PDP-K Technical Memorandum # 3-1

Title: Extension of the PDP-11 Address Space

Author(s): Robert Gray

Index Keys: Memory
Paging
Segmentation

Distribution Keys:

Revision: 1

Obsolete: Technical Memorandum #3

Date: March 12, 1970

ABSTRACT

This memo discusses the limited address space in the PDP-11 architecture.

The PDP-11 16-bit processor can address a maximum of 32K (indirect or indexed) 16-bit words. A 32-bit version of the PDP-11 could address 16K 32-bit words.

This aspect of the present PDP-11 architecture is examined as a potential problem in larger versions (32-bits) of a proposed PDP-11 family. The 16K limit is compared with medium size computers offered by other manufacturers. It is also compared with the expected user requirements of machines in this performance category. Finally, this memo discusses briefly the limit and advantages of two possible memory expansion techniques: Paging and Sgementation.

It is concluded that the 16K limit in a 32-bit PDP-11 would be a severe competitive handicap. It is further concluded that neither paging nor sgementation offers an efficient way to run procedures that exceed 16K 32-bit words.

Direct Addressing Capability of Other Computers

A survey was conducted to determine what direct addressing capability other manufacturers offered in their computers. The purpose of this was to give a perspective on the "competition" and to discover, through their hardware, their estimates of the amount of addressable memory needed in medium computers.

In a family of computers, it is the viability of the upper end processor (32 or 36-bit words) that would be most limited by insufficient address space. Hence, the survey covered only processors with word size above 16 bits.

Representative 18, 24, 32, and 36-bit processors are presented in the chart below, as well as the present PDP-11.

		MAXIMUM WORDS ADDRESSABLE		
Processor	No. Bits	Direct	Indirect	Indexed
PDP-11 (KA11)	16	8	32K	32K
PDP-15	18	4K	32K	128K
DPD224	24			64K
XDS 940	24	16K	16K	64K
Sigma 5 & 7	32	128K	128K	128K
System 86	32	128K	128K	128K
KP Omega	32	Not Available		
IBM 360 Series	32	4K	4000K (24 Bits)	4000K
Univac 1108	36	64K	64K (16 Bits)	256K
PDP-10	36	256K	256K	256K
GE 635/645	36	256K	256K (18 Bits)	256K

As can be seen in the chart above, the 16K of address capability in a 32-bit machine is far below that of other machines in this performance class. In the next section, we will examine the need for address capability greater than 16K. This will be broken down into two classes of systems: the single user system and the multi-user (time-sharing) system.

The "Market" Need for Address Capability Greater than 16K

It is simple to say, "everybody knows 16K is too small in a large machine." But it is much harder to determine the correct number.

Certain facts, however, tend to indicate larger computers need greater than 16K. Such system programs as the PDP-15 Background/Foreground Monitor "barely" runs in 16K. More orders are being received for PDP-15's with 32 to 64K. PDP-9 customers are asking how they can put more than 32K on their machines. There was such a market for the Ampex 128K memory system that the PDP-10 group was forced to market it. The PDP-10 time-sharing system requires about 48K to be "respectable" and the average PDP-10 system being sold today is ordered with 64K.

Single User Market

DEC's traditional market has been in the scientific market. This market often uses the real time capability of the processor and has very large problems: simulations, array manipulations, real time control, etc. It is this class of users who probably are most sophisticated in using the computer and in evaluating competing processors. They are also the group that is often under-funded and, hence, try to get every last bit of processing power out of their machine. In the future, this group can be expected to attempt to write ever larger, more complex programs as the complexity of the problems they try to solve increases.

Consider the implications of 16K of 32-bit words on a user doing matrix manipulations: assume that the user desires a high-precision solution and, hence, chooses to use a quadruple word floating-point format (64 bits) to store the matrix points. Assume that 8K of the 16K 32-bit addresses available to the user are to be occupied by procedure and the other 8K is to be occupied by 3 matrices (allowing [A] + [B] = [C]). He can then have a total of 4K (of quadruple words), or 1,333 words per matrix. This will allow him to have 36 X 36 matrices (1296 points per matrix). In certain classes of programs, this is not sufficient (e.g., linear and dynamic programming).

Paging can be used to extend the amount of core on the PDP-11 as is being done on with the KT11A Paging Box. However, as will be shown in the section describing paging, it will require many instructions every time a user tries to reference a location outside his 16K (and, assuming the monitor allows such) to accomplish the transfer.

