
2. The provision of conceptual tools, such as multidimensional arrays, subroutines, and arbitrary meaningful operand names
3. The ability to transfer programs from one machine to another without extensive change, and without retraining programmers
4. Reduced manual labor in program writing
5. Programs are easy to read and require little or no external documentation
6. Compilers are easy to write and are easily designed to produce efficient object code.

Although many professional programmers subjected them to ridicule, programming languages are now firmly established. With modern compiler-generating systems it is fairly easy to turn out a language for some particular class of users, and a translator from that language to some particular machine language. It is not so easy to produce a translator that can operate in a small memory, and that can produce efficient object code. Many such language systems operate by assuming the existence of a special machine for which

efficient object code is easily produced; an interpreter program then simulates the pseudo-machine on a real machine.

Computer users, emboldened by the power made available to them through programming languages, have ventured forth to tackle increasingly challenging problems. This has caused them to make increasingly challenging demands upon systems and systems programmers. A system these days is expected to have a large software set available in on-line storage. It is expected to perform multiprogramming and perhaps multiprocessing. It is expected to multiprogram jobs written in various languages, so that many functions formerly considered a part of the language system must now be shared among several language subsystems. The modern system is expected to present a familiar face to the batch user, and at the same time look like a remote batch system, a general-purpose time sharing system, and a dedicated on-line application system. The modern system must contain a comprehensive file subsystem (whatever that means!). It is a long way from the language of "LOAD A" to the language of "A[I,J]=SIN(B+X**P);". It is yet another giant step to a job control language which allows one to say, in effect, "execute the program named DAILY RUN using the file named TRANSACTIONS as input and updating the file named MASTER FILE." Since most present-day computer hardware is still working at the "LOAD A" level the computer user is separated from the machine by many layers of software. The hardware machine might be regarded as simulating a different virtual machine, which is itself simulating still another virtual

machine, ad infinitum. With all this simulation going on it is not surprising that rather few of the instructions executed by the machine make any direct and obvious contribution to solving the user's problem.

There have been several rather ill-advised attempts to build machines which can execute FORTRAN or some other language directly in hardware.[1,5,19,20] Few of these were ever built, and none was successful in the marketplace. Probably the main obstacle facing these early designs was their complexity, at a time when hardware cost and unreliability dominated the thinking of computer center managers. Too, the languages that were implemented were not very comprehensive, lacking the flexibility of more traditional machine languages, and were disdained by professional programmers, who would have regarded the machines as interesting toys but not as real computers. Today's environment might be much more receptive to a language-in-hardware approach; but on more fundamental grounds these designs overlook some very sound reasons for introducing a stage of compilation between the user language and the hardware. For example, parentheses in an arithmetic expression serve to indicate the order of performing the evaluation operations, but in addition they usually have a 'grouping' meaning to the programmer. This higher meaning does not exist for the machine, which is required only to evaluate the expression correctly. If an expression is to be evaluated only once it does not matter whether the order of the operations is discovered at run time or in an earlier compiler run; but if the expression is evaluated repeatedly it is definitely

a waste of effort to extract this invariant information anew at each iteration. But a version of the expression which is efficient for computation would be decidedly unpalatable to most human beings. Thus the direct-execution machine operates under quite an efficiency handicap in any iterative process.

We now find ourselves in the following situation. Computer software in the form of programming languages and operating systems is an essential ingredient in the way we use computers. With very few exceptions computer hardware does not differ much from its desk-calculator ancestors, in spite of advertising claims to the contrary. Software design remains largely a craft, caught as it is between the ever-advancing concepts of users and hardware that is modern in construction but primitive in concept. It is not surprising that newly-introduced machines are plagued for years by vastly expensive software that does not live up to its promises. Nor is it surprising that some conceptually simple and powerful things we would like to do with computers get bogged down in a morass of LOAD A's.

We must not ignore a few bright spots in this generally dismal picture. The NELIAC project[17] furnishes an early example of the use of a high-level language to write a compiler for a high-level language (in this case, for NELIAC itself). Both NELIAC and JOVIAL are offshoots of the work that led to ALGOL; but they developed in the direction of systems programming languages rather than scientific languages. The family of machines that began with the Burroughs

B-5000[3,6,8,9,10] emphasized both machine design geared to high-level languages and the practice of writing all programs in such languages. The Burroughs extensions to ALGOL 60 produced a language in which compilers could be written. A very similar language, ESPOL, provided the few extra handles needed for the operating system program: principally access to mechanisms in the hardware that are invisible to the ALGOL programmer. The Rice University computer [12,15] provides special hardware features for handling structured data and simplifying software. These features are improved and enlarged in a new machine now under construction at Rice, and in the I.C.L. Basic Language Machine of Iliffe.[13,14] The recent Burroughs B-6500 represents a considerable enlargement of the B-5000 concepts (in which the existence of the earlier machine played no small part - much of the B-6500 software was written and tested in B-5500 ALGOL). The MULTICS project at M.I.T. has undertaken the writing of a very large and complex operating system in a subset of PL/1. There has been some unfair judgement of this system because of its slow progress; we must realize that in this project research is at least as important as development. Too, the MULTICS group does not have the advantage of a machine designed expecially to implement the PL/1 language.

Software will continue to be a problem as long as man's reach exceeds his grasp; but it need not be a disaster. The remainder of this paper is devoted to an exploration of two notions: that all software should be written in appropriate high-level languages, and that the

hardware should be designed for easy implementation and efficient execution of the language. Although a few original ideas may turn up here and there, what follows is essentially an appreciation of the work that has already been done by the pioneers previously cited.

Additional References

Reference [30] shows that the idea that the machine should have something to say about its own programming arose quite early. Later writers emphasized that the needs of programming should have a powerful influence on computer design[11,4,18]; while others wisely suggested that in the search for computer improvements the greatest payoff could come from a consideration of what the machines really do, rather than how they do what they do.[22] Several other papers of general philosophical interest have appeared.[7,18,23,28,32] Other interesting work is known to be in progress; so perhaps we are finally beginning to achieve a body of knowledge about computer organization.

II

System Functions

"Machines should work; people should think." - IBM

The modern information processing system seems to have at least the following features:

1. A large software set residing in on-line storage
2. A multiprogramming operating system
3. All user and systems programming is done in high-level languages
4. Variable hardware configuration: both statically variable, to allow for growth, and dynamically variable, to accomodate maintenance while the system is running
5. Built-in data communication capability
6. A large main memory, to accomodate the multiple simultaneous users
7. An elaborate file system

We begin by considering the programming languages. We might say in passing that there are many ways to approach the problem of computer system performance: fast circuits, pipelining, associative memory, arrays of processors, vector processing, etc. We believe that the language approach to computer design is fruitful simply because it is through languages that computer users express their desires to the computer system. A user should begin by inventing a language, if necessary, in which he can say what he wants the system to do. Only after this step should he be required to say

how the system might accomplish his desires by fabricating his own what out of a set of more primitive whats. Once a user has expressed a willingness to live with his own set of whats the machine designers can move in to implement this set as efficiently as possible.

Although the invention of new programming languages is a favorite pastime of computer scientists we choose for this project to use an outgrowth (or undergrowth) of ordinary PL/1. PL/1 was designed with both scientific and business data processing in mind; accordingly we expect that a system which can implement PL/1 should present no great impediments to the implementation of FORTRAN, COBOL, and ALGOL 60. Additionally, PL/1 contains facilities which look as if they would be useful in writing an operating system: particularly multitasking and storage allocation features. In choosing to talk with a PL/1 accent we do not intend to pick fights with those who feel that LISP, ALGOL 68, or Iverson notation are "better"; our real concern is with the data processing activity which is called for by the statements of the language rather than with the style of expression. And we do not hesitate to introduce non-PL/1 constructs where they seem to be needed; nor do we intend at this time to be committed to the design of a "PL/1 machine".

"Systems", says Mealy, "resemble the organizations that produce them." Systems also resemble the languages in which they are written. FORTRAN and assembly language are "flat" languages in which all subroutines have equal rank and all identifiers are either strictly local or strictly global. ALGOL and PL/1 by contrast allow

arbitrarily many levels of nesting. A flat language encourages the writing of a flat operating system in which any subroutine can be called from any point and need not terminate with a return to the calling point. Recursive calls and multiple executions of routines require special handling. Any instruction can reference any location in memory, unless special memory protection hardware is present. The availability of a nested language suggests that nesting can be applied to several practical system problems. For example, a form of storage protection results from the use of disjoint scopes of declaration. We may regard the entire system as a single large program in which the various jobs are concurrent tasks. Independent jobs cannot disturb one another deliberately because their respective sets of local variables and procedures are mutually invisible. The system library subroutines are declared at a level global to the jobs and thus are available to all. Any job may, however, contain a procedure declared with the same name as a system procedure. This causes the declaration to apply in a natural way to that job alone. System routines which are declared globally but which should not be accessible to all jobs automatically can be screened by the automatic insertion of dummy local-to-the-job declarations of their names at compile time. The fact that all programming is done in high-level languages is itself a protection, since the compilers can refuse to compile any instructions that are not intended to be available to users. Accidental interference among programs is not precluded by these measures; a user's program might attempt to index beyond the end of an array, or might cause operands to be fetched where instructions are expected. The abnormalities must be prevented

by hardware checks of index quantities and instructions. These checks can be quite simple in nature, and are applied exactly where they are needed. In the system to be described index quantities are checked against array extents before any access to array data takes place. Instructions are marked as such and are execute-only, while operands are marked as such and are non-executable. Programs are pure procedures, so that any sequence of instructions which has been compiled correctly cannot be changed into something dangerous during execution.

It is sometimes necessary to share data among independent processes. It may be objectionalbe to declare such data globally, because this would make it equally available to all processes. One solution to this problem is to declare a block to contain the shared data and all jobs that are to be allowed access to it. A more general solution is to declare a global "message-center" procedure which contains the declaration of the shared data. All accesses to the shared data are then made as calls on its containing procedure, which may require any desired activity as a condition for granting access to the data. In fact the user procedure does not access the data at all; it requests that the message-center procedure perform some access function on its behalf. The scope-of-declaration rules of ALGOL and PL/1 assumed here are not the only ones possible, and they may not be the most desirable for our purposes, but they appear to be adequate.

A serious problem with nested languages for systems work is the

necessity to use subroutines compiled at different times. For example, we cannot afford to recompile the operating system whenever a new job enters. And even if our compilers are very fast we do not wish to recompile a large element of software or a large user program when some small portion of it is changed. In flat languages the same problem really exists, but it is much less apparent because rather simple artifices can be used to solve it. There are really two aspects to the problem:

1. When a procedure is compiled it will in general contain references to names that are not declared locally. These will consist of names that are declared in some containing block (which may or may not have been compiled already). There may also be undeclared names which are simply programming errors.
2. The name of a procedure must be known at the point where that procedure is to be called, even if that procedure has not yet been compiled.

