The Design of a Programmable Computer:

A Qualitative and Quantitative Analysis

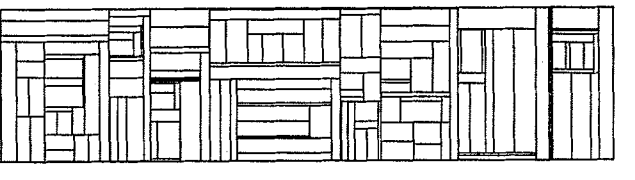
by

Michael J. Manthey

DAIMI PB-31
JUNE 1974

This work was conducted with the support of NSF grant # GJ-993.

Institute of Mathematics University of Aarhus
DEPARTMENT OF COMPUTER SCIENCE
Ny Munkegade - 8000 Aarhus C - Denmark
Phone 06-12 83 55



Abstract

A survey is made of the contemporary state of program construction and subsequently the isolation of the areas where errors and programming difficulties creep in. A machine architecture which attacks these difficulties is then presented, first in its larger outlines, and then a more detailed exposition of its operation. Finally, this machine architecture, which is based on block structure, is quantitatively compared to two "standard" machines (the IBM S/360 and the DEC PDP-10). The result of this comparison is that one should expect superior performance from the block structured machine herein presented, perhaps by as much as a factor of two.

Keywords: block structure, machine architecture, microprogramming, tag bits, delayed binding time, stack machine, dope vector.

Contents. page

1. 0 CONTEMPORARY COMPUTER USAGE

1. 1 Historical Perspective

The usage of computers has progressed far from the days when it was estimated that five Univac I's could totally saturate the national requirements for computation. In large part this advance is the result of the realization that a computer is a general purpose symbol processor capable of far more than just floating point calculations. Indeed, the evolution of operating systems and concomitant software (loaders, assemplers, compilers) parallels the development of the computer as symbol processor. There can be no doubt that the early pioneers in the field would gasp in amazement at the sophistication and complexity of a modern computing system.

The development of these software complexes has not come cheaply, however. The history of computer usage is littered with failures large and small, and even the most glorious software triumphs have sprung phoenix-like from near disaster e. g. the SABRE airline reservation system, the B 5000/5500, and most currently MULTICS (whose Resurrection was by no means assured). Even when software projects have been successfully completed, they have most often been the product of cost overruns and downward-revised performance specifications. Part of this checkered success record is the result of the fascination of computerniks with their toys, their unshakeable faith in the potential of their 'children'. A second influence has been the unfortunate alliance between software's unprecedented malleability (the bell and whistle syndrome) and the nascent development of software construction as a science instead of an art. [The fact that Knuth could contemplate gathering all of programming technology in a unified work and then entitle it " The Art of Computer Programming" is a capsule statement of this situation.] A third influence on software technology's stuttering progress is the hardware, the machines on which the dreams are to be realized: that hardware advances were each time hoped to solve 'the software problem' indicates the position that hardware was felt to hold in the minds

of implementors.

The current status of these problems is as follows:

1. While computerniks are still entranced by the computer's potential, 20 years of experience now blends this fascination with realism.
2. Fundamental theoretical knowledge of software primitives (e. g. mass storage queueing, grammars) is slowly being felt, but the seat-of-the-pants approach is still commonly found.
3. The hardware supplied by manufacturers remains fundamentally unchanged but is garnished with certain features which are obviously required by the software plans e. g. bounds registers.

Looking back over the history of computing i. e. computers, one finds that computers were the creations of electrical engineers, who, until faced (as they are now) with the limitations of the speed of light, were in the vanguard of advancing computer technology. However, if we survey the current state of computing, it appears that software, and not hardware, is the stumbling block. It therefore makes sense that hardware must do more that just 'be there on the floor', but rather must take an active role in supporting the environment which software complexes demand.

Unfortunately, 'hardware types' are still much in control of the design process. The following true life parable illustrates the type of problems this causes.

> K is a brilliant digital logic designer who is also an excellent programmer. He was told of the inconvenience of performing programmed I/O on devices such as teletypes, paper tape gear, etc. when one would rather relegate such business to a data channel. K himself had experienced this as well. Unfortunately the minicomputer involved did not allow this, so K set out to design a multiplexed data channel which would relieve the software people of their worries. And a wonderful design it was! The programmer had only to set up a table in core which contained the

necessary parameters and then fire off the channel. Better yet, K was able to take advantage of some already existing hardware features and thereby build the device very cheaply. At this point it should be mentioned that logic designers are (rightly) obsessed with cost [$] due (if only) to their formal education. True to form, K decided to build the device with the restriction that the parameter table must begin on a memory page boundary! The software people were horrified, as this caused severe memory allocation problems and restricted dynamic memory management. K's reply was that the restriction saved about $20 since the low order bits of a register would be zeroes and thus buildable with much cheaper components.

The basic problem was that while K is an excellent programmer, he understands little about the environment in which programs execute. Grounded firmly in his knowledge of programming, he maintained that the restriction represented only a minor inconvenience. K won the day, and due (in large part) to the restriction, the device never saw use.

Another common hangup of hardware designers is speed - any operation which takes more than X micro (nano) seconds is " too slow ". When faced e. g. with the requirement for a set of semaphore primitives (which might have to make several accesses to memory and perform several different functions), the designer replies

(1) it's too slow,

(2) it costs too much,

(3) and besides, all you really need is a " replace add 1 " which is both cheap and fast !

As in the preceding parable, this kind of reasoning indicates a lack of awareness of the environment in which e. g. semaphores are used. This environment is the dynamic and complex world of the operating system, a world where experience has shown that what can go wrong will go wrong, and hence hardware constructs which diminish the universe of potential bugs are valuable regardless of their speed or other such myopic qualities. After all, a program which runs with blinding speed, but usually crashes due to unfindable bugs, is of no value to anyone.

The use of hardware constructs which directly reflect higher-order processes clearly has smaller potential for error than the simulation of these processes via software, which can be subject to unforeseen effects such as interrupts, mis-association of parameters or other names, violation of usage convention, or accidental overwriting. Thus what is needed is for the hardware to be structured in such a way as to create a comfortable environment for programs to execute in, an environment in which whatever basic operations are necessary can be stated as hardware primitives. If all the basic software operations e. g. expression evaluation, indexing, semaphores, etc. have counterparts in the hardware environment, then the 'distance' between the source language program and its implementation in hardware is greatly diminished, and concomitantly the universe of potential errors which enter as the result of improper bridging of the gap.

This 'distance' between the source language and the object computer can be seen to be proportional to the size of the so-called 'runtime environment': if the distance is great then the runtime environment is large, and vice versa. Thus a true e. g. Fortran machine would have no code devoted to runtime support. The seemingly ideal situation of completely hardware-based environmental support is not realistic, however, in view of the fact that contemporary computing requirements are for multiple languages on the same computer, and hardware which is suitable for Cobol I/O or Fortran addressing may not be for Algol or PL/I. In addition, certain aspects of the environment are most cost effectively accomplished in software e. g. I/O formatting. It can thus be concluded that completely hardware-based environment is not practical.

Impinging on the discussion of substitution of software by hardware is the much bandied about concept of 'efficiency'. Efficiency is usually associated with speed, as well as the converse: inefficiency equated with slowness. We define efficiency as accomplishment of certain specification with the maximum speed, given the available tools. Thus the complaints about early OS/360, bugs aside, are not totally justified since the slowness was only partly due to poor coding or whatever, the remainder coming from the unsuitability of the given tools: the 360 instruction set, etc. A second example is Snobol 4, which runs slowly enough, to be sure, but compared to what?

Efficiency also applies to hardware design, since all hardware is built out of the same logical building blocks. Thus if two groups are given the same specification and one builds it faster (costs being equal), that one is clearly more efficient. But what if the winner were also more expensive? Or what if the specification was to produce a system that minimized software costs? How does one measure?

The preceding discussion reveals that efficiency is a term that everyone understands, but whose meaning is very slippery. Indeed, defining 'efficiency' is not unlike Plato's defining 'the good'. Faced with this impasse, and yet unwilling to abandon the concept, we shall henceforth utilize the working definition that " X is inefficient if it could be done (in some global sense) better". Clearly this definition is more efficient that not using the word at all!

With these conclusions in mind, we next ask what contribution should the hardware make toward the environment? The answer is in the form of two choices which delimit a choice spectrum : (1) the hardware does what is minimally necessary with software 'filling in' the gaps, and (2) the hardware does as much as possible while still not restricting a particular language's needs. The former is characteristic of 2nd and 3rd generation computers, while the latter has been approached by several atypical architectures [I2], [B3], [B5]. The principal problem posed by the second choice is the isolation of those primitive features required by all languages and systems, and the following sections examine contemporary programming practicees on the assumption that problem areas grow out of the lack of good environmental support. Isolating the problem areas is therefore the first step in the derivation of (hardware) environmental primitives.

1.2 Problems in Applications Programming.

Contemporary applications programming (by which is meant programs which fulfill a specific user need e. g. data processing, equation solving) is done almost exclusively in higher level languages (HLL's) such as Cobol, Fortran, Algol, and PL/1, the primary exception being real time applications.

Unfortunately, in spite of the popularity and proven effectiveness of HLL's, construction of even relatively simple programs is often a painful process. Even assuming (possibly incorrectly) that the chosen HLL is appropriate for the problem statement, the programmer still faces the difficulties of debugging, as well as interfacing his program to other programs, data files, and the ubiquitous System.

1.2.1 HLL Suitability

The applications programmer faces a dual problem in implementing a program in an HLL. The first is choosing (or learning) an HLL in which to write the program, and the second is adapting his problem to that language. The choice begins with " Cobol, Fortran, or PL/1 ? " versus a whole menagerie of less common and more special purpose languages. The choise is usually in favor of one of the Big Three since these are the best supported by, and also somewhat standard across, vendors. This situation exists primarily because even supporting three language processors is an awesomely expensive endeavor.

Having chosen one of the common (and hence familiar) languages, the programmer is faced with stating his problem within the syntactic and semantic confines of that language. Herein lies the fundamental issue: the program executed by the computer is doubly distant from the original application problem since

(1) the problem was translated (by the programmer) into the HLL, and
(2) the HLL was translated (by a compiler) into machine code.

The first translation's effectiveness is a function of the programmer's skill and cleverness and the language's suitability, which are more or less user controllable factors. The second, however, directly confronts us with the 'distance' between the source program language and the actual hardware machine.

1.2.2 Debugging

Program bugs can be classified as arising out of (1) the programmer's

translation from the problem to the HLL, and (2) the compiler's translation from the HLL to machine code. While the former is an intangible, the latter is (theoretically) amenable to action. This action could be in the form of better documentation or improved runtime diagnostics, but the real crux of the problem lies in the 'distance' mentioned earlier which seperates the source program from the object (machine) program. In effect, documentation explains how the compiler has bridged the gap (via the runtime environment and generated code) between the source and machine program, while the runtime diagnostics may or may not inform the programmer that he has violated certain explicit or (worse yet) implicit assumptions of that environment. Most often, these assumptions are based on the raw physical realities of the particular computer involved e. g. word length, type of arithmetic, and addressing peculiarities.

1. 2. 3 Summary of Applications Programming Problems

The preceding sections have discussed the problems faced by the applications programmer : choosing a language and debugging the program. The major conclusion is that the basic problem is the gap between the source language and the object language, the gap being filled by the user-unknown machinations of the compiler and a large and mostly mysterious runtime environment.

1. 3 Problems in Systems Programming.

This writer's defination of a 'system program' is a program whose data are other programs. As such, system programs perform in the general areas of

(1) generating programs e. g. compilers,
(2) providing services to programs e. g. runtime environment,
(3) manipulating programs e. g. loading, scheduling.

We now examine each of these areas in greater detail.

1. 3. 1 Programs that Generate Programs

Compilers represent a magnified view of the problems which faced the applications programs: choosing a language, debugging, and system interfacing. The source of this magnification is that a compiler must

simultaneously be aware of two environments: its current executing environment, and that of the program it is generating.

For example, a compiler is a large complex program which utilizes broad range of system services and is usually written in assembly language. But because of the known [productivity, documentation, correctness proving] disadvantages of a large program written in assembly language, the contemporary trend is toward HLL's with special features [the nature of the data manipulations (character and bit strings, packed tables) is such that to state these manipulations in an HLL without such special features implies intolerable inefficiencies.] Since additionally compilers are being written in their own language (implying-therefore that the compiler and user runtime environments are one and the same), the question now arises of the extent to which the 'special features' should be available to ordinary users - in view of the fact that system integrity could thereby be compromised. Yet such features as runtime availability of the symbol table and the ability to build code at runtime are obviously desirable. [It should be observed at this point that the severity of this dilemma is lessened if a compiler is not 'special' to the system i. e. the hardware-software complex is such that even misuse of system services affects only the user program.]

The gap between the source language to be compiled and the target machine also has the same doubled effect mentioned above. A compiler written in an HLL is itself remote from the object-form compiler, and must also generate code which plugs this gap. The wider the gap, the more difficult it is to bridge and the less efficient is the resulting compiler.

The issue of system interfacing in a compiler is particularly severe since the compiler must generate calls on service routines which will be executed by the generated program in a manner which is not completely predictable at compile time. Therefore the service routines must be general, extremely self-protective, and able to deal with a myriad of possible situations (this last being particularly true of time dependent functions such as I/O).

In summary, compilers are faced with the following problems:

- the nature of the compilation task versus the capabilities of the hardware host machine
- the nature of the compilation task versus the capabilities of the target machine
- the extent of user access to compile time information and abilities.

1.3.2 Service Programs

Service programs fall into two categories - those which can be viewed as subroutines (e. g. conversions) and those which must be viewed as co-routines (e. g. buffered I/O). The former tend to be small and logically simple functions which are very concerned with the conventions ruling themselves and their fellows; the latter are distinguished by their logical complexity and intimate relationship to the operating system. Likewise, the service subroutines are closest to the user and should be concerned with divining the cause of errors so that the user can be informed of the error in terms that relate to his source program; in contrast, the coroutine services tend to be invisible to the user and exclusive of simple user exceptions (e. g. end of file), any errors which they detect represent bugs or actual errors in the object code, whether due to the compiler or user self-destruction. Experience has shown that determining how code came to be destroyed is extremely difficult and we should therefore strive toward an environment in which such is not the case.

We should also take note of the peculiar nature of almost all service programs in that they give the user the ability to temporarily step outside of his formal execution environment i. e. they are a system approved means to 'break the rules' . As an example, consider type conversations : they are defined as an automatic mechanism in the Fortran, PL/1 etc. machine, but must necessarily be accomplished outside that environment. Often the System provides analogous services to the environment itself e. g. storage requests.

Since the fulfillment of such requests without error is so critical to the System, it must ensure that these routines are protected from accidental damage. In 3rd generation machines this has been accomplished through the Supervisor Call mechanism which asks the System to exe-

cute a piece of code which is not in the user's address space. This scheme is easy to implement on contemporary machines but in turn imposes an interrupt burden on the System. It is important to note that the purpose of this mechanism is not to prevent the user from invoking the service routine, but merely to ensure the integrity of the code. This scheme is used because the storage protection mechanism on most contemporary machines have insufficient resolution to allow multi-user access to code while ensuring entry only at designated entry points and forbidding write operations therein. Clearly it is desirable to relieve the operating system of the interrupt burden so long as the other constraints are met.

1.3.3 Programs that Manipulate Programs.

This category of system program is essentially the nuclear operating system plus allied functions such as the spooling and file management subsystems. The nuclear operating system includes the linkage editor/loader, traffic controller and scheduler, memory and I/O device manager, and interrupt handler; these functions usually communicate with each other through a number of global tables which contain the status of all entities in the system.

The nuclear system is itself not very large, but is logically intricate due both to the complexity of the job it is doing and the presence of interrupts. It would be very helpful to be able to write the operating system in an HLL, but two considerations have traditionally discouraged this : the requirement for object code efficiency, and the lack of source language constructs for e. g. interlock, interrupts, and multi-tasking. Additionally, if the operating system is written in an HLL, then it is actually executing on an HLL machine and must move outside this machine when it wishes to break certain rules, as an operating system is wont to do. Note that this situation is exactly analogous to that of a compiler written in its language.

The reasons for the desire to write in an HLL are the usual ones - clarity of statement, and ease of maintenance and debugging; these considerations are especially important when one considers the magnitude of coding represented by a full system. The requirements of efficiency and necessary language constructs have been the stumbling

blocks; nevertheless, several outstanding examples exist of systems written in HLL's,notably the Burroughs MCP's and MULTICS. In the case of Burroughs, the hardware has been structured so that compilation is quite efficient, whereas MULTICS uses a PL/1 subset which contains only what is needed coupled with a compiler and documentation which together yield acceptable object efficiency.

The question of 'whither HLL' aside, operating systems in general can be characterized as having overlong childhoods and being rather fragile in adulthood. OS/360 is a case in point, but is by no means alone. They can also be characterized as being too large, too complicated, and too slow - this latter in spite of being written in assembly language. One exception tothis bleak picture, but chiefly with respect to efficiency, is Burroughs who claims [D4] system overhead on the order of 5-10 % while most systems are in the 15-25 % bracket.

1.3.4 Other System Trouble Spots.

1.3.4.1 Linkage/Loading.

The initial concept of a loader was a program which simply loaded an object program into memory. Unfortunately this happy state of affairs was quickly followed by a large number of demands on the loader's abilities: satisfaction of external addresses, overlay structures, memory (data) initialization, partial linkage etc. The result is that on most systems the loader is an exceedingly complex and creaky program, one that takes an inordinate amount of time to get rolling even for a trivial job; and heaven help the user who attempts to use the more exotic so-called 'standard features' - if they work, it usually is different from the manner specified in the documentation.

The root of the problem lies in the type of address space used by most computers: linear. The linear address space forces the System to demand that a program lie in a (address-wise) contiguous area of memory, and thus the loader must build a monolithic load module. Since the same loader is usually used for user jobs and the entire system, it must be prepared for even the most incredible eventualities relative to table space and options. Additionally, all external addresses must be resolved , whether or not they are ever invoked.

It is safe to say that if loaders did not have to resolve external addresses and link everything together all at once, they would shed most of their problems. It is possible to structure hardware to help in this endeavor, as we shall see.

1.3.4.2 Interrupts.

An interrupt can be viewed as a time-random procedure entry in a program; as the result of some hardware generated condition, a procedure entry is forced to some environmentally-defined location, where execution is resumed. [In most computers, this is accomplished by saving the program counter in a register, then setting it to the interrupt routine address. If the register contents should be destroyed, there is absolutely no way to either recover or possibly even to figure out why you are where you are.] Presumably the interrupt handler does whatever it deems necessary and then causes execution to proceed once more from the point of interruption. As far as the typical user program is concerned, there are no problems.

However, if the interrupt should occur while the System is itself performing some state-dependent function, and the interrupt routine unknowingly modifies the state, disaster invariably ensues. The obvious solution is to allow interrupts to be disabled and enabled under program control, which works fine as long as interrupts are not disabled too long - after all, an interrupt often represents the occurrence of a time dependent event, and as such, cannot be ignored over long. Furthermore, some interrupts are more time critical than others, and many computers allow interrupts to be enabled/disabled on a proirity basis.

Especially in older systems, the interrupt handler always operated with interrupts disabled due to the difficulty of design which would otherwise be necessary. Indeed, most contemporary operating systems are either partially or entirely nonreentrant, and rely on interrupt discipline and coding conventions to prevent reentry to nonreentrant code.

We can conclude that if interrupts

(1) could be forced in such a way that the path was unlosable, and

(2) weren't time dependent, and
(3) all code was reentrant, and
(4) code which shouldn't be interrupted or reentered absolutely couldn't be,

then the impact of interrupts on operating system design and stability would be minimized.

1.3.4.3 Lockout and Events

The fourth item in the list above - code which can't be interrupted-is the subject of a classic paper by Dijkstra [D2], who denotes such code 'critical'. An example of critical code would be a routine which is responsible for updating a multiple entry table: if an interrupt occurs before all the entries have been made, and in the course of processing the interrupt this routine is reentered, the table will hereafter contain invalid information. Dijkstra demonstrates that this problem can be solved by the use of 'semaphore' variables i. e. variables which can be updated without possibility of being 'seen' in their transitory state by another piece of code.

While Dijkstra's paper assumes that multiple processors are executing cooperating pieses of code which must be coordinated by the use of semaphores, a single interruptable processor executing on logically distinct processes interrupts poses the same problems. In traditional computing systems, the situation has been resolved either by disallowing interrupts throughout the (logical) vicinity of critical code, or by using the increment memory (" replace add 1") feature, or a combination of the two. The increment memory instruction constitutes the necessary primitive to accomplish the semaphore strategy. In actual operation, what happens is that a process will increment and test the semaphore, and if it finds the semaphore 'busy' must either (1) continue to poll the semaphore until it becomes 'unbusy' (very wasteful of compute time) or (2) ask the System to place it in a wait state until the semaphore clears. Current terminology for this situation is 'waiting for an event', although events are not restricted to being semaphores e. g. an interrupt.

While it is true that Dijkstra's semaphores are useful primitives, they demand more than 'increment memory' to be truly efficient e. g. suppose

process A locks semaphore S, and then process B 'blocks' on S and is queued by the System. Now process A unlocks S; if the unlocking action neglects to inform the System that S is now unlocked, B is blocked forever. This can be avoided with today's hardware, but multiple instructions and programming conventions are required; in addition, the semaphore is associated with its target variable only by convention, not by hardware, and therefore bugs can creep in via misassociation. The problem of 'deadly embrace' [2] is a meta-problem in this context, and as such is outside this scope of this paper [see H3].

Another inefficiency deriving from the need to use multiple instructions to test and react to semaphores is that several different types of critical code are involved i. e. producer/consumer, non-reentrant code , and data lockout. While it is true that all can be accomplished using simple semaphores, usage becomes clearer and more efficient if the different types of lockout are available [R2], [H1].

Thus far the discussion has been directed toward the System's use of semaphores. However, contemporary usage e. g. real time subsystems, cries out for allowing user programs as well as the System to spawn subtasks coordinate their activity via semaphores and event variables. Furthermore, these facilities are most needed in HLL's since users prefer to write in HLL's and also because the System can then use the compiler to enforce adherence to conventions. Clearly if an arbitrary number of users in a large multiprogramming system wish to declare semaphores, the operating system will swell in size if it is demanded that semaphores be known a priori thereto; therefore we postulate that semaphores should operate in such a fashion that they need not be known before-hand to the System, and need only be of concern to the System when a task blocks or unblocks.

In summary, it is desirable that both system and user programs be able to spawn and coordinate parallel processes in a simple, clear, and flexible manner.

1.3.5 Microprogramming

This section deals with microprogramming as it impacts systems (and user) programming i. e. microcode as a resource, and not microprogramming as an implementation technology. We include it, besides for

its own merits, to point out that a System must be forward looking relative to new and unique demands for allocation and control; the discussion also illustrates that often times what may seem new is really an old problem in disguise : in this case multiprogramming and block structure.

When the IBM Corporation announced System/360, it also provided emulators for prior systems (1400 and 7000 series machines). Although these systems were (usually) implemented 'on the 360' via microprogramming, there was no capability provided to multiprogram these emulators i. e. if the 7090 emulator was in use, this precluded the use of the hardware as e. g. a 360, with consequent disruption of job stream scheduling. System/370 attempts to remedy this problem by allowing emulators to be multiprogrammed within certain limitations, but in essence, when a given emulator is in control, the entire system acts as if it were e. g. a 1401. Upon occurrence of an interrupt or when control is transferred to the 370 machine, the 1401 machine logically disappears and the system is a 370. The barrier between the 370 and the other emulators is arranged to preclude any real communication between them.

At the present time, IBM and other vendors prohibit the user from tampering with or adding to the microcode. This is done for reasons of propriety and maintenance, but pressure from users of smaller computers which encourage experimentation with microprogramming will eventually force the large manufacturers to allow the use of microprogramming as a resource, and eventually all operating systems will be expected to allow multiprogramming of emulators. Unfortunately, the emulators which users are likely to construct will be for machines which have hitherto not existed. The following problem now arises : a computer is of no use without software-compilers, I/O drivers, loader, storage and file management disciplines, etc. Whereas the IBM emulators execute problem program cum operating system within the emulator partition (thus supplying the problem program with a ready made software set), new emulators will not have such a ready made system to greet them. The choice now facing the owner of a new emulator is between the nontrivial difficulties and expense of constructing a new software complex or finding a way to bridge the gap between his emulator and an existing system.

At this point it is useful to distinguish between two types of microcode

a user might write: simple extensions/replacements to the existing instruction set, or a complete subsystem (although the latter can be viewed as an extreme case of the former). The destinction is useful because it enables one to draw an analogy with extensible languages.

The discussion on Application Programming stressed the gap between the user's problem and the language in which it must be stated. Extensible languages are seen in many quarters as the bridge across this gap. In general, such languages allow new operators to be defined upon entry to a block which become undefined upon exit from the block. Such extensions constitute an execution environment which is elaborated by cascading the extended statement into calls on procedures in outer blocks. In figure 1.3.5-1, the 'less than' operator can be looked upon as such a procedure, to say nothing of the routine which coerces integers to reals.

Clearly the example of the figure represents a trivial case of the type of extension which can be specified, and more extensive definitions would result in a more deeply nested sequence of procedure elaborations. Thus the user of such language extension features must pay for his 'convenience' in increased execution time, which is to say: emulation of a computer which simulates the desired execution environment. If however the extensions were implemented using microcoded subroutines, then the user could have the best of both worlds.

Unfortunately, the user with his microcoded language extensions still expects to be able to multiprogram jobs written in his augmented language, with perhaps different jobs employing different microprograms. Hence the liberal support of a microcode-extended language is tantamount to multiprogrammed emulation.

We have now gone full circle on the subject of multiprogrammed emulation in the sense that we have argued that a demand for multiprogrammed emulation arises out of the seemingly disparate areas of extensible languages and emulation of new computers. However the digression has paid a dividend if we notice that such language extensions, whether accomplished through procedures or microcode, always operate on a block basis. On the other hand, emulators are also implemented using micro-

```
begin
        op min = (real a, b)  real: (a<b|a|b);
        :
        :
        :.....statements using min of two reals.......
        :
        :
        :
end
        ¢ above min operation now no longer exists ¢
begin
        op min = ([1:] real a1) real :
                 (real x := max real;
                 for i to upb a1 do (a1[i]<x|x:=a1[i])
                 x)

        :
        :
        :
        :.....statements using min of an entire vector.......
        :
        :
        :
end
        ¢ examples from [L1] ¢
```

Figure 1.3.5-1

Algol 68 Program Skeleton Illustrating the Declaration of Operators within Block Structure Scope Definition Regulations.

code. Language extensions operate on a block basis, so should the computer emulator. If we therefore complete the analogy and state that computer emulators should also be viewed as blocks, we have the key to multiprogrammed emulation, since multiple tasks can be thought of as parallel blocks.

The next chapter delves more deeply into the relationship between block structure and multiprogramming and it will be shown that a computer architecture that supports block structure (and therefore multiprogramming) well would be amenable to the inclusion of the ability to support multiprogrammed emulation. See [3.11].

1.4 Summary of Contemporary Computer Usage

The preceding sections have dealt with the contemporary problems of application and systems programming. With regard to the applications programmer, we saw

(1) most programming is done in HLL's,
(2) these HLL's do not correspond well either to the user's problem or the hardware on which they execute,
(3) this lack of correspondence is a breeding ground for programming errors and bugs.

With regard to the systems programmer we found

(1) a need to program in HLL's which is frustrated by the unsuitability of the hardware,
(2) system software is very complicated, yet it seems that much of this complication is due to the unsuitability of the hardware,
(3) this complication causes system software to be slow to stabilize, bulky, and inefficient,
(4) even operating system primitives such as storage protection and semaphore processing are only minimally supported by the hardware,
(5) there is a need to be able to easily write reentrant code,
(6) there is a need to support user-program subtasking,
(7) there will be a need to support multiprogramming of user microcode under existing software.

In conclusion we can state that in general, programs seem overly difficult to write and debug (particularly system software), and that ultimately the blame falls on an environment which is too much software and not enough hardware.

The remainder of this paper describes and evaluates a computer architecture which attacks the problems heretofore described by providing the supportive environment which has been lacking. The design is not perfect, but experience in writing a compiler and operating system for this machine indicates that it <u>is</u> eminently and efficiently programmable in a higher level language. That in itself is progress.

2. 0 DESIGN CRITERIA FOR A PROGRAMMABLE COMPUTER

This chapter presents design criteria which derive from attacking the software problems described in Chapter 1. In general, we are looking for a computer architecture which is a good host for divers higher level languages and system software. The latter therefore implies that our computer must be a good host for multiprogramming, subtasking, time sharing, and new developments such as user microcode. Although PL/I has demonstrated the problems of trying to be all things to all users, we shall demonstrate that the environment requirements of [1.0] can be accomplished while the overall structure remains fundamentally simple.