Even with paging, a program greater than 32K would have to be split into blocks with an absolute minimum of cross references between the blocks to run efficiently. This makes it more difficult to write programs and results in very poor performance when they are finally operating.

The Conventional Time-Sharing Market

This market is characterized by a different set of requirements than the single user. In these systems all users are using high level languages. In these systems, no attempt is made to put all of a user's program in core at once. The core is allocated among many users and parts of a given user's program are moved into core as the program requests them. This technique obviously does not provide the real-time capability that a single user system can and a "job" in time-sharing mode (assuming that the job contains few I/O instructions) will take much longer to be executed than if the job had all of the computer to itself. A time-sharing system will often process a job for a finite time slice or until it is waiting for some I/O process. At that time, the computer transfers to another job and, if necessary, moves the first job from core onto some secondary storage, usually a high-speed disk.

It is usually assumed that to have an economically operating system, this swapping must occur as it cannot be economically justified to have enough fast memory to have all of the active jobs totally in core. This philosophy has led to studies to determine how to allocate the limited amount of core among many programs. The studies give the impression that the first rule is to attempt to divide core equally among the jobs -- given equality in the jobs. The result of this memory allocation is that program execution speed is limited by the sharing of facilities and by the time taken as users are swapped in and out of secondary storage.

In this situation, the fact that a program exceeded the direct address capabilities of the machine probably would less significantly increase the time it took to run (compared with a single user system), since the execution time is already limited by the program swapping, which is external to the program itself.

To allow programs greater than 32K, however, means that the compiler and monitors must be structured to allow writing programs larger than the address space and must break them up into parts that can be overlaid as the computation progresses. This is probably a very difficult task to accomplish and would make large complex programs even more difficult to document, debug and would certainly increase their time to run.

Paging and Segmentation

There are two well-known techniques to expand memory space. One, paging has previously been mentioned. This and the second technique, segmentation, will be explained and examined for possible ways to efficiently run procedures exceeding 16K on the PDP-11.

General Explanation of Paging

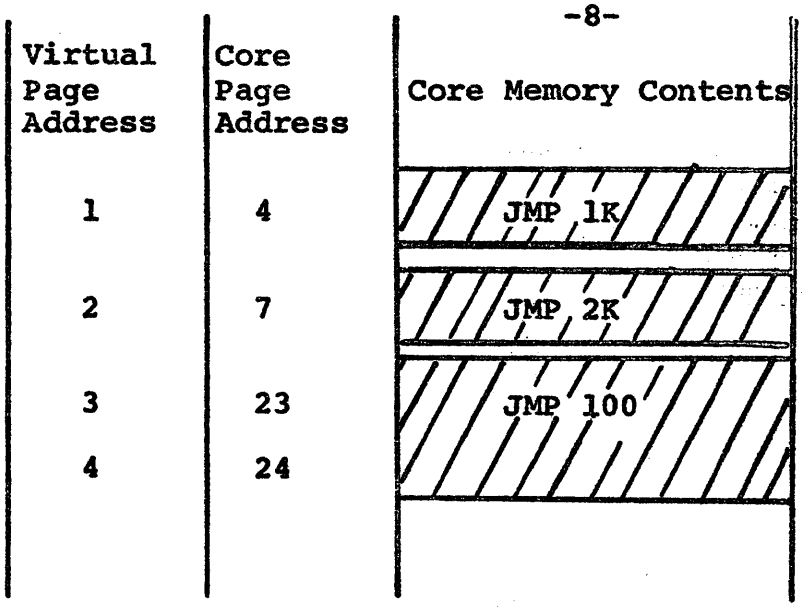
We can conceptually divide a core memory into sections. For purposes of example, suppose we divide a memory system into sections of 1024 words (sections of 2048, 4096, 512, and 128 words are also commonly used.) We call each 1024 word section a "page." A nominally 32K memory system will then have 32 pages. The 10 least significant bits of the address generated by an instruction will tell the location of a reference within a page. We say that the least significant 10 bits tell the location in terms of a "displacement" from the lowest address of the page. The most significant 5 bits of the address determine the page number of the address.

PAGE #	DISPLACEMENT
5 bits	10 bits

In a non-paged computer system, the 15 bit address is sent unchanged to the memory from the processor. In a paged computer system a hardware box is placed in the memory bus between the processor and the memory. This box passes the 10 bit displacement address directly to the memory. The box, however, changes the page number (most significant 5 bits) it receives and substitutes a new page number which it sends on to the memory, assuming no page fault. The manner and reason for this substitution requires explanation.