The first aspect of the problem has often been solved simply by keeping all unknown names in their character-string representation and then performing a search at load time or execution time for their owners. Another popular scheme is to keep all such operands in a common area, the layout of which is known to all program writers. Then a symbolic name can be replaced at compile time with a reference to the appropriate location in the common area. Neither of these schemes preserves the nested structure of a program. We might note

in passing that this aspect of the problem is entirely absent when the procedure to be compiled contains only local names and parameters. A general solution to this part of the problem might be to have a compiler output containing all symbol tables resulting from compilation, together with an indication of their nesting relationships. Then to alter a procedure the compiler would be preloaded with the applicable set of symbol tables and nesting information. Addition of a procedure would require some means of indicating what scope of declaration should be chosen by the compiler. To the user the system might resemble conventional assemblers and compilers having alteration facilities controlled by line numbers. We intend to have the compiler symbol tables partially preloaded anyway with the names of the system library subroutines so that user programs can reference these. A compiler option will then be needed to overwrite the preloading so that the system library routines themselves can be compiled. Aside from the separate compilation problem, symbol tables have to be saved somewhere for the diagnosis of run-time errors.

The second aspect of the problem requires that the name of a separately-compiled procedure be available to the statements which call it. Many languages include a feature for this purpose; in PL/1 it is the ENTRY declaration. This in itself solves only part of the problem. It provides a location within a program to which references to the separately-compiled procedure can be directed. A separate mechanism must further direct those references to the

actual location of the procedure. Yet another form of this problem shows up when the name of the separate procedure isuunknown when the calling statement is compiled. A particular instance of this concerns the user jobs within the system, which to the system executive are procedures to be called as multiple tasks. The call may take place at any time, and has the effect of creating a process which is executable. Actual execution is delayed until a processor becomes available. More will be said about this later; for the present it is sufficient to note that an entry can be a variable to which a value can be assigned. Thus the language readily allows one to write calls on procedures when the actual names of the procedures being called are not known until the moment that the call takes place.

The operation of most contemporary systems is strongly influenced by interrupts. The prompt interpretation and servicing of these interrupts is of major concern in systems programming. We propose a different philosophy under which interrupt servicing becomes much less crucial. First we make a distinction between interrupts and traps. A trap is an exception condition in a processor which is the direct result of the activity in progress there (e.g., exponent underflow in a floating-point arithmetic operation). The appropriate activity when a trap occurs is an automatic call on a procedure which takes some fix-up action and returns to the program that was in execution. The fix-up procedure does not have any particularly difficult addressing environment problem. It can have locally declared references and system global references. If it has any parameters they are likely to be the operands at the current point

of execution. An interrupt, on the other hand, results from activity outside the current process and usually means that the process cannot continue forthwith. Our first attack on the interrupt problem is to legislate against problems of patience; that is, those situations in which an interrupt must be serviced within some time interval else some dire event takes place. This principally means that I/O processors must be designed to complete autonomously whatever operations they start. On completion of an I/O operation the desired activity is that some process which was roadblocked until completion of the operation is to be unblocked. This is normally accomplished by having the I/O processor make an entry in a queue which is periodically examined by the operating system executive process. Periodic examination is guaranteed by an interval timer which periodically interrupts whatever process is currently in execution. We computer designers have been far to anthropomorphic in our views of interrupt priority and urgency. We imagine ourselves speaking to the system as we might to a subordinate employee, "Drop everything, here's a hot one!". At the machine's time scale dropping everything (in a way that allows it to be picked up again) is itself a time-consuming assignment not to be given lightly. When there really is a need for different priorities of access to a processor the solution is to have a different queue of ready-to-run tasks for each level of priority - effectively a queue of queues, or a vector of queues. Then it is necessary only to make the time interval between timer interrupts short enough to assure an adequate grade of service to the queues. The only interrupt we propose in addition to the interval

timer is a voluntary relinquishment. This activity carries with it the concept that the current process is to be suspended and placed in a queue of some kind, from which it will be reactivated later. We might even convert the timer from an interrupt to a trap, with the idea that a timer trap would call a procedure which normally executes a relinquish operation immediately, placing the process in a ready-to-run queue. This would allow certain system procedures to be invulnerable to timer interruptions simply by calling a different trap procedure which does nothing. However it would then be necessary to have the compiler guard against unauthorized attempts to bypass timer traps. Probably in the interests of system security it is best to make the timer interrupt irresistible. Note that an urgent situation like imminent power failure has not been established as an interrupt. The most appropriate response to imminent power failure would seem to be the sole responsibility of each processor acting alone, and would consist of storing everything volatile and then ceasing all activity. The recovery upon reapplication of power might be to simply resume operation. These objects could be most easily achieved with a small nonvolatile store local to each processor. The power failure activity might consist simply of a voluntary relinquish operation which places the current process in a ready-for-processing queue and safe-stores only the location of the queue. For a deliberate manual shutdown of a processor the same system can be used.

We believe that input/output is a very poorly understood subject at the present time. It seems terribly complex, both to the user

programmer and at the system level. I/O is used for many different purposes: "original" input, "final" output, storage during execution of a program, passing data from one program to another, etc. A comprehensive file system makes these things a little more orderly by attempting to free users from concern with specific physical devices. Data communication is in some respects quite different from more conventional I/O; on input characters simply arrive without being requested by the system. Some input characters call for action, while others are simply to be added to a string being accumulated. It is best to view the input from a remote terminal syntactically, with the idea that a parsing program examines each new character as it arrives. The parsing program may be executed by either a central processor or an I/O processor. In the former case the I/O processor places the incoming character in a buffer and enables a parsing process for the main processor, while in the latter case the I/O processor must itself be able to parse.

In some instances we may be able to spare the user explicit concern with I/O altogether. The simplified languages SPL and SPL use pseudo variables instead of conventional reading and writing facilities. INPUT may appear in the right-hand side of an assignment statement. Each reference to INPUT yields the next operand from the input stream. OUTPUT may appear on the left-hand side of an assignment statement. Each value assigned to OUTPUT is appended to the output stream. This scheme could be extended to any sequential file by the addition of mechanisms to declare file names and to rewind and backspace. Some conventions would have to be established about alternate reading and

writing. If F is a file in the program:

```
F = 1;
A = F;
```

there might or might not be an automatic backspace so that the value of F is always that which has just been assigned to it. Thus A might receive the value 1, or it might be given the non-value undefined.

The view of the system as a single large program suggests that the use of a file to pass data from one program to another could be allowed simply by declaring the file at a level visible from all of the job programs. This suggests a job control language identical with the programming language. Consider a job using a file and having two programs to be executed in sequence. The user might write the simple program:

```
DECLARE F FILE, (P1,P2) ENTRY;
CALL P1;
CALL P2;
```

and submit this program for compilation and execution. The object code for P1 and P2 would of course have to be supplied and would be inserted into the resulting program using the alter facility. Suppose this job is to be run every day, with P1 writing daily transactions on F and P2 processing them at the end of the day. Then the calls on P1 and P2 can be put into an endless loop. P1 will go into execution and sooner or later hang waiting for input. As input arrives P1 processes it and again hangs waiting for more. At the end of the day some special code in the input is recognized by P1, causing it to terminate with a RETURN. The main program then calls

P2, which does its job and returns, causing P1 to be called again. P1 then again hangs until some input for it arrives. Because of the automatic storage allocation facilities of the system there is no great penalty for leaving P1 hanging, as it occupies only a word or two in memory when it is not in active execution. If it happens that P1 and P2 in this example have different names for the file this can be handled by declaring two variable file names to which F is assigned at appropriate times between calls.

We would like to make it unnecessary for the user to be concerned with files just because his program is too large to fit into the amount of main storage available to him. We have in mind here both data files and procedure files, although in some cases a data file may be the most natural way to handle data in an algorithm. The term "folding" has been applied to any activity performed on a large program tomake it fit into a small memory.[29] We may distinguish between preplanned folding and automatic folding. In preplanned folding the user must plan in advance exactly what parts of his program are to occupy memory at the same time. This can get so complicated that automatic folding can be just about as effective, in spite of its lack of intelligence. Part of the work necessary for preplanned folding comes naturally to the user who programs in a nested language and makes good use of the subroutine and blocking facilities of the language. Segmentation is an automatic folding scheme which manipulates the blocks of the user's program independently. Paging is an automatic folding scheme which ignores these natural entities of a program and instead divides the entire extent of a program into fixed-size pages.

With either scheme the decision to move a segment or page from backing store to main memory is usually deferred until the program attempts to reference the missing information; hence the term "demand paging". The much harder decision in automatic folding is where to put the information when there is no free space in main storage; something already there has to be overlaid. Algorithms of varying complexity have been devised for making these decisions, which in preplanned manual folding would have been made in advance by the user. The goal of all overlay algorithms is to maximize system performance by avoiding unnecessary data movement. There is some experimental evidence to suggest that simple overlay algorithms work almost as well as complicated ones and give better overall results. Experiments in this area tend to be controversial, because an experiment can always be devised which will make any particular algorithm look good or bad at will.*

An important concept with either paging or segmentation is that of the working set, which may be regarded as a threshold number of segments or pages present in main storage at the same time. If a program is given enough main memory to accomodate its working set it will run fairly well; while with less memory it sill spend most of its time moving data between main memory and backing store and will get very little done between moves. When there are parallel independent processes, as in multiprogramming, there is a working

*The literature on memory management has become so large that we are herein avoiding all references except to Denning's survey [29], which in turn references almost all of the original works.

set for each process. Thrashing is a mode of system behavior in which no process can get a working set into memory; the available memory is spread too thin. Thrashing can be avoided by limiting the number of simultaneous processes so that all can have their working sets in memory. This is easier said than done, since the size of the working set may be unknown when a process presents itself for execution, and since the size of the working set may vary dynamically as execution proceeds. It also points up a considerable difference between batch multiprogramming and time sharing. With batch processing it is optimum to multiprogram a number of jobs just equal to the current capacity of the system. With time sharing the goal is to service a given number of users simultaneously; this number is in general larger than the capacity of the system in a batch processing sense, so that efficiency in job processing is sacrificed to get and keep users happy.

An anti-thrashing measure which can be applied easily is to allow a process to overlay only its own segments; it can add to its allocation of main memory only by capturing free space, and that perhaps only with the permission of the operating system. This rule guarantees that a process which cannot secure its own working set at least cannot disturb any other process in its attempts. It seems necessary to have the operating system monitor the accretion of free space to processes. Otherwise the processes currently in execution would gobble up all free space as it became available and prevent the creation of new processes. We conjecture that space would tend to

accrue to the process already having the most space, as it is this process which can proceed longest between missing-segment events and thus which has the highest probability of being in execution when free space becomes available.