Our first topic is block structure, as well it should be, since the majority of extant HLL's are block structured. Clearly an architecture which is a comfortable host for HLL's would be of great value to applications programmers, but it would be unfortunate if such an architecture were unsuitable for the systems programmer.

2. 1 Block Structure and Systems Programming.

This section explores the relationship between block structure and systems programming (or rather the logical structure of systems). Let us begin by stating some of the characteristics of block structure:

(1) the nesting of blocks yields a tree structure,
(2) data and code entities are known only to subsidiary portions of the tree,
(3) dynamic allocation of storage is inherent,
(4) recursion is inherent (given the existence of procedures).

The analogous characteristics of a running system are:

(1) the nesting of tasks and subtasks yields a tree structure,
(2) data and code entities are known only to subsidiary portions of the tree,
(3) dynamic allocation of storage (to tasks) is inherent,
(4) recursion is inherent (given the existence of interrupts).

One can see that the analogy is very close, as Figure 2. 1-1 illustrates. We now consider more carefully the application of block structure con-

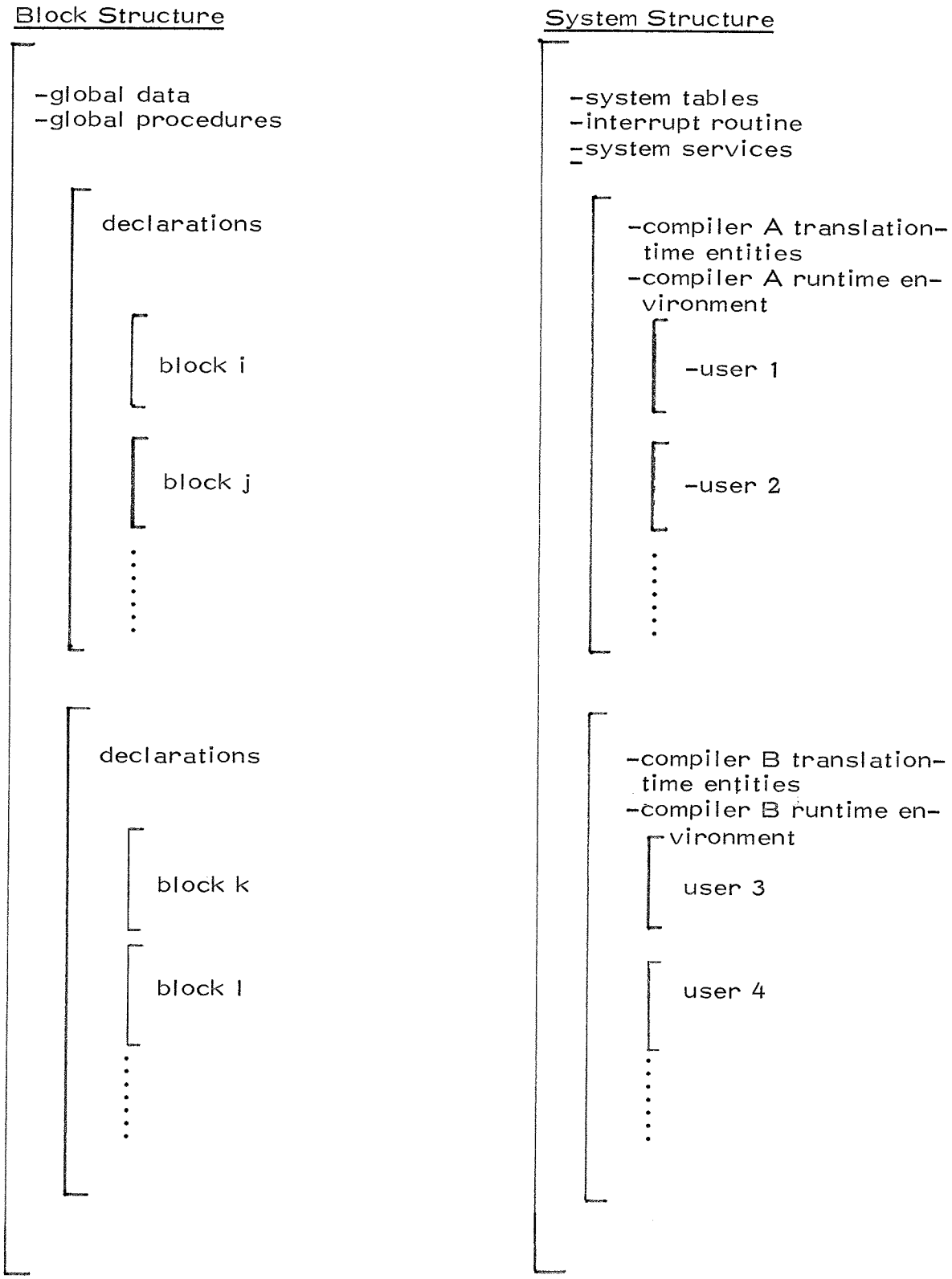


Figure 2.1-1

Illustration of the Similarities

between Block Structure and System Structure.

cepts to system structure.

2.1.1 Interrupts.

As shown in the figure, the operating system is global to all other entities in the system, and therefore so is the interrupt processibg procedure. Thus processing an interrupt can be regarded as a (block structured) entry to a global procedure, with the interrupt type, etc. as parameters to this procedure. Nested interrupts merely cause the interrupt routine to be entered recursively. A subsidiary dividend is that subtasks can directly invoke the interrupt procedure, whether to signal 'soft' interrupts or external interrupts (e. g. for testing purposes).

2.1.2 Multi-tasking.

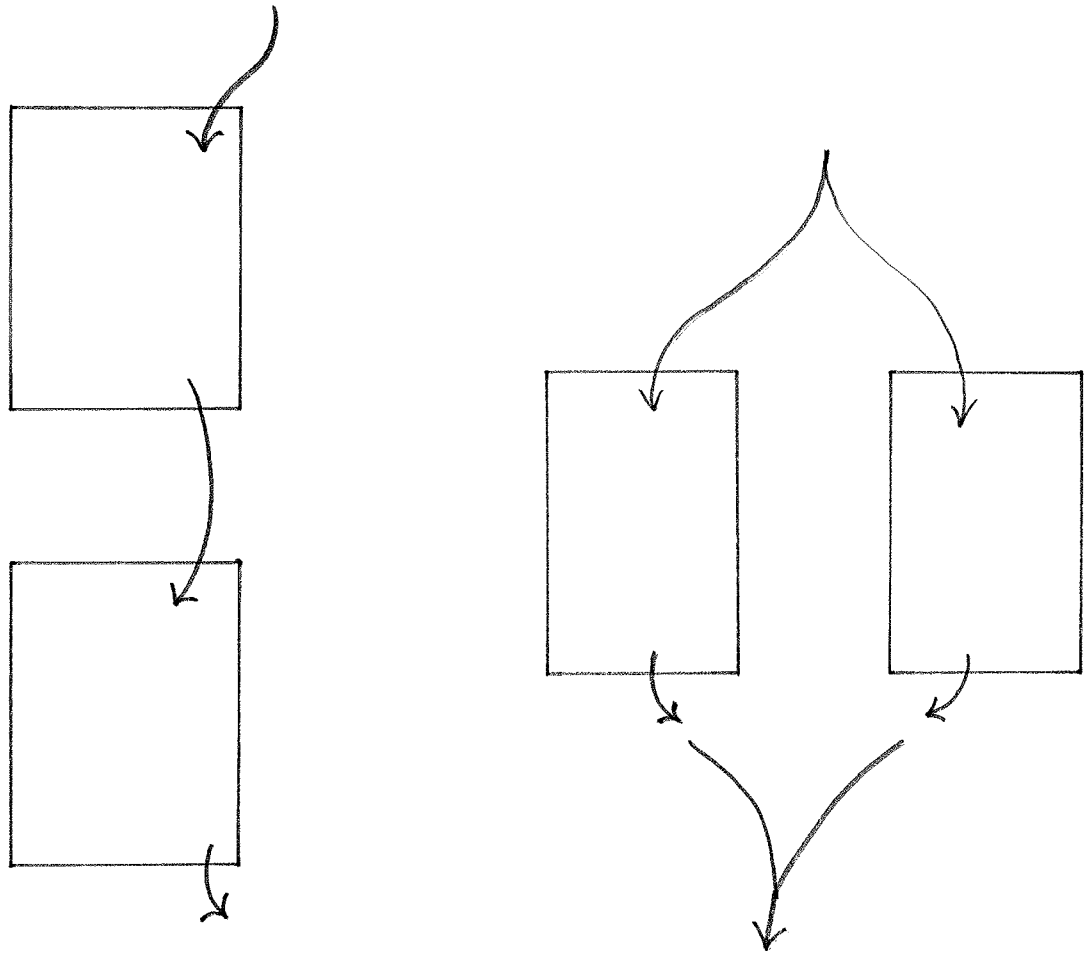
Consider a program in which two blocks exist on the same lexical level. Normally the two blocks would be executed in serial fashion - first one, then the other. If however it made no difference in which order they were executed, then they could in fact be executed in parallel. See Figure 2.1-2. The ability to execute in parallel is not restricted to blocks which have no common data; however, the sharing of data between parallel processes requires some mechanism such as semaphores to prevent ambiguity. Looking now at our two blocks executing in parallel (sharing or not sharing data), they have the same characteristics as two parallel tasks. Thus it would seem that an architecture which supports blocks ought to be able to support tasks with minor extensions.

2.1.3 Storage Protection

The scope rules of block structure provide exactly a type of logical storage protection that is lacking in modern systems: if we accept the analogy between parallel blocks and parallel tasks, then disjoint tasks (jobs) cannot 'see' each other, subtasks can see their parent but not each other, and all can see their global environment, i. e. the runtime environment and the operating system. Furthermore, since only the procedure head (and not its interior) is 'visible', it is impossible to enter any procedure (especially crucial system routines) except at the designated entry point.

2.1.4 Conclusions on Block Structure

The close analogy between block structure and system structure; the nat-



The arrows denote the path of execution.

Figure 2. 1-2
Two Independent Blocks (a)
Can also Execute in Parallel (b).

ural application to interrupts, tasking, and storage protection; coupled with the ubiquity of block structured languages; altogether comprise a compelling argument for hardware which conforms closely with the requirements of block structure.

2. 2 Data Representation.

The basic data which programs manipulate are numbers, characters, and bits; furthermore these entities are often processed in groups which are commonly referred to as arrays, character strings, and bit strings. The more closely the environmental support of these latter entities resembles the source language semantics, the easier it is for the compiler to generate good code and the more likely that the environment can detect a violation of the source semantics. A particular case is array/string bounds violations: automatic hardware recognition of this condition not only aids during the formal debugging phase, but also recognizes that a program is never really ever completely bug-free.

A fourth type of data is addresses or pointers. Until recently, pointers were not supported by HLL's, and this lack only enhanced their unsuitability for systems programming. It is therefore helpful to both applications and systems programmers to include pointers in the HLL's, and given that a pointer appears to be merely an address, one would think that hardware support of this construct would be easy.

The first inkling that such is not the case appears when one realizes that the mapping of data onto the address space (1) is unknown to the user, and (2) often does not correspond to the lexical ordering of entities in the source program i. e. add one to an address, and you may not get a pointer to the expected variable.

Worse yet, there is great impact on the System if user programs can retain absolute addresses since this precludes the System from moving programs around in memory. Therefore if we want pointers in our source language, it is imperative that the environment support them in a relocatible form.

We must also note that pointers have the characteristic of being used as

a pseudonym for either a single datum or an entire set of data. Either the compiler (via its generated code) or the hardware must be cognizant of this distinction, since data arrays require indexing and simple data do not.

We have thus far discussed how the environment ought to support numbers, characters, bits, arrays, character and bit strings, and pointers. What about linked lists, associative arrays, array crossections, etc. ? At this point we must consider the tradeoff between supporting them directly in hardware and forcing the compiler to generate code which implements them in terms of the existing structures. Clearly neither alternative is particularly desirable. Forinstance, besides the fact that a hardware implementation would be rather complicated, there are the two remaining problems of (1) choosing a particular set of semantics from the many possibilities (e. g. singly or doubly linked lists?), and (2) melding these constructs into the existing environment so as to retain consistency and generality (e. g. linked code ?).

In addition to choosing between compiler and environment support for such data structures, we can beg the issue by doing neither but providing the ability to extend the environment to allow later inclusion. The discussion in [1.3.5] of user microprogramming bears on this topic, and [3.11] discusses the required environmental 'hooks' for convenient microcode extension.

2.3 Evaluation of Expressions.

The sequential nature of (contemporary) computers dictates that the parenthesized manner in which humans write arithmetic expressions be linearized. This linearization is performed by the syntactic parsing of the compiler to produce an intermediate form called polish notation [G1] (although in many compilers, code is generated with the polish string's existing only implicitly). At this point the compilers' treatments (of the polish string) diverge due to differences in the target hardware.

If the target machine contains multiple general purpose registers, then the compiler must keep track of which registers are in use and deal with the movement of temporary results between the registers and memory. This inefficiency at compile time is (hopefully) recovered during exe-

cution since movement of temporaries between the registers and memory occurs only when necessary. If however the target machine is capable of executing the (reverse or suffix) polish string directly (KDF 9, B5500/6500 computers), then the compilation speed increases and execution speed decreases according to the number of temporaries and the number of top-of-stack registers on the machine [W6].

The most commonly heard arguments against direct execution of polish strings revolve around this very point of memory accesses. We now examine them.

Argument #1

Consider the expression "A: = B+C;". The reverse polish coding of this expression if "aAvBvC+:=" where 'a' denotes 'address of', 'v' denotes 'value of', and ':=' is the assignment operator. A machine which executes strict reverse polish would require the following code:

-load the address of A
-load the value of B
-load the value of C
-add the top two items in the stack
-store the topmost item in the stack into the location specified by the second item in the stack.

[Each hyphenated line above represents one instruction with appropriate stack maintenance automatically performed by the hardware.]

The argument says that on a register machine, the STORE instruction contains the target address, and therefore the loading of the address of A onto the stack and subsequent (automatic)maintenance is pure overhead. The origin of the argument is probably the notion that operators (e. g. :=) must be zero address instructions, and therefore the address must be preloaded onto the stack.

The refutation of the argument is to postulate that the STORE operation shall be a single address instruction, just like the LOAD instruction. The object code now becomes

-load the value of B

-load the value of C
-add the top two items in the stack
-store the top-of-stack item into A

In terms of the actual instructions generated, this is no different code from a register machine (although, depending on the state of the top-of-stack registers, there may be an extra stack manipulation).

Argument #2

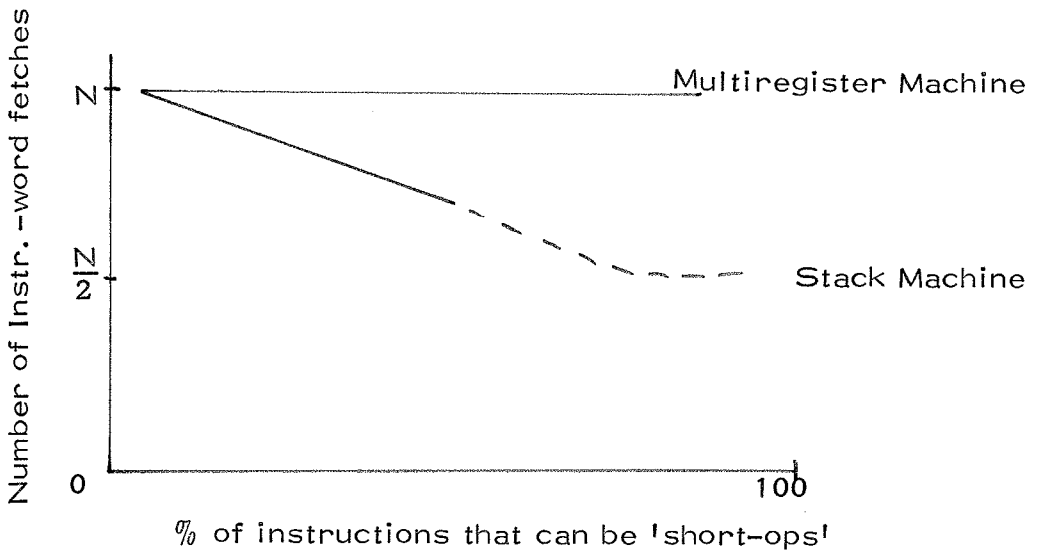
"But wait." say the opponents of stack evaluation. "If the expression is complicated, then there must be movement of temporaries between the top-of-stack registers (of which there are usually 1-3) and main store. Even though the hardware does this automatically, it still takes time to read/write memory."

The argument is valid so far as it goes, but it neglects to consider that the movements can be overlapped (i. e. done at the same time as) the fetch and decode of the next instruction (given memory interleave and/or multiple instructions per word). Moreover, if the next instruction is a single (instead of zero) address operation, there is even more time since an effective address must be calculated and possibly several memory accesses made.

Affirmative Point # 1

There is an argument in favor of 'polish hardware' which is often overlooked. Since there are a number of registers which can hold operands, an instruction for a multiregister machine must contain therein one (or several) register address(es), thereby increasing the length of the instruction (in bits). In contrast, there is no need for register addresses in a stack machine instruction since the source/destination of all operands is known by the hardware to be the top-of-stack; therefore stack machine instructions tend to be shorter than their multiregister counterparts. Since the instructions are shorter, more instructions can be packed into a machine word, thereby decreasing the number of instruction fetch cycles which must access memory (see Figure 2. 3-1).

Figure 2. 3-2 from [H4] (and see also [W3]) demonstrates that indeed code for stack machines is more compact.



The graph illustrates the fact that a multiregister machine must always specify the source and destination addresses for operands. The short operations of a stack machine [e.g. +, -, ∧] on the other hand require no such addresses since source and destination are known to be the stack. The multiregister machine is unable to take advantage of the potential shortening, and consequently cannot achieve the code density of the stack machine.

Notes:

1. Long instructions are memory reference or register-instructions, while short instructions are zero-address stack-oriented operations.
2. The graph assumes
   (a) long instructions for both machines are the same length
   (b) short instructions are half as long as long instructions.
3. The reader should be aware that the figure oversimplifies considerably from the discussion in the text.
4. N/2 is an asymptote, since long instructions must be used to load operands onto the stack.

Figure 2.3-1
Number of Instruction-Word
Fetches on Multi-register and Stack Machines.

Machine Types:

MRR3: arith metic operations only between registers.

MRS2: arithimetic operations either between registers or between a register and memory.

STCREG: multiple general registers plus a stack.

STCMOD: stack machine with ability to address non-Top-of-Stack elements, relative to the top.

STC: stack machine which can only address Top-of Stack element.

OAM: classic one-address, one-accumulator machine.

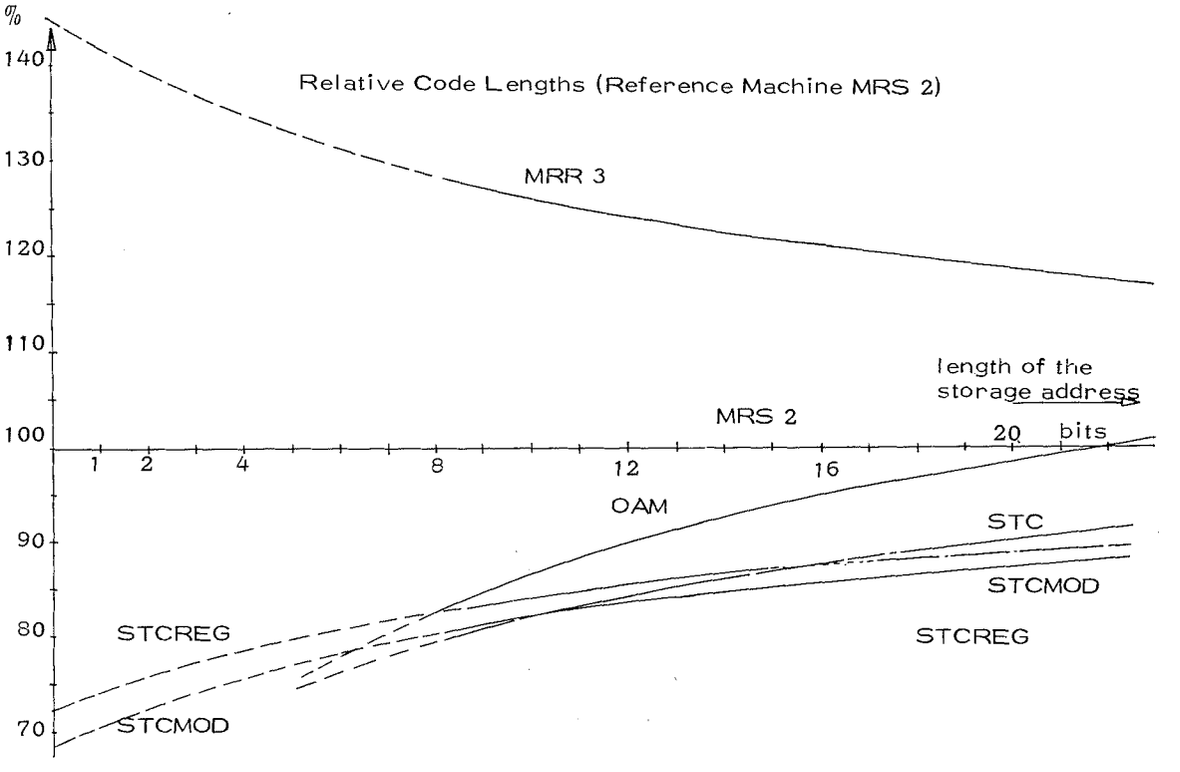


Figure 2.3-2

Relative Code Compactness of Algol-like Expressions and Assignments for Stack and Multiregister Computers. (from [H4]).

It must be pointed out that multiregister machines coupled with multiple parallel arithmetic units (e. g. CDC 6600) do an extremely effective job in evaluating expressions, whereas the means of accomplishing this on a stack machine is not apparent. The only counter-arguments are the relatively small number of such applications, the expense of such hardware, and that compilers which can take advantage of such hardware are both difficult to write and slow (e. g. CDC 6600 FTN compiler).

Affirmative Point # 2

Because all operands, including parameters and return addresses, can be stored in the stack, object code for a stack machine is inherently re-entrant. A stack is also the usual means of accomplishing the dynamic allocation of activation records [W2] required by e. g. Algol 60 and PL/I AUTOMATIC.

Affirmative Point # 3

Because the hardware knows where the stack is and how to store and retrieve items therefrom, it can use the stack to hold hardware generated items e. g. return addresses. This is in contrast to the typical register machine wherein the hardware cannot arbitrarily decide to use some of the registers for its own purposes because it doesn't know which ones (if any) are unused. Even if some registers were designated as belonging to the hardware , the many possible recursive situations that exist (e. g. interrupts) would cause the hardware to attempt to reuse its registers when they were already full.

We shall also see in [2.6] that the ability of the hardware to use temporary storage has great impact on the deferral of binding time.

Conclusion

The two arguments against direct hardware execution of polish strings are refuted above. In favor we have the ease of compilation, code compactness and reentrancy, and applicability to dynamic storage. Thus the decision would seem to favor a stack type architecture from the point of view of expression evaluation. See [I2], [M1], [B6] and[W6] for additional discussion.

2.4 Procedures and Parameters

Procedures (or subroutines) constitute a doubly potent programming tool. In the first place they represent a means of structuring a program so that it can be comprehended, written, and debugged. [Incidentally, one wonders if an equally convenient tasking facility would not have the same benefits!] Secondly, procedures imply object code compactness by allowing a body of code to be executed 'out of line' rather than being repeated in-line at each needpoint.

We now postulate that the environment should support reentrant and recursive procedures unless this causes undue inefficiency in the resultant object code. This seems reasonable in view of the recursive nature of the operating system, compilers, and artificial intelligence applications such as pattern matching, problem solving, game playing, and list processing in general.

The fact that procedures are out-of-line code means that they require parameters to particularize them for each invocation. The problem that now arises is how changes to the parameters inside the procedure body should reflect back to the point of invocation. Three schemes are in use: call by value, call by reference, and call by name. The first allows no changes to be reflected back to the caller; the second that changes will be reflected, but if the actual parameter is an arithmetic expression, then the expression is called by value; the third that changes will be reflected, and if the actual parameter is an arithmetic expression, references to the value of the parameter will(each time) cause the expression to be reevaluated in the environment of the caller. This last requirement (underscored) turns out to have some rather subtle implications, the resolution of which has had great (deleterious) impact on object code efficiency; it was for this reason that PL/1 replaced call by name with call by reference.

The prevailing attitude toward call by name is that while it has its (really nice) uses [K2], [N1], for the most part the inefficiencies outweigh the benefits. Certainly this attitude had much to do with the PL/1 decision to call by reference. By far the thorniest problems with call by name occur when a procedure is passed by name, the problem being that the (block structure) environment at the time of the call must be saved and used with

the procedure whenever it is invoked. However, besides the sort of 'tricky neat' uses of procedure call by name, there is one extremely useful one, demonstrated by the so called 'ON conditions' of PL/I.

The idea behind ON-conditions is to allow the programmer the ability to specify certain processing when and if certain exception conditions occur. Call by name has a perfect use here since the user can pass his error procedures by name to the System (at which time the current environment is saved) which can then invoke them when the condition occurs. Since the procedure's original operating environment is applied, the user gets exactly the results he wants, regardless of how far afield he has strayed from the point at which the ON-condition was established.

Since ON-conditions can also be associated with semaphore and other 'events' [B4] the aggregate of usage possibilities for call-by-name argues for its support by the environment.

2. 5 Compiler Considerations.

This section deals with a rather nebulous concept - that since a compiler tends to be somewhat myopic relative to the piece of source code it is currently processing, the eventual environment architecture should strive to operate in such a fashion that the compiler needn't care wether a particular variable is a formal parameter, actual parameter, name or value referenced, fullword, halfword, byte, bit, loop index (even expression), or 'involved' with pointers. Admittedly this is a stupendous wish, yet the payoffs in both compiler size and speed and object time efficiency are equally large.

Clearly if the compiler doesn't worry about these matters, then the environment must. Thus the crux of the problem is that the environment must somehow be able to make the appropriate decision at runtime, which therefore implies that it must have a means figuring out what it is it is supposed to do i. e. there must be a means for the environment to distinguish between several possible courses of action.

Section [3. 3] discusses how this can be accomplished.

2.6 Binding Time.

The preceding section was actually talking about something called binding time. Thus, [2.5] was saying that the compiler's life would be considerably simplified if the time at which action (i.e. code) is bound to variables could be deferred to runtime. In fact, binding time has cropped of several times in the preceding pages e. g. [1.3.3]. Linkage/ Loading, [2.2] pointers, [2.3] Affirmative # 3, [2.4]ON-conditions.

One can trace the advances in computing by the increasing deferral of binding time e. g.

1st	generation	absolute code
2nd	generation	relative code and subroutine libraries
3rd	generation	relocation hardware, virtual memory, incremental compilation, languages like Snobol 4 and APL.

Wegner states [W1] p. 18 that "Binding attributes as early as possible sometimes results in more efficient program execution, since it saves repetition. However, binding attributes as late as possible allows the decision regarding the bound attributes to be delayed and thereby allows greater flexibility in specifying the attribute". Using the reasoning that efficiency of use for programmers is more important than super-efficient execution by the hardware, we postulate that our environment shall strive toward maximal deferral of binding time.

2.7 Input/Output.

Two types of operating system bugs which are exasperatingly difficult to track down are (1) a lost interrupt due to an unexpectedly long lockout of interrupts, and (2) an error in the parameters to an I/O channel which results in its writing beyond the buffer limits. Both of these bugs have the characteristic that by the time they result in a detectable error (usually unrelated to the real culprit), all trace of the cause is gone.

The solution to the lost interrupt problem is to make all interrupts non-time dependent, though this seems to be a contradiction. If this can be

accomplished, however, then we are in a much better position to make use of semaphores on the System (having resolved Wirth's dilemma in [W5] in which he can't seem to decide whether or not he really wants interrupts). The efficacy of using a semaphore approach is reported by Wirth [ibid] and Dijkstra [D3].

2.8 Conclusion

The design criteria established in the preceding paragraphs are

(1) the hardware should support block structure,

(2) the hardware should support the source language data types directly,

(3) expression evaluation should be accomplished by direct execution of reverse polish strings,

(4) call by name has valid uses and should be supported by the hardware,

(5) the hardware should allow maximal deferral of binding, implying it can 'fill in' the semantics that the compiler left unbound,

(6) I/O channels should be cognizant of the logical structures residing in the PC's memory,

(7) interrupts should be time independent insofar as is possible.

The next chapter presents the actual design of a computer which meets these criteria. Since these criteria were derived from considerations of software "programmability", the computer described in [3.0] is called the Programmable Computer (PC).

Preface to Section 3

This section presents the actual design of the PC in a narrative fashion which explains the motivation behind the various constructs that are considered primary or highlights. As a result, some of these are treated in detail while other points are only mentioned.

The knowledgeable reader may become frustrated at the ordering of topics or the apparent missing details. However, the material has been organized for expository clarity to the unitiated and hopefully all the loose ends will have been gathered together by the end.