We will first consider a paging system where the "paging box" hardware substitutes the same size page number as it receives. In our 15 bit address example, this means that the paging box sends a 5 bit page number and 10 bit displacement to the memory. Thus, the address transmitted by the paging box can directly address a maximum of 32K as could the original address sent into the paging box.

In a time-sharing system, several programs may be in core at one time. Because each program was written at a different time and because segments of the memory were already occupied, when a program was written one program may be physically in several separate non-connected pages.



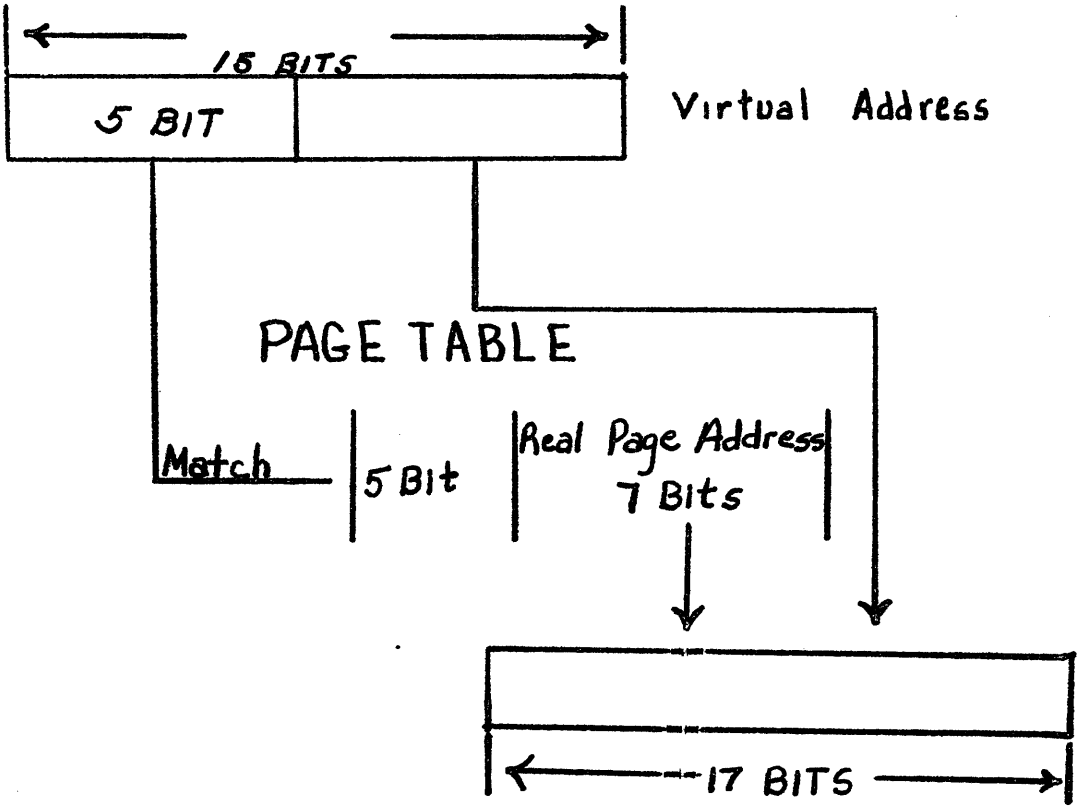
RUN??

In the above diagram the program is located in core at page addresses 4, 7, 23 and 24. Suppose the program was written for pages 1-4 (note JMP instructions). It would seem that there is a problem! The program won't write. We could reshuffle the programs in core to place the program in core pages 1, 2, 3, and 4. This, however, would be very time-consuming and is not necessary with paging.

Suppose that whenever the paging box receives a page 1 address from the processor, it sends to the memory page number 4 address. When it receives #2, it sends #7, receives #23 sends #3, receives #4 sends #24. Then the program would access the correct core memory location even though its physical page location was different from the page address sent by the program. In this manner, the program never "knows" that it was loaded into the physical core in several separate places. The program addresses pages 1-4 and operates normally as if the instructions were really located at physical core locations 1-4. The physical core addresses are sometimes called REAL addresses and the program addresses the VIRTUAL addresses.