Paging in principle seems inferior to segmentation because of its Procrustean-bed nature which makes no use of the natural blocking within a program. This could be partly overcome by providing an assortment of page sizes, but then the page swapping mechanism would become as complicated as one which handles segments of completely arbitrary size. A combination of these two techniques, in which only the very large segments are paged, may offer an improvement over either technique used alone; for this would allow a user to define segments even larger than the capacity of main memory if this happened to suit his purposes.

In a paging or segmentation system the pages or segments of a process may be assigned to fixed locations in backing storage for the life of the process. No attempt is made to use the backing store space formerly occupied by a segment or page when that entity is moved to main memory; the space is reserved for the eventual return of the segment or page. This seems reasonable because backing storage is much larger than main memory; and at most only an amount of backing storage equal to the capacity of main memory is held in reservation. With a rotating backing store we might wish to move a segment from main storage into the first available location in backing storage rather than into a fixed position so as to avoid rotational delay. This is fairly easy to do with paging because all pages are the same

size; but with segmentation it requires finding a space that is big enough to accomodate the segment to be moved. In a system which is large enough to have several requests for movement between main memory and backing storage pending at the same time the rotational delay can usually be washed out anyway by arranging the requests in a shortest-access-time-first order.

Another problem with segmentation is the fragmentation of main storage. Usually a segment will be moved into an area which is somewhat oversize, leaving a small space. Opportunities to make use of the small space may be few and far between, so that such spaces tend to lie unused for a long time. Yet the total amount of space devoted to such fragments may be quite large. If the space-finding algorithm attempts to find the best possible fit for a segment to be moved in the potential for creating short fragments is greatly increased. For this reason Knuth advocates choosing the first available space that is big enough in preference to the one which most nearly fits.[16, p. 437] When a very small fragment results from the fitting process it is best just to move the segment into the oversize space, as the extra space just cannot be used anywhere else. Hence the space allocation mechanism tends to produce chips of space just larger than some minimum size. Eventually all of memory is chopped up this way and is practically useless. One solution to this problem is a "garbage collection" mechanism which from time to time merges adjacent chips into larger spaces, and perhaps moves in-use segments around, until all free space has been turned into one contiguous area. If the garbage collection program

is called in too late there may be no space large enough to contain it, and the whole system breaks down. A different approach is a sort of continuous automatic garbage processing in which the adjacent free areas are recognized and merged immediately. This is most easily done just as an in-use space is evacuated. The spaces on either side are examined and, if free, are merged with the in-use space.

With paging there is no fragmentation of this kind because storage is always allocated in page-size blocks. The wasted space in a paging system is less visible, and consists of in-use pages which contain inactive material in addition to their active material. Hence a process in a paging system may appear to have an artificially large working set. This effect is minimized by having small pages, but small pages demand a relatively large and expensive page table (memory map).

Paging creates for the user a very large contiguous virtual memory. It succeeds if this memory is always larger than what the user needs; otherwise the user has to resort to manual folding anyway. One way to use the large virtual space is just like that used in an ordinary memory with an ordinary loader program. That is, the various blocks of the user's program are packed into the available space with internal addresses relocated as necessary in the process. Then the storage efficiency is reduced as just noted by the fact that many pages in main memory will contain both active and inactive material. An alternative is to load each block of the program into as many pages as necessary, starting each new block on a fresh page.

Unused space at the end of a block on the last page of that block remains unusable for the life of the block, which makes this scheme totally unacceptable if the page size is larger than the average block size. However it makes the relocating loader unnecessary since each block of the program can begin at address zero of its first page. The availability of a large low-cost associative memory would allow paging of segments in this way with a very small page size, so that a compromise might be effected between wasted space and the complexity of segmentation.

III

Data, Data Structures, and Data Processing

"Machines should work; people should think." - IBM

An important feature of the system design philosophy is the deliberate concealment of certain features of the implementation. There are two reasons for this. The first is to prevent users from employing clever tricks which use the equipment in some unplanned-for manner. The second reason follows from this; it is to preserve the freedom of the designers to change details of the implementation at will, provided that the "official" description of the system is not violated. Hence the following descriptive material must be taken with a grain of salt whenever such details as the number and arrangement of bits in an operand are given.

Early FORTRAN provided only for integer and floating-point data; character strings could be quoted, but not manipulated. Although the integers are in a class by themselves, where numbers are concerned, their inclusion in FORTRAN may have resulted more from machine-dependent thinking than from real need. Identifiers were limited to six characters because six characters in the code employed would exactly fit into one machine word. Even in some contemporary machines the memory word length exerts a strong influence on all matters of data representation. Operands requiring fewer bits than an entire word are either stored one per word, wasting space, or packed several to a word, requiring special hardware instructions or programming to do the packing and unpacking. In the latter case a word may not hold an integral number of operands, so that there must still be

some waste space unless rather complicated software or hardware is employed to work across the word boundaries. These considerations explain the popularity of 36 bits as a word length. Aside from the acceptability of 36 bits in a numeric operand, 36 can be divided evenly more ways than any of its near neighbors. (36 has nine factors; 24, 30, 40, and 42 each have 8 factors; and it is not until we reach a length of 48 with 10 factors that 36 is bettered.) An alternative to this preoccupation with word lengths and packing is offered by the variable-field-length machines which allow arbitrarily-long strings of small (4-8 bits) data elements. This can be quite successful in a small machine in which one is willing to expend one memory cycle time per byte accessed, but it can be quite discouraging when high performance is desired.

We might like to push aside the matter of memory word length entirely, taking the single bit as a primitive data item and allowing an operand to contain any number of bits. PL/1 contains some rather dubious features along these lines, allowing fixed and floating point operands to be declared with any number of digits in either binary or decimal base. Bit and character strings of any length are also allowed, but these are data structures rather than elementary items. There are several drawbacks to the strict variable-by-bit length specification.

1. Small operands require big addresses. A memory of 2^N bits contains 2^N distinct locations, each addressable with an N-bit address. A process which ultimately references all locations must manipulate $N*2^N$ bits of address data to

access 2^N bits of operand data. The number of addresses which must actually be generated or stored in the course of a computation may be small or large, depending on the extent to which a single description can apply to many distinct data items. Thus the address length factor may or may not be important.

2. An efficient encoding for the operand length specification is needed. A 1-bit number is sufficient to distinguish two different operand sizes, while a very large number is needed to distinguish among a very large number of operand sizes. It may be important for storage economy to represent a one-bit operand in a one-bit notation instead of using 2 bits; but there are rather few situations in which a 38-bit operand is definitely large enough and 39 bits is definitely too large. Hence a more efficient encoding of operand length might be had by offering a limited assortment of useful operand lengths and representing each length by a short code group. A variable-length code might well be employed, in which short operands receive short length codes and long operands get the long codes.

3. It is difficult to build fast hardware for a completely variable-length scheme. This may be regarded as the manifestation in hardware of the problem of manipulating a large number of large addresses (which must be done in hardware, even if the addresses are invisible to the user). An operand located anywhere with respect to the memory word boundary must in general be moved to a particular position for processing. The easiest but slowest way to perform alignment is to place the bit string containing the operand into a shift register and shift it to the desired location, masking off any superfluous

bits that may remain. To store into memory is somewhat more complicated, because we must preserve the bits on either side of the operand in question unchanged. To make either of these processes go faster we must replace the iterative shifting process with a faster process. An N-fold increase in shifting hardware will not yield the expected N-fold increase in speed because of second order effects such as wire length and logic circuit loading.

In the machine under study (herein called the HW machine) a compromise is made: the memory is addressable to the level of 8-bit bytes, and all data elements are composed of an integral number of bytes. If this turns out to lead to an unacceptable waste of storage for small operands the scheme can be revised to use smaller bytes. The following primitive data elements have been defined.

a. Alphanumeric character - 1 byte
b. Ordinal - 2 bytes
c. Address - 3 bytes
d. Numeric operand - 5 bytes
e. Double-precision numeric operand - 9 bytes
f. Instruction - various lengths

The present plans are to store all operands smaller than 8 bits as alphanumeric characters. There is no clear need for packed decimal data (4 bits per digit), so this has been omitted. The standard character code for the system will be an 8-bit representation of ASCII, although other codes can be used.

The ordinal is a small integer which cannot be manipulated directly

as a program variable. Ordinals are usually used as relative addresses, and as such are created automatically in the system whenever a relative address is required. An address, which may be used for purposes other than addressing, is similar to an ordinal in that it is beyond the reach of the user. A 24-bit address is adequate for addressing more than 16 million bytes of memory.

The numeric operand serves for both integer and floating point data. Its internal structure is a mantissa of 27 bits plus sign and an exponent of 11 bits plus sign. Both components are binary based. The double precision operand provides 54 bits plus sign for the mantissa and 16 bits plus sign for the exponent. These operand sizes are purely arbitrary and can be changed rather readily if they turn out to be objectionable. Likewise a quadruple-precision operand can easily be added if the need arises. There are no plans to allow these operand sizes to be altered by a console switch or under program control. A change in operand size necessitates at least a cold start of the system, because it affects the memory mapping; and it may even require recompilation of all programs. Even these objections could be overcome by additional hardware; but if one is really concerned about changing operand sizes it would probably be better to employ a completely variable-length machine from the start.

Most instructions occupy one or two bytes, but some contain literal constant data and thus are longer. The length of an instruction is implied by a portion of its leading byte. Still other data types exist and will be introduced later. A sort of grammar has been

used as an aid in the description of data structures. (See Appendix I) An implication of this is that if the grammar is unambiguous a particular data structure can be recognized by the hardware and decomposed into its constituents. This aids in run-time determination of data types.

There seem to be exactly three ways to fasten data elements together to form structures: concatenation, pointers, and content association. The elements of a concatenated structure are located adjacent to one another in memory. Adjacent here means in the usual sense of consecutively-numbered addresses, so that the address of a byte n places away from a given address is obtained by adding n to the given address. Also, only one-dimensional concatenation is intended here since physical memory addresses are only one-dimensional. Multidimensional concatenation can be represented on a one-dimensional address space by using storage mapping functions, or by means of more complicated structures to be discussed later.

In a concatenated structure each element has exactly two neighbors, except for the two special elements occupying the ends of the structure. The ends must be so identified in some way, either internal or external to the structure. The sizes of the elements in the structure must also be made known so that from the address of a given element the addresses of its neighbors can be calculated. If all elements in a concatenated structure are the same size the structure is said to be homoomeral. Only in a homoomeral concatenated structure is it possible to index; that is, to select the n-th

element of the structure directly by using the starting address of the structure, the element size, and n. A relative ordering of elements is implied by their relative positions in the structure; but the user may or may not regard the structure as ordered. A concatenated structure can expand or contract conveniently only at its ends. Any insertion or deletion affecting the middle of the structure, or any rearrangement of its elements other than a one-for-one swap, requires that other elements be moved.