3.0 THE DESIGN OF THE PROGRAMMABLE COMPUTER.

Let us summarize the problems outlined in [1.0] and distilled into design criteria in [2.0]. The Programmable Computer (PC) attempts to optimize throughput by the following considerations:

(1) ease of program creation via HLL's,

(2) ease of debugging,

(3) recognition of the relationship between block structure and multiprogramming,

(4) recognition of the Dijkstra coordination primitives both between cotasks and between the PC cpu and the I/O processes.

These considerations result in the following design objectives:

(1) The PC's instruction repertoire should be 'close' to the languages which execute on the PC, thus allowing easy compiler generation, efficient code, and execution time diagnostics relatable to the source program.

(2) The PC hardware should automatically oversee all program execution (including the System) and trap all questionable semantics. This implies the existence of primitive features which allow the hardware to distinguish among different data types, addresses, and code.

(3) Hardware support of the Dijkstra primitives implies an explicit hardware recognition of task structures i. e. block structure.

The resulting design can be summarized as a machine with a block structured address space, hardware supported dope vectors, hardware regulated pushdown stack, and timeindependent and PC-cognizant I/O channels. As it turns out, variable size page virtual memory is a (nearly) free dividend of the dope vectors, so this is also included in the design along with several special instructions to help maintain storage.

The following paragraphs discuss these features and related topics.

3. 1 Pushdown Stack.

The hardware maintained pushdown stack is discussed first because it is basic to the evaluation of expressions and the block structured nature of the machine as a whole.

A pushdown stack can be implemented most easily in hardware with the stack itself residing in main store (hereinafter referred to as 'memory'), and the controlling registers in a fast store. Each element of the stack is one machine word; the limits of the stack are maintained in two registers called BOS (base of stack) and LOS (limit of stack); the address of the word currently on the 'top' of the stack is contained in TOS (top of stack). Normally, TOS is in the range BOS $\leq$ TOS $\leq$ LOS; stack underflow occurs when TOS becomes less than BOS, and stack overflow when TOS exceeds LOS. When an item is to be pushed onto the stack, TOS is incremented and checked against LOS, and then the item placed in the memory location pointed at by TOS. An item is removed (popped) from the stack by decrementing TOS and checking against BOS. Figure 3. 1-1 illustrates.

A load-type instruction causes an item to be pushed onto the stack, while a store-type instruction causes an item to be popped off the stack and stored elsewhere in memory. Binary operations e. g. ADD, LESS THAN, cause the two topmost operands to be manipulated and the result placed on the stack i. e. as if the operation had performed POP, POP, PUSH.

The stack maintenance hardware assures that operands are always available to operations, and that TOS never violates the bounds of BOS and LOS.

Also associated with the stack are the B or display registers, which are used to maintain the block/task structure of the running program. These registers point into the stack at those points where the storage for particular blocks are located, and are numbered 0, 1, 2, ... ; the register CLVL denotes the maximal B[i] currently in use. Figure 3. 1-2 illustrates and the next paragraph explains their use further.

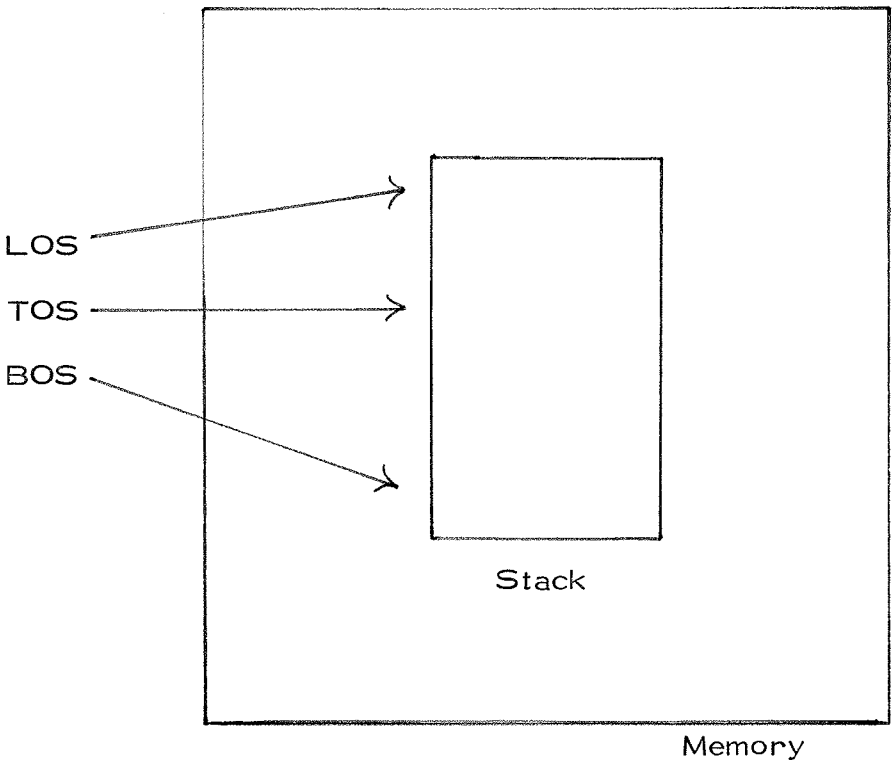


Figure 3.1-1
The Stack and Controlling Registers.

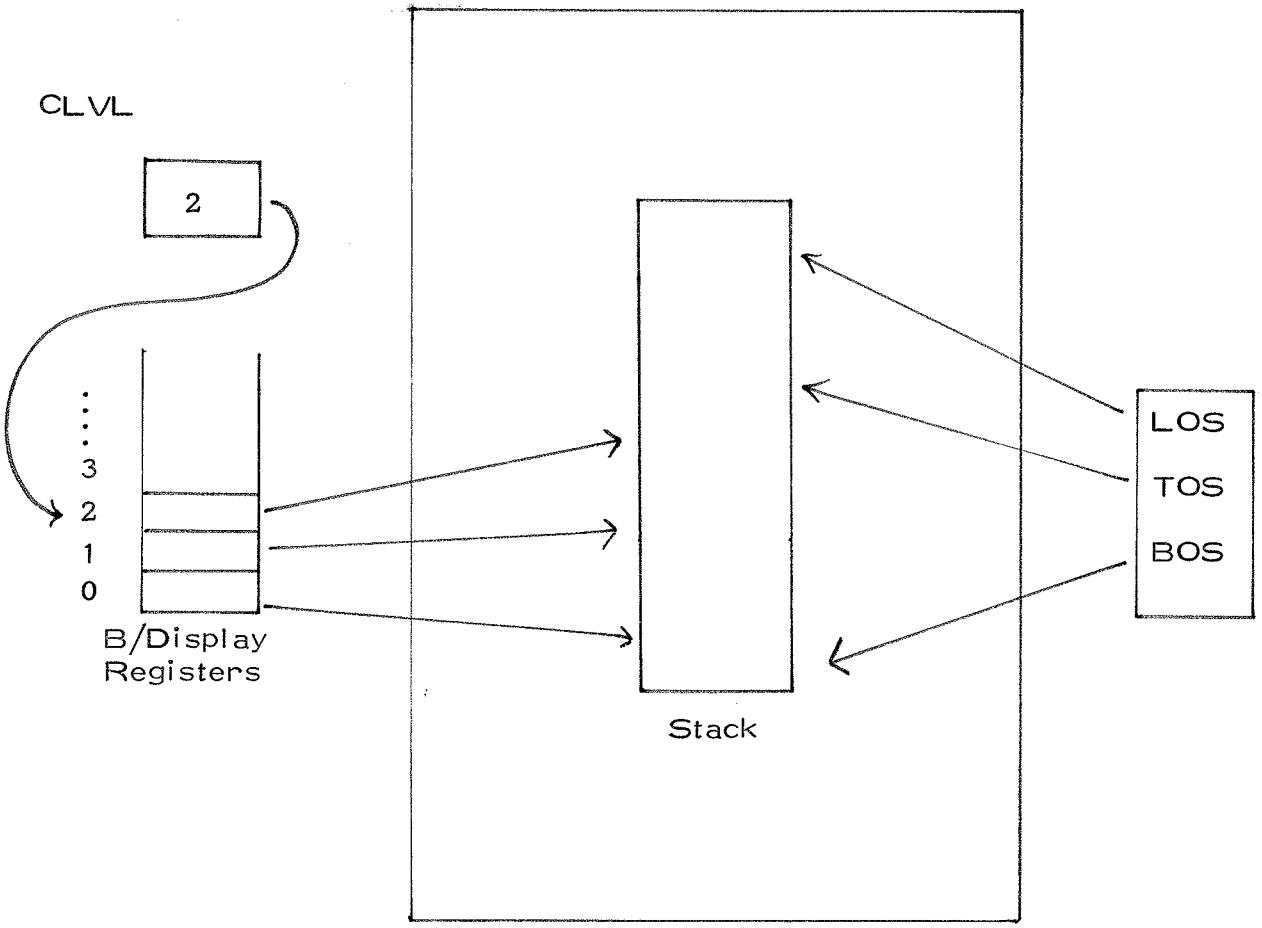


Figure 3.1-2
Stack with B/Display Registers

3.2 Block Structured Addressing.

Figure 3.2-1 shows a simple Algol 60 program. Note that the various block have been dilineated and their nesting level labeled. We can now label the variables according to this nesting level e.g. I is in the first (b=0) block level, and is the first (d=0) variable therein decalred. Thus I is denoted [0,0]. Similarly, J is denoted [0,1]; A is [0,2]; and P is [0,3]. The bracketed pairs [b,d] are called address couples [R1], and Figure 3.2-2 lists the address couples for the entire program. Observe that there are three I's in the program (I1, I6, I10), and that I6 and I10 both have the same denotation [1,1]. This lack of uniqueness poses no problems however if we remember that one cannot simultaneously 'be' in two block on the same level, according to the definition of Algol. Thus within the definition of block structure, our [b,d] notation provides us with a method of uniquely addressing all variables, and furthermore, this method of addressing has the property that it reflects the source program natation within the variable's address, a valuable debugging tool.

Another way to view address couples is as addresses in a tree. Figure 3.2-3 illustrates this point af view.

The problem now facing us is how to map our tree space addresses into the linear address space found in computer memories. The mechanism by which this is accomplished utilizes the B-registers mentioned in the preceding section. An address couple [b,d] is mapped into an absolute linear address 'a' via the B registers by the algorithm:

$$a := B[b] + d.$$

Hence, so long as the address in B [b] is the address in the stack where the variable of block (nesting level) 'b' begins, this mapping preserves the intent of the source program. We can observe that the B registers are really nothing more than index registers, and thus the price paid for having block structured addressing is that all memory references

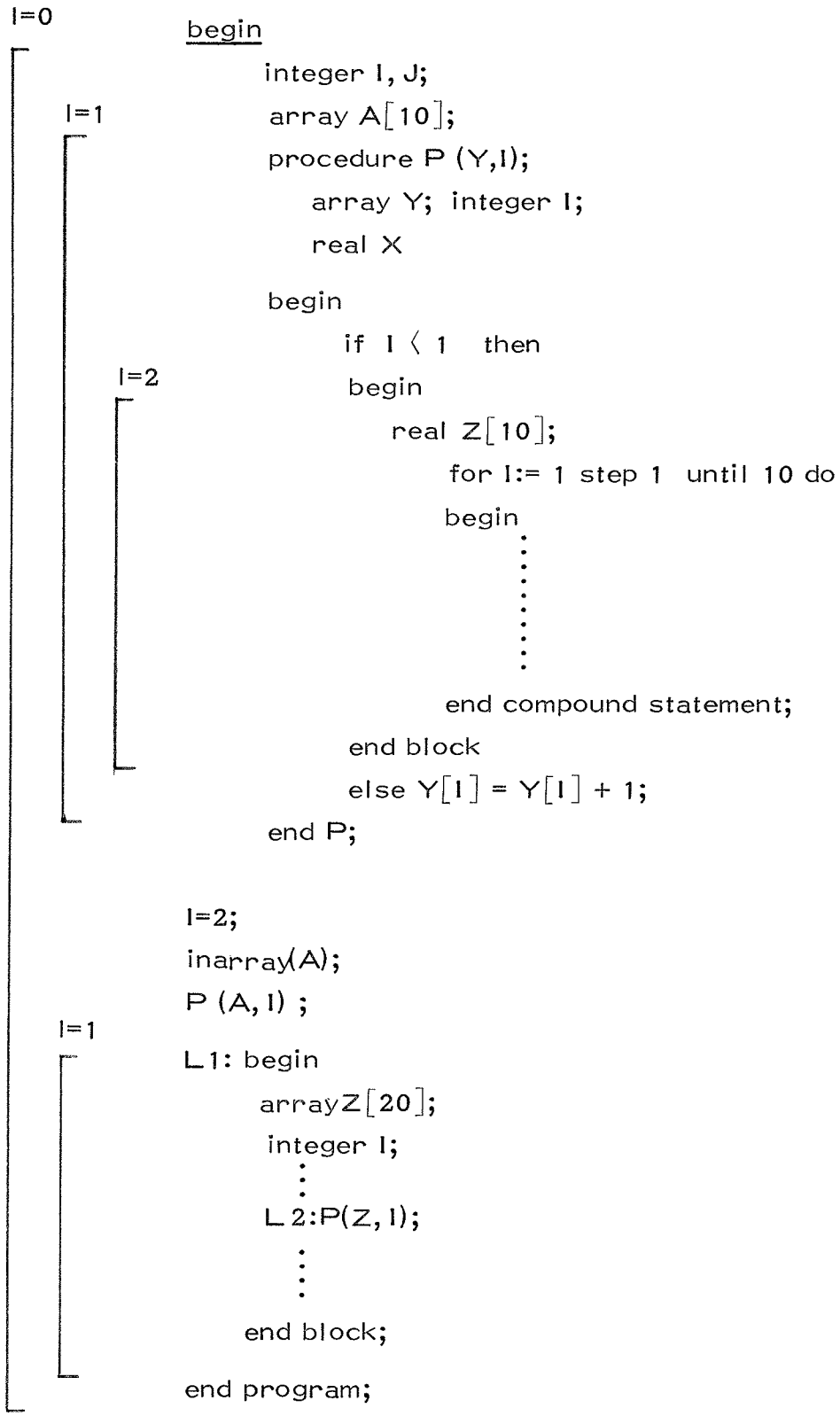


Figure 3. 2-1

Block Structure

of a Simple Algol 60 Program called SIMPLE.

	variable	block level	block displacement	address couple
1.	I	0	0	[0, 0]
2.	J	0	1	[0, 1]
3.	A	0	2	[0, 2]
4.	P	0	3	[0, 3]
5.	Y	1	0	[1, 0]
6.	I	1	1	[1, 1]
7.	X	1	2	[1, 2]
8.	Z	2	0	[2, 0]
9.	Z	1	0	[1, 0]
10.	I	1	1	[1, 1]

Figure 3.2-2
Address Couples for Program SIMPLE.

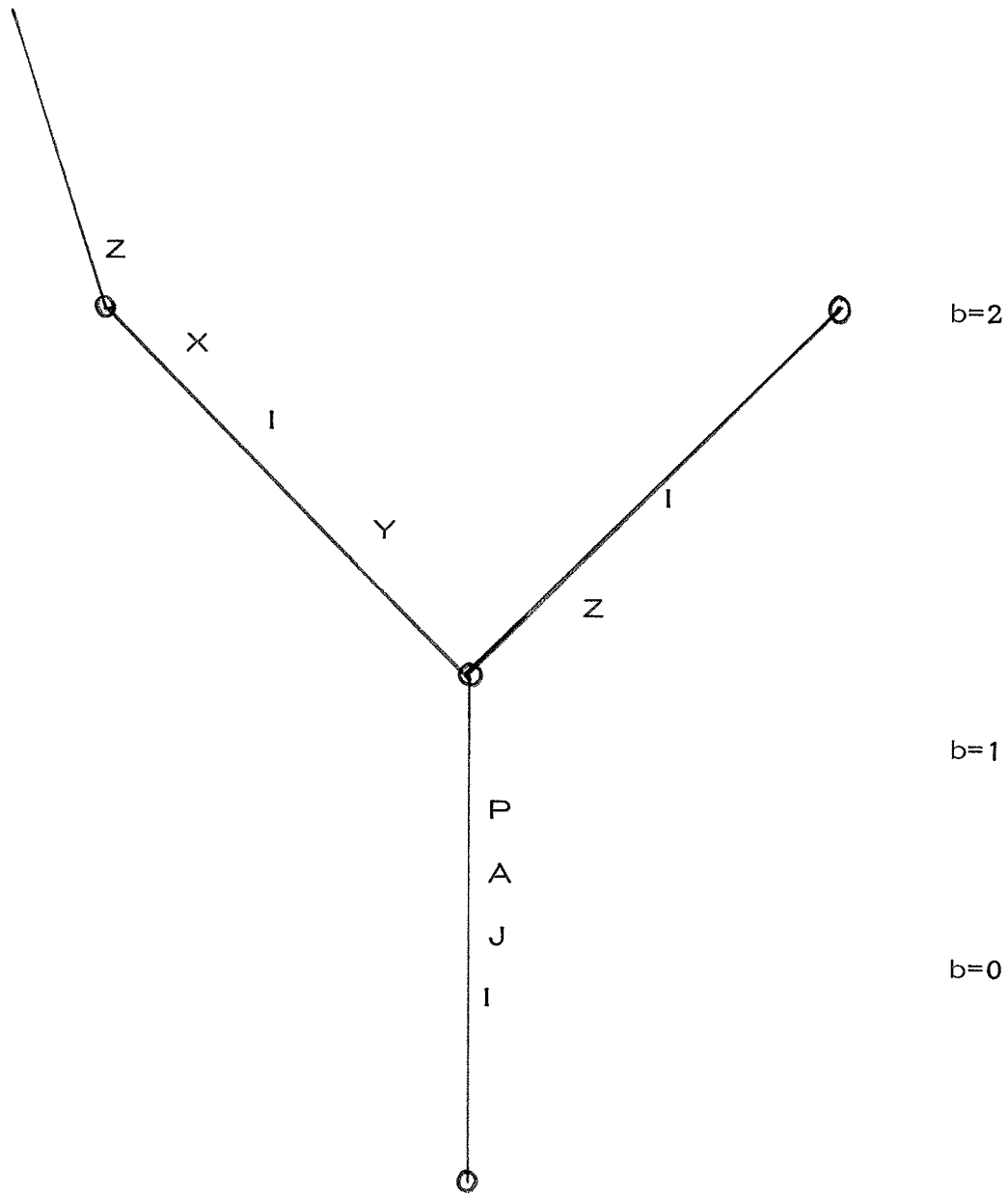


Figure 3.2-3
Address Couples as Tree Addresses (Program SIMPLE).

require indexing. Figure 3.2-4 illustrates how this addressing scheme is applied.

The next question which arises is how many B-registers should there be? At first glance one might think that an infinite number are necessary if recursion is allowed, but this is not true since a recursive procedure always executes at the block level at which it was declared (thus causing the same B[i] to be continuously updated). The only real requirement on the number of B's is whatever one considers to be a 'reasonable' maximum for block/task nesting. Clearly 8 is too few and 64 more than necessary, and therefore either 16 or 32 is probably a reasonable number.

Having decided on the number of B registers, we now must fix upon the maximum displacement (i.e. the maximum number of declarations at a given block level). It is necessary to choose a maximum because the [b,d] will be the address field of the PC's (single address) instructions. Employing logic similar to the above, 64 is probably too few, and 1024 too many. Thus we will choose among 128, 256, and 512. This decision has distinct impact on the System since the outermost block contains the declarations of all the system library routines and utilities. In effect, too small a d-field restricts the total number of 'externals' that can exist. There is, however, another approach to these field calculations, described in [W3, W4].

Two primitive operations are required to maintain the B registers: enter block (NTRB), and exit block (XITB). Figure 3.2-5 gives the function which each must perform. The attentive reader will note that upon entering a block, the contents of the associated B-register are pushed onto the stack. This is done, though it may at first seem unnecessary, because execution may be retracing the tree structure as the result of an outer block procedure call, and the hardware must save the register 'just in case'. [It is also crutial to realize that in the final machine, the contents of the B's cannot simply be pushed onto the stack (as shown in Figure 3.2-5 ab) because they contain absolute addresses, the bane of relocation. Hence they are converted to a relocatible form called [s,d] (explained later) and then pushed.] XITB undoes the ef-

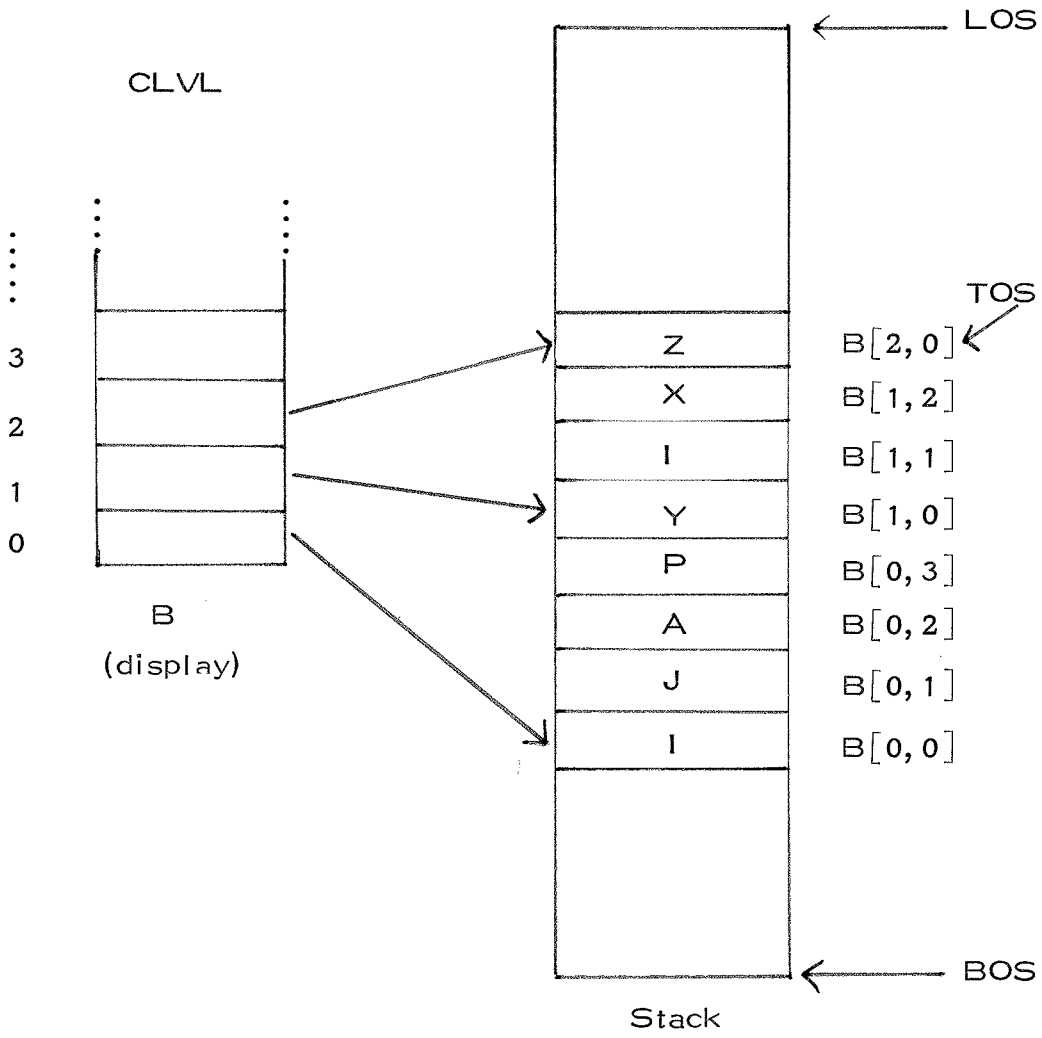


Figure 3.2-4

Execution Snapshot of Program SIMPLE.

```
procedure enter;
begin
     comment Interchange code page logical address and stack[f-1];

     old f:=get field (stack[f], full word);
     unpack 2 (stack[f-1], recent mscw, displacement);
     pack return control word(stack [f-1], cpra, cplam, cplad);
     comment see comment on cpra, cplam and cplad in exit;
     cplam:=recent mscw; cplad:=displacement; cpra:=0;

     comment follow pointers to code reference word and unpack;

     address:=bottom of stack+recent mscw+displacement;
     follow pointers (address, recent mscw);
     unpack 2 (stack[address], current lexical level, displacement);

     comment unpack codedescriptor;

     address:=bottom of stack+displacement;
     unpack 2(stack[address], pageplace, pagelength);

     comment build new mscw and update f;

     pack mscw(stack[f], upper mscw, current lexical level,recent mscw);
     upper mscw:=f; f:=old f;

     comment update display;
     lexlvl :=current lexical level;
     display[lexlvl ] :=next mscw:=upper mscw;
     lexlvl:=lexlvl-1;
     for next mscw:=get field (stack[next mscw], previous mscw)
     while display[lexlvl] ≠ next mscw do
     begin
          display[lexlvl ]:=next mscw; lexlvl:=lexlvl-1
     end while
end procedure enter;
```

Figure 3. 2-5a

Enter Block/Procedure Action.

```
procedure exit;
begin
        top of stack:=upper mscw-2;

        comment unpackreturn control word;
        unpack 3(stack[upper mscw-1], cpra, cplam, cplad);
        comment where cpra is locationcounter inside codepage, and
        cplam and cplad are address of a mscw and displacement such
        that cplam+cplad is address (relative to bottom of stack) of
        a pointerchain to code reference word;

        comment follow pointers to code reference word and unpack;
        address:=bottom of stack+cplam+cplad;
        follow pointers (address, recent mscw);

        comment unpack codedescriptor pointed at;
        address:=get field (stack[address], relative cdsc address)+bottom
                                                              of stack;
        unpack 2 (stack[address], pageplace, pagelength);

        comment update upper mscw and display;
        upper mscw:=get field (stack[upper mscw], previous mscw);

        lexlvl:=old lexical level:=current lexical level;
        current lexical level:=get field (stack[upper mscw], lexical level);

        display [lexlvl]:=next mscw:=upper mscw;
        lexlvl:=lexlvl-1;
        for next mscw:=get field (stack[next mscw], previous mscw)
        while lexlvl > old lexical level ∨ display[lexlvl] ≠ next mscw do
        begin
              display [lexlvl]:=next mscw; lexlvl:=lexlvl-1
        end while
end procedure exit;
```

Figure 3.2-5b
Exit Block/Procedure Action.

Explanation to preceding algoltext:

unpack <i>(parameter 0, parameter 1,....., parameter <i>);
unpacks parameter 0 into the following parameters.
Note that the tagfield of parameter 0 defines the packing.

get field (p1, p2);
A field described by the mask p2 is extracted from p1 and rightshifted.

follow pointers (address, recent mscw);
Selfdescribibg. On call address is the address of a pointer-chain. On return address is the address of the last object in the chain, and recent mscw is the address of the last mscw encountered before the end of the chain.

fects of NTRB. Rather than explore the gory details of block entry and exit here, the reader is refereed to [S1, O1].

3.3 Memory Structure.

The discussion in Chapter 2 relative to binding time states that the hardware must be able to distinguish among different data types, addresses, and code. The mechanism which the PC uses to accomplish this is called 'tag bits' [H2]: several bits appended to each word of memory which denote or tag the contents of the word. The hardware on which the PC is implemented has 18-bit wide memory, which given the word size for the PC as 32 bits, leaves 4 bits for use as tags.

Four tag bits per 36-bit word means 11% of memory is devoted to tag storage, which might seem somewhat excessive. Two rationalizations are given:

(1) Since a 32-bit word size was chosen rather than violate the US de facto standard, and because 8-bit characters are also a forward looking stand, the extra four bits are either wasted, or must be gathered together eight at a time to form extra words of memory at a ridiculous processing cost.

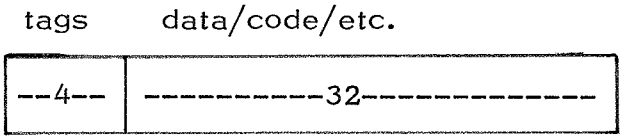
(2) The original PC design postulated only two tag bits, which actually were quite sufficient, but allowed no room for expansion, hence the decision to go to four, especially since four were available.

A much more global rationalization for the apparent luxury of tag bits is represented by the arguments of the first two chapters. Besides allowing deferral of binding time, tags can tell the hardware that what it's been asked to do is definitely illegal. Further support is given by [F1].

The tag bit designations for the PC are shown in Figure 3.3-1.

3.4 Descriptors, Pointers, and Indexing.

Tag type two (TT2) entities are those which (with the exception of



(a) a word of the PC memory

Tag Value	Hardware Interpretation
0	data: word, byte, bit
1	data: nonstandard e. g. multiprecision
2	non data: descriptors, pointers
3	memory links, hardware created entities, object code
4	continuation i. e. this word is a continuation of another
5	semaphore-either ptr, code descriptor, or buffer descriptor
6	emulator storage descriptor
7-15	unallocated

Figure 3. 3-1
Tag Bits in the Programmable Computer.

code descriptors) contain information about data, but which themselves are not. TT2 words fall into two categories: descriptors, and relocatible addresses.