Suppose that we wish to have a memory system of nominally 128K of core and that we wish to have 4 different programs in core at once on our 15-bit address computer. We will have one program in the first 32K, another in the second 32K (the assumption that each program is in a continuous group of pages is made to simplify this section. Each program could be in any 32 pages with the separations as explained previously.), etc. Since we only have a 15-bit address capability in the proposed computer, how can we address all 128K? First, recall that any one program will address a maximum of 32K -- never the entire core. Suppose our paging box, when it receives

a 5-bit page number substitutes a 7-bit page number which it sends to the memory. This 7-bit page number can then select 32 of the 128 pages. Obviously when programs are switched, the paging box must be changed so that it substitutes a different page number corresponding to the relationship between the VIRTUAL and REAL page #'s of each program.



Paging Box Substitution Technique

Each program has a table as part of it that contains the relationship between its REAL and VIRTUAL pages. For simplicity suppose that the page table is located in memory locations 0-31 of the program. In location "0" is the REAL page number where program (VIRTUAL) page "0" can be found. Hence, in operation, everytime a memory reference is performed, the paging box first does a memory cycle to get the correct REAL page number, which it sends out to memory. This is clearly inefficient as it increases every memory cycle by 1 memory cycle.

A newer technique is to place the page table in a special memory called an associative memory. The associative memory is usually a semiconductor memory that has very fast access. For our system we would need a maximum of 32 words in this associative memory. First, let's consider the properties of an "associative" memory. Recall that a given set of cores in a stack has fixed address. The address is determined by how the stack is wired. In an associative memory word register, there is a place to store the "data" and a place to store the "address" assigned to that data.

Associative Memory Word

Hence, both the address and the contents are variable. The contents of the address part of the word is said to contain the address of the data "associated" with it. Suppose that instead of 32 such registers, we only have 8 such registers. Let's consider first what happens when the page number referenced by the program is one of the addresses in one of the 8 registers.

Suppose that page 4 is requested by the processor. Let us assume that his VIRTUAL page is located in page 24 of Real core. Then one of the associative memory registers in the paging box contains:

Address	Data
4	24

The 4 from the processor is compared, in parallel, with the contents of the address part of each associative memory register. If a match occurs, as it will here, the contents of the data section of that associative register will be sent out to the memory as the REAL page address - in this example "24." Since semiconductor logic is used here, the time to make the comparison and then place the REAL address on the memory lines is on the order of 100 nsec. This is about 1/10 the amount of time it would have taken to do the memory cycle if the associative memory had not been there.

Finally, let us consider what happens when the VIRTUAL page requested by the processor does not match the contents of any of the address parts of the 8 associative registers. This section will also illustrate how paging can be used to provide an automatic "overlay" system to make very large programs run in computers with a small amount of core.

When no match occurs, the paging box will address the core location equal to the page address it received from the processor. This will contain (since it is the page table previously defined) the REAL page address (and also some "control bits"). The box will then substitute the new VIRTUAL page number and REAL page number associated with it into one of the 8 associative registers. Then the REAL page number is sent to the memory. From that time on any reference to that page will be processed without a special reference to the core page table. (This assumed that the page was presently in core.) Suppose that the program was larger than the amount of core available. Assume that there is 4K of core and there is an 8K program. Obviously, not all of the program can be in core at once. We could set up the page table so that VIRTUAL pages 0, 1, and 2 were associated with real pages 0, 1, and 2 respectively. VIRTUAL pages 3, 4, 5, 6, and 7 could all be associated with REAL page 3. In this case, when a reference to VIRTUAL page 4 is made (and some other virtual page is presently in REAL core page 3) the computer first swaps out the current content of REAL page 3, next rolls in the virtual page 4 and then places the new address and data in the associative memory; finally, the original program reference continues and the program continues. To accomplish this, "control" bits are usually part of the "data" in the page table in addition to the REAL page numbers.

In most of the above description of paging, simplifications have been made. This is done in order to stress the system implications of paging on the addressing structure of a computer system. All aspects of "memory protection" are omitted.

Paging Advantages

The efficiency of a multiprogramming system can be increased if several programs can be kept in core at all times. Without paging (or relocation), the total amount of core to be shared by all programs is limited to that directly addressable by the processor. Hence, each program is allowed 1/n of the total core where "n" is the number of programs in core. With paging the total amount of core can be increased to the point where several programs can have all the core they can directly address. This scheme though, requiring much core, allows a given user to take place by merely storing the "machine state" and bringing in the previously stored state of the next program. The first user is not swapped out and the next does not have to be rolled in from the secondary storage.