A pure pointer structure is not very interesting, because the pointers are the elements. Each element can be only a pointer to the next. The structure may have two ends or none, as the pointer of the "last" element may point to the "first". For the present we do not distinguish between pointers which are absolute addresses and those which are relative to something.

More interesting and useful structures are combinations of concatenated structures and pointer structures. These include both concatenated strings of pointers and pointer structures having concatenated structures as elements. A string of concatenated pointers is homoomeral and can be used to make a non-homoomeral concatenated structure indexable. The pointers simply point to the beginnings of the elements of the non-homoomeral structure. This kind of mechanism can also be used to change the ordering of a concatenated structure without moving the elements; only the pointer values have to be changed. A pointer structure having concatenated structures as elements can be quite complex. Each element may contain any number of pointers, in addition to other data, so that the

structure may be linked in several different orders. Not all pointer chains have to link the same sets of elements. The most common use of one of these pointer-linked structures is as an ordered structure which allows insertions and deletions at arbitrary points along the structure. Pointer structures can never be indexed.

A pure content-associated structure is uninteresting because all of its elements are exactly alike; otherwise the elements are themselves structures of some kind. In useful content-associated structures the elements have their similarities and their differences. We may wish to determine all of the elements of a structure which have some common property, or we may wish to determine whether exactly one element is related to an item outside the structure. With ordinary technology these operations require an item-by-item search of the structure. Usually this is acceptably fast only if the extent of the structure is known and is small,or if most of the elements do have the property of interest. With associative memories an entire structure can be examined at once. It is hard to imagine a really general application for content associated structures alone, because it is hard to pin down just what kinds of elements might make up such a structure and what constitutes membership. Hence most uses of content-associated structures involve tightly defined operations upon tightly defined sets of operands.

The most common concatenated structures in the HW machine have elementary items as their elements and are preceded by a four-byte head. The homoomeral concatenated structures are called strings.

The first byte of a string head contains a unique type code for strings and a code for the size of the elements in the string. The other three bytes contain the extent of the structure, in bytes. A concatenated structure containing instructions is a procedure. A procedure head is just like a string head except that the size code implies a procedure instead of a size. The third kind of concatenated structure is the stream. A stream may contain a mixture of different kinds of elementary items. Each element must be preceded and followed by a one-byte type code which specifies its type and by implication its size. Thus a stream is a concatenation of typed elements.

A pointer consists of a type code, control bits, and an address. A pointer may appear wherever any other typed element or structure is allowed; in this context the pointer is simply an indirect address. The pointer control bits convey information about the object pointed to, such as whether the object is present in memory, whether it has ever been written, etc. A controller, of which there are several different kinds, is similar to a pointer with additional information appended, usually in the form of one or more ordinals. In a program controller, for example, the address points to the procedure head, while a single ordinal points to the next instruction to be executed. A stack controller likewise contains one ordinal. The address points to the head (base) of a stack, and the ordinal points to the current top-of-stack element. Stack overflow can be detected by comparing the stack controller ordinal with the extent value given in the stack head. "Stack head" here is used colloquially, since what is meant is really a stream or string head. A queue controller

contains two ordinals which point to the current first and last elements of the queue. A queue is circular; when an ordinal passes the last address in the queue it is set to zero, and when it passes zero in the opposite direction it is set to the last element. When the ordinals are equal the queue may be completely empty or completely full, depending on how they got to be that way. This distinction is recorded in the control bits.

Program Structure:

As noted above a program controller contains a pointer to the head of a procedure and an ordinal pointing to the next instruction to be executed. On entry to a block of instructions a program controller with a zero ordinal is automatically created. To fetch an instruction the machine adds the address and ordinal of the program controller, and then adds 4 (to space over the head). At the same time the value of the ordinal can be compared with the extent in the procedure head to be sure that the resulting address is actually within the procedure body. Each time an instruction is fetched the program controller ordinal is incremented so that it points to the next instruction. The size of the increment cannot be determined until the first byte of the instruction has been partially decoded, because of the variable length instructions. A conditional or unconditional jump within the procedure is accomplished in the obvious way: by altering the value of the ordinal. The ordinal of the program controller currently being used thus corresponds to the instruction counter of a conventional machine.

A subroutine call may result from an explicit CALL statement or by

encountering a function name used as an operand. In either case the processor will find itself holding, as an operand, a program controller pointing to the procedure being called. The logical structure for handling procedure calls is a pushdown stack. Hence the program controllers currently active are collected into a string, called the control stack, which is under the control of a stack controller. The program controller in the position of an operand is pushed into the control stack and becomes the source of the next instruction to be fetched. On return from a procedure the control stack is popped, discarding the old program controller and making the former one again available. This mechanism allows subroutine calls to any depth of nesting; it requires no special features to handle recursive calls. The operation of a trap consists of forcibly stacking a program controller which points to the trap servicing routine. For reasons to be discussed later a pointer may be found at the address pointed to by a program controller address. In this case the address taken from the pointer is used to locate the procedure head.

A problem is created by a jump from within a procedure to a labelled point outside that procedure. Many languages allow this, although we may always wonder just what the programmer really had in mind in writing such a jump. A jump into the middle of a procedure or block is even more casual. Our immediate impulse is to disallow such jumps altogether. The resulting source language restriction will not bother the FORTRAN user because all of GO TO's are local and do not come under this restriction. People who are sophisticated

enough to write meaningful non-local GO TO's in other languages are probably wise enough to know why they should not do so. One kind of jump into a procedure which is sometimes desirable and often used is exemplified by the procedure with multiple entry points. In this case the activity which takes place is a genuine procedure call-entry rather than a jump. Extra procedure entry points can be created simply by creating extra program controllers in which the ordinals are initialized to some value other than zero. This alone is not sufficient, however, for there is usually some invisible (to the user) activity associated with procedure activation which in a single-entry procedure is accomplished by instructions preceding the user's first executable statement. In a multiple-entry procedure this activity must be made a separate, invisible sub-procedure which can be called from any entry point.

There are some obvious difficulties in trying to achieve high performance with such a complicated instruction fetch mechanism. (And other mechanisms yet to be described are equally complicated.) Our approach thus far has been to seek logically correct machine structures, and afterwards to seek engineering solutions to the performance problems that they create. At least the logically correct machine can be expected to work, albeit slowly. The alternative of designing a fast simple machine and pushing the implementation problems off onto the software designers does not guarantee that the system can ever be made to work at all. Even if it can, it will not have high performance after the software designers have transformed it by programming into the machine it should have been all along.

Computational Structure:

The mechanism for expression evaluation is based upon Polish suffix notation, and uses a pushdown stack to hold operands during evaluation. The action takes place in a stack called the value stack. The value stack is a stream pointed to by a stack controller. The operation can best be illustrated by examples of expressions and the resulting procedures.

Example 1.

expression	instruction list
A = B + C;	Location A Value B Value C Add Store

The Location A instruction causes the address of A to be formed on top of the value stack. The Value B instruction fetches the current value of B to the top of the stack. Value C likewise places the current value of C on top of the stack. ADD requires that the two top-of-stack operands be values; since they are, it performs the addition and leaves the result on top of the stack. The values of B and C are consumed in the process, so that the stack now contains the value B + C on top and the location of A just beneath. The Store instruction requires a value on top and a location next, so it can now be performed immediately. Both the value and the location are consumed, leaving the stack empty (or at least with whatever contents it may have held before the execution of A = B + C;).

Example 2.1

expression	instruction list
A = (B+C)*D;	Location A Value B Value C Add Value D Multiply Store

Example 2.2

expression	instruction list
A = B + C*D;	Location A Value B Value C Value D Multiply Add Store

Example 2.3

expression	instruction list
I = I + 1;	Location I Value I Literal =1 Add Store

The lack of identifiable registers for operands means that there will sometimes be heavy traffic to and from memory (which may or may not mean actual main memory cycles); but the compiler never has to decide how best to use a limited number of hard registers, no matter how complicated the expressions become.

In the source language notation there is no distinction made between a function call and a subscripted variable. A theorist might argue that a subscripted variable is really just a kind of function anyway; but in the machine rather different actions are called for in the two instances. This is unfortunate if at code generation time the compiler must examine the meaning associated with the function or

array name before it can decide what code to generate.

Horrible Example 3.1

```
expression list
F: PROCEDURE (X,Y) RETURNS (FIXED);
   X = X+1;
   Y = Y+1;
   RETURN (X+Y);
   END F;
I = 1;
J = 2;
A(I,J) = F(I,J);
```

When F is called the values of I and J are 1 and 2 respectively. Clearly the value returned by F should be 5. But is it A(1,2) or A(2,3) which receives this value?

Horrible Example 3.2

```
expression list
F: [as defined in above example]
I = 1;
J = 1;
A(I,J)=F(1,I+J);
```

When X+1 is encountered in the body of F this expression clearly has a value of 2. But is the assignment of 2 to X permitted? And does F return the value 3, 4, 5, or something else?

We do not attempt to say what the evaluation rules in these cases "ought" to be, for this is very much a matter of personal taste. Some users favor very simple rules such that programs are essentially self-explanatory and side effects of procedures are not allowed. Others delight in constructing innocuous-looking procedures which use side effects in an obscure and devilishly clever way to accomplish astounding results. The original ALGOL 60 position was the "copy rule"; procedures should behave as if their actual parameters were copied into the procedure body in place of the formal parameters.

When simple variables are used as actual parameters this results in a call-by-name evaluation procedure. Call-by-name can be defeated by declaring the formal parameters to be call-by-value. When a constant or expression is used as an actual parameter the substitution rule fails in those statements in the procedure body which assign a value to the corresponding formal parameter. The most devastating implication of the copy rule is that an expression used as an actual parameter must in general be calculated anew for each reference to the corresponding formal parameter; for the variables in the actual parameter expression may be assigned new values in the course of execution of the procedure. Some related problems of ALGOL 60 have apparently never been laid to rest. It is not difficult to construct expressions involving functions in which the final values obtained depend upon some subtleties of the order of evaluation in the expressions. No one seems to be bothered by these problems simply because no one has ever intentionally attempted to use the kinds of expressions which evoke them.