3.4.1 Descriptors.

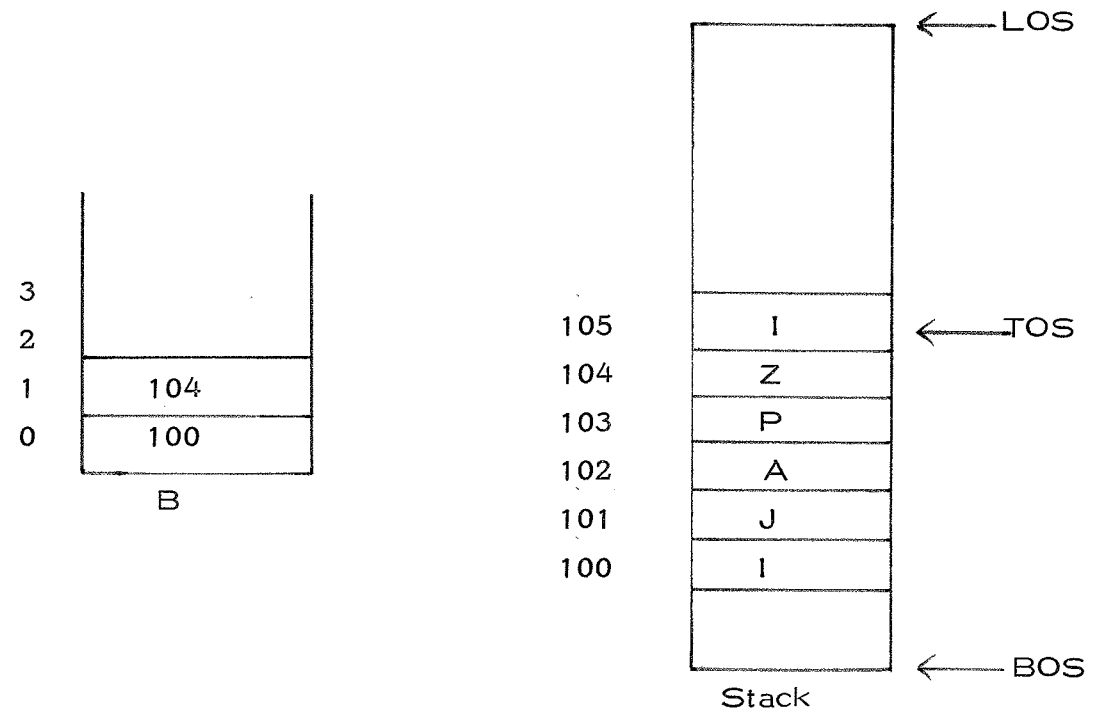
A descriptor is another name for 'hardware supported dope vector', and describes an area memory outside the stack. This description includes the absolute address of the area, and the type and number of data items in the area. Aside from the PC's registers (B-registers etc.), descriptors are the only PC constructs which contain absolute addresses. The type field denotes the type of data to be bit, byte, halfword, or word data. The number of items is held in the length field which is checked by the hardware to assure that the bounds are not violated by indexing.

One other field of the descriptor is important: the socalled 'presence bit'. This bit in the descriptor tells the hardware whether or not the data area associated with the descriptor is present in main memory; if not, an interrupt is generated so that the System can bring the area (i.e. page) into memory from auxiliary store.

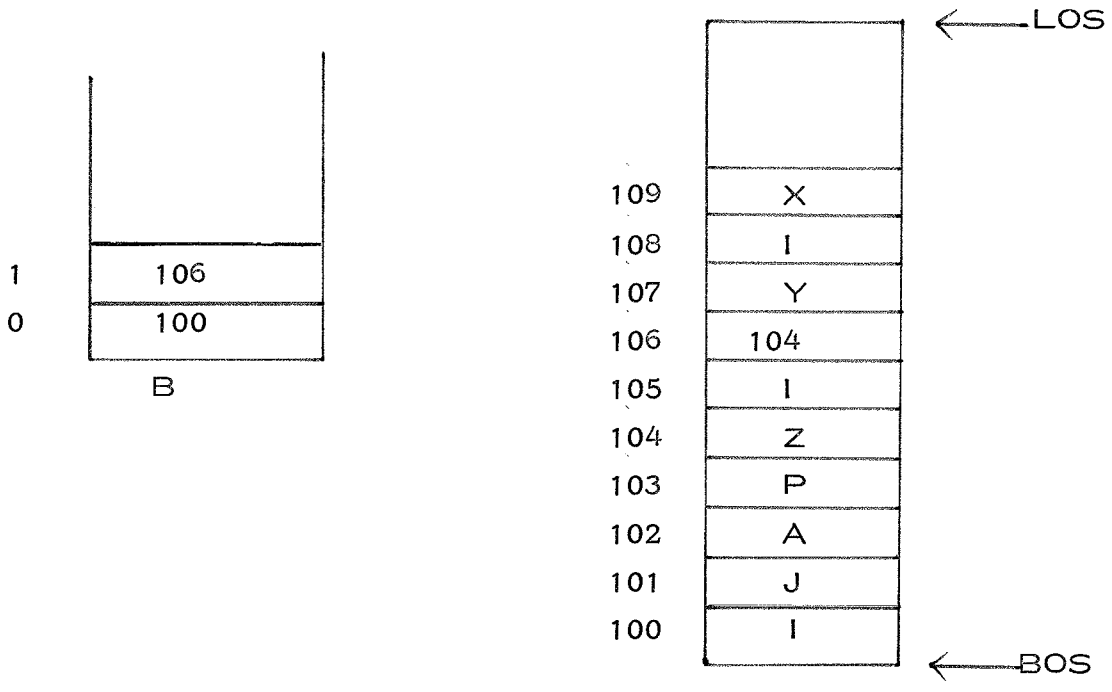
3.4.2 Relocatible Addresses.

In keeping with the overall philosophy that the occurence of absolute addresses should be minimized, and yet that the use of addresses in programs is well established, we compromise on two constructs: pointers and logical addresses. Although both have the same form, pointers are interpreted to mean 'automatic indirect' whereas logical addresses say 'this is the last indirect'. Essentially logical addresses are the means by which pointers can be 'gotten hold of' since they (ptrs) are otherwise automatically passed through.

One at first might think that an address couple constitutes a perfect form for a relocatible address, but this is not the case. Figure 3.4-1 illustrates this by pointing out that procedure calls cause the block structure to become folded upon itself, thus in a sense destroying the definition of block structure which states that one cannot 'be' in two blocks at the same level at the same time. Procedure parameters (which the



(a) SIMPLE's display while in the block labeled L1. Address of (L1's) Z is [b,d] = [1,0]



(b) SIMPLE's display after calling P from within block L1. Note that now [1,0] is no longer the address of Z; in fact, Z cannot be addressed using the display (use of B[0] to address Z totally violates the sense of [b,d] addressing).

Figure 3.4-1

Two Snapshots of SIMPLE Illustrating the Inadequacy of [b,d]'s as General Purpose Pointers.

figure illustrates) are the evidence of this violation, since they allow one block to reference variables (in a disjoint block) which ordinarily (by the scope rules of block structure) would be unknown to it. Since the scope rules are violated, and these scope rules are the basis of [b, d] addressing, it is therefore not surprising that address couples are unsuited to such addressing 'outside' of block structure.

The solution to this addressing problem is to consider a new form of address which is relative to the base of the stack. Given which stack and the relative displacement within that stack, the hardware can find any particular stack location; this [stack, displacement] form of address (denoted [s, d]) is the form used for pointers and logical addresses, as well as for relocating B registers which must be pushed into the stack. Figure 3.4-2 shows the format of TT2 words.

3.4.3 Indexing and Contiguous Descriptors.

Most structured data e.g. arrays and strings, utilize this type of descriptor (reference Figure 3.4-2), which informs the hardware that the data region described is L elements in length, beginning at (PC) absolute address A. When any data descriptor (or structured pointer/ logical address) is encountered, the value at MEM [TOS] is assumed to be the required index (i.e. TT0) and first checked against L, then added (with appropriate adjustments for data type in the case of byte, or bit) to A. If the index is not in [0:L] then an index out of bounds interrupt is generated. If the TOS element is not TT0, an invalid index interrupt is generated. Note that since L is a 10 bit field, the maximum size of any one dimension of a structure is 1024 elements, be they words, halfwords, bytes, or bits. Multidimensioned structures are accomodated by using one descriptor for the entire structure which points to a vector of descriptors describing the next layer of the structure etc. Multiple indexing from the stack as required for such structures is supported. See Figure 3.4-3.

The 'appropriate adjustments' referred to in the above description of indexing proceeds as follows (assuming no bounds or index type violations) :

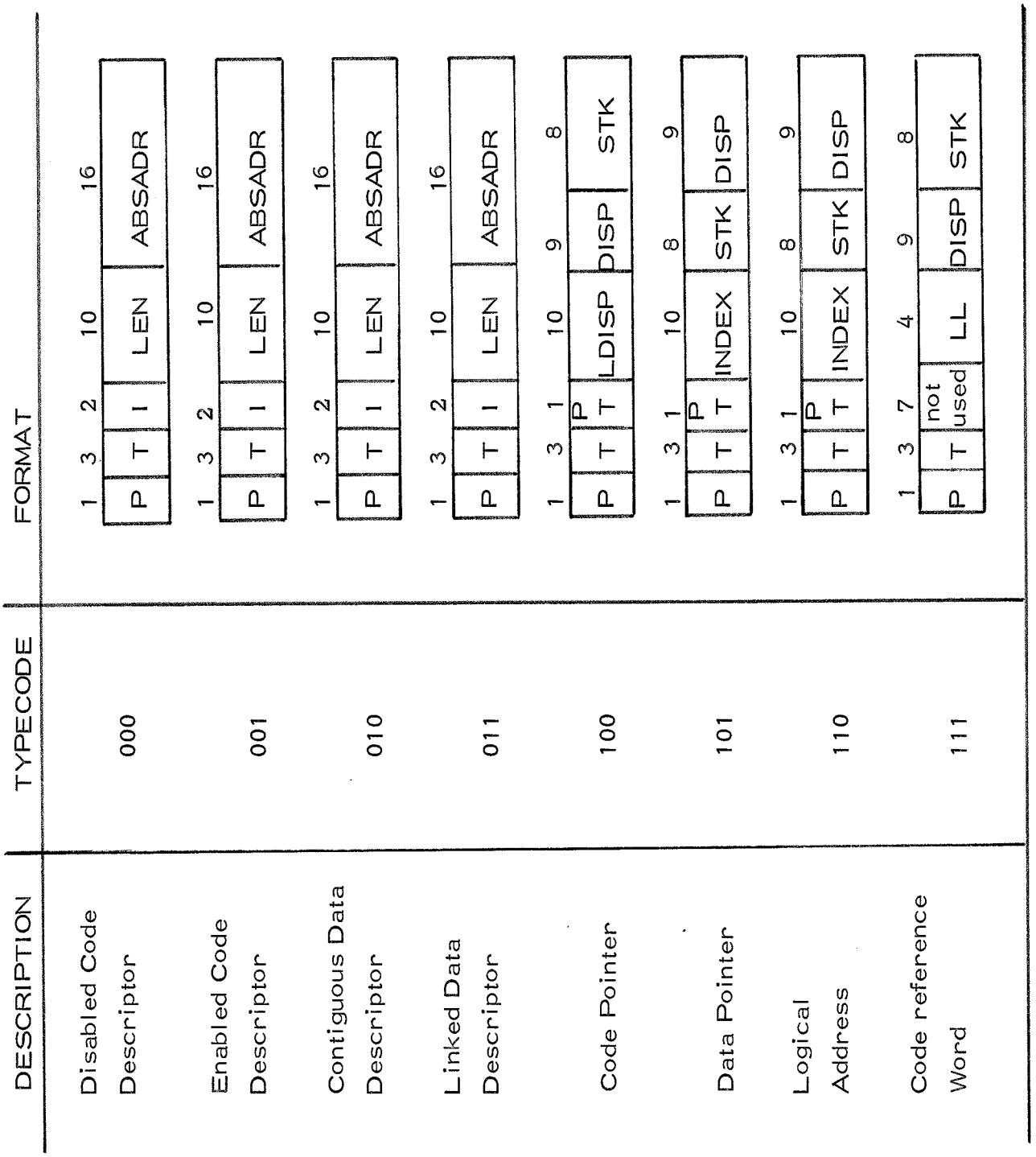


Figure 3.4-2

Format of Tag Type Two (TT2) Words.

P	presence bit
T	type code bits
I, IDX	indexing type
LEN	length
ABSADR	absolute address
PT	pointer type
INDEX	local index
STK	stack number
DISP	stack displacement
LDISP	local displacement
LL	lexical level

if=0, then area is not "present"
as shown in the middle column
none, word, byte, bit indexing on the data
length by item type (IDX) of the data area
absolute core address of the data area
simple (=0) or structure (=1) pointer
used to index the found descriptor
used to index the system stack vector
used to index the stack descriptor
displacement from found MSCW to target word
the lexical level at which the proc. shall execute

Figure 3.4-2 (cont.)
Legend of Abbreviations.

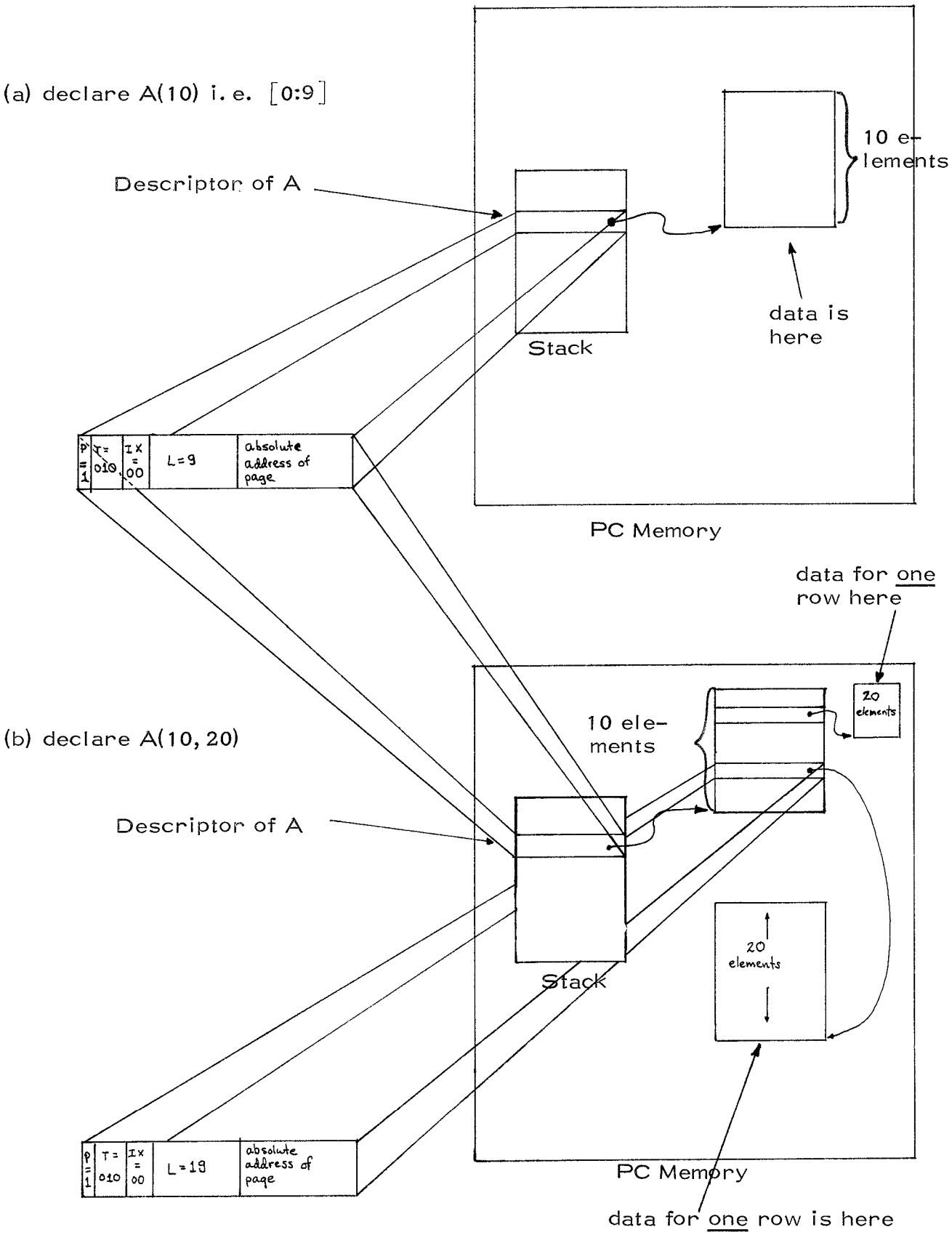


Figure 3.4-3.

Implementation of Homogeneous Arrays on the PC.

(1) If the indexing type of the descriptor is 'word', then the element on TOS is added to the absolute (page) address and that location i. e. MEM [A+MEM [TOS]] fetched and the stack popped. If the location thereby referenced is TT0, it is copied and pushed onto the stack. If the location is not TT0, but rather TT2 e. g. another descriptor or a pointer, the fetch operation continues analogously, consuming indicies from the stack as required.

(2) If the indexing type of the descriptor is not 'word', but rather halfword, byte, or bit, the index from TOS is shifted right the appropriate number of bits (2 or 5 respectively) and the result added to A. If the word found thereby i. e. at MEM [A+shr (MEM[TOS])], is not TT0, this is an error since only TT0 words can be bytes etc. From the found word is now extracted the subfield indicated by the low order (1, 2, or 5) bits previously ignored; the extracted subfield is right adjusted and pushed onto the stack, the index having previously been popped.

Figure 3. 4-4 illustrates, although insufficient information about the PC's instruction set at this point makes the example somewhat obscure.

3. 4. 4 Pointers and Logical Addresses.

Referring back to Figure 3. 4-2, we see that a pointer has three numeric fields - an index, a stack number, and a (stack) displacement. The latter two define the location in memory of some item, while the index field supplied an index (possibly in addition to TOS) to the item if it is a descriptor. If several pointers are passed through before encountering a descriptor or datum, the respective index fields are simply accumulated by the hardware in a working register known as 'cumulative index'.

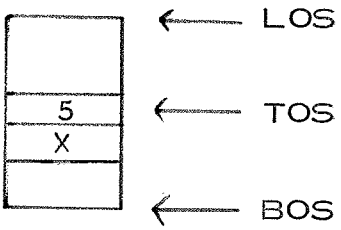
There are two types of data pointer: simple and structured, which are destinguished by the pointer type bit. The only place where the two types are distinguished is when the object pointed at is a data descriptor; for all other objects the actions are as outlined below:

declarations: declare A(5, 5) fixed;
declare S(10) character;

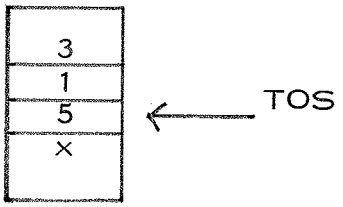
example: A(5, 1) = S(3);

```
1.    LITC 5        ; push a 5 onto the stack
2.    LITC 1        ; push a 1 onto the stack
3.    LITC 3        ; push a 3 onto the stack
4.    VALC S        ; 'value call' on S(3)
5.    STC  A        ; 'store and clear' into A(5, 1)
```

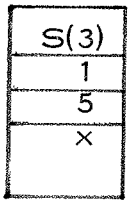
stack after 1st instruction:



stack after 3rd instruction:

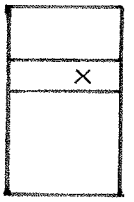


stack after 4th instruction:



; the 3 on TOS was consumed by S's descriptor. The value of S(3) is automatically placed on TOS.

stack after 5th instruction:



; the STC instruction hits S's descriptor which consumes the 1; then the next(column) descriptor consumes the S; the implied location is TTO, so S(3) is stored there and the stack cleared.

Figure 3.4-4
Example of Indexing Using Descriptors.

[Note: cum index is initially zero].

item encountered by pointer	actions
pointer (either type)	cum index:= cum index +index field; continue.
TT0 word	cum index is ignored; this is the desired object.
code descriptor	use cum index as a byte address into the page.
data descriptor	see below.

In the case of a data descriptor, the type of access is checked, and the actions taken are:

path to descriptor	actions
direct (no pointers)	descrip index: MEM[TOS]; TOS:=TOS-1.
via simple ptr	descrip index:=cum index.
via structured pointer	descrip index:=cum index+ MEM[TOS]; TOS=TOS-1.

'Descrip index' is then applied to the descriptor (biased appropriately by the type bits) and the implied item fetched. Actions are now as described in the preceding paragraph on descriptors.

The motivation for having two types of pointer is that it allows the PC to reference either a particular data element in the first level of a structure, or the entire structure, through pointers; a single pointer type would lead to ambiguity as to which form of reference is desired.

Example:

Assume that the following extended XPL declarations are in effect:

```
declare X fixed initial (5);
declare A(5) fixed initial (1, 2, 3, 4, 0);
declare P1 simple ptr initial (X);
declare P2 simple ptr initial (A(5));
declare P3 structure ptr initial (A);
declare P4 structure ptr initial (A(4));
```

Then all of the following have the effect of assigning the value 5 to A(5) :

1. P2 = 5;
2. P2 = P1;
3. P3(5) = 5;
4. P3(X) = P1;
5. P3(P1) = P1;
6. P4(1) = 5.

The fifth assignment above shows a simple pointer being used to index an array pointed at by a structured pointer. Example 6 coupled with the declaration for P4 shows how a structured pointer can also define an initial offset to indexing to the first level of its associated data structure.

A logical address is identical in form and interpretation to a pointer except that the indirect reference must be terminated at the next word, even if that word is a pointer. Another exception to the similarity to pointers is if the logical address is 'simple', then the index field is ignored when a descriptor is encountered (i.e. the descriptor is not indexed), and the descriptor itself is returned. In contrast, if a structured logical address encounters a descriptor, an element of the described page, not the descriptor, is retrieved.

3.4.5 Summary of Descriptors, Pointers, and Indexing.

Descriptors provide a means for uniform treatment of different types of data, automatic bounds checking, and virtual memory (via the presence bit). Pointers allow the source language to treat address variables without affecting the System's ability to relocate pages; structured pointers permit the creation of pseudonyms both for the entire structure and (front end offset) substructures. Logical addresses are the means by which pointers and descriptors themselves can be manipulated.

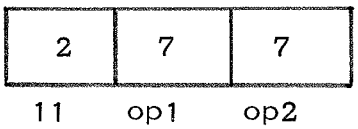
3.5 Opcode Structure and Code Stream Maintenance.

The two types of instructions in the PC are single address and zero address, the former being 16 bits in length and the latter 7 bits. They are known as long and short operators respectively.

3.5.1 Short Operators.

These operators consist of the common arithmetic and less common and more specialized operations. See Figure 3.5-1. In general, these operators finds their arguments on the stack, consume them, and place their result (if any) on the stack.

The short operators are packed two per 16-bit halfword, and such a halfword is identified by having the leftmost two bits '11'.



Seven bits are used because of the following reasoning: six bits are probably enough (the current PC uses about 35 short ops) but seven bits (=128 opcodes) are certainly sufficient; furthermore, the seventh bit is 'stolen' from the long operators leaving a maximum of twelve of these, which appears to be sufficient.

3.5.2 Long Operations

These operators are generally the most commonly used, and which also require a single argument. A given long operator could just as well be coded as a short operator, but then it would always require the execution of a prior (long) operation to place its argument on the stack. Clearly in commonly used operations this is inefficient.

The long operators fall into three categories:

Literal Call	Places its immediate field on the stack.
Local Branches	Uses its immediate field to update the code stream registers.
B-reg References	Immediate field is a [b, d].

Mnemonic	Opcode(dec)	Description
ADD	0	add two elements of stack
SUB	1	subtract top two elements of stack
LT	2	test top two elements for LT
GE	3	" " " " GE
EQ	4	" " " " EQ
NE	5	" " " " NE
GT	6	" " " " GT
LE	7	" " " " LE
DUP	8	duplicate TOS element
REV	9	reverse top two elements
POP	10	pop one item off stack
MKSK	11	mark the stack
NTRP	12	enter block/procedure
XITP	13	exit block/procedure
XITPV	14	exit block/procedure with value
BTOS	15	branch top-of-stack
HALT	16	halt
NOP	17	no operation
IGEN	18	initiate generation
GENS	19	generate single
GEND	20	generate double
STASK	21	swap task
PRAJ	22	prepare return address and jump
STOS	23	store TOS via TOS-1
LTOS	24	load TOS via TOS-1
MUL	25	multiply top two stack elements
DIV	26	divide top two stack elements
MOD	27	find modulus of top two stack elements
SHCL	28	shift circular left
AND	29	logical and
OR	30	logical or
EOR	31	logical exclusive or
DINT	32	disable interrupts
EINT	33	enable interrupts
DOIO	34	initiate I/O

LT, GE, EQ, NE, GT, LE: boolean result returned to stack

Figure 3.5-1

Short Operators.

Their format is:

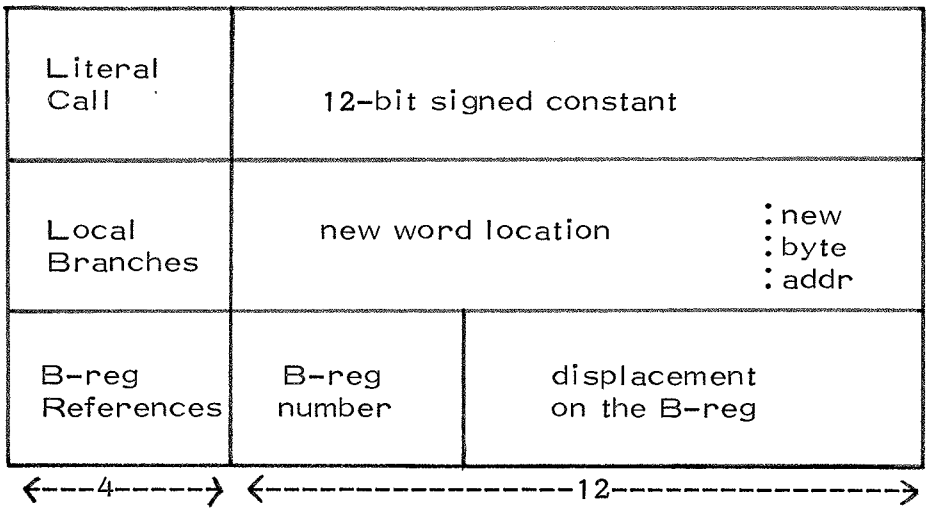
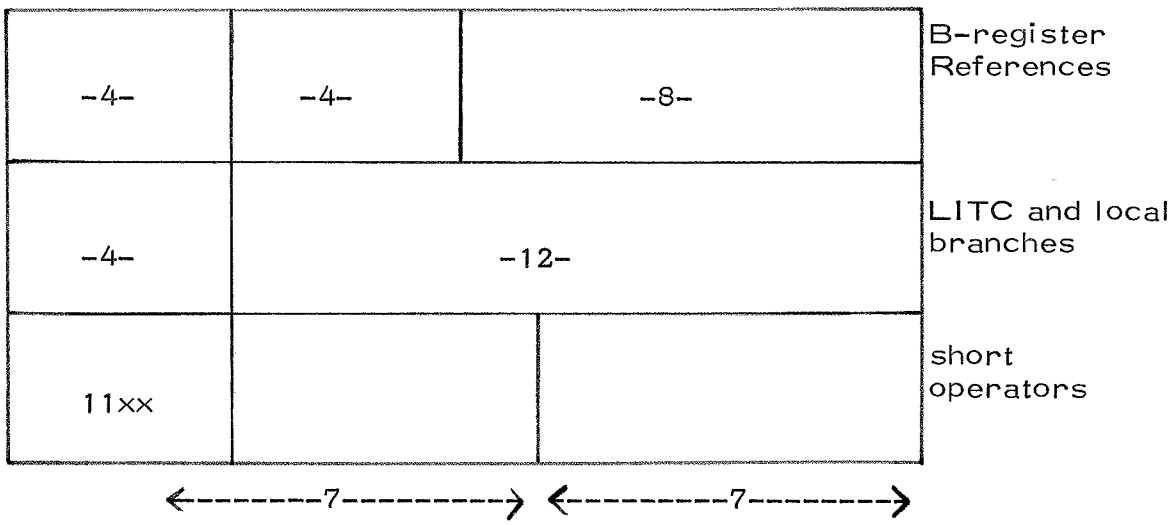


Figure 3.5-2 lists the twelve long operators, but their explanations are deferred to later.

3.5.3 Summary of Opcode Structure.

Figures 3.5-1 and 3.5-2 imply the following complete opcode structure:



It should be noted that 16 B-regiaters and 256 displacements are provided by this design, in agreement with the comments in [3.2].