Also, in small systems with minimum core, paging provides an automatic overlay technique so that a small core can be made to appear as the maximum addressable core. When a "non-core resident" page is referenced by the program, a resident page is swapped out and the new page is rolled in. The program then continues as if the computer actually had the larger core. Naturally, it results in a very slow execution of the program if there is lots of overlay.

Using paging to allow procedures of greater than 16K

We have shown that paging can be used to increase the memory address space (allow connection of several times the amount of addressable core to the processor . We have, also, shown that the address space of the PDP-11 is insufficient for a 32 bit computer. Is there a way that we can use paging to run procedures greater than 16K in the PDP-11?

The answer is yes. Procedures of greater than 16K could be run on the PDP-11, but they would run very inefficiently and would increase the complexity of the systems programs.

Assembly Language Programs

For programs in assembly language, the assembler could be designed to accept programs greater than 32K. When such a program is encountered, it could flag the user, informing him that his program ran over 32K and that the assembler had broken it up into "sub-programs." The sub-programs would each have a separate page table and the assembler could provide a monitor call to change page tables whenever a branch or reference was made outside of the current sub-program. The monitor call would involve passing the new sub-program name and starting address in the new subprogram. In the KT11-A, this involves changing the User Base Register and the User Program Counter and perhaps certain other registers. Followed by a RR instruction of the KT11-A. This would be much like changing jobs, except that some of the bookkeeping tasks associated with changing jobs would not be necessary (saving the state of the machine, etc.). The human user could examine his assembled program and reorganize the sub-programs to minimize the number of expensive jumps over sub-program boundaries during execution.

High Level Language Programs

In higher level languages, however, Fortran, Cobol, Basic, etc., there is no opportunity to "manually re-organize" the sub-programs. The compiler, unless it is extremely sophisticated, can place the sub-program boundaries anywhere in the program. It could easily place the boundary in the middle of a frequently used "DO" loop. Since it would probably require 10 to 50 memory reference cycles of overhead every time a sub-program boundary was crossed. The result would be a very long execution time for the program.

Desirability of using paging to run procedures greater than 16K.

In the 32/26 bit class of machines, about the only programs written in assembly language will be the operating systems. We would not expect these to run greater than 16K and hence, we do not have to worry too much about them. Most of the applications programs will be written in FORTRAN or perhaps, BASIC or COBOL. For any large program in a higher level language, the use of paging to get the large address space invites disaster.

Because it is clear that many programs appropriate to a 32/36 bit class of machine will run over 16K and since most of them will be in higher level languages, the use of paging cannot be recommended as a solution to the limited address capability of the PDP-11.

General explanation of Segmentation

Segmentation, like paging, is a technique for dividing the address space into parts. However, in segmentation the two parts (segment, address) are usually each as long as the total address space provided normally in the machine. We will again consider a 16 bit processor as the basis of further explanation.

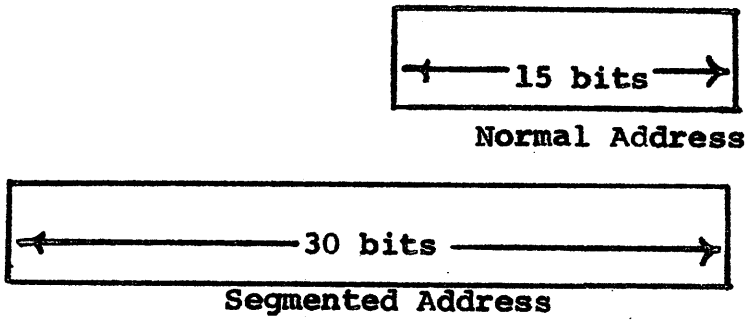


Figure 3.1 Normal and Segmented Addresses.

The segmented address is usually divided into two or three parts. For this discussion, we will consider only two parts: Segment number and Address number. Again usually the address number is about 3 bits less than the normal address.

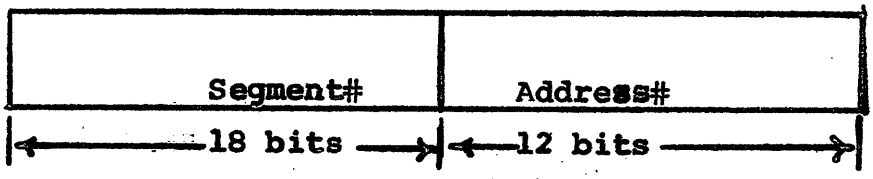


Figure 3.2 Segmented address divided into two parts.