In PL/1 the rules governing parameters are terribly complicated because of the many different kinds of objects that can be used as parameters. The actual rule of execution seems to be simpler than the copy rule of ALGOL: any actual parameter which is simply a name is called by name, while anything else is called by value. A simple name can be forced into the call-by-value mode by enclosing it within parentheses. Hence any actual parameter which could not logically be assigned a value in the course of execution of the procedure is regarded as a call-by-value parameter. Expressions used as actual

parameters are evaluated only on procedure entry. The effect of the copy rule could always be achieved if required by writing a function which evaluates the expression to be used as a parameter, and then using that function as the parameter instead of the expression. The formal parameter in this case would have the ENTRY attribute rather than an ordinary operand attribute. (In the HW machine these attributes are irrelevant.) All things considered, the PL/1 parameter mechanism seems to be more natural than the ALGOL 60 copy rule; for it prohibits the contradiction of assigning a value to a constant, and it makes the user responsible for writing in tricky side effects if he wishes to use them.

It appears possible to implement an evaluation rule of the PL/1 kind in the HW machine rather easily and in a manner which allows the compiler to generate the same instructions for both function calls and subscripted variable references. The proposed mechanism is to compile all names appearing in subscript lists and actual parameter lists as pointers to the values or procedures represented by those names. The appearance of an arithmetic operator in a parameter or subscript list causes the pointers to be followed, the values to be fetched, and then the arithmetic to be done, leaving the resulting value in the parameter or subscript list. The body of a procedure begins with allocation of storage for the local variables and formal parameters. Then the actual parameters left in the value stack are stored into the formal parameter locations. At compile time a simple name enclosed in parentheses (for call-by-value) must be distinguished from a name not so enclosed; and a Value instruction must be generated instead of the Location instruction.

A one-dimensional array variable can be stored as an indexable string. A common practice with multidimensional arrays is to employ a storage mapping function which maps the elements of the array into a one-dimensional array. For example, if A(0:1,0:2,0:4) is an array the physical one-dimensional array would contain 2*3*5 = 30 elements and the mapping function for accessing A(I,J,K) might be I+2*J+6*K. Thus A(0,0,0) would be stored in location 0 of the one-dimensional array, while A(1,2,4) would be stored in location 29. The compiler must generate instructions for computing the storage mapping function, taking the actual subscript ranges into account. Storage mapping functions can be employed in the HW machine, but a more general method is preferred. We may think of an array as a special case of a tree structure, in which the root node of the tree represents the name of the entire array and the edges leading out of the root represent the different possible values of its first subscript; or in other words a partitioning of the array in one of its dimensions. At the next level of the tree the array is partitioned in another of its dimensions, and so on until the leaves of the tree, which represent the individual elements of the array, are reached. For a rectangular array the number of levels of the tree above the root is equal to the number of dimensions; and all nodes at a given level have the same number of edges emanating from them. Selection of an element of the array is therefore a matter of traversing the tree, using one subscript at each level to choose the route to the next level. In the machine representation the root node is a pointer which points to the head of an indexable string, the elements of which represent the nodes of the next level of the tree. In traversing the tree each string head encountered demands an ordinal from the

value stack. The value of the ordinal selects an element from the string. If the selected element is not an elementary item it will be a pointer to another string head, which calls for another ordinal. All of the ordinals needed in the process can be created in the value stack from the subscript values supplied (assuming that the correct number of subscripts has in fact been supplied). When subscripts may have aribtrary origins these must be taken into account in generating the ordinals. The Rice University computer does this automatically, as the subscript-origin values are stored in the codewords which are the equivalents in that machine of our pointer and string-head objects. Origins can be handled in software by compiling the literal values of the origins and subtract instructions whenever a subscript is used; but this has the disadvantage of requiring that the compiler distinguish between subscripted-variable references and function calls at compile time. Even this can be avoided by compiling functions which take on the names of the array variables and which perform subscript-origin adjustment before making access to the real array variables. These must be rather strange functions, since array variable names can appear on either side of an assignment statement. These functions can be compiled at the time the array variable declarations are processed; whereas correction of subscript origins at the point of reference to a subscripted variable requires that remotely-located information about the variable must be considered in compiling each reference. Hence the functional treatment of arrays clearly saves compilation time and instruction storage space at run time, but at the expense of going through the function call mechanism at each array variable reference at run time. Further study will be needed to determine which method is ordinarily preferable. In

some languages a different out is implied: simply ignore the lower subscript bound. This is about all that can be done in languages which do not use array dimension declarations at all, and it is probably advisable in languages which allow only 1-origin subscripts.

Note that the tree structure built of arrays of pointers is not limited in its application to rectangular arrays. It can be used for any sort of tree, including one in which the edges leading out of a node terminate in nodes nearer to the root of the tree. Such recursively-defined trees do not lead to an endless loop at execution time because an ordinal from the value stack is consumed each time a node of the tree is passed. Sooner or later the stack will run out of raw material for the manufacture of ordinals even if a leaf of the tree has not been reached.

Example 3.3

expression	instruction list
X = A(I,J);	Location X Location I Location J Value A Store

If A were known to be a subscripted variable, and if the subscript ranges had already been adjusted to zero origin, Value instructions could have been used instead of the Location instructions for I and J. The instruction sequence given assumes that it is not known at the point of compilation whether A is a subscripted variable or a function name. Shown below is the contents of the value stack at the time the Value A instruction begins execution, and also an example of a function which might be used to access the array A

after adjusting the subscript values.

value stack	access function
Location of J	Literal = Subscript_2_Origin
Location of I	Subtract
Location of X	Exchange [the top two items]
	Literal = Subscript_1_Origin
	Subtract
	Location %A
	Return

In this example the execution of Value A will not reach the root pointer of array A; instead it will find a program controller which points to the access function. %A is a made-up name which really contains the root pointer of array A. The Subtract instructions find the location of I or J where values are expected, so these location pointers are automatically followed until values are reached. The Exchange instruction here is admittedly an artifice to make this example work. (The problem is that stacking has buried the subscript locations which have to be adjusted.) What is needed in general is either an operator which can transfer locations found in the value stack to local operands of the function or a generalized operator which rotates the n top items of the stack. In this example the Location %A instruction puts the root pointer of the physical array A into the value stack, on top of the adjusted subscript values. On return from the function the following Store operator finds:

Location of A root pointer
Value of I, adjusted
Value of J, adjusted
Location of X

The Store instruction requires a value on top of the value stack and a location under it. Since the stack is not in this condition the root pointer of A will be followed automatically to the location of A(I,J), the values of I and J being consumed in the process. Since a value is called for the value of A(I,J) will be placed into the

value stack rather than the location of this element. At this point the conditions necessary for the Store instruction to proceed are satisfied.

The array access function might be simplified slightly if we take note of the fact that an array name on the left side of an assignment operator is syntactically distinct from a function name, even if one on the right side isn't. Hence a compiler could generate in-line code for subscript adjustment in processing a left-side array reference. Then access functions would be needed only for right-side array references; and since these always return a value it would be unnecessary to return a location and subscript values from the access function. The access function itself could obtain the required value and return that. Note in passing that when a location of an array is in the stack all pointer following and use of the subscript values is deferred until either a value is required or a Store instruction is encountered. An alternative would be to follow pointers as soon as a subscript value is available, leaving in the end a pointer to an element value in the value stack. In a simple system this might be attractive; but with automatic swapping between main and backing stores going on in the background it would lead to some potential booby-traps. A pointer to an array element on the left side of an assignment statement would be developed before evaluation of the right side. During this evaluation the program might be interrupted; and during the interruption the array variable might be moved. The pointer remaining in the suspended process would then be entirely incorrect. One way out of this trap would be to make the root pointer of the array non-overlayable at the time the pointer

is brought into the value stack, and to restore the former overlayability of the pointer after the Store operator has been executed. All things considered, it seems preferable simply to defer all pointer following until use of the information developed thereby is clearly imminent. Then even if the overlaying mechanism marks a root pointer non-present just after it has been used to develop an element pointer, the operation involving the element pointer can be completed before the overlaying mechanism can possibly start moving the data.

At the time of a function or subroutine call, if we adopt the PL/1 evaluation rule, the value stack will contain the locations of all actual parameters which are ordinary identifiers not enclosed in parentheses, and the values of all other parameters. The first action taken by a procedure when it gains control will be to copy the parameters from the stack into its local variables. This will require a special store instruction, which might more properly be called a copy instruction. The copy instruction might contain the local name into which the top-of-stack operand is to be copied literally (without any automatic pointer-following); or it might use a Location instruction to put the name of a local variable on top of the stack and then simply copy the next-to-top item into the location indicated by the top item. The former seems to involve less lost motion. This same scheme can be used for the array access function. There is really no need to store array subscripts into local variables, but this may be easier than trying to implement a general stack-rolling operation. If the roll operation is omitted from the machine design there are still two options for array subscript adjustment open to the compiler writer. The compiler can

perform this adjustment either in an array access function or in the subscript list at each reference to the subscripted variable.

Example 4

expression list

```
DECLARE A(2:24) FLOAT;
A(I) = F(1,X+Y,A(J));
...
F: PROCEDURE (P,Q,R) RETURNS
   (FLOAT);
   DECLARE (P,Q,R,S) FLOAT;
   S = 1;
   P = P + 1;
   I = I + 1;
   R = R + 1;
   RETURN (P+Q+R+S);
   END F;
```

instruction list

```
[main program]
 Value I
 Literal = Sbscr_Org = 2
 Subtract
 Location A
 Literal =1
 Value X
 Value Y
 Add
 Value J
 Literal = Sbscr_Org = 2
 Subtract
 Value A
 Value F
 Store
```

instruction list

```
[procedure F body]
 Copy R
 Copy Q
 Copy P
 Location S
 Literal = 1
 Store
 Location P
 Value P
 Literal =1
 Add
 Store
 Location I
 Value I
 Add
 Store
 Location R
 Value R
 Literal =1
 Add
 Store
 Value P
 Value Q
 Add
 Value R
 Add
 Value S
 Add
 Return
```

The Copy instructions in the body of F seem awkward, since they must be arranged in reverse order with respect to the formal parameter list. Perhaps a better technique would be to pass the number of parameters in an actual parameter list as an invisible parameter following the list. This number would be at the top of the value stack as the procedure is activated. A single Copy Parameter List instruction could then copy all of the parameters into the local storage allocated to formal parameters automatically, reversing their

order in the process. An equivalent mechanism would be to pass a pointer to the beginning of the parameter list in the value stack as the final invisible parameter. Formal parameter references then could be made relative to this pointer into the value stack itself, or the actual parameters could be copied into the formal parameter local storage. Working with the parameters directly in the value stack has the difficulty that they must be protected against being gobbled up by instructions, and that they remain in the value stack at the end of the procedure. The copying process takes more time and storage, but it leaves the value stack clean at the end of the procedure evaluation, with only a returned value there if the procedure returns a value.