Mnemonie	Opcode (dec)	Description
VALC	0	value call
NAMC	1	name call
LITC	2	literal call
STC	3	store and clear TOS
STP	4	store and preserve TOS
BU	5	branch uncond. (same page)
BGE	6	branch TOS GE 0 " "
BF/BZ	7	" " " " " EQ 0 " "
BT/BNZ	8	" " " " " NE 0 (true)"
BLT	9	" " " " " LT 0 "
BUN	10	branch uncond. to new page
PRDC	11	predecessor call (1 level dereference)

Figure 3.5-2
Long Operators

3.5.4 Code Stream Maintenance Registers

The following registers are used to control the execution of the code stream :

CPBASE	absolute address of current code page base,
WDCTR	absolute address of current instruction word,
SYLCTR	location of current operator within the instruction word (range = 0, 1, 2, 3),
CPMAX	absolute address of the end of current page,
CPLA	[s, d] address of the current code page (for construction of relocatible return addresses).

3.5.5 Local Branches.

The local branch operators (BU, BGE, BF/BZ, BT/BNZ, BLT) place the low order two bits of their immediate field into SYLCTR, and add the top ten bits to CPBASE to yield a new WDCTR. Before commencing execution, the new WDCTR is checked against CPMAX and an interrupt is forced if the bound is exceeded.

3.5.6 Non-Local Branches.

Branch Unconditional New Page (BUN) interprets its immediate field to be a [b, d] and branches to the implied location, including following pointers etc.

3.6 Procedures and Parameters.

Close support by the PC of the procedure mechanism is demanded by the requirements previously established:recursion, call by name (ON-conditions), call by value. This support will be reflected by a detailed discussion of the PC operators which support procedure entry, parameter passing, parameter accessing, and exit.

3.6.1 Disabled and Enabled Code.

It is anticipated that most object code in the PC system will be disabled i. e. it is not 'enabled' to allow entry thereto except by direct invocation e. g. branches to or 'called by' an explicit procedure call. In contrast, enabled code can only be invoked indirectly i. e. on a load or store. The

enabled code construct was originally invented to help out the compiler when compiling 'name' parameters - it automates the 'thunk' mechanism of [13] and works as follows: when a load or store operation encounters a code descriptor (through whatever devious path of pointers etc.), it ascertains whether or not the code descriptor is enabled or disabled; if disabled an error interrupt is forced. If enabled, a procedure entry is automatically forced to the code page under the assumption that it is a procedure.

Whenever the hardware forces a procedure entry (including interrupt), it passes a count of the number of parameters being passed. Thus if enabled code is encountered by a load operation, a count of 1 (the count itself is a parameter) is passed; if encountered by a store operation, the item to be stored, which is known to be on TOS, is passed as well, causing the count to be 2. The count is included as a debugging aid and to allow the enabled code body to determine if it is to load or store. The presumption made by the compiler is that the enabled code procedure will return a value (to TOS) if called by a load operation, and store the value away if called by a store operation; thus enabled code is assumed to be the 'tail' of the invoking instruction and the instruction counter is advanced as usual, and the return address constructed by the hardware points to the next instruction.

Enabled code has greater uses than call by name, however. It can be used to implement data structures not directly supported by the hardware e. g. queues. In this application, an enabled code procedure could be used to place values into an array and another to retrieve them, using queue disciplines; maintenance of the queue parameters and error checking are part of the code body.

Enabled code can also be used to effect a dynamic branch i. e. direct the path of subsequent program execution under program control. This is accomplished by performing a branch within the code body to whatever other point in the program is desired. Indeed, since a (block structured)entry is made to the enabled procedure, the procedure may do anything it pleases, even recurse the entire program or itself.

The discerning reader will realize that enabled code is a means by which binding time is deferred to the ultimate limit: execution time. Use of enabled code to build complex data structures and effect dynamic branching allow the compiler to avoid the decisions involved at compile time. The possible uses for enabled code are limited only by the imagination e. g. array elements which are expressions, variable tracing.

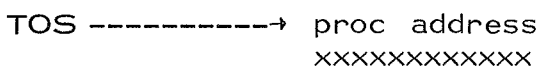
3. 6. 2 Preparing to Enter a Procedure.

Whether enabled or disabled procedures are involved, the following sequence of actions is required to enter a procedure:

(1) specifying the destination address and other set-up,
(2) supplying the parameters,
(3) preparing a return address,
(4) branching to the procedure code page,
(5) entering the procedure (in the block structured sense).

This paragraph discusses the first of these - the preparatory actions.

The first instruction which is executed is a name call (NAMC), which is a single address instruction having a [b, d] as its immediate field. The [b, d] is converted to an [s, d] (actually, a code pointer) and pushed onto the stack. The pointer is assumed to point to a path to a code reference word, which points to a code descriptor. At this point the stack looks like:



The next action is to 'mark the stack' (MKSK). This operation pushes the current contents of the mark stack register (MKST) onto the stack (in pointer form). The purpose of the MKSK operation is to reserve a slot in the stack in which will later be placed the pushed down B-register (á lá NTRB). The true reason for the need for the MKSK operation was discussed earlier: the fact that procedures allow variables that are otherwise invisible (because of scope rules) to be referenced within a procedure. While parameters are actually being passed, the B-register envi-

ronment must allow them to be referenced as usual, yet they must be placed on the stack in such a way as to be 'in' the procedure's block when it starts executing. By reserving a slot on the stack underneath the parameters, we remain in the current environment, yet allow a B-register to be pointed to the beginning of the block.

The MKSK operation must push (i. e. save) the current value of MKST because at the current time, we may be half finished entering another procedure e. g. CALL P(Q(R)). At the conclusion of the mark stack operation, the stack looks like:

```
TOS, MKST ----------→ old MKST value
                      code ptr to code ref word
                      xxxxxxxxxxxxxxxxxxxxx
```

3. 6. 3 Supplying the Parameters.

This phase of procedure entry must consider whether the parameters are to be passed by value or name (reference being an easy case of name). The compiler is responsible for generating the appropriate code i. e. code which will place either values or pointers on the stack. The former case is no different from normal expression evaluation, but the latter requires special consideration.

At first glance one might think that a simple NAMC on the variable would be sufficient to build a pointer to that variable. However there is the consideration that if the variable is itself a parameter, then a NAMC will create two levels of indirectness (pointers) between the called procedure and the actual variable. In fact, every successive time a variable is passed as a parameter, another level of indirectness is interposed. Execution time could become intolerable. Clearly, changing NAMC to minimize the length of pointer chains nullifies its original purpose - the building of pointer chains. The resolution of the dilemma is to coin a new (single address) operator 'predecessor call' (PRDC).

PRDC converts its [b, d] to a pointer ([s, d]) and examines that location.

If the location is also a pointer, it pushes a copy of it, otherwise it pushes the original pointer, onto the stack.

Thus PRDC looks only at the immediate predecessor in most cases, and the way it works is such that no matter how deeply a simple variable is passed as a parameter, the number of indirects is exactly one. In the case of a procedure passed by name, the reader should convince himself that the enter/exit mechanisms described below have the desired effect of saving/restoring the appropriate environment. Another way of describing PRDC is that it performs a single level de-referencing [L1].

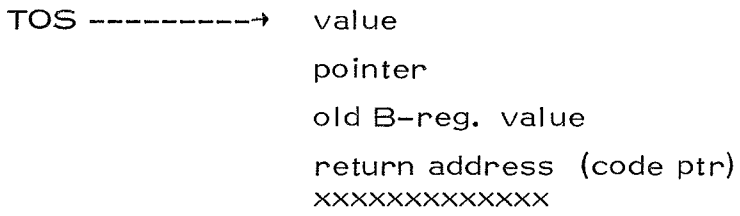
At the end of the parameter passing phase, the stack looks like:

TOS ------→	value	param if call-by-value
	pointer	param if call-by-name (built by PRDC)
MKST ------→	old MKST	
	code ptr	
	xxxxxxx	

3.6.4 Entering the Procedure.

Once the stack has been marked and parameters stacked, the last step is to perform the transfer and adjust the B-registers to reflect the entry into the block which is the procedure. Since the hardware cannot know which one of these is to be modified, a parameter is necessary. Thus the 'level' field of the code pointer supplies the necessary information. One other job of NTR is to restore the value into MKST which was saved in the slot that now holds the old value (in pointer form) of the designated B-register. NTR also packs the current value of CLVL into the index field of the slot and replaces it with the level found in the code pointer.

At the end of NTR, the stack now looks like (see also Figure 3.2-5a):



3.6.5 Exiting a Procedure.

Once the procedure has finished its job, control must be returned to the point of call with appropriate readjustment of the environment (B-regs). Two short operators, XIT and XITV, accomplish this change , which is considerably simpler than the entry process; the two operators differ only in that XITV moves whatever is at TOS to the new TOS after all the procedure entry related items have been cleared off. Figure 3.2-5b gives an algorithm for XIT.

3.6.6 Summary of Procedures and Parameters.

Figure 3.6-1 shows an example of the code generated by the (extended) XPL compiler for the PC for procedures. It should be noticed that no matter how subtle the semantics involved, the code generated by the compiler for both the caller and the callee is very straightforward.

3.7 Overall Structure of the Programmable Computer.

Sufficient information has now been given to present Figure 3.7-1, which shows how all the pieces previously mentioned fit together. A number of loose ends can now be tied together.

3.7.1 Tasks and Blocks.

The figure shows the stack of the operating system and the stack of a task (job). We have heretofore talked about the relationship between blocks and tasks, but the figure clearly shows the B-registers pointing into two different stacks - the same B-registers which we have discussed only in the context of blocks. The fact that the interpretation of the B's does not differ whether or not they are used to point to blocks or tasks solidifies the previously theoretic relationship.

The function of the Stack Number Registers alluded to earlier now is re-

```
begin integer N;
    procedure P(X, C); value C;
            procedure X; integer C; begin
            begin
                procedure R
                begin
                    N:=N+C;
                    X
                end ;
                if C > N then X else P(R, C+1);
            end;
    end;
    procedure Q ;
              N:=N+1

    N:=2;
    P(Q, 2);
    print (N)
end
```

(a) This program is program SAM from [J1], slightly modified. The result is N=5, is explained in detail in [M2].

Figure 3.6-1
Sample Procedure and Object Code.

```
        LITC    0       ; declare N     code page main
        NAMC    0,2     ; declare P
        NAMC    0,3     ; declare Q
        LITC    2       ; assign 2
        STC     1,1     ; to N
        NAMC    1,2     ; pointer to P
        MKSK            ; ready to enter P
        PRDC    1,3     ; first param
        LITC    2       ; second param
        NTR     2       ; enter P
        HALT            ; end program
        END             ;               end main

        NAMC    0,4     ; declare inner block(I.B.) code page P
        NAMC    2,3     ; pointer to I.B.
        MKSK            ; ready to enter I.B.
        NTR             ; enter inner block
        XIT             ; finish P
        END             ;                               end P

        VALC    1,1     ; fetch N       code page Q
        LITC    1       ; fetch 1
        ADD             ; add 1 to N
        STC     1,1     ; and assign to N
        XIT             ; finish Q
        END             ;               end Q

        NAMC    0,5     ; declare R     code page inner block
        VALC    2,2     ; fetch C
        VALC    1,1     ; fetch N
        GT              ; compare C and N
        BF      16      ; if not (C>N) then goto 16
        NAMC    2,1     ; pointer to X
        MKSK            ; ready to enter X
        NTR             ; enter X
        XIT             ; finish  I.B.
16:     NAMC    1,2     ; pointer to P
        MKSK            ; ready to enter P
        PRDC    3,1     ; first param
        VALC    2,2     ; fetch C
        LITC    1       ; add one to get
        ADD             ; second param
        NTR             ; enter P
        XIT             ; finish I.B.
        END             ;               end inner block

        VALC    1,1     ; fetch N       code page R
        VALC    2,2     ; fetch C
        ADD             ; add N and C
        STC     1,1     ; and assign to N
        NAMC    2,1     ; pointer to X
        MKSK            ; ready to enter X
        NTR             ; enter X
        XIT             ; finish R
        END             ;               end R
        FIN             ;               end program
```

Figure 3.6-1 (cont).

Object Code for the Program.

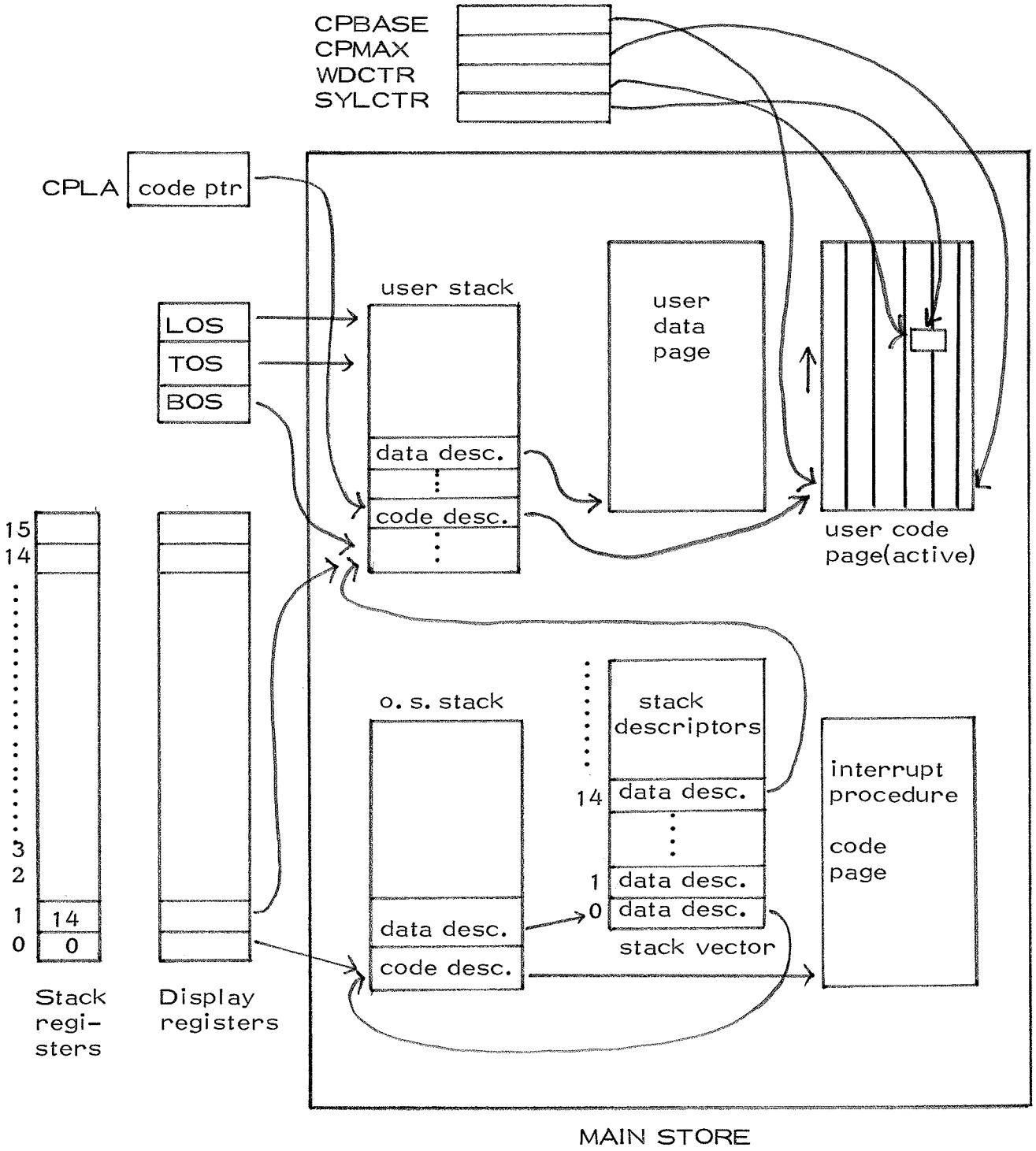


Figure 3.7-1

Overall Picture of the PC Architecture.

vealed: since a given B-register can point into an arbitrary stack, the ability to form a (reloctible) [s, d] out of the absolute address in the B hinges on knowing the absolute address of the base of that stack. [Recall that the 'd' in [s, d] stands for displacement, the distance between the location itself and the base of the stack in which it resides]. Given the stack number, the hardware need only consult the stack vector (see below) to find the necessary information. It can also be observed that pointers with their [s, d]'s can as easily point to data in parallel tasks as its own or its parent.

3.7.2 The Stack Vector.

The stack vector is a one-dimensional array of ordinary contiguous data descriptors which describe all the stacks in the system, the reader noting that a stack and a task are now synonymous. The stack vector itself has a descriptor which resides in [b, d] = [0, 2], which fact is known to the hardware. Since the elements of the stack vector are themselves descriptors, they have a presence bit which can be turned off, which allows the System to easily page out jobs, even the parent of subtasks, without great trouble or side effects; any reference to data or code in the missing stack will be signalled by a presence bit interrupt.

The stack vector has another salutory effect on the System. Since the stack vector descriptor and the elements of the stack vector are ordinary data descriptors, a reference to SV[i, j] constitutes a reference to the jth location in the ith task. This fact can be used to advantage since the System can store all task related data e.g. priority, queue list links, etc. with the job itself, thereby minimizing the size of System tables and freeing the System from any inherent limitation on the number of jobs (aside from the fact that the descriptor length field forces a limit of 1024).

3.7.3 The Interrupt Procedure.

The interrupt procedure (code) descriptor is known to the hardware to reside at location [b, d] =[0, 1], and interrupts (as mentioned before) are treated as entries to this procedure and the interrupt type information as parameters. This scheme has two interesting effects.

One: The System stack, after dead start, will probably never be used

for actual execution since work will be done by the system subtasks thus rendering the parent stack inactive. Therefore interrupts will occur while some job is executing (i. e. its stack active) and the interrupt procedure entry will be built on the user stack. The result is a very well defined path from the user into the System. If the system routines encountered should decide to swap tasks, it makes no difference to the interrupted user, who when reactivated will wend his way out of the system without any special attention, using the normal procedure return mechanism.

Contrast this scenario with the events which transpire in an ordinary (register) machine. Upon occurrence of the interrupt, the state changes from 'user mode' to 'supervisor mode', which on many machines e. g. 360 implies saving all the registers. Next some number of system routines are entered and the return addresses carefully saved away in temporary locations. If at this point a task switch is desired, it is very messy to accomplish, and even if accomplished, to allow the user task to once again find his way out of the system using those (saved elsewhere) return addresses is more difficult yet. Worse, this machinery would find most of its use in processing supervisor calls, the most common type of interrupt.

Two: The reader may have noticed that there is no instruction to 'change state' from user mode to supervisor mode. This is made possible for the reasons stated in [2. 1. 3] and the closing paragraphs of [1. 3. 2]. The effect of the state change (among other things) is to change the address space for subsequent execution. The analogous event in the PC would be to change stack vectors! Normally, the user communicates with the System by directly invoking the desired procedures, but if he had his own stack vector, he would be totally insulated from everything else in the system.

This speculation has two benefits:

(1) It sharply dilineates the difference between computers which use global states to isolate jobs from the system. The difference shows clearest when one considers the

vastly different software implied such as discussed in One.

(2) It intrigues the mind as to what use such a scheme might have, since it represents an even higher and thicker wall between users, and between users and the System. Such walls are for isolation and protection, but what could require so much isolation? Perhaps a different system ?

3.8 Virtual Memory and Paging.

The availability of virtual memory is essentially a bonus dividend of descriptors. The fact that even the system stack and the stack vector are both themselves pages described by descriptors emphasizes the point that everything in the PC's memory (except unused memory) is a page. Thus theoretically the only parts of the System that need be present are the interrupt handler and the paging module. The fact that virtually the entire system can float leads to a more rational approach in that designers of a system are not burdened with the requirement that a system to do thus-and-such must not take more than N words of memory. This type of constraint leads to monolithic code modules which sacrifice generality by their nature. Even with a non-virtual memory computer, the operating system must allow for transient code e. g. OS/360's A, and B transients [11].

However, even having a virtual memory does not spell the end of the designer's problems, but merely their movement to a different locale. In the case of virtual memory, the problems arise in the paging module: garbage collection, compaction, thrashing, etc. The logic applied in the PC's design was that a virtual memory facility (especially variable length pages) without some hardware support for managing it would succeed only in accumulating mountainous overhead. But the issue of exactly how to support it is not obvious.

Given the volumes of literature on paging strategies e. g. [D1], [B1],

[K1], [K3], the decision was made to support the following:

(1) Hardware maintained list of availble space (LAVS), singly linked in order of size.
(2) Hardware maintained list of pages in use, doubly linked.
(3) Hardware compaction of memory.
(4) Hardware flagging of every page that is accessed during program execution.

These features are realized through three short operators:

(1) GETPAGE - given the address of the potential page's descriptor, LAVS is search on a first-fit strategy. If a page is found, the space is removed from LAVS and the page linked into the in-use list. The in-use list also contains a back pointer to the descriptor which describes it (corollary: there is a one to one correspondence between pages and descriptors). If no space is available, it returns a failure flag.
(2) RETURN PAGE - given the address of a descriptor, the page described is linked back into LAVS in such a fashion that LAVS will never contain entries for contiguous blocks.
(3) COMPACT - given a base memory address, all in-use pages above that address will be compressed, resulting in all available space being collected together in one big LAVS cell at the top of memory.

To aid the System in deciding whether or not to compact, a time consuming process, the LAVS head cell contains the total amount of available space, however scattered. If compaction still will not gain the needed space, the System must choose a page to write on disk, and to aid in the selection of a page, the hardware turns on a bit in the second link word of every page whose descriptor is accessed, whether by read, write, or branch. Every time a page is thrown out, the System can clear the usage bits in anticipation of the next choice point. The memory overhead for this mem-

ory management system is one word per LAVS cell and two words per in-use cell.

Clearly it would be superior to flag reads and branches versus writes to save the system from writing out an unmodified page (assuming there's already a copy on disk). It would also be helpful to have several in-use lists of varying priority. Hardware limitations and practical considerations precluded a more elegant approach, some of which are discussed in [L2], [L3].

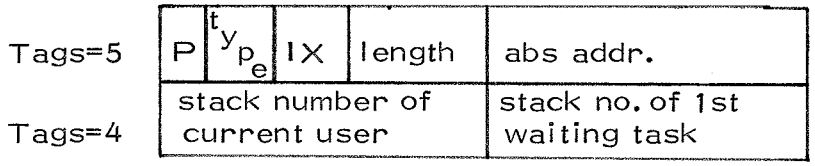
3.9 Semaphores.

Three types of semaphore-like constructs are supported in the hope that convenience will encourage use. All are double-word (64-bit) constructs with the first word being identical (except for tags) to the non-semaphore counterpart, and the second word containing the semaphore fields and 'continuation' tags. Figure 3.9-1 shows the format of these constructs.

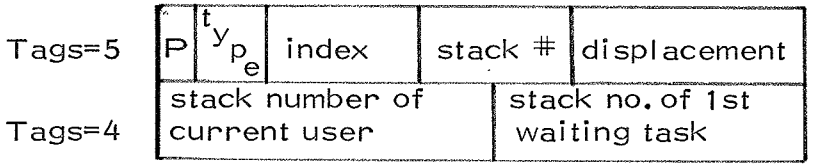
3.9.1 Semaphore Code Descriptor.

This mechanism is restricted to procedure entries to code pages (as opposed to branches, to avoid the 'honor system' of using critical code, which can result in someone forgetting to unlock and thus blocking the system). The locking of the code page is done upon entry (NTR) and the unlocking upon exit (XIT, XITV) and is not under programmer control except insofar as he specifies the use of a code semaphore in the procedure declaration.

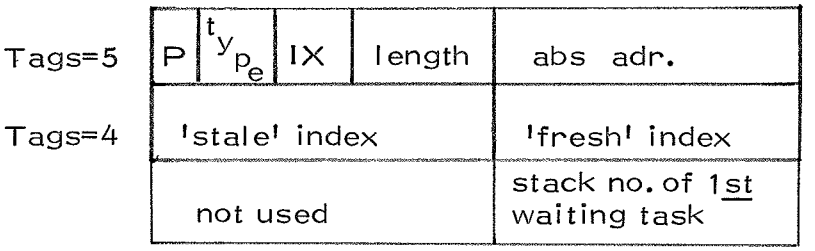
This construct is expected to be more convenient than the semaphore pointer (below) since it is very automatic. Consider short sensitive (since they may manipulate absolute addresses) routines which are used frequently yet are of sufficiently minor strategic importance not to warrant explicit queueing mechanisms in the System. In fact, the automatic queueing of other tasks on the semaphore's wait list can be easily pushed down into the hardware. However, whether done by hardware or software, such wait-queues can use the ready-list link found in the base of each task's stack. See [O1] for a description of this elegant structure.



(a) code semaphore



(b) pointer semaphore



(c) data (producer/consumer or circular buffer) semaphore

Figure 3.9-1

Format of Semaphore Constructs.

3.9.2 Semaphore Pointer.

The semaphore code descriptor provides a means of locking code, and could therefore be used as a gatekeeper for data shared between tasks. This carries with it however considerable overhead for such a simple need. On the other hand, the semaphore pointer can lock out anything to all tasks save the one that locked the semaphore. The beauty is that once the semaphore has been locked, multiple accesses can be made to the referenced object with only the price of an indirect.

This beauty has its beast however, in that if the semaphore pointer is used to lock out code, the code is locked out to other tasks, but reentry by the same task is not. This is the price paid for the efficiency of the access and the fact that the locale of the access can be far removed (i. e. not restricted to be in the procedure) from where the LOCK was made. The semaphore pointer also has the disadvantage that the honor system is used to ensure that what is LOCKed is eventually UNLOCKed. The field in the semaphore which contains the stack (task) number can be used by the system to remove a job which has violated its honor.

3.9.3 Semaphore Data Descriptor.

This semaphore construct's intent is to allow shared access to data in an automatic fashion, while the intent of the previous two semaphores is to restrict shared access. The principle usage of the semaphore data descriptor is foreseen to be producer-consumer applications, and it therefore is realized as an automater of circular buffering, with the producer supplying fresh items and the consumer removing stale ones.

The price paid for the convenience is that neither producer nor consumer have access to the fresh and stale buffer pointers, and hence every access to the circular array updates one index or the other. Every load (VALC) from the buffer causes the stale pointer to be advanced and every store (STC, STP) causes the fresh pointer to be advanced. Thus the programmer, if he wishes to 'see' a buffer item for longer than the one event in which it is moved to (or from) TOS must save it away in some other location.

phore emphasizes the automatic sharing, but probably makes it more difficult to undo a deadly embrace, since the embracees are not therein reflected.

3.10 Generators.

As any former IPL-5 [N2] programmer will tell you, the generator constructs are an extremely useful programming tool. With this in mind, the PC supports its own version of generators which is adapted to the linear (or other yet to be implemented) data structures of the PC instead of only list structures. Figure 3.10-1 shows the logic of the generator mechanism, which has two variations implemented on the PC: generation of elements off a single structure, and generation of pairs of elements from corresponding structures.

Figure 3.10-2 gives an illustrative example of the use of a double generator to compare two strings for equality (which could actually be done more efficiently using a loop). A better example might be a generator driving a lexical analyzer which drives a parser, but the principle point is that, like semaphores, generators automate a common process in programming and therefore encourage cleaner programs and improved overall logical structure.

3.11 Support of Sub-emulators.

Section 1. discussed the necessity for microcode extensions to be supported as inner, rather than disjoint, blocks in a system; the reasoning behind this conclusion is to allow these microcode extensions to avail themselves of existing global software structures such as the operating and file systems. It is important, however, to merge this capability into the PC architecture without destroying its inherent cleanliness and generality. The technique described below accomplishes this by defining a new type of descriptor and structually consistent operators for entering and exiting the microcode.