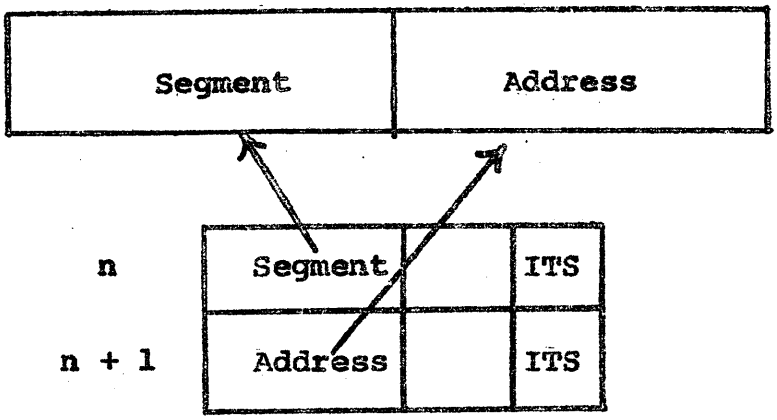
This scheme allows 256K, 2K segments. If we examine the 15 right half bits, it is clear that this includes 12 bits of address and 3 bits of segment number. We could then say that we could directly address 8 segments of 2K each.

The goal of the segmentation technique is to provide a convenient way to address the other 255, 998 segments. If this can be obtained, then the size of programs for all practical purposes can be unlimited.

Present Segmentation Schemes

GE645/MULTICS

This system accesses the other 255,998 segments by using a special form of indirect addressing. The low order 18 bits of the indirect word are not normally used as the indirect word of 36 bits contains only an 18 bit address. In a special ITS (Indirect Thru Segment) mode, several of the normally unused bits contain a code which cause the 18 bit address to be considered a new segment number and cause the following word to be fetched and interpreted as the address.



The PDP-11 indirect word has only a single bit that is free, and that bit is free only in word oriented instructions. The bit is the Byte bit in the indirect address. In all word instructions, this bit must specify a word boundary.

Consider the implications of using this bit to create a special ITS (Indirect thru Segment) operation. First of all, none of the Byte instructions could be used across segments because that bit is used for addressing. MOVB could not be used to transfer Bytes from one segment to another. Likewise, the other double operand/byte instructions could not operate across segments. The single operand byte instructions could reference only within their segment. Finally, in a compiler environment the choice would have to be made between a multipass interative compiler or one that generated ITS indirect addresses for all indirect references. This would be necessary as every time an ITS instruction were

inserted into the procedure section, it would bump some other instruction out of the segment. The compiler would then have to go back and make ITS instructions out of all references to that instruction. This would have to continue until all out-of-segment references were made with ITS formats. The only way to avoid this would be for the compiler to assume that all references were out-of-segment and create the double length address for all memory references. This in the extreme case could nearly double the program size and the execution time. In addition, it appears that there would be considerable problems with stacks crossing segment boundaries. Because of the problems just described, this approach to segmentation cannot be recommended to extend the PDP-11 address space.

IBM 360/Spectra

The IBM and RCA manuals call their addressing techniques a segmentation system. As previously defined, the IBM/RCA system is not segmentation. The normal indirect address on these computers is 24 bit long (with space left to go to 32 bits). All they have done is call the high indirect address bits the "segment" bits. This is a matter of semantics and the technique is of no use whatsoever to the PDP-11 problem. Applying this would be to call the high order bits of the PDP-11 address the segment bits - the result being the same address space as before. This scheme will not be considered further and, of course, is rejected as a possibility for the PDP-11.

Other Schemes

Several other segmentation schemes are mentioned in the literature. None, however, seem to attack the problem of extending the basic address space of the machine. They are not discussed as their basic rational was for core allocation or the technique was so interlocked with the architecture of the particular computer that it was not applicable.

Segmentation for the PDP-11

We are, hence, left with creating a special instruction for the PDP-11 that says: "Change Segments." Such an instruction, except perhaps in a very expensive version, would be a monitor call which would replace a current segment with the new one. This is identical to the earlier scheme which we called "paging" and it has the very same disadvantages.

We are, hence, forced to conclude, unless a different technique can be demonstrated, that the PDP-11 architecture does not lend itself to segmentation for address space expansion.

Conclusion

The address space problem is serious in the PDP-11. Neither the paging or segmentation techniques examined seemed to be practical solutions to this. Since in order to be competitive and to be able to efficiently run large problems are requirements for a machine in the 32/36 bit class, the PDP-11 architecture must be rejected as a candidate for this market.