Reference Structure:

In the foregoing examples we have used instructions such as Value A and Location X without stating just how these operands are to be obtained. The task of the reference mechanism is to supply named operands to these instructions. In early programming systems each operand was simply assigned to a unique memory location at compile time. This obviously fails when procedures can be activated recursively and by parallel independent processes. Within the lexicographical structure of programs we note that all operands are local variables of some block (which may be a procedure or a BEGIN block). The reference mechanism must satisfy essentially two constraints:

Dynamic Constraint: Every activation of a block requires a fresh set of local variables for the duration of the activation.

Static Constraint: The sets of variables accessible to a process must at every moment be exactly those which are made accessible by the fixed lexicographical structure of the program.

The constraints apply to sets of local variables, which suggests that machine addresses be divided into at least two parts. One or more parts of an address identify the particular set of variables containing the desired operand; and one part locates a particular operand within that set. The simple variables, root pointers of structured variables, and pointers or controllers referencing procedures might as well be made a concatenated structure (a stream). Thus the location of a particular operand can be states as its offset or displacement relative to the beginning of the stream. Since the operand sizes are known at compile time (if not, the operands themselves are replaced with pointer of known size) displacements can be expressed directly in bytes. The reference mechanism then has to provide the starting address of the set of operands containing the one of interest, after which that one is located by adding the given displacement to the starting address.

Our first attempt to design a reference mechanism might be to give each block in a program a distinct block number; and to have a vector of pointers indexed by block number point to the various streams containing the local operands of the corresponding blocks. This approach fails in several areas.

1. Since procedures can be activated recursively, there must in general be a stack of sets of local variables for each

block, rather than just one set.

2. Since the entire system is a single large program all blocks of all jobs must have distinct block numbers; yet all job procedures are not compiled at the same time, so that we cannot know in general which block numbers are available.
3. Since several activations of a block can exist in parallel simultaneously even the stack of operands sets for each process is not adequate.
4. Out of all the block numbers that are assigned, only a few operand sets are accessible at any time. Thus most of the numbers and their associated pointers and stacks of operand sets are wasting storage space most of the time.

Some of these objections can be overcome by allowing each independent process to have its own vector of pointers to operand set stacks. Each such vector need contain only the stacks of operand sets which pertain to its process. Since the vector of pointers serves only to render the stacks of operand sets indexable by block number we may speak logically of vectors of stacks, even though the physical structure involves the pointers. With independent vectors of stacks we do away with the need for unique block numbers among all blocks. At any level of nesting only the relatively global blocks must have predetermined block numbers. The relatively local declarations are invisible globally and may freely make use of in-use block numbers that are mutually invisible.

This condition on the block numbers if formalized by adopting the lexicographic level of a block as its block number. For example:

```
A: PROCEDURE;                          lexicographic level = 0, operand 1
   DECLARE VA;                                  "          "  = 1,    "    1
   B: PROCEDURE;                                "          "  = 1,    "    2
      DECLARE VB;                               "          "  = 2,    "    1
      C: PROCEDURE;                             "          "  = 2,    "    2
         DECLARE VC;                            "          "  = 3,    "    1
         D: PROCEDURE;                          "          "  = 3,    "    2
            DECLARE VD;                         "          "  = 4,    "    1
            VC = 2;        [refers to l.l. = 3, operand 1]
            CALL B;        [refers to l.l. = 1, operand 2]
            . . ..
            END D;
         E: PROCEDURE;                 lexicographic level = 3, operand 3
            DECLARE VE;                         "             = 4,    "    1
            VC = 1;        [refers to l.l. = 3, operand 1]
            CALL B;        [refers to l.l. = 1, operand 2]
            . . . .
            END E;
         CALL D TASK;      [refers to l.l. = 3, operand 2]
         CALL E TASK;      [refers to l.l. = 3, operand 3]
         . . .
         END C;
      CALL C;              [refers to l.l. = 2, operand 2]
      . . . .
      END B;
   CALL B;                 [refers to l.l. = 1, operand 2]
   . . . .
   END A;
```

(In this example the operands were simply numbered, beginning with 1, whereas in machine code these numbers would be replaced by displacements in bytes.) The non-uniqueness of these block numbers is immediately apparent; there are two operands with l.l. = 4, operand 1. These two operands are mutually invisible, according to the scope rules of the language, so that their block numbers will be unique during execution. This example illustrates both an indirect recursion (B is called from D and E after being called from A while this earlier activation is still in effect) and parallel processing (D and E are called as concurrent tasks).

Imagine for the moment that the task option is absent from the calls on D and E, so that we have only a single recursive process. We have assumed that a single vector of stacks of operand sets will

process this program correctly; let us see how this works out in practice. When A is activated the reference vector contains a pointer to the head of A in position zero, and operand VA and a pointer to the head of B in position 1. A calls B via the latter pointer, causing the operand VB and a pointer to the head of C to be placed into the stack at position 2. B calls C, using the last-named pointer; this causes VC and (via pointers) D and E to be placed on the stack for position 3 of the vector. C calls D, placing VD in the level 4 stack. D references VC, which is found in the level 3 stack. It then calls B via the pointer in the level 1 stack. At this point VC, D, and VD become invisible simple because the compiler cannot possibly compile references to them; they are still lying in their respective stacks. Now the new activation of B stacks a new VB and pointer to C on top of the level 2 stack, making the prior contents of that stack invisible because of the stack mechanism. B calls C via this new pointer. C stacks a new VC, D, and E on level 3 and then calls D via this new pointer. D stacks a new VD on level 4 and references the top VC rather than the one which was referenced previously. We now assume that statements not shown in the example cause the recursion to terminate; assume a RETURN from D. This causes the top set of operands at level 4 to be discarded, revealing the previous set (if any). Now C will execute the call on E, which will proceed to stack its own operand VE on the level 4 stack. Then E references VC in the most recent activation of C and calls B, leading to another recursion. Again we assume a RETURN that is not shown in the example. The return from E causes the most recent set of level 4 operands to be discarded. The return from C similarly discards a set of level 3 operands; the return from B a set of level

2 operands. Now we are back inside D, or perhaps E, and the not-shown return mechanism causes another round of throwing off operand sets. This continues unravelling things until the return from B to A causes the last set of level 2 operands to be discarded, and then the end of A discards the level 1 operands.

The lexic levels do not have to be numbered from the outside in. Some further insight into the workings of the reference mechanism can be gained by using self-relative lexic level numbers. That is, the variables just inside the current block are at level 0; the current block and other operands declared at the same level are level 1; operands further outward are given levels 2, 3, etc. These self-relative numbers are easily converted to absolute numbers by recording the current absolute lexic level in a counter, from which the self-relative numbers are then subtracted. It has not yet been decided whether to use absolute or self-relative level numbering in the HW machine; each has its compile-time advantages.

Now let us return to the full-blown example with multi-tasking. Everything works as before up through the call on D. C then calls E without waiting for a return from D. E cannot just stack its own operand set on top of the level 4 stack, for D is still using its own operand set located there. Further, both tasks attempt to assign a value to VC. This is probably not what our obviously unskilled programmer had in mind; but the point is that the two tasks are not synchronized in any way, so there is no telling what <u>generation</u> of VC will be on top of the level 3 stack when the assignment takes place. It is clear that when there are multiple

concurrent tasks (either actually concurrent or potentially concurrent) each must have its own vector of local variable sets. Yet each must continue to reference the common set of variables that are global to all such tasks. This is accomplished by giving each independent task its own reference vector, but filling in the lower levels (lower than the task) with pointers to the reference structure of the calling program. If the calling program then is recursively called while the concurrent subtasks are still in execution the latter will continue to have access to the same operands, even though these operands have become invisible to the calling task. Further, if one of the subtasks makes a recursive call on some global procedure its own reference vector entries will be perhaps covered up, but the calling task will not be disturbed. The rules of PL/1 require that all currently active subtasks be destroyed, whether or not they have finished executing, when the calling task terminates. This means that pointers into the reference vector of the calling task cannot remain in existence invalidly after the calling task has ceased to exist.

The foregoing discussion has been concerned mainly with the dynamic aspects of the reference mechanism. Clearly every activation of a block will create a fresh set of local operands for that block; for this is built into the block entry operation. Exit from a block will turn up the previous set of operands, for the block exit operation is built that way. What has not been shown is that the proposed reference mechanism will correctly preserve the static lexicographic structure of a program. In fact it will not, as shown by the following example taken from McKeeman.*

*A Compiler Generator, page 69.

```
DECLARE C FIXED;
P: PROCEDURE (F);
   DECLARE A FIXED, F ENTRY;
   A = 3;
   CALL F;
   END P;
Q: PROCEDURE;
   DECLARE B FIXED;
   R: PROCEDURE;
      DECLARE DUMMY FIXED;
      C = B;
      END R;
   B = 2;
   CALL P(R);
   END Q;
CALL Q;
```

All seems well until P calls R via the name F. When R attempts to reference B the set of local operands containing B will be buried beneath the set of local operands of P. With lexicographic levels used as block numbers the result will be that A is actually referenced where B is intended, simply because A occupies the same relative position in the local operands of P that B occupies in the local operands of Q. Note that this problem would not have come up if we had elected to use unique block numbers for all blocks. The solution to this problem to be employed in the HW machine has not been decided. Some possibilities, in addition to that of numbering all blocks uniquely, are:

1. Legislate against the kinds of procedures and calls which cause the problem. This non-solution derives some moral support from the view that such side-effects are a means of circumventing the scope rules built into the language and are thus rather easily misused anyway.
2. When a procedure name appears in a CALL statement, create a vector of pointers to the current tops of the reference vector stacks. Pass this vector (by means of a pointer) with the procedure name in the call. When the procedure is

then called, use the vector of pointers for references instead of the existing reference structure.

3. Whenever a procedure name is introduced into a set of local variables on some reference stack, create a chain of pointers linking its scopes as of that time. On any call of the procedure use the chain links instead of the normal reference mechanism.

4. Employ a modification of the unique-block-numbers scheme, in which the compiler is expected to minimize the consumption of block numbers by detecting when a number can be re-used with absolute safety. This obviously places a burden on the compiler; but if the compiler is to make this kind of analysis of the program anyway for code optimization purposes the results will be available for block number conservation.

5. As in 2. above, create a vector of pointers at call time to the current reference environment of a procedure. On entry to that procedure, simply stack the pointers from the vector on top of the current reference stacks. This will re-create the proper addressing environment while the called procedure is active, and will restore the reference structure on exit from the called procedure.

IV

System Functions Revisited

"Machines should work; people should think." - IBM

Process Structure:

Throughout the foregoing discussion we have used the term "process" rather loosely, with the connotation that a process is something that can be executed more-or-less independently. We now apply the name process to a very specific collection of information, which is exactly the information needed to specify such an independent activity. A process consists of:

a. The stack controller for the control stack

b. The stack controller for the value stack

c. A pointer to the reference vector of stacks

d. An ordinal containing the process number.