3.11.1 Emulator Storage Descriptors.

The first aspect of the sub-emulator problem we will attack is that of

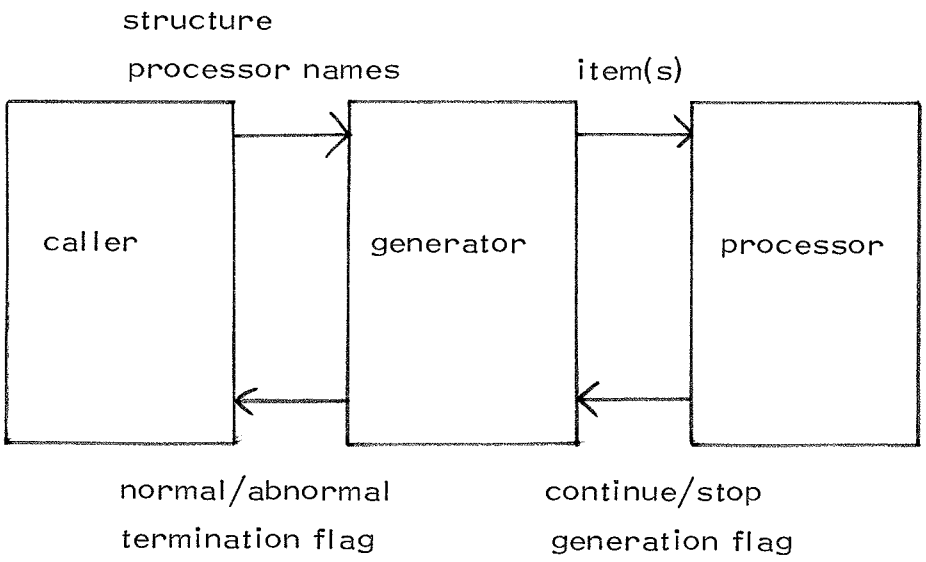


Figure 3.10-1
Overall Generator Logic.

```
COMPARE:  procedure(item 1, item 2) boolean;
              declare item 1, item 2 character;
              if item 1=item 2 then
                return true; /* stop generating*/
          end COMPARE;

          declare string 1(100), string 2(100) character;

          if generate (string 1, string 2, COMPARE) then
          /* strings are identical*/  ------;
          else /* strings were different */
```

Figure 3.10-2

Use of Double Generator to Compare Two Strings.

storage for it and the object code it is interpreting. One must describe where the emulator itself is - in Main Store (MS) or Control Store (CS); the code to be interpreted will be assumed to always reside in MS. The resulting descriptor, is

tags = 6

P	E I	MS CS	Length (words)	Absolute Address

where

P = presence bit

E/I = emulator code or interpretable code

MS/CS = the storage described is located in Main Store or Control Store.

An emulator is treated, formally, as a procedure requiring one parameter: the location of the code it is to interpret. Thus the sequence is analogous to that of normal procedure entry.

3.11.2 Entering an Emulator.

Given that an emulator treated formally as a procedure, we can see therefore that the entry sequence is

a. NAMC on the emulator storage descriptor which describes the location of the emulator;

b. MKST;

c. construction of an emulator storage descriptor which describes the location of the code to be interpreted. In the interests of efficiency, the storage for the descriptor should be allocated and initialized before the next step;

d. NTR the emulator (which also builds a return address to the PC code).

Step (d) above is the critical one in preserving the architecture. It is

crucial that the emulator be viewed as a block nested within the system, and thus a block-structured entry (via NTR) must be made. 3. 11. 3 on Input/Outputgives a good illustration of why this is necessary. Another dividend is that the emulator can use the stack for scratch storage, thus paving the way for reentrant use of emulators.

3. 11. 3 Life within an Emulator.

Having entered the emulator, we next examine what goes on while the emulator has control. Most likely, the first thing the emulator will do is to transfer the absolute addresses locating its code to some Local Store (LS) registers. It then proceeds to fetch and interpret instructions in the usual fashion. Two things can happen which disturb this happy scene, however: (1) the necessity of perhaps accessing data items non-local to it, e. g. in PC's semantic space; and (2) the necessity of invoking non-local object code e. g. the operating system.

Both of these problems have in common the characteristic that reference to code in the PC's emulator (or hardware) is implied. In the simpler case of data accesses, the emulator must construct PC-type addresses and feed them as parameters to the e. g. VALC instruction. The result is the appearance on the top of the stack of some datum which must, in the general case, be converted to a form compatible with the emulator's data types. Thus the emulator must contain routines which convert in both directions between the two sets of data types.

The case where the emulator wishes to contact PC object code is a little trickier, since a transfer of control impacting PC's semantic space is involved. However, the problem is considerably clarified when we remember that the emulator is really an inner block, and we wish to call an outer block. Thus what is necessary is for the emulator to build and execute (perhaps using single invocations of PC instructions) a block structured entry to the appropriate PC procedure. Exit from the PC procedure, by implication, thus requires PC's XIT operator to be cognizant of the fact that it is exiting to an emulator instead of to PC code, and transfer machine control appropriately.

Input/Output can be expected to pose problems, as usual, but these are, in this context, manageable. It is first necessary to postulate the existence of an I/O nucleus which is in charge of doing physical I/O, whether to/from the PC machine or a subemulator. Given that I/O is accomplished within the semantic confines of the particular emulator, it is clear that the nucleus must already be made aware of (1) which I/O events are of interest to which emulator, and (2) MS accessing routines for each emulator. These two items are "givens" in a multiemulator environment. The only mechanism which is missing is a means for the I/O nucleus to invoke the PC (i.e. the ruling machine) for interrupts of interest to it. Thus it is additionally necessary for the nucleus to be aware of the subemulator routine which accomplishes entry to the PC machine. This mechanism fortunately already exists, as described in the preceding paragraph.

3.11.4 Exiting an Emulator.

We have now discussed how one enters an emulator, and how one, from that emulator, can enter and return from the PC machine. What remains is to exit from the sub-emulator, but this can now be seen to be an application of the same principles. What is necessary is to construct a block-structured exit, at the same time returning control to the PC machine. But this is really the same mechanism as returning from the PC to a sub-emulator. The unifying concept is to assume that the PC has all the appearances of any other emulator - it is distinguished only by the fact that it is the 'boss' machine i.e. the machine on which the bulk of the system executes.

3.11.5 Some Final Observations on Emulators.

The foregoing discussion leads one inevitably to consider a multi-emulator system to be structured as shown in Figure 3.11-1.

The structure shown in Figure 3.11-1 accommodates all the mechanisms postulated in the preceding paragraphs. It also demonstrates a pleasing symmetry wherein individual instructions are viewed merely as procedure calls. Most important, by its very form one can see the (previously established) necessity and utility of recognizing a subemulator as a block.

```
begin
        array MEMORY [0:flex];
        procedure SWITCH EMULATORS; -----
        procedure PC(location of code to be interpreted)
            .
            .
            .
            .
            do case OPCODE-FIELD (MEMORY(location counter));
                    VALC: -----
                    NAMC: -----
                    LITC: -----
                    NTR:  -----
                    XIT:  -----
                      .
                      .
                      .
                      .
                      .
        end PC machine;
        procedure SUB-EMULATOR 1(location of code to be interpreted);
            .
            .
            .
            .
            do case OPCODE-FIELD (MEMORY(location counter));
                      .
                      .
                      .
                      .
                      .
        end SUB-Emulator 1;
        .
        .
        .
        .
        .
        ┌
        │   I/O NUCLEUS
        │
        └
end;
```

Figure 3.11-1

The Structure of a Multi-Emulator System.

Only by accepting this recognition can one achieve the necessary generality to accommodate multiple emulators. Furthermore, if one cares to make a final 'leap of faith', one can say that once more block structure has demonstrated its stubborn consistency in reflecting the structure of the things we want computers to do.

3.12 Input/Output.

The design criteria established in [2.7] imply:

(1) Interrupts should not be time dependent;
(2) I/O channels should be cognizant of the memory structures e.g. buffer limits;
(3) I/O should if possible make use of the Dijkstra coordination primitives.

The first of these is satisfied by (a) arranging I/O requests such that only error and completion interrupts are allowed, both of which are non-time-critical, and (b) stacking all interrupts, even multiple occurrences of the same interrupt type, up to some limit. The former requires that channels be rather intelligent so they can be given the specifications for the entire I/O operation and perform it without central processor supervision. The latter requires that the hardware contain some amount of local memory in which to accumulate interrupts until they are reenabled e.g. one 'slot' for each device plus a number of slots based on the expected number of active tasks.

The second and third design criteria, channel recognition of the PC's memory structures and semaphores, is accomplished by postulating that channels will use the non-TOS related portions of VALC and STC as subroutines in connection with semaphore data descriptors. The use of semaphore data descriptor buffers implies recognition of buffer limits, automated maintenance of buffer pointers, semaphore-based coordination of the I/O process, and the ability of the I/O-requesting task to achieve a producer-consumer relationship (if needed) to optimize transfer rate. The only exception to this scheme is the disk, whose transfer rate is sufficiently high so as to cause the use of circular buffers, to be

such that the Fresh and Stale pointers are updated at the end of the transfer.

The only I/O instruction is DOIO, which expects the following information on the stack:

Input or Output
Device number
Transfer count
Buffer (desc) address

As stated above, the buffer is a circular one, and the transfer count can also achieve the value infinity for such devices as teletypes, which can initiate input independently. The infinite count serves to keep an open pipeline between the buffer and the device.

One other instruction, FIO (function I/O) is a catch-all for testing device status, both logical and physical, although most of the traditional 'function' operations are automatically taken care of by the channel.

3.13 Summary of the PC Design.

The preceding sections have described an architecture which attacks the problem outlined earlier. The architecture rests on three major concepts, and like a three-legged stool, it collapses if any one of them is missing. These concepts are:

(1) Self-identifying 'Data' i.e. tag bits, code and data descriptors, pointers, semaphores.

(2) Hardware-known Scratch Storage i.e. the stack, which allows expression evaluation, interrupts, and enabled code.

(3) Block Structured Addressing, which confers memory protection, easy translation, and close runtime support of a source language.

The interdependencies can be summarized:

(1) There is a little need for hardware known scratch storage without self-identifying data;

(2) Block structured addressing and dynamic storage allocation without a stack is nearly meaningless;

(3) The generality of e. g. VALC with its address couple is lost without self-identifying data; most generated code consists of VALC, LITC, and STC.

The discussion which surrounds the introduction of each of the PC's constructs contains the rationale for it and the manner in which it approaches the design objectives of [3. 0]. Rather than repeat the discussion, we here reproduce those objectives and ask the reader to judge if the PC will :

(1) Ease program creation via HLL's;

(2) Ease debugging;

(3) Recognize the relationship between block structure and multiprogramming;

(4) Recognize the Dijkstra coordination primitives both between co-tasks, and between the central processor and the I/O processes;

or as restated immediately thereafter

(1) The PC's instruction repertoire should be 'close' to the languages which execute thereon, thus allowing easy compiler generation, efficient code, and execution time diagnostics relatable to the source program;

(2) The PC's hardware should automatically oversee all program execution (including the System), and trap all questionable semantics. This implies the existence of primitive features which allow the hardware to distingguish among different data types, code, and addresses;

(3) Hardware support of the Dijkstra primitives implies an explicit hardware recognition of task structures i. e. block structure.

4. 0 COMPARATIVE ANALYSIS OF THE PC.

This section analyzes the architecture of the PC with respect to two existing computers: the IBM S/360 and the DEC PDP-10. The former was chosen because of its wide distribution, and the latter because it has a rich instruction set including simple stack manipulation operators.

4. 1 Rationale for the Comparison.

The general argument which has been made in the previous chapters is qualitative and runs:

A. Multiprogramming has been demonstrated to be an effective method of increasing the throughput of a computer system.

B. Even in a batch system, the presence of I/O activities represents a multi-process environment.

C. Multiprogramming and block structure give rise to identical data and control structures.

D. Therefore a machine architecture which directly supports these (block structure) data and control structures will yield superior performance (throughput).

This chapter will also yield the same conclusion, via the following syllogism:

A. Certain primitive data fetch and transfer of control operations must be accomplished in any multiprogramming (i. e. block structure) system.

B. The "cost" of these operations is generally lower for the PC than for the S/360 or PDP-10.

C. Therefore the PC will yield superior performance.

In a sense, the 'cards have been stacked' for the PC in this argument because a number of the primitive operations measured (e. g. block entry) have been the object of optimization in the PC architecture. The counter argument to this objection is that:

A. The PC performs equivalently to the standard architectures in the non-block structured areas.

B. The PC would perform very poorly in an environment which was contrary to block structure. (It is the author's contention that such an environment is rare compared to the complement.)

C. It is therefore fair to compare PC to other architectures in those attributes where they differ and which also see wide multiprogramming application.

4.2 The Criteria and Data for the Comparison.

The first criterion for the comparison is that it be independent of the ingenuity of the hardware engineers e.g. speed. Clearly such devices as faster logic and memories are capable of making any comparisons based thereon vacuous.

A second criterion is that the comparison be independent of the ingenuity of the programmers and of the overall software environment in the computer. This is much more difficult to accomplish than hardware independence, and some amount of personal judgement was involved in deriving the data presented below.

On the basis of these criteria, it was decided to measure the amount and number of cycles of main store required to accomplish various primitive operations.

While the "number of MS cycles" measurement is reasonably intuitive, the "amount of MS required" deserves further clarification. In essence, it is the number of bits residing in MS which are required by the hardware to accomplish the given operation; the measure thus encompasses the bits in the instructions, tags, pointer words, etc. It does not, however, include the bits in the manipulated data item(s) encountered by the hardware, since these do not directly affect the semantics of the hardware operation.

With respect to the data actually gathered, several of the data reference

and transfer of control operations were drawn from constructs usually associated with Algol i.e. block structure. The two bases for this decision are (1) the PC by its nature has only block structure oriented operations and therefore this representents the only common ground for comparison with other architectures, and (2) given the underlying presumption of multiprogramming and the identity of the data/control structure arising in Algol and multiprogrammed environments (as discussed in chapter 2), measurement of operations within such a structure is meaningful.

With these considerations in mind, the block structure related operations in PC are associated with their counterparts in the Algol runtime packages of the S/360, PDP-10, and the measurements derived from examination of the assembly code. In order to make the comparisons valid, certain conventions were established:

For "Amount of MS Required!"

1. All data items are normalized to 32 bits (e.g. pretend that PDP-10 data is 32 bits, not 36).
2. All MS word addresses are normalized to 16 bits. (Thus the S/360, with byte addressing, has an 18-bit address field; the PDP-10 instructions are therefore 34 bits(not 36) since they have an 18-bit word address field).
3. Defaulted registers such as base registers used (always) for relocation purposes are not included.
4. Only those bits directly needed for the operation are included; fields, instructions, or registers used for error checking or detection (e.g. the length field in a PC descriptor), or formally unused (blank fields in a word) are excluded.

For "Number of MS Cycles Required".

1. Any necessary set-up has been done e.g. registers already loaded or saved, indices stacked.
2. MS cycles for instructions are biased by the number which can be packed into a word on the machine in question e.g. on S/360 a BR instruction is 0.5 cycles. This bias is not

applied to data packed into a word since unlike an instruction word, the remaining contents of the data word will not be used.

3. Any object code not directly concerned with accomplishing the function e. g. testing for error conditions or other cases, is not included.

Example: Data Fetch via Simple Dijkstra Semaphore.

[Assume that the semaphore will be found "unlocked".]

IBM s/360		#bits to rep	#MS cycles
1.	Instruction to load semaphore word into register.	26	1
2.	Load the sem word itself (only the 18-bit address counts semantically).	18	1
3.	Instruction to add 1 to sem wd	13	.5
4.	Instruction to test sem $\neq 0$	13	.5
5.	Instruction to store sem back into memory	26	1
6.	Store the sem word to memory	--	1
7.	Instruction to load the datum	26	1
8.	Load the datum into register	--	1
		122	7

DEC PDP-10			
1.	Instruction Replace Add 1 and Skip if$\neq 0$.	34	2
2.	Instruction to load datum	34	1
3.	Load datum into register	--	1
		68	4

PC

1.	Instruction to load a datum	16	.5
2.	Sem tags (=4)+ ptr type (=3)+ [s,d] address (=17)	24	1
3.	Load the sem word	--	1
4.	Replace the sem word	--	1
5.	Load the datum	--	1
6.	Push the datum	--	1
		40	5.5

4.3 Comparison of Data Fetch Operations.

Table 4-1 lists the measurements for the three machines with regard to data fetches. It should be noted that any judgemental considerations in these derivations were biased to show the S/360 and PDP-10 to their advantage. Figure 4-1 is a graph of the "bits to represent" versus the "number of MS cycles" from the table, and characterizes the differences among the three architectures.

The graph shows that the two register machines require generally more bits to accomplish their function, while the PC pays for its compactness with generally more memory references. The PDP-10's rich instruction set allows most operations to be accomplished with only a few instructions, but this flexibility is achieved at the expence of a huge menu of instructions and having only full word operators. The S/360 exhibits a more 'middle of the road' approach and appears to pay for its compromise in both bits and storage cycles for most nontrivial operations.

An ideal computer would have all of its data points as near as possible to the origin i.e. taking as little memory and memory cycles as possible. A simple computing of the center of gravity of each machine's graph about the origin yields 110, 252, 257 for PC, S/360, and PDP-10 respectively. One could therefore naively conclude that the PC is about twice as close to the ideal as the others. If, for example, the lesser used reference types (nos. 3, 6, 7) are weighted a tenth as much, the moments are 76,

	PC		S/360		PDP-10	
	# bits	# cycles	# bits	# cycles	# bits	# cycles
1. Data Fetch -via pointer	40	3.5	122	4	122	2
2. Data Fetch -simple (word)	16	2.5	26	2	34	2
3. Data Fetch -via semaphore	40	5.5	122	8	122	4
4. Data Fetch -8 bit byte	41	4.5	26	2	68	3
5. Data Fetch -via indexing	39	4.5	44	2	50	2
6. Data Fetch -via double indexing	62	6.5	120	4	152	4
7. Data Fetch -1 bit byte	41	4.5	130	5	68	3
8. Data Fetch -simple (value) parm ref	16	2.5	792	76.5	68*	3*
9. Data Fetch -simple (name) parm ref	42	3.5	536	55	136*	5*

*best estimates - data not available

Table 4-1

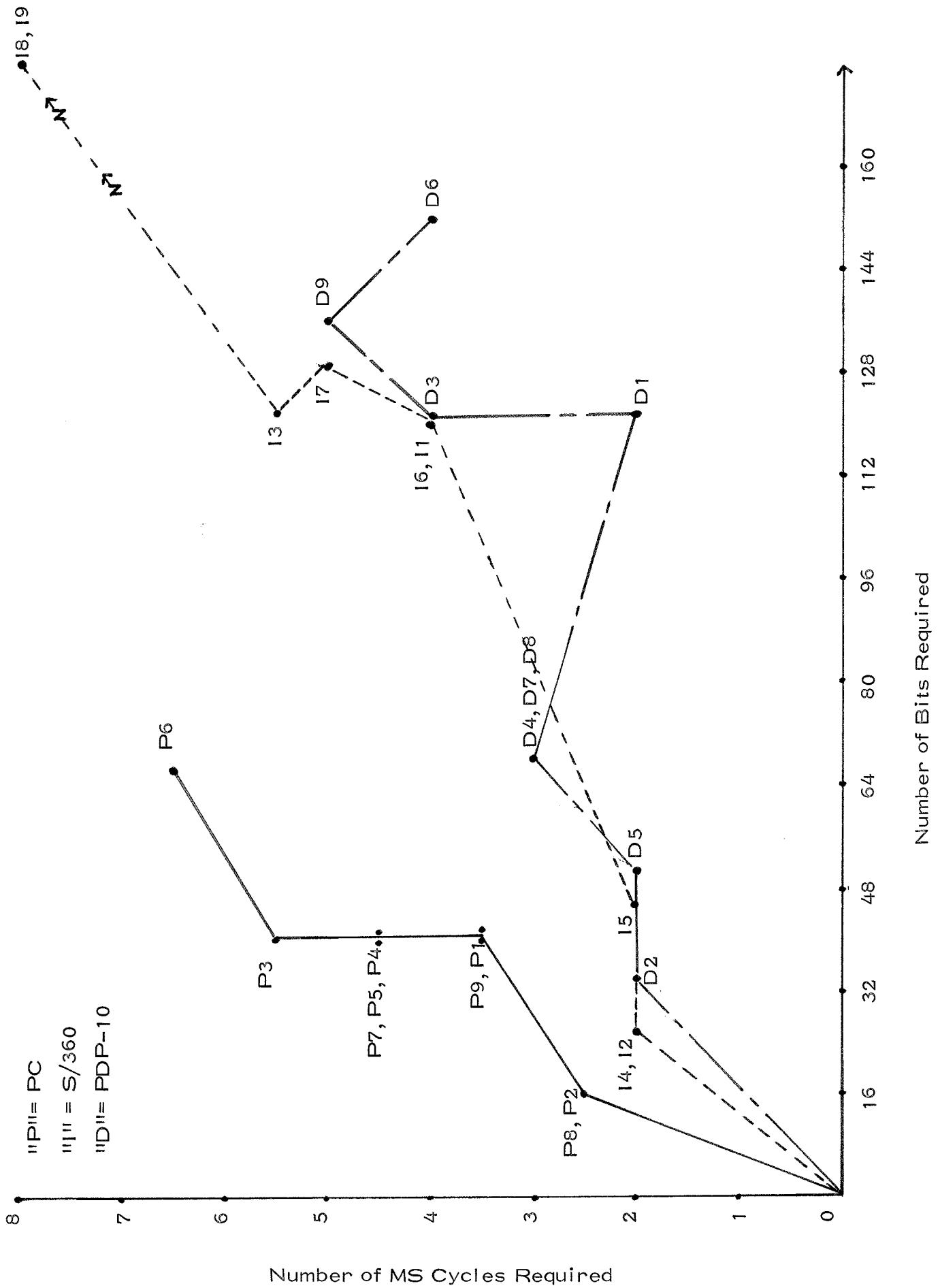
Data Fetch

Measurements for Several Machines.

Figure 4-1

Memory Cycles us. Memory Bits:

Data Fetch Operations.



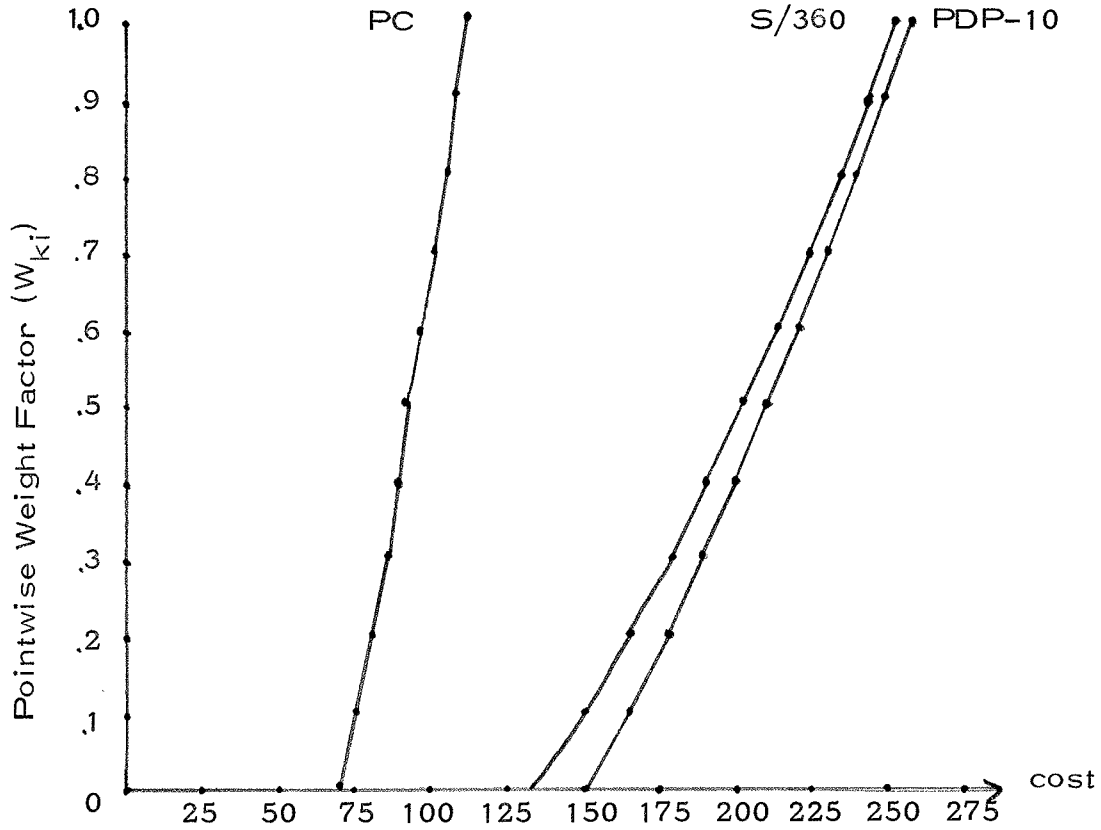
152, 167. Figure 4-2 shows the moment relationships as this weighting factor moves from 1.0 to 0.0, and we can therefore see that such "unusual" operations have no real effect on the relative comparison scale: the factor of two still remains.

Another way of looking at the significance of these data is to decide on the importance of consuming storage versus consuming storage cycles. Figure 4-3 shows the effect on the moment depending on this relative importance from which we can see that for the quadratic cost function considered here, once again the relative rankings of the machines remains unaffected until minimization of storage cycles becomes of paramount importance.

At this point one might raise the objection that the particular cost function used [relating storage and storage cycles] is not necessarily the 'correct' one i.e. perhaps some other cost function would exhibit strikingly different behaviour. Therefore a number of other possible cost functions were tried, with their only common characteristic being that the increase in favor of minimizing storage cycles be complementary to the decrease in weight on storage bits. The results, displayed in Figures 4-4abc....h, indicate that these cost functions exhibit almost exactly the same characteristics as the more intuitive second moment used in Figure 4-3.