Depending on the outcome of the procedures-called-as-parameters question the pointer to the reference vector may turn out to be a stack controller instead. The point is that a process contains in its 18 or 20 bytes all of the information needed to place a task in execution or to preserve a task for later reactivation. A process is small enough to be conveniently "handed-about" from processor to processor via queues; or it can remain in one place while a pointer to it is moved about. Still another possibility would be to arrange processes into a vector and refer to them by ordinals. This would make it unnecessary to have the process number as part of the process; but there would be holes in the process vector from time to time as processes terminate. The reason for

the process number is to allow the process to identify itself to the operating system, both for processor time charging and for memory allocation. With process numbers stored in the process, rather than implied by the location of the process in a vector of processes, it is possible for more than one process to have the same process number. This may be useful, since it automatically causes processor time for all processes having the same number to be charged to that same number, and for all processes running with the same number to share a common allocation of memory. A process is created as a result of a CALL statement containing the TASK option. It is then placed into a ready-to-run queue, from which the processors take their orders. If a process in execution is interrupted by timer runout it is simply placed at the tail of the ready-to-run queue. If the process is suspended by a voluntary relinquish the queue which is to receive it will be designated in the relinquishment. This might be a queue of processes waiting on some event variable or a queue of processes requesting service of an I/O processor. A process is liquidated when it has nothing more to do. This can be detected by an attempt to return from its own outer block procedure, which will cause the control stack tounderflow. We may also wish to allow a process to commit suicide without returning from its outer block. When a process is liquidated the operating system must be informed so that storage allocated to that process can be set free and the process number used again. If a procedure creates sub-tasks and then terminates before its sub-tasks terminate the latter should be terminated by force. Presumably this kind of thing will not be programmed to happen deliberately, since it leads to unpredictable results; and unpredictable results are rarely if ever useful.

Storage Management:

A variable-length segmentation scheme is proposed for the HW machine, with the added possibility that large segments will be paged. The segments belonging to a process are chained together in a linked list by pointers. The heads and tails of chains are locatable by means of a memory allocation vector, which is indexable by process number. Process number zero is reserved for the free chain. Non-overlayable operating system segments may be gathered together when the system is initialized and omitted from any chain, since their storage allocation is permanent. In an existing machine simulator program the chains are doubly-linked and can be traversed in either direction; but this is not absolutely essential.

Suppose that the system has been running for some time, so that several in-use chains have been established and the memory is quite checkerboarded with segments belonging to different chains. We will first consider the release of allocated storage to the free list, as this provides some necessary background for an understanding of the complementary activity of requesting more space. Also we shall assume for now that each segment contains just one structure: procedure, indexable string, or stream. This structure is pointed to by a pointer, program controller, queue controller, stack controller, etc. There is only one such controller containing the presence information for the segment; if there must be more than one controller controlling the segment a single pointer is created to contain the presence information, and all other controllers are referred to that pointer for access to the segment.

The procedure which releases space is called with either the location of the segment to be released or the location of its controller as a parameter. The segment itself contains an address pointing to the controller; this is called its control link. The segment also contains the predecessor link and the successor link, which are links to the rest of the in-use chain. The extent of the segment body is contained in the head of the structure residing in the segment. This enables one of the neighbors of the segment, the rightward segment, to be located from the head of the segment. The other neighbor, the leftward segment, ends just before the segment begins; its head can be located by using an address, the segment tail, which is located just beyond the end of the data proper. The segment releasing procedure does the following.

1. Changes the control-link to zero, to indicate a free segment
2. Cuts the segment out of the in-use chain and links the loose ends of the in-use chain back together.
3. Examines the leftward segment to see whether it is in-use or free. If free its control link will be zero. In this case the two segments are merged by increasing the extent of the leftward segment and changing the tail address of the segment to be released.
4. Examines the rightward segment to see whether it is free. If so, it is cut out of the free list and merged into the segment being released.
5. If the segment being released was not merged with its leftward neighbor it is now added to the end of the free list.

These details are open to change. For example, implementation may be easier if the segment being released is always attached to the tail

of the free chain. Then any free neighbor can be cut out of the free chain and merged into the segment being released.

A process requesting more space must furnish its process number and the amount of space requested as parameters to the space-procuring procedure. This procedure may in principle deny the request at once, and cause the requesting process to be delayed, suspended, or killed; this part of the procedure is up to the systems programmer. Assuming that the request is accepted the free list will first be searched for a sufficiently large segment. Note that segments being released could have been placed in the free list in order by size, so that on searching the list from beginning to end the first fit found would be the best fit. Instead the free list is ordered at random, so that the algorithm is first-fit rather than best-fit. This is as recommended by Knuth.[16, p. 435 ff.] If a fitting segment is found it may be entirely too large, so that it is worthwhile to split it into two parts. One of these is cut out of the free list and given to the requesting process by attaching it to the tail of the in-use chain for that process. The leftover segment, if any, remains in the free list. Once the found segment is attached to the in-use chain its control link is set to point to the controller involved in the request and then the requesting process is allowed to continue. If there is no space in the free list large enough to satisfy the request this will be indicated by a successor-link value of zero, which indicates the end of the free list.

If there is no free space the user's own in-use list may be examined next in search of something that can be overlaid. Codes in the

controllers of in-use segments indicate those which are non-overlayable, and perhaps also those which have never been written into and thus do not have to be written out to the backing store before they can be overlaid. The in-use list can be searched just like the free list, but by using a more complicated procedure the search can be made more rewarding. This procedure consists of considering the neighbors of a too-small in-use segment to see if one of them might be free, and then if the two segments combined might be large enough. It is probably not worhwhile to consider whether both neighbors are free and the three segments combined are large enough, but it will take statistical studies to answer this question more definitely. When an in-use segment is to be overlaid, with or without combining with a free neighbor, its controller is first marked non-present. Then if the segment has been written a backing-store transfer is set up and the process is temporarily suspended for completion of that transfer. Once this is out of the way the segment can be merged with a neighbor, if this is to be done, and then considered for subdividing. Subdividing will create the desired segment, to be placed at the end of the in-use chain, and a free segment to be added to the free chain (after checking for free neighbors with which it might be merged). Placing the found segment at the end of the in-use chain leads to a first-in, first-out swapping algorithm. This may be modified by the desirability of choosing non-written segments for swapping. It would be possible, for example, to keep all procedure segments at the head of the in-use chain; and since procedure segments are never written into this would increase the probability of finding

an overlayable unwritten segment to overlay. Similarly non-overlayable segments could be kept at the tail of the in-use chain, or even in a separate in-use chain, so that no time is wasted in examining them for possible overlay use. Most of the more sophisticated overlay selection algorithms require that some usage statistics be recorded. These could be worked into the segment heads or controllers; but for the present we are inclined to believe that the simpler first-in, first-out algorithm is about as good as any other, all things considered.

Since the use of the memory allocation system does entail some overhead it may be undesirable always to store one program structure per segment. It is not difficult for the system to store more than one structure per segment, although this does require some additional machinations in the compilers. Consider a tree operand such as an array. If the dimensions of the array are small all of its operands and pointers can be packed into a single segment. The root pointer acts as the controller for the entire segment. The other pointers are permanently marked present, since if any one of them can be accessed at all the entire structure must be present. The extent of the segment, as stored in the segment head, must be artificially large, as it applies to the entire structure rather than to the outermost vector of pointers. This precludes automatic bounds checking of the first subscript; but it does not invalidate protection. Bounds checks will at least insure that the subscript points somewhere within the segment. The additional complexity of these packed segments arises from the fact that the compiler must somehow force

the internal pointers or controllers to indicate present, and must never try to release the storage allocated to some part of the structure; only the entire structure can be released. Instruction strings that are nested lexically might also be nested into one segment. This is more difficult than packing an array because of overlaying considerations. The root pointer of an array is examined at every access to any element of the array. Thus the whole array can be swapped into and out of main storage at any time so long as the presence information in the root pointer is kept up to date. Instruction fetching references only the current program controller. The system might overlay an outer block and all of its contents while an inner block was still being used. This can be prevented by an appropriate mechanism, although the prevention may be more trouble than it is worth. For example, an inner block might contain instructions on entry to make its containing block non-overlayable, and to restore its former overlayability on exit.

It is easiest to talk about the operation of the system in terms of processes that are in execution. Other matters that must be considered include the complexities of getting processes started from scratch. At compile time a program consists of executable statements, literal constants, and variables that have not yet received any values. (Variables which are to receive initial values by an INITIAL attribute can receive these values at run time by instructions which store literal constants into the variables.) In principle a compiler can pre-evaluate some expressions of a program and in general bring the program to a point beyond which it cannot proceed without run-time input. Such a pre-evaluated program could be expressed by

transforming it into a program with INITIAL declarations, or it might have values already assigned to its variables. The compiler output thus may contain instruction strings, undefined variables, and variables having values. To get a job started a process must be created for it. The control stack must contain a single initialized program controller pointing to the outer block of the job by the time the first instruction is to be fetched. The value stack will be empty; and the reference vector must contain pointers to the global reference vector. Meanwhile all the procedures of the job presumably reside in data segments of the compiler, or in some other non-executable form in mass storage. For a procedure to be executable its head must indicate that it is a procedure, and it must have a pointer or controller in a reference stack somewhere from which it can be called. If the procedure segment is present this pointer will point to it; otherwise the address in the pointer will point to a control word containing the backing-store address of the segment and its extent. The big problem is that reference stacks come and go with procedure activation, while the instruction strings of procedures endure at least for the life of a process. These instruction strings are quite similar to the own variables of ALGOL or the STATIC variables of PL/1. One way to implement STATIC variables is to ignore their actual positions in the nesting structure of programs and to treat them as if they were declared in the outer blocks. It is up to the compiler to work around any naming conflicts and programming errors of scope violation which might result from this change in declaration. In other words, the compiler doesn't tell the user where it really put the operands and doesn't allow him to reference

them globally unless they are really declared that way. Hence it seems reasonable to treat instruction strings in the same way, hiding their declarations in the outer block. Since the outer block lasts for the life of the job this means that the process of connecting the procedure segments with their declarations has to be done only once. It is now reasonable to imagine a loader system procedure which performs the special function of creating outer blocks for jobs. Note in passing that this method of declaring procedures effectively results in giving each procedure block a unique block number. Also, each block must now contain instructions to set up the correct reference stack on block entry and to restore it on block exit; no longer can the reference stack number be implicit in the block's own level of declaration. The term "loader" is used only by analogy to more conventional systems. It does not really have to load anything into main storage since this will be done dynamically by the overlay mechanism. The loader simply obtains an empty segment, fills it with pointers to the instruction strings of a job, and stacks this segment on the appropriate reference stack in the process that has been created for that job. Some sort of unloader will also be needed to release the backing storage of a job at its termination, and to destroy its outer block. Further, it will often be desirable to call the loader procedure from inside a program. This will allow a prototype compiler, for example, to run as a user job and to have its output placed into execution. Indeed, an entire prototype operating system could be run as a user job under an existing operating system. This would also be advantageous if it is desired to run time-sharing concurrently with batch processing; the time-sharing system would run as a single batch job.