Summarizing the various considerations of the preceding paragraphs, we see that:

1. The PC has a generally more compact representation for the various types of data reference.
2. This compactness advantage remains even after the less common data reference types have been eliminated.
3. The issue of storage versus storage cycles indicates that one must place all emphasis on minimization of storage cycles before the relative distances between PC and the other machines changes.
4. Regardless of the outcome of #3, PC's virtual memory tends to eliminate pressure for storage, but more importantly, since it should by now be clear that the vast



$$\text{Cost} = n \times sc = \sum_{i=1}^{n} w_{ki} \sqrt[2]{M_i^2 + B_i^2}$$

where

n = number of points

$$w_{ki} = \begin{cases} 1 & i = \{1, 2, 4, 5, 8, 9\} \\ k/10 & \text{otherwise and } k = 0, 1, 2, \ldots\ldots 10 \end{cases}$$

sc = point-weighted cost

Figure 4-2

Movement of Centroid for Data Fetch Operations

(with varying weight on less common operations)

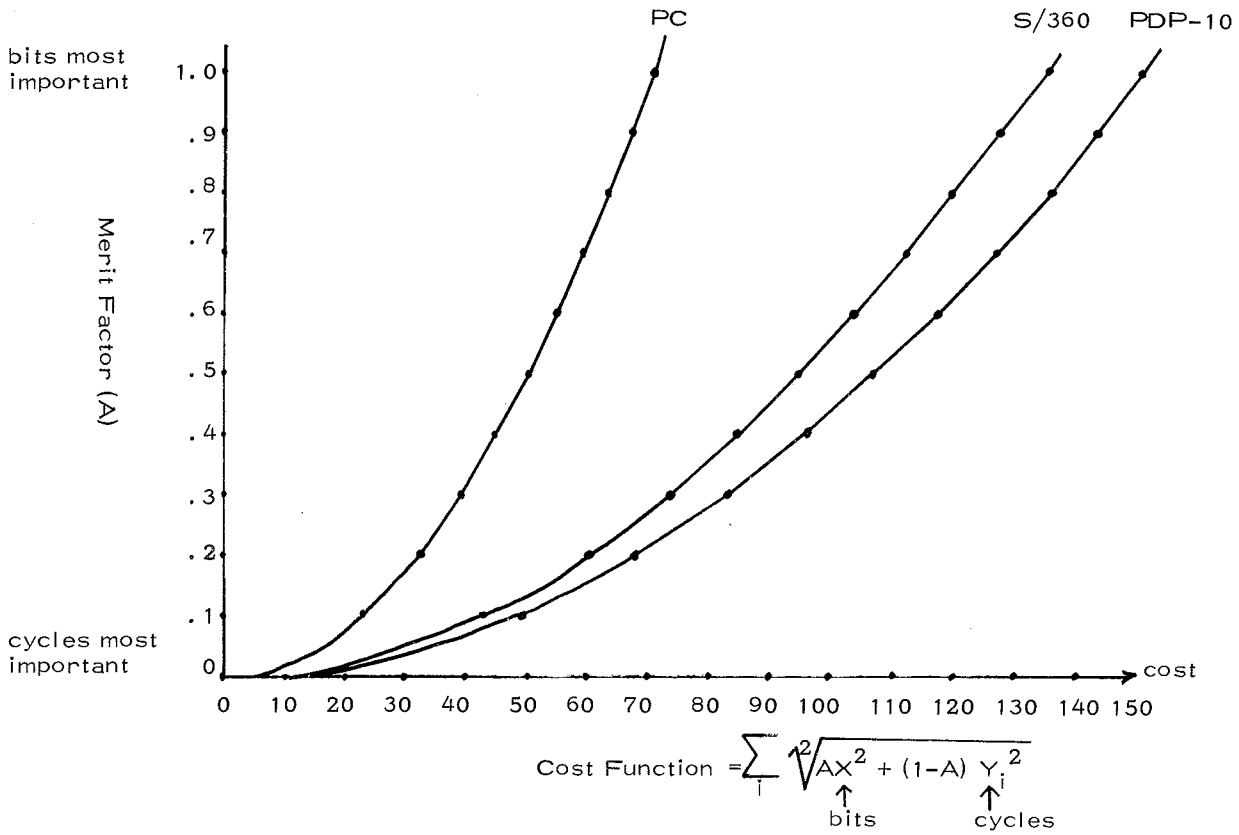
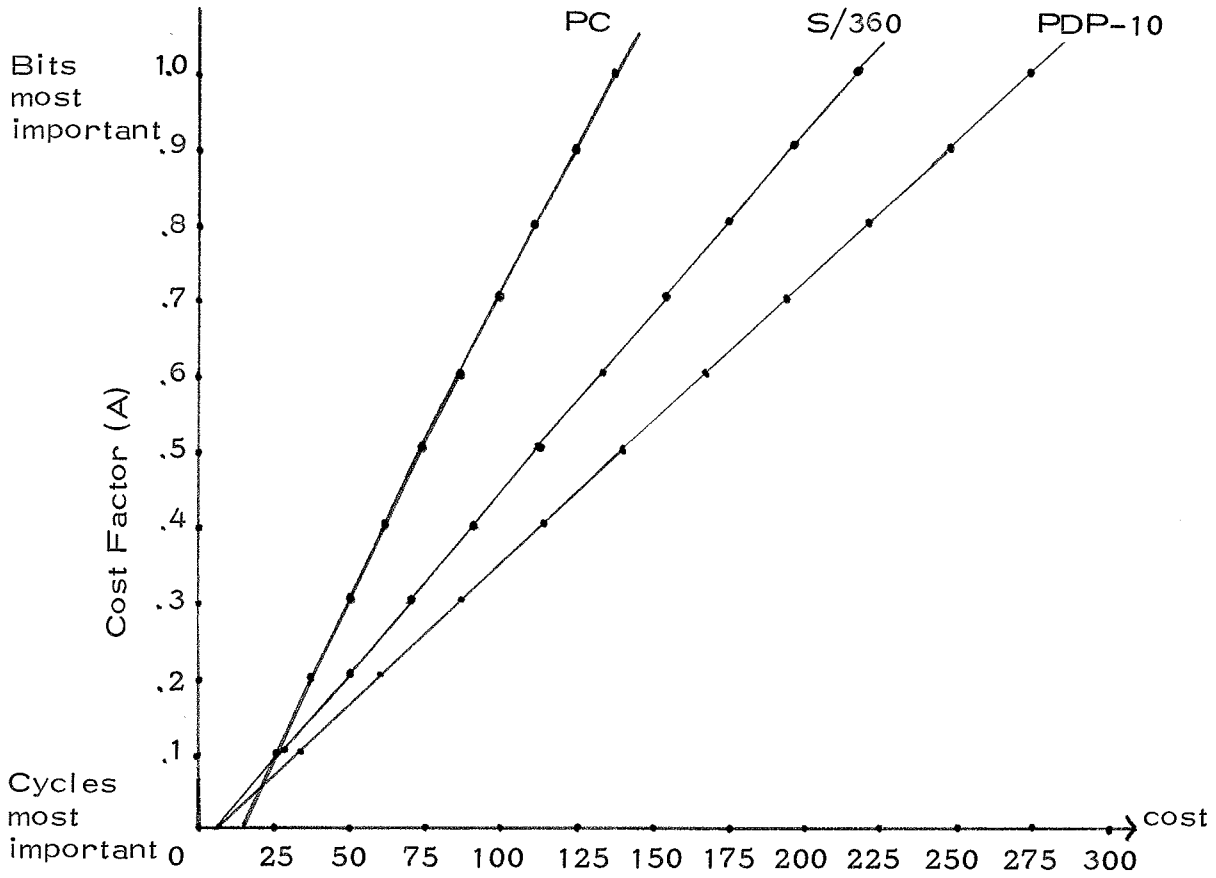


Figure 4-3

Centroid Movement for Data Fetch Operations

(with varying weight on storage vs. storage cycles)

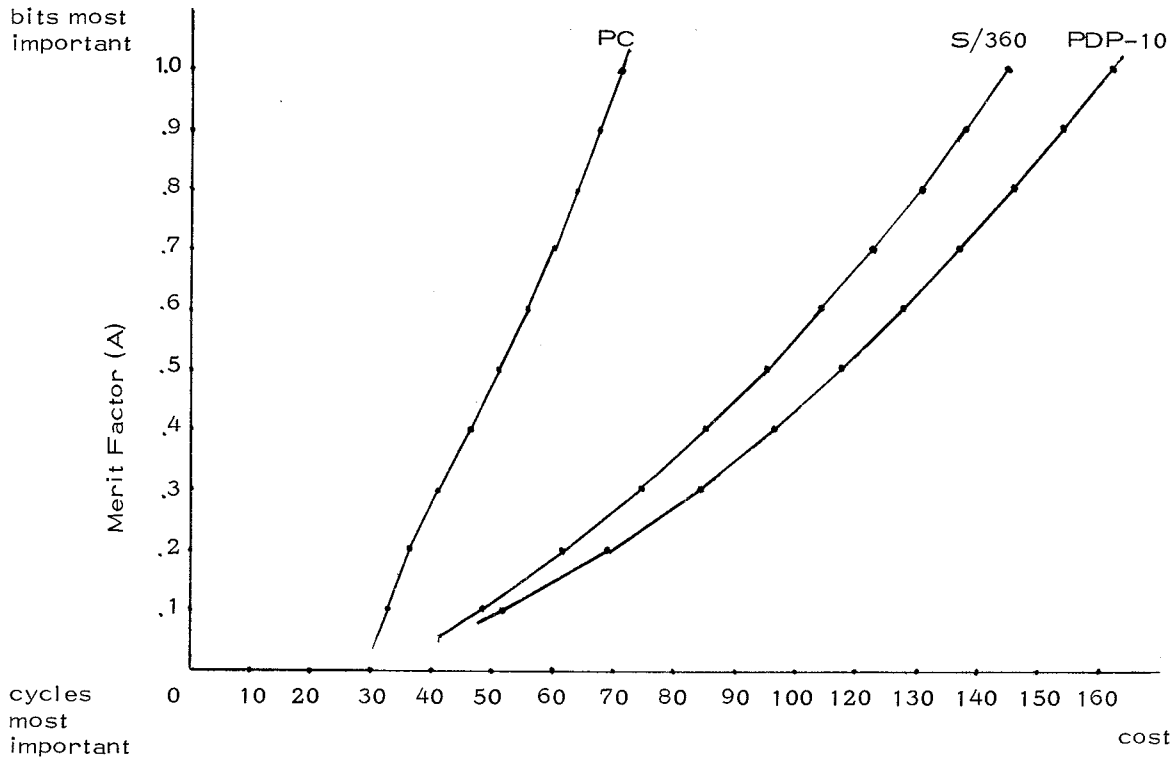


$$\text{Movement of Cost Function} = \sum_i (AX_i + (1-A)\ Y_i)$$

Figure 4-4a

Movement of Cost Function for Various

Types of Cost Function.



$$\text{Movement of Cost Function} = \sqrt[2]{\sum_i (AX_i^2 + \frac{1}{A} Y_i^2)}$$

Figure 4-4b

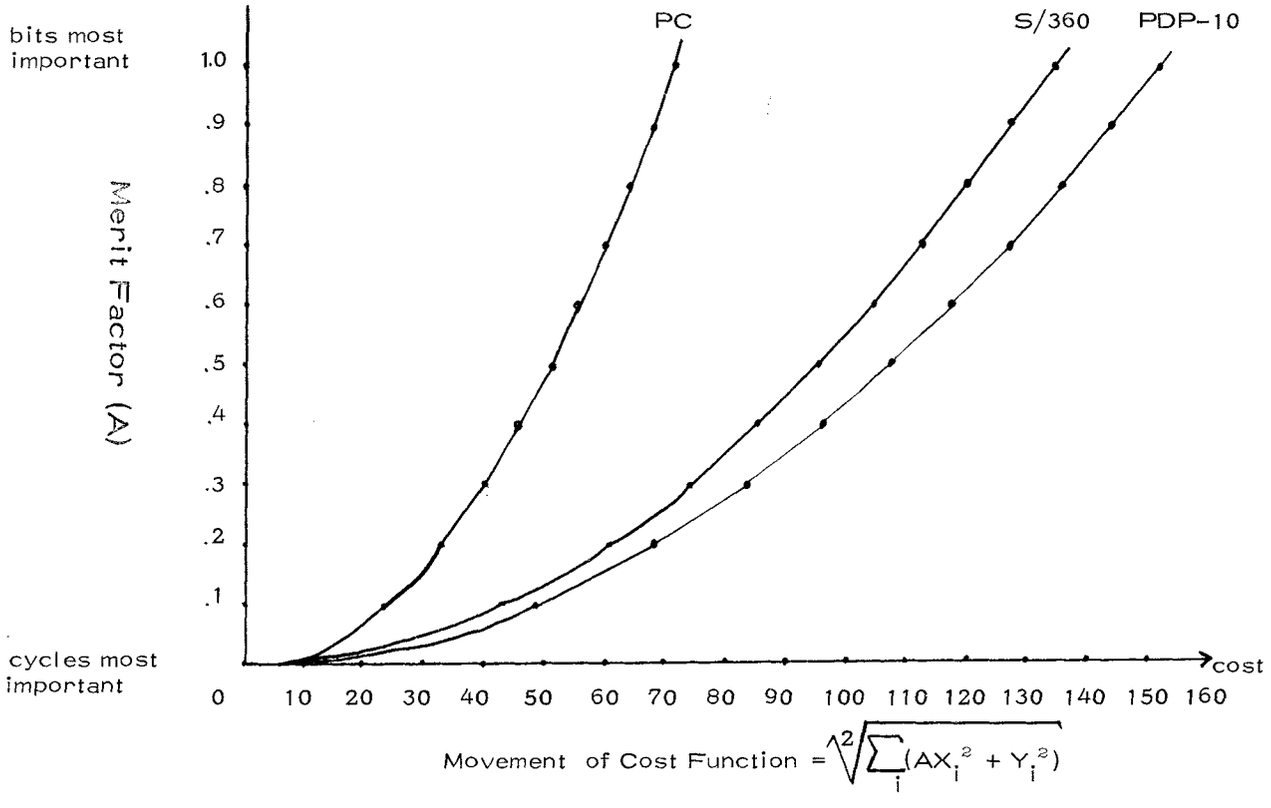

bits most important
Merit Factor (A)
cycles most important
1.0
.9
.8
.7
.6
.5
.4
.3
.2
.1
0
10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160
cost
PC
S/360
PDP-10
Movement of Cost Function = $\sqrt[2]{\sum_i (AX_i^2 + Y_i^2)}$

Figure 4-4c

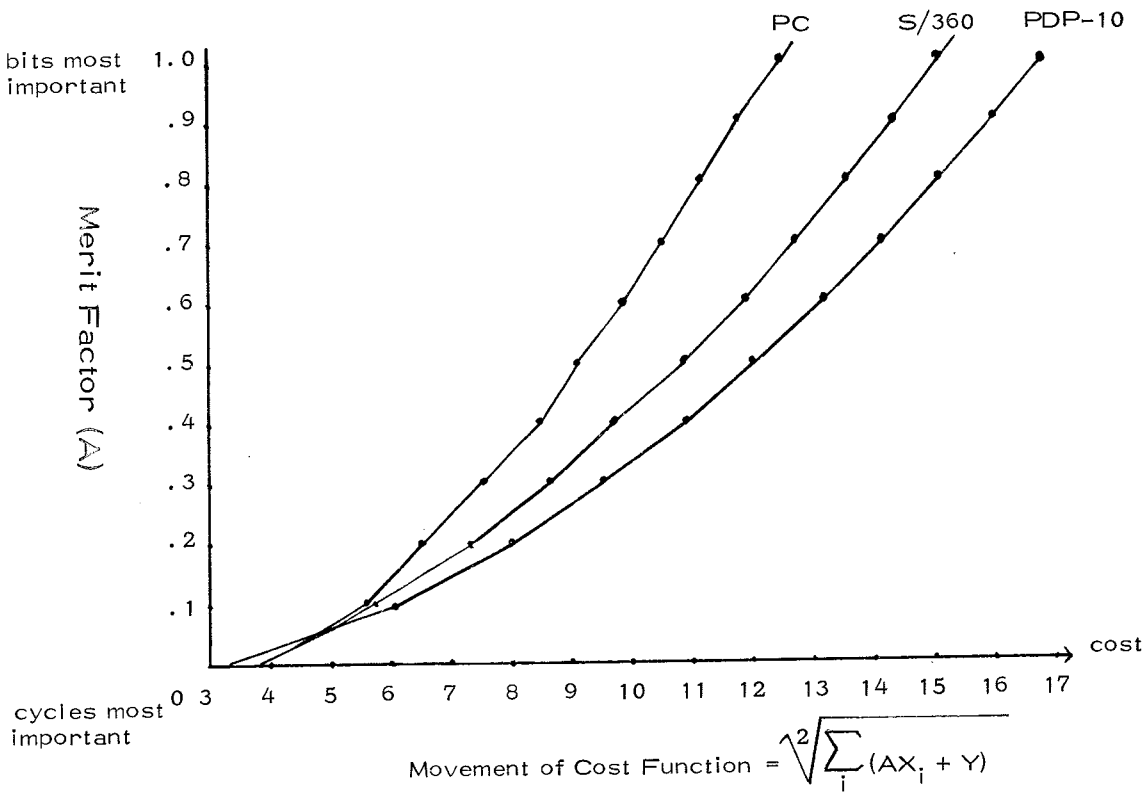

PC
S/360
PDP-10
bits most important
1.0
.9
.8
.7
.6
.5
.4
.3
.2
.1
Merit Factor (A)
cost
0 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
cycles most important
Movement of Cost Function = $\sqrt[2]{\sum_i (AX_i + Y)}$

Figure 4-4d

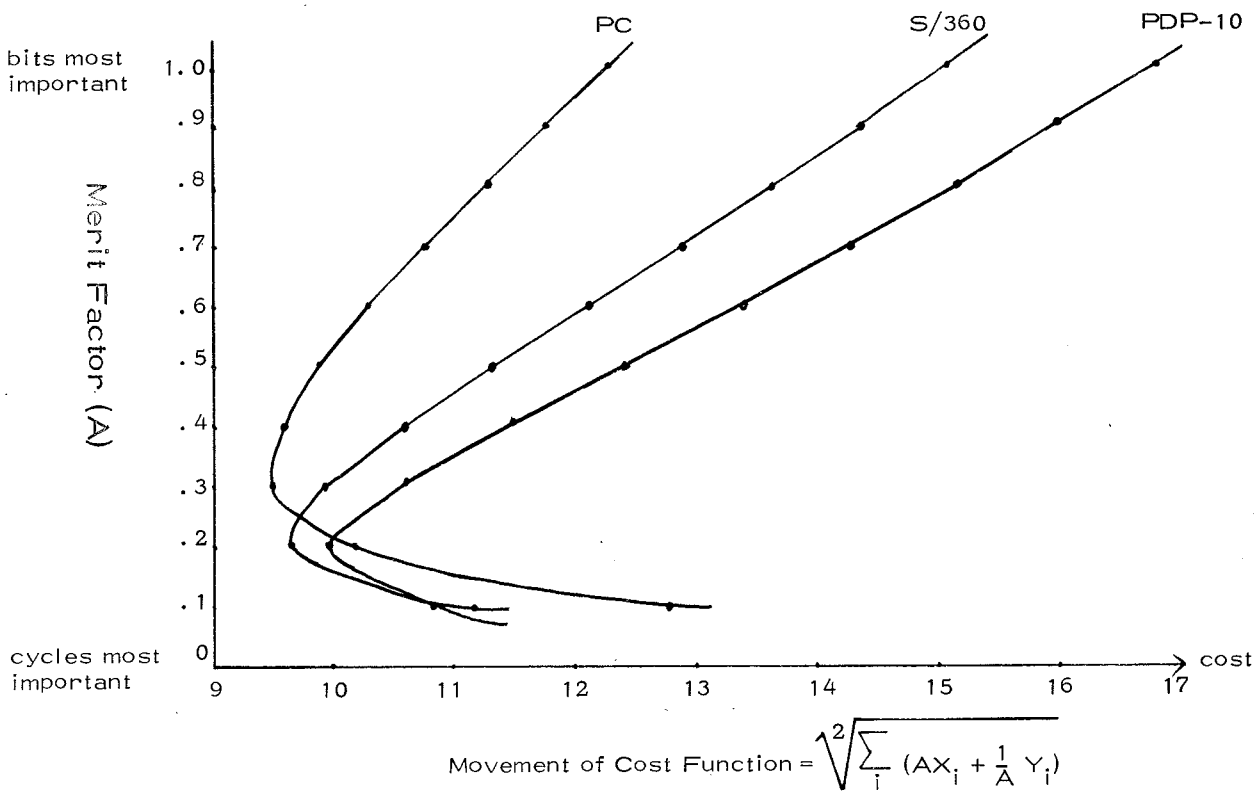

PC
S/360
PDP-10
bits most important
Merit Factor (A)
1.0
.9
.8
.7
.6
.5
.4
.3
.2
.1
0
cycles most important
9
10
11
12
13
14
15
16
17
cost
Movement of Cost Function = $\sqrt{\sum_i (AX_i + \frac{1}{A} Y_i)^2}$

Figure 4-4e

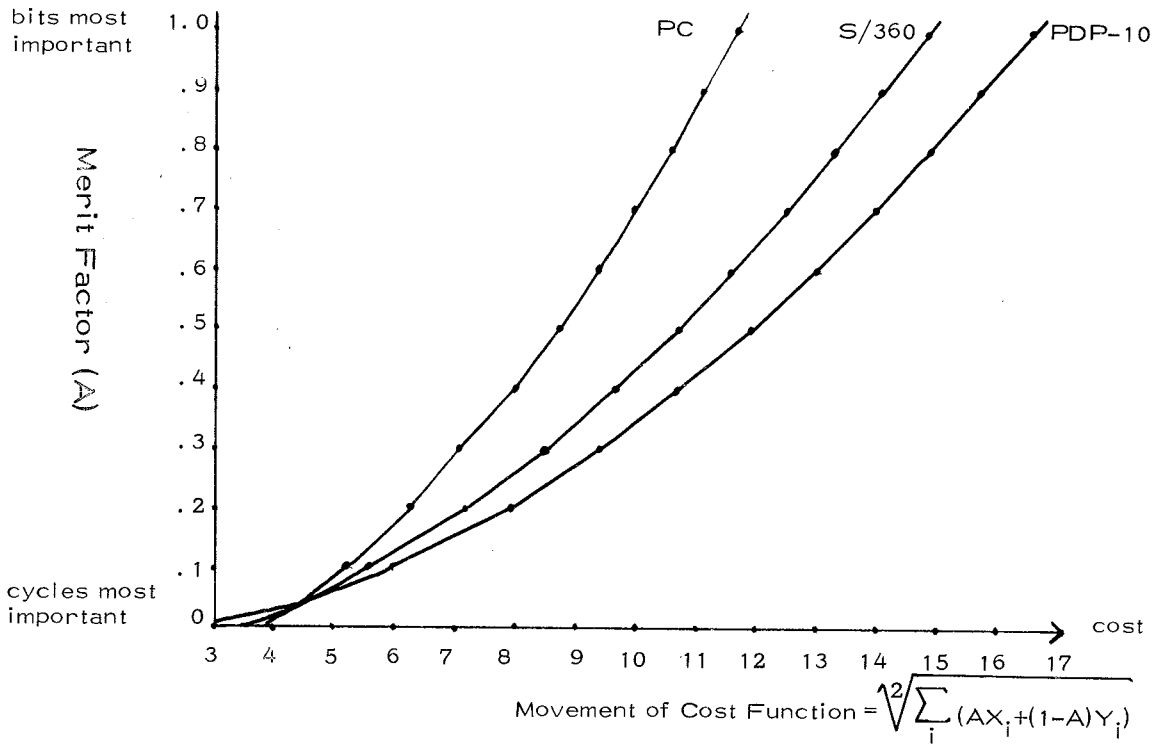
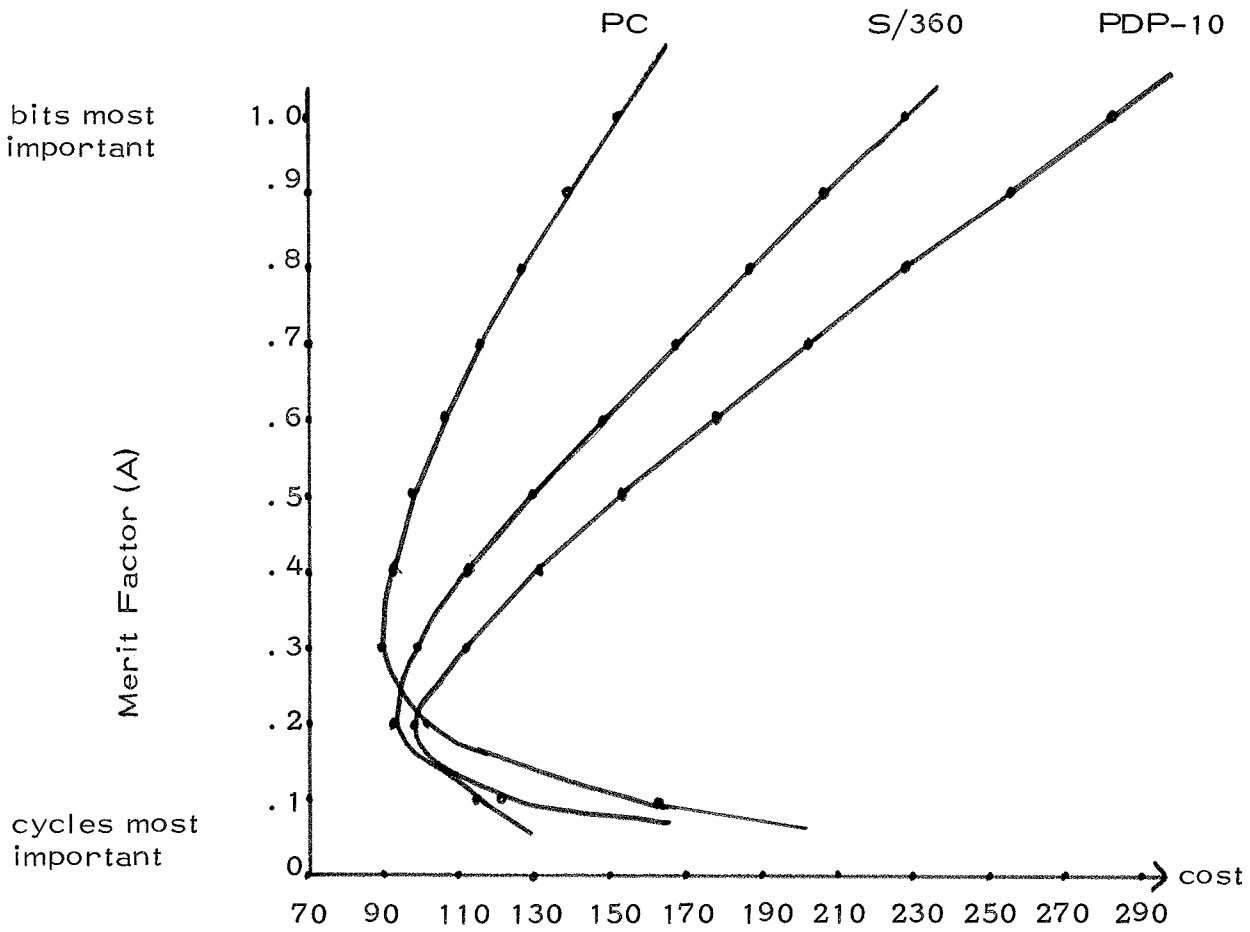

bits most important
1.0
.9
.8
.7
.6
.5
.4
.3
.2
.1
cycles most important
0
Merit Factor (A)
PC
S/360
PDP-10
cost
3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
Movement of Cost Function = $\sqrt[2]{\sum_i (AX_i+(1-A)Y_i)}$

Figure 4-4f



Movement of Cost Function= $\sqrt[2]{\sum_i (AX_i + \frac{1}{A} Y_i)}$

Figure 4-4g

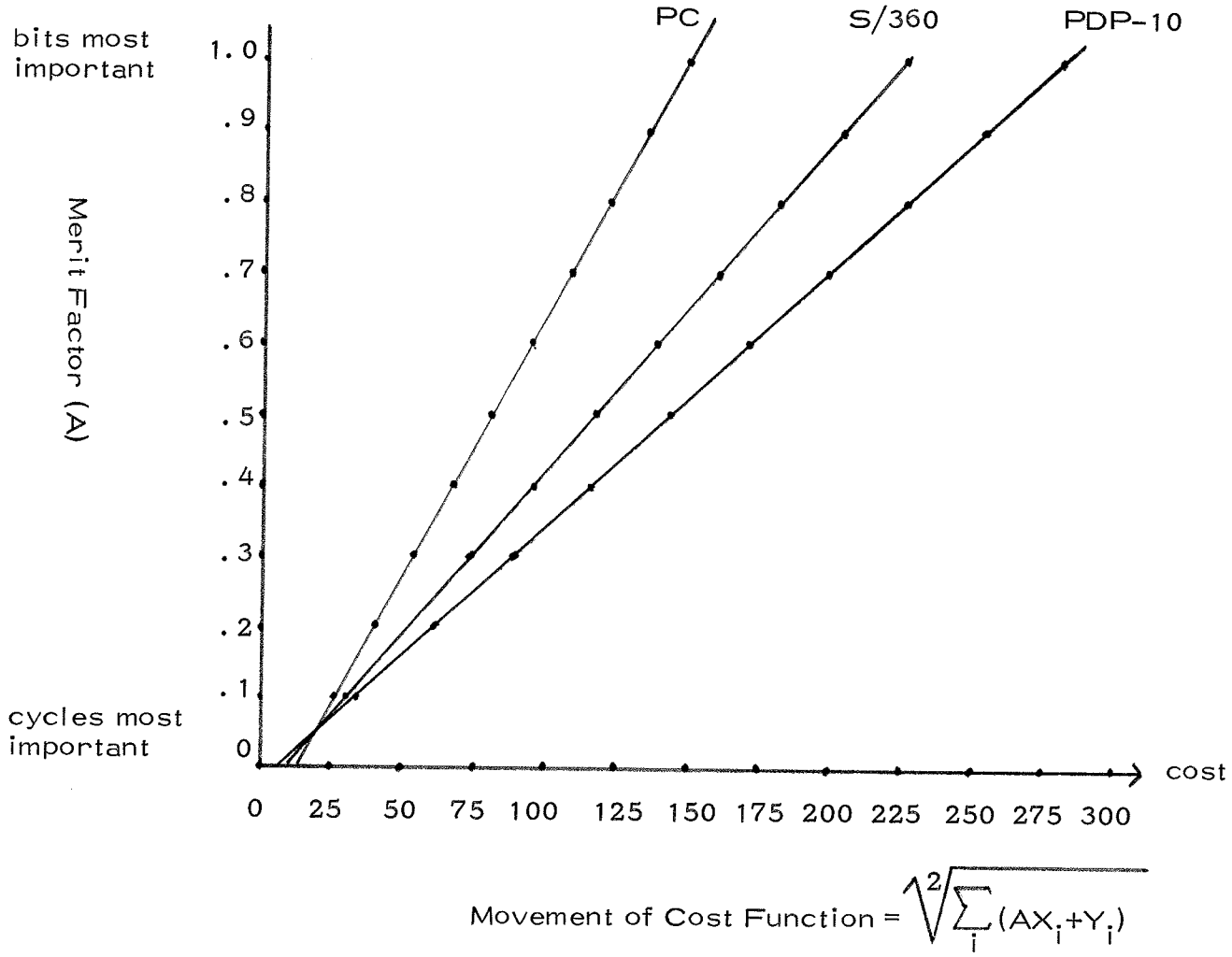

PC
S/360
PDP-10
bits most important
Merit Factor (A)
cycles most important
1.0
.9
.8
.7
.6
.5
.4
.3
.2
.1
0
cost
0 25 50 75 100 125 150 175 200 225 250 275 300
Movement of Cost Function = $\sqrt[2]{\sum_i (AX_i+Y_i)}$

Figure 4-4h

majority of data references (perhaps 99%) are to the user's stack, movement of the current stack to a cache memory would immediately relieve contention for storage cycles. The reader should also note that moving a single stack to a cache is considerably more straightforward than a generalized cache such as found on S/360.85 .

4.4 Comparison of Transfer of Control Operations.

Table 4-2 lists the measurements derived for various types of transfer of control operations, from simple branches to block entry. Figure 4-5 graphs (once again) "MS cycles" versus "Number of bits to represent the function" on log-log axes. Even with this compression, the quantum jump previously noted in Figure 4-1 is quite evident as the function moves from the very simple to the complex.

Both the PDP-10 and S/360 are highly optimized for those functions which minimally impact the environmental context (see [4.6]). The PC exhibits about a 50% greater compactness for these same operations but a slightly higher storage cycling rate due to the indirect referencing through the code page descriptor.