APPENDIX I

A BNF Description of Data in the HW Machine

The following is intended only for data description; it is not a true grammar. Semantic explanations are enclosed in [square brackets].

```
<primitive element>::=<character>          [1 byte]
                     |<ordinal>            [2 bytes]
                     |<address>            [3 bytes]
                     |<number>             [5 bytes]
                     |<d-p-number>         [9 bytes]
                     |<instruction>        [1 or more bytes]

<character string>::=<character string><character>
                     |<character>

<ordinal string>::=<ordinal string><ordinal>|<ordinal>

[and so on, for strings containing primitive elements of only one kind]

<pointer>::=00110<pointer control><address>

<pointer control>::= 111 [absolute address, present, non-overlay]
                     |110 [presence unknown: follow pointer]
                     |101 [present, overlayable, written]
                     |100 [present, overlayable, never written]
                     |010 [a ghost: storage has not been allocated
                           yet, and the address is the size needed]
                     |000 [non-present: address points to control
                           word containing location]

<pointer string>::=<pointer string><pointer>|<pointer>

<homoomeral string>::=<character string>
                     |<ordinal string>
                     |<address string>
                     |<number string>
                     |<d-p-string>
                     |<pointer string>
                     |<program controller string>
                     |<stack controller string>
                     |<queue controller string>

<program controller string>::=<program controller string>
                                  <program controller>
                              |<program controller>
```

[stack and queue controller strings defined in the obvious way]

<typed element>::=<typed character>
|<typed ordinal>
|<typed pointer>
|<typed number>
|<typed d-p-number>
|<typed stack controller>
|<typed queue controller>
|<typed program controller>

[for use in streams]

<typed character>::=00010000<character>00010000

<typed number>::=00010001<number>00010001

<typed d-p-number>::=00010010<d-p-number>00010010

<typed pointer>::=<pointer>00110xxx

<typed stack controller>::=<stack controller>00011xxx

<typed queue controller>::=<queue controller>00101xxx

<typed program controller>::=<program controller>00100xxx

<indexable string>::=<index head><homoomeral string>

<index head>::=0000<size code><extent>

<size code<::= 0001 [character - 1 byte]
|0010 [ordinal - 2 bytes]
|0011 [address - 3 bytes]
|0100 [pointer - 4 bytes]
|0101 [number - 5 bytes]
|0110 [program or stack controller - 6 bytes]
|1000 [queue controller - 8 bytes]
|1001 [d-p-number - 9 bytes]

<stream>::=<stream head><typed string>

<stream head>::=00000000<extent> [an index head with zero size]

<typed string>::=<typed string><typed element>|<typed element>

<procedure>::=<procedure head><instruction string>

<procedure head>::=00001011<extent> [an index head with size 11]

<program controller>::=00100<pointer control><address><ordinal>

<stack controller>::=00011<pointer control><address><ordinal>

<queue controller>::=00101<queue control bits><address><ordinal 1>
<ordinal 2>

<queue control bits>::= 111 [present, non-overlayable, non-empty]
|110 [present, non-overlayable, empty]
|101 [present, overlayable, non-empty]
|100 [present, overlayable, empty]
|010 [non-present ghost, address is size]
|000 [non-present, address points to control word containing location]

<ordinal 1>::=<ordinal> [the head of the queue]

<ordinal 2>::=<ordinal> [the tail of the queue]

::=<predecessor link><successor link><control link>
<segment body>

<predecessor link>::=<address>

<successor link>::=<address>

<control link>::=<address>

<segment body>::=<indexable string><segment tail>
<stream><segment tail>
<procedure><segment tail>

<segment tail>::=<address>

<process>::=<control stack controller><value stack controller>
<reference vector pointer><user number>

<control stack controller>::=<stack controller>

<value stack controller>::=<stack controller>

<reference vector pointer>::=<pointer>

<user number>::=<ordinal>

Summary of type codes:

0000 index, stream, or procedure head
00010000 typed character
00010001 typed number
00010010 typed d-p-number
00011 stack controller
00100 program controller
00101 queue controller
00110 pointer

The distinction among indexable strings, streams, and procedures

is based on the size code part of the head. If the sizes of elements in indexable strings are limited to values between 1 and 15, excluding 11 and 13, an efficient hardware operation of indexing can be realized. The beginning of any item, relative to the beginning of the string, is located at N*S, where N is the index value (zero origin) and S is the item size. This multiplication can be performed by a two-input adder (or less) for sizes of 1-15 except 11 and 13. Hence these awkward size codes are tentatively reserved for non-indexable strings.

A different typing system has also been considered and will be described briefly. This system assigns a type code to every element, even though those elements stored in indexable strings do not require individual type codes. It is based on a memory addressable by nine-bit byte, and uses a variable-length code. The variable length code minimizes the use of type bits where they are hard to come by, and uses them lavishly where they are abundant. For example, a character is represented by a leading type bit of 1 and 8 bits of data. Any other item has a leading type bit of 0. An instruction has a leading type code of 01 and 7 bits of data - or more if a multi-byte instruction is involved. All remaining elements have leading bits other than 1 or 01. Leading codes of 0011, 0010, and 0001 are available for three more elements which can use these sizes conveniently. Thus a 14-bit ordinal could be made with two bytes, the first of which has a code of 0011; and a 23-bit address can be made with three bytes and a leading code of 0010. Leading codes of 0000111, 0000110, 0000101, and 0000100 accomodate four more elements, after which the codes begin with 00000 and so on. By juggling the

codes and data elements like this it is usually possible to arrive at satisfactory elements which have enough bits for data and then fill out an integral number of bytes with code bits. The coding technique is similar to Huffman coding, except that the goal is not to minimize the total expenditure of bits so much as it is to minimize the expenditure of bytes while providing operands of appealing length.

Bibliography

1. Anderson, James P.:"A Computer for Direct Execution of Algorithmic Languages"; Proc. Eastern Joint Computer Conference, 1961, page 184.
2. Anderson, James P.; Hoffman, Samuel A.; Shifman, Joseph; and Williams, Robert J.:"D825 - A Multiple-Computer System for Command and Control"; Proc. Fall Joint Computer Conference, 1962, page 86.
3. Barton, R. S.:"A New Approach to the Functional Design of a Digital Computer"; Proc. Western Joint Computer Conference, 1961, page 393.
4. Barton, R. S.:"The Interrelation Between Programming Languages and Machine Organization"; Proc. IFIP Congress 65.
5. Bashkow, Theodore R.; Sasson, Azra; and Kronfeld, Arnold: "System Design of a FORTRAN Machine"; I.E.E.E. Transactions on Electronic Computers, Vol. EC-16, No. 4, August, 1967.
6. Bock, R. V.:"An Interrupt Control for the B5000 Data Processor System"; Proc. Fall Joint Computer Conference, 1963, page 229.
7. Bryant, Peter:"Levels of Computer Systems"; Communications of the A.C.M., December, 1966, page 873.
8. Carlson, C. B.:"The Mechanization of a Pushdown Stack"; Proc. Fall Joint Computer Conference, 1963, page 243.
9. Cleary, J. G.:"Process Handling on Burroughs B6500"; Proc. Fourth Australian Computer Conference, 1969, page 231.
10. Hauck, E. A.; and Dent, B. A.:"Burroughs' B6500/B7500 Stack Mechanism"; Proc. Spring Joint Computer Conference, 1968, page 245.

11. Hopper, Grace M.;"and Mauchly, John W.:"Influence of Programming Techniques on the Design of Computers"; Proceedings of the I.R.E., Vol. 41, No. 10, October, 1953, page 1250.

12. Iliffe, J. K.; and Jodeit, Jane G.:"A Dynamic Storage Allocation Scheme"; The Computer Journal, October, 1962, page 200.

13. Iliffe, J. K.:Basic Machine Principles; London, 1966, [U.S.A. distribution through American Elsevier Publ. Co. Inc.]

14. Iliffe, J. K.:"Elements of BLM"; The Computer Journal, August, 1969, page 251.

15. Jodeit, Jane G.":Storage Organization in Programming Systems"; Communications of the A.C.M., November, 1968, page 741.

16. Knuth, D. E.:The Art of Computer Programming, vol. 1, "Fundamental Algorithms"; Addison-Wesley, 1968.

17. Halstead, M. H.:Machine-Independent Computer Programming; Spartan Books, Washington, D.C., 1962.

18. McKeeman, W. M.:"Language Directed Computer Design"; Proc. Fall Joint Computer Conference, 1967, page 413.

19. Melbourne, Alan J.; and Pugmire, John M.:"A Small Computer for the Direct Processing of FORTRAN Statements"; The Computer Journal, April, 1965, page 24.

20. Mullery, A. P.; Schauer, R. F.; and Rice, R.:"ADAM - A Problem-Oriented Symbol Processor";Proc. Spring Joint Computer Conference, 1963, page 367.

21. Randell, B.; and Russell, L. J.:ALGOL 60 Implementation; Academic Press, London, 1964.

22. Richards, R. K.:"New Logical and Systems Concepts"; Proc. Eastern Joint Computer Conference, 1958, page 51.

23. Rosen, Saul:"Hardware Design Reflecting Software Requirements"; Proc. Fall Joint Computer Conference, 1968, page 1443.

24. Thompson, Rankin N.; and Wilkinson, John A.:"The D825 Operating and Scheduling Program"; Proc. Spring Joint Computer Conference, 1963, page 41.

25. Weber, Helmut:"A Microprogrammed Implementation of EULER on IBM System/360 Model 30";Communications of the A.C.M., September, 1967, page 549.

26. Wegner, Peter (editor):Introduction to System Programming; Academic Press, London, 1964.

27. Wirth, Niklaus; and Weber, Helmut:"EULER: A Generalization of ALGOL and its Formal Definition"; Communications of the A.C.M., January, 1966, page 13, and February, 1966, page 89, and errata December 1966, page 878.

28. Wirth, Niklaus:"On Multiprogramming, Machine Coding, and Computer Organization";Communications of the A.C.M., September, 1969, page 489.

29. Denning, Peter J.:"Virtual Memory"; Computing Surveys, September, 1970, page 153.

30. Huskey, Harry D.:"Semiautomatic Instruction on the Zephyr"; Proceedings of a Second Symposium on Large Scale Digital Calculating Machinery; Annals of the Harvard Computation Laboratory, Harvard University, 1951, page 83.

31. McFarland, Clay:"A Language-Oriented Computer Design"; Proc. Fall Joint Computer Conference, 1970, page 629.