The major point of interest is the PC short (local) branch which uses only 16 bits and 0.5 MS cycles. Empirical data gathered by Saal [S2] using a 7094 emulator on an IC-6000 establishes that 70% of all branches are ± 32 words from the current point of execution, and virtually all branches are within about 250 words. One can interpret this phenomenon to mean that code bodies tend to represent a localized function and the longer branches are only used to move the locus of execution from one function to another. Hence the PC is nicely optimized to take advantage of this characteristic of program behaviour.

If we once again consider that the origin of Figure 4-5 represents the ideal of no storage and no storage cycles for any function, then Figure 4-6 shows that the migration of the center of gravity of the architectures as the less common transfer types are weighted less in such that PC is

	PC		S/360		PDD-10	
	# bits	# cycles	# bits	# cycles	# bits	# cycles
1. Block/Proc Entry	105	3.25	368	40.5	175*	5*
2. Block/Proc Exit	105	3.25	224	17	175*	5*
3. Long Branch (unconditional)	43	1.25	64	1	50	1
4. Short Branch (unconditional)	16	.5	48	.5	50	1
5. Simple Subroutine Entry	33	2.25	48	.5	50	1
6. Simple Subroutine Exit	33	1.25	48	.5	50	1

* best estimates - hard data not available

Table 4-2

Transfer-of-Control Measurements for Various Machines.

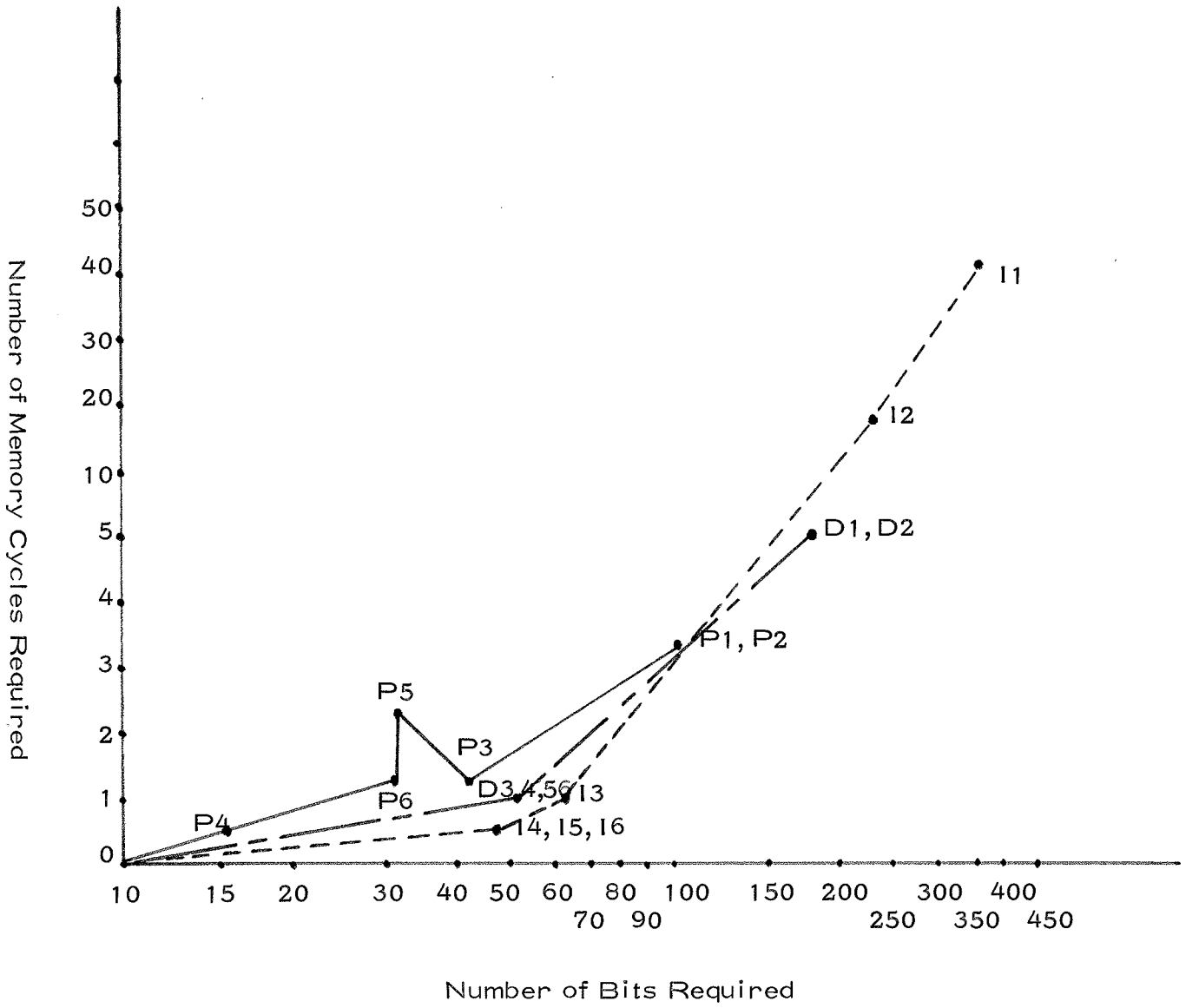
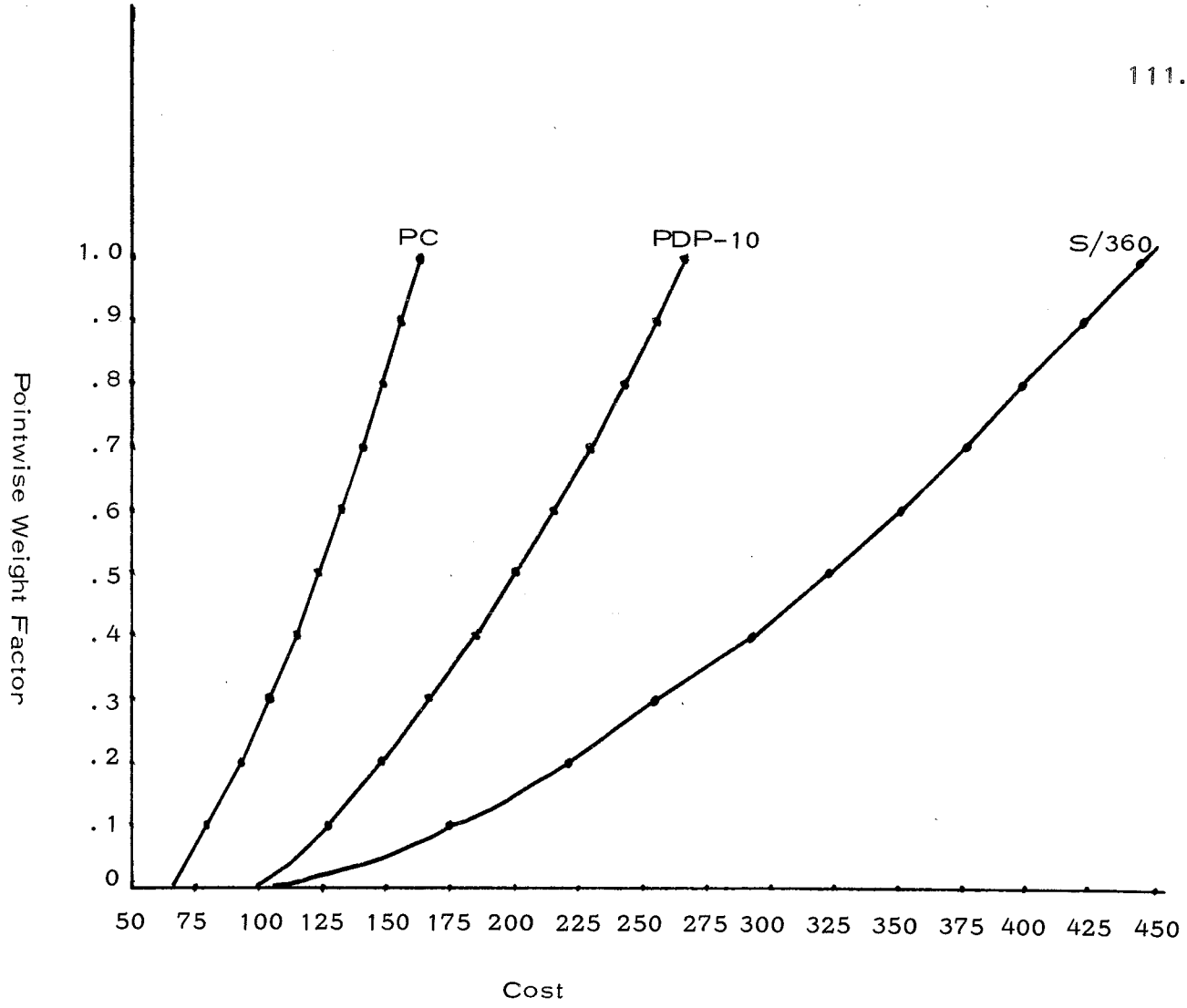


Figure 4-5

Memory Cycles vs. Memory Bits:

Transfer of Control



$$\text{Cost} = n \times sc = \sum_{i=1}^{n} w_{ki} \sqrt[2]{M_i^2 + B_i^2}$$

where

n = number of points

$$w_{ki} = \begin{cases} 1 & i = \{3, 4, 5, 6\} \\ k/10 & \text{otherwise} \end{cases} \quad \text{and } K = 0, 1, 2 \ldots\ldots 10$$

sc = point-weighted cost

Figure 4-6

Movement of Centroid
for Transfer of Control Operations.
(with varying weight on less common operations)

again closer to the ideal by a factor of two. As we did for data references, we can examine the movement of the moment based on the relative importance of storage versus storage cycles; the results of this exercise are comparable to those of Figures 4-4abc.....h, but not shown.

4. 5 Information Theory Approach.

Information theory concerns itself with [among other things] the probability of occurrence of certain events as opposed to their meaning. In the case of computer architecture, we can consider the instructions to be events ("messages"), and now we can examine the redundancy of these messages. Clearly, an architecture which minimizes this redundancy will show superior utilization of memory, and spend less time performing memory fetches.

In view of the fact that at the time of this writing, sufficient code bodies by which to measure the necessary parameters for the PC do not exist, we will now summarize a similar analysis performed by Wilner [W3] for the Burroughs 1700 computer [B2, W2].

The implementations of a number of different types of processors on the B1700 were subjected to an information theoretic analysis, with the result that the emulated instruction repertoires were found to be nearly minimally redundant. This compaction was achieved by Huffman coding traded off against consideration of instruction decoding efficiency, and some rather impressive performance improvements were noted: "From a set of twenty ANSI COBOL programs of diverse application and varying size, we have concluded that COBOL programs tend to occupy 70% less memory on the B1700 than they do on a S360/30. Such a drastic reduction in the memory also improves running speed, which averages about 60% faster than the 360/30. The B1700, when interpreting its COBOL S-machine, even seems to out-do the B3500 system, whose hardware was designed to execute and compile COBOL programs. Program storage requirements are 60% less, and execution times are comparable". [W3].

The graphs and arguments of the preceding section take on added significance in the light of this analysis.

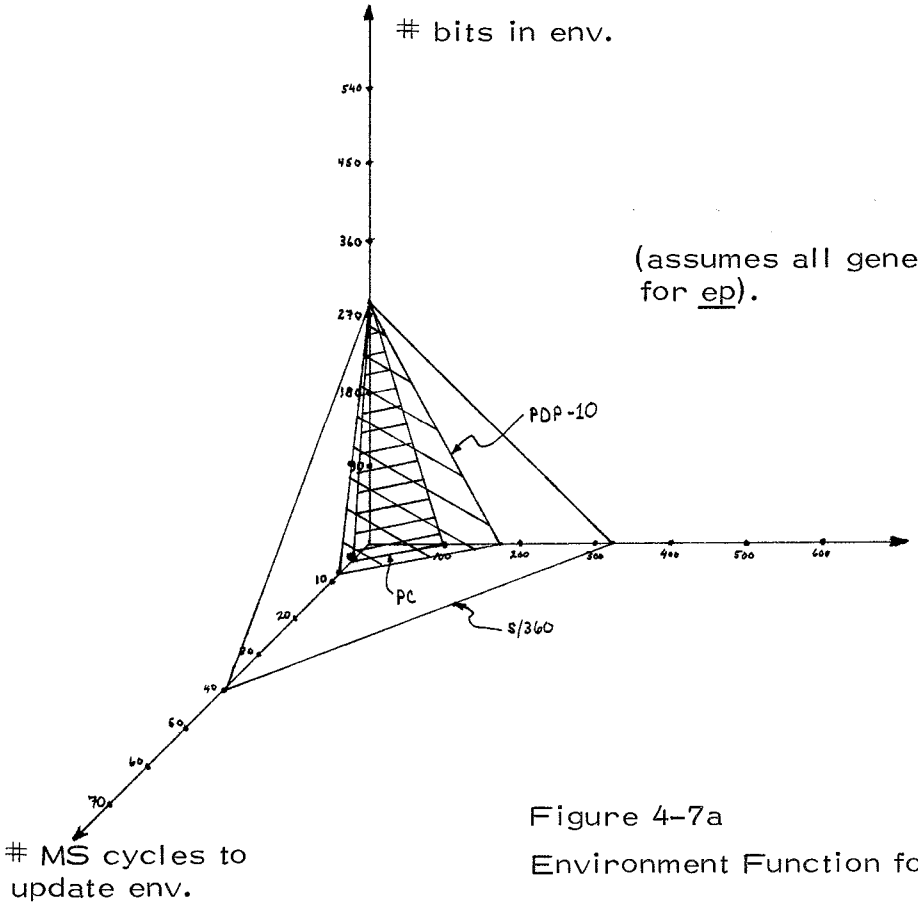
4.6 The Environment Pointer (ep).

The environment pointer (ep) is a conceptual construct [J1] which encompasses all the addresses which together define the data accessing environment at a particular point of execution in a program. When combined with the ip (instruction pointer), the ep + ip completely define the execution state of a program. This section analyzes the ep concept with respect to the three architectures.

The starting point of the analysis is to calculate the size of the ep for each of the three machines. For the PC, the endeavor is straightforward: the display registers plus TOS and MKST for a total of 18 registers times 16 bits/register = 288 bits. However the S/360 and PDP-10 present a more difficult problem since most of their registers do not have a hardware defined function. Clearly, any registers used as base registers belong to the ep, but in general the usage to which the general registers are put by the programmer or compiler is unpredictable. If all the registers are indeed used to hold data area base addresses, then the entire register file belongs to the ep, and on this basis, the ep for S/360 is 18 b/r × 16r = 288 bits; for the PDP-10 it is this same 288 plus the user level relocation registers = 304 bits.

It should be intuitively apparent that the richness and extent of the data accessing environment formed by the registers has a direct effect on program structure and efficiency. Pursuing this thought, it follows that the ability to manipulate the environment in a meaningful and efficient manner is also important. Returning to the "ep size" calculations, it would appear that all three machines possess about the same size ep. However this is true only if all the general registers are used for the ep, a situation which would heavily impact the efficiency of the PDP-10 and S/360. Hence the effective ep for these machines is much smaller than the PC's, yet the manipulation of this ep in a multiprogrammed environment, i.e., block entry, is much poorer. Figure 4-7 illustrates this situation by charting a function we call the "environment function".

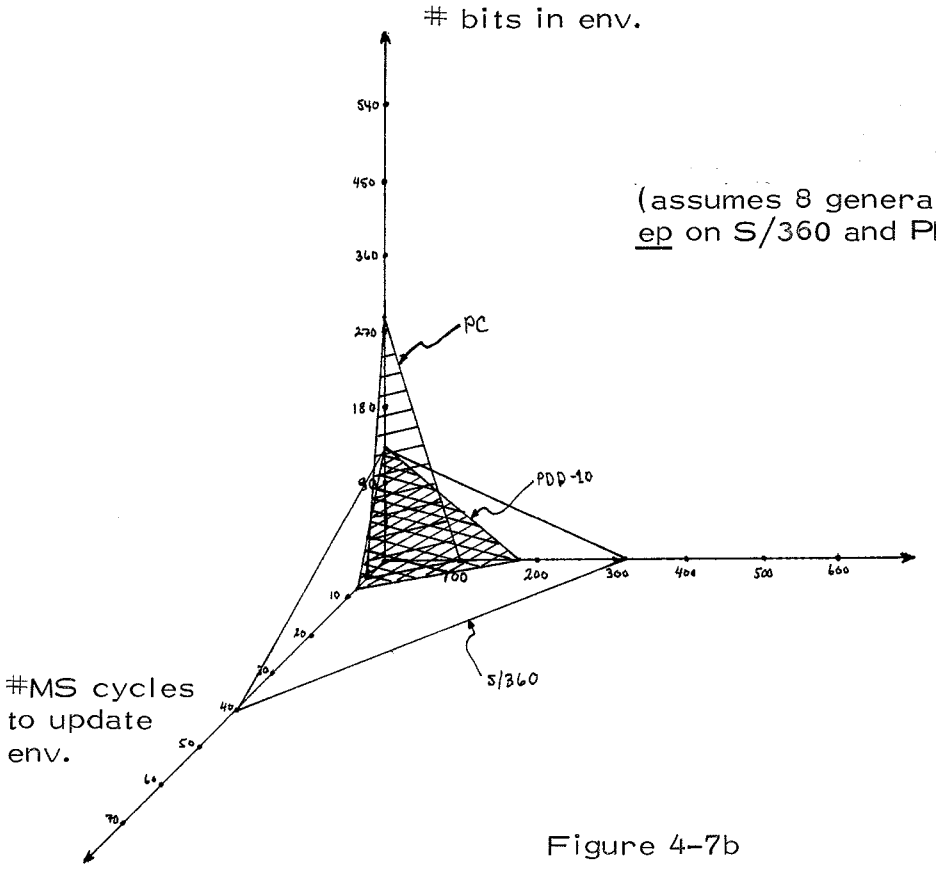
Env Fcn = F1 (size of env) × F2 ("cost" of maintaining this env)
= f (#bits in env, # MS cycles to update, #bits to specify update).



(assumes all general registers used for ep).

Figure 4-7a

Environment Function for PC, S/360, PDP-10



(assumes 8 general registers used for ep on S/360 and PDP-10).

Figure 4-7b

Environment Function for PC, S/360, PDP-10

Clearly the ideal situation is to have an immense ep which costs nothing (in cycles or bits) to update. When we examine Figure 4-7 with this reasoning in mind, we can see that the PC comes much closer to achieving this ideal than either the S/360 of the PDP-10. It would also appear that the PDP-10, on this point at least, represents a better set ofarchitectural decisions than S/360.

4.7 Conclusion.

This chapter has compared the PC architecture to those of the IBM S/360 and the DEC PDP-10. This comparison was based on data derived from examining how each of these machines accomplishes data referencing and transfer of control, particularly in a block structured environment. This type of environment was selected because (1) it is the only common ground between PC and the other machines, and (2) it is the same environment as that required for multiprogramming.

The argument made is that efficient operation in a block structured environment is desirable, and the PC exhibits superior performance in both data referencing and transfer of control. This superiority is maintained even when the less common and generally more complex operations are not considered; and also when memory space is traded off against memory cycles.

The third point was that if one considers an instruction to be a message and measures the redundancy of this message, significant gains in speed and storage can be achieved by minimizing said redundancy. The PC design embodies this concept in two ways: (1) instructions with zero-address and no register fields, and (2)[b, d] address which are shorter (yet capable of addressing the same or larger space) than the address fields of S/360 or PDP-10.

The fourth area of analysis was that of data access environment. As in the preceding arguments, the concept of an ideal computer which consumes no storage or storage cycles was postulated. The PC's generally larger ep was shown to be much closer to this ideal than either the S/360 or the PDP-10 when one consideres the cost of maintaining the ep.

5. 0 MAJOR CONCLUSIONS AND FUTURE DIRECTIONS.

The preceding chapter presented an abstract argument in favor of block structured as opposed to (traditional) general register computer architecture. The data presented showed that block structured architecture has distinct advantages in the basic areas of data reference, transfer of control, and data access environment.

One should not lose sight, however, of the other features of this type of design, in addition to these measurable aspects. The presence of automatically reentrant code, efficient recursion, architecturally supported debugging aids, logical and time independent I/O, the semaphore constructs, and variable-size-page virtual memory argue strongly in favor of the PC's being an excellent host machine for contemporary (i. e. complex) software systems. Construction of an (enhanced) XPL compiler and an operating system (in XPL) for the PC have benefitted greatly from the architecture, as evidenced by the compactness of their code and general design cleanliness.

On the other side of the coin, however, there are several facets of the PC's architecture which require further development:

1. Pointer Structure. The (i, s, d) relocatable pointers are adequate as long as pointers into only zero-or one-dimensional structures are needed. However, the current design does not support pointers to elements of multi-dimensional structures, and extension of (i, s, d) to (i1, i2, ... in, s, d) does not seem very attractive on the basis of both storage and storage cycles. Whatever design is proposed will have very direct impact on the virtual memory management as well.
2. Stack Discipline. The PC currently supports only the traditional LIFO stack discipline, and such concepts as based or heap storage, retention of storage after block exit, and pointer reference counts are at present not supported by the architecture. A generalization of the role of the stack vector may provide a solution to the problems, and indeed such generalization is not obviously incompatible with the

stack-cum-retention strategies outlined by [B8] [B7].

3. I/O and Interrupts. The PC logical time-independent and memory-structure dependent I/O, and the handling of interrupts as global (block structure) procedure entries, seem to be a step in the right direction. There is still room for improvement in the direction of integrating I/O more completely into the 'event structure' as described in [O1] and implemented in the PC and B6700 operating systems.

4. Multiprogrammed Emulation. The Enter/Exit Emulator instructions described in Chapter 3 represent only a first step in this direction. At the current state of development, the PC architecture seems to be amenable to the inclusion of this type of function, but only further development will solidify or destroy this feeling. One matter which is clear is that the structure of the various microcode subsystems must be very well thought out if such capabilities are to be supported.

5. In spite of the object code compactness demonstrated in the previous chapter, there is still room for improvement. The minimal coding technigue of the B1700 (and the underlying hardware strategy which made it possible) should be viewed as major developments in the tuning of the hardware to meet software (instead of hardware)needs. There is also room to better tune instruction sets to fit the languages, examples being the B1700 and Doran's Tree Machine [D5].

5.1 Acknowledgements.

My sincerest gratitude to extended to my advisor and friend Prof. Robert F. Rosin; to Prof. Gideon Frieder for his many helpful ideas; to my colleagues and coworkers on the PC (alias Buffalo Stack Machine) project Mike Lutz, Rowan Snyder, Herb Kleinberger, Mike Brenner, and Denis Lynch; to the National Science Foundation; and to my former wife Bobbi who cared for and supported me through much of this work. Special mention should also be made of Arne Tolstrup Madsen and Edel Jensen without whose help this paper would never have been printed.

BIBLIOGRAPHY.

B1. Belady, L. A.: A study of replacement algoritms for virtual storage computers. IBM Syst. Jour. 5, 2 (1966), pp. 78-101.

B2. Burroughs B 1700 Manual.
Burroughs Corp., Detroit, Michigan.

B3. Burroughs B 5500 Information Processing System Reference Manual (1964)
Burroughs Corp., Detroit, Michigan.

B4. Burroughs B 6500 Information Processing System ESPOL Reference Manual
Burroughs Corp., Detroit, Michigan, 1970.

B5. Burroughs B 6500 Information Processing System Reference Manual (1969)
Burroughs Corp., Detroit, Michigan.

B6. Barton, R. S. Ideas for Computer Systems Organization: A Personal Survey. Software Engineering, Vol. 1, 1970. Academic Press, N. Y.

B7. Berry, etal. On the Time Required for Retention. ACM-IEEE Symposium on High Level Language Computer Architecture. SIGPLAN/SIGARCH/IEEE-TCCA Nov. 1973.

B8. Bobrow, D. G. and Wegbreit, B. A Model and Stack Implementation of Multiple Environments. CACM 16, 10. Oct. 1973.

D1. Denning, P. J.: Virtual Memory. Computing Surveys, vol. 2, no. 3, sept. 1970.

D2. Dijkstra, E. W.: Cooperating Sequential Processes. Technological University. Eindhoven, 1965.

D3. Dijkstra, E. W.: The Structure of the THE-Multiprogramming System. Comm. ACM 11, 5 (May 1968), pp 341-346.

D4. Dent, B. A. : Personnel Communication.

D5. Doran, R. W.: A Computer Organization with an Explicitly Tree-Structured Machine Language. Australian Computer Journal, Vol. 4, No. 1 , February, 1972, pp 21-30.

F1. Feustel, E. A.: On the Advanteges of Tagged Architecture, IEEE Transactions on Computers, vol. C-22, no. 7, July 1973.

G1. Gries, D.: Compiler Construction for Digital Computers. John Wiley and Sons, Inc. New York-London-Sydney-Toronto, 1971 pp 247-251.

H1. Hansen, P. B.: A Comparison of Two Synchronizing Concepts. Acta Informatica 1, pp 190-199, Springer-Verlag 1972.

H2. Hauck, E. A. and Dent, B. A.: Burroughs' B6500/B7500 stack mechanism. Spring Joint Computer Conference, 1968, AFIPS, pp 245-251.

H3. Haberman, A. N.: Prevention of System Deadlocks. CACM 12, 7 July 1969, pp 373-377.

H4. Hager, K.: Organization of Central Registers-A Comparison of Code Length Quality. ACM Conference Procedings 1970.

I1. IBM System/360. 360D-05. 2. 005 CP-67. CMS Program Logic Manual.

I2. Iliffe, J. K.: Basic Machine Principles. MacDonald-London and American Elsevier Inc. -New York. 1968.

I3. Ingerman, P. Z.: Thunks - a way of compiling procedure statements with some comments on procedure declarations. Comm. ACM 4, 1 (Jan. 1961), pp 55-58.

J1. Johnston, J. B.: The Contour Model of Block Structured Processes. Proc. of a Symposium on Data Structures in Pro-

gramming Languages. Ed. Tou and Wegner. ACM/SIGPLAN Feb. 1971.

K1. Knuth, D. E.: The Art of Computer Programming, vol. 1, Addison-Wesley, Reading, Mass., 1968, pp 435-455.

K2. Knuth, D. E. and Merner, J. N.: Algol 60 Confidential. (9. An Innerproduct Procedure). Comm. ACM 4, 6 (June 1961), p 71.

K3. Kuck, D. J. and Lawrie, D. H.: The use and performance of memory hieraachies: a survey. Software engineering(ed. Tou), vol. 1, 1970, pp 45-77.

L1. Lindsay, C. H. and van der Meulen, S. G.: Informal Introduction to Algol 68. North-Holland Publishing Co. Amsterdam, London. 1971.

L2. Lutz, M. J. and Manthey, M. J.: A Microprogrammed Implementation of a Block Structured Architecture. Fifth Annual Workshop on Microprogramming (ACM-SIGMICRO). 1972.

L3. Lutz, M. J.: The Design and Implementation of a Small Scale Stack Processor System. AFIPS vol. 42, 1973.

M1. McKeeman, W. M.: Language directed computer design. Fall Joint Computer Conference, 1967, AFIPS.

M2. Manthey, M. J.: Execution Snapshots of Program SAM. SIGPLAN Notices Vol 8, 7 July 1973.

N1. Naur, P.: Some uses of Jensen's device for Algol 60 procedures. Automatic Programming Information, no. 7(May 1961), pp 12-13.

N2. Newell, A. etal: Information Processing Language -V Manual 2nd Edition. The Rand Corporation. Prentice-Hall Inc. 1966 pp 68-72.

O1. Organick, E. I.: Computer System Organization-The B5700/B6700 Series. Academic Press, New York, London. 1973. (ACM Monograph).

R1. Randell, B. and Russell, L. J.: Algol 60 Implementation. Academic Press Inc.-New York, 1964.

R2. Rice Computer-2, General Specifications, Dept. Electrical Engineering, Rice Univ., Houston, Texas, Sept. 1970.

S1. SIGPLAN Notices, vol. 8, no. 5, 1973 July, pp 4-20, (letter by Mike Manthey).

S2. Saal, H. J. and Shustek, L. J.: Microprogrammed Implementation of Computer Measurement Techniques. Fifth Annual Workshop on Microprogramming (ACM-SIGMICRO). 1972.

W1. Wegner, P.: Programming Languages, Information Structures and Machine Organization, McGraw-Hill Book Company, New York, 1968, p 18.

W2. Wegner, P.: Programming Languages, Information Structures and Machine Organization, McGraw-Hill Book Company, New 1968 2, pp 8-23.

W3. Wilner, W. T.: B1700 Memory Utilization. Fall Joint Computer Conference, AFIPS, 1972, pp 579-586.

W4. Wilner, W. T.: Design of the B1700. Fall Joint Computer Conference, AFIPS, 1972, pp 489-497.

W5. Wirth, N.: On Multiprogramming, Machine Coding, and Computer Organization. Comm. ACM 12, 9 (Sept. 1969), pp 489-498.

W6. Wirth, N.: Stack vs. Multiregister Computers. SIGPLAN Notices, March 1968, pp 13-19.