CMU-CS-71-106

C.ai(P.L*) -- An L* Processor for C.ai

D. McCracken
G. Robertson

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

October 11, 1971

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107) and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

Although this report and the basic design were principally carried out by the listed authors, the primary idea of L* is due to Allen Newell. A group composed of Allen Newell, Peter Freeman, Don McCracken, and George Robertson has developed some of the concepts of L*-like kernel systems, implemented several, and experimented with their use.

Robert Chen provided some very valuable assistance during the final stages of the design effort, and both he and Peter Freeman contributed generously to the editing of this report.

ABSTRACT

The results of a preliminary design study for a specialized language processor (P.ℓ) for L* are presented. The objective of the study is to give an example of a specialized processor for C.ai.

The L* processor is to run 20-30 simultaneous L* users with very large address spaces at a speed improvement of better than 10 times a typical PDP-10 L* system. Its cost should be low relative to the memory resources of C.ai.

The design presented is that of an L* central processor (Pc.L*) with a low-level instruction set (about the level of typical micro-code). Pc.L* is time-shared by a mini-computer that sits to the side, so that each L* user sees himself as running on a base L* processor. User contexts are switched by swapping processor status information in Pc.L*.

Each L* user has complete access to the central processor status through his address space. His machine code (microcode) can appear anywhere in the large address space, but executes out of a fast cache memory. It thus runs at microcode speeds. L* programs and data being interpreted by the machine code are accessed explicitly from a second cache memory. The initial L* kernel system will consist of ~ 1K of machine code, with some initial data and available space.

The low-level instruction set of Pc.L* does not contain any of the more complex instructions (such as floating point arithmetic and byte manipulation) that usually exist on large general purpose computers.

These capabilities are meant to be written in machine code as needed by each L* user. He thus gains considerable flexibility in the exact nature of these higher level operations at the cost of increased programming effort and somewhat reduced efficiency compared to hard-wired implementations.

The results of this preliminary design effort, although still unclear in spots, shows that a specialized processor could run very large L* systems on C.ai at 20-40 times the speed of a PDP-10.

1. INTRODUCTION

Our objective is the design of a specialized processor to run L* systems on C.ai. We call our processor P.L*. A thorough understanding of. the context in which we are designing requires familiarity with C.ai as presented in reference 1. Much of what follows, however, can be understood with the knowledge that C.ai provides a processor such as P.L* with (1) a port to a primary memory of up to 2^{22} 296-bit words of 550ns cycle time accessible as 74, 148, 222 or 296 bits per access, (2) transfer capability to and from the outside world, and (3) transfer capability to and from large on-site secondary and tertiary memories.

Familiarity with kernel systems (reference 2) and L*(F) on the PDP-10 (references 2 and 3) is essential and assumed throughout. Without attempting to summarize these papers, it is worth noting that the essential idea of L* is the growth of arbitrary programming systems from a small kernel of machine code (on present implementations) that permits rapid acquisition of higher-level language facilities and system tools.

Throughout this report we use and assume familiarity with the PMS and ISP notations as presented in reference 4.

2. DESIGN CONSTRAINTS FOR AN L* PROCESSOR

Several important design constraints for our L* processor are listed below to provide a framework for the design.

1. The system running on an L* processor should be consistent with L* design philosophy. This is actually a set of constraints, such as a small sized L* kernel system, accessibility to the complete L* machine as seen by its user, etc. A more complete enumeration of the constraints is given in reference 2.
2. The L* user's address space must be large ($> 10^6$ words), and large L* systems must not experience drastic performance degradation relative to small systems.
3. L* should run much faster than L*(G) on the PDP-10 (at least 10 times faster).
4. A single L* processor must support up to 20-30 simultaneous L* users in a time-sharing mode with an allocated memory of 64k $\sim$ 1024 k words.
5. An L* processor should be inexpensive and simple to construct relative to the cost ($\sim \$10^7$) of the C.ai large memory resources. Subsequent L* hardware processors should be possible in the same fashion as software versions are possible.
6. The L* processor should not be so complex that reliability is low.
7. The final design and building of the L* processor system must be done in parallel with the rest of C.ai.

In connection with constraint 1, we had first to decide what it meant to build an L* computer. The L* philosophy originally addressed building systems on a given powerful machine (e.g. PDP-10) which has high level capabilities already built into the instruction set. The decision we made was to design toward a very fast low level instruction set, and then allow more powerful capabilities to be built along with the growth of the rest of an L* system. This basic approach is compatible with the hardware technology (i.e. microprogramming). For example, floating point and byte manipulation capabilities will have to be coded in the low-level machine code of the L* processor.

Within this idea for growth of high-level capabilities lurks the danger that certain desired advanced capabilities will be very difficult to grow or will be grossly inefficient as compared to an equivalent hardware implementation. Of course, this danger, if it is relatively insignificant as we suspect, is favorably balanced by the freedom of the user to specify the high level operations himself.

We will not state the effect each of the constraints had on the design of the L* processor, but many such effects will be obvious as we describe the design.

3. OVERVIEW OF P.L*

A PMS diagram (Figure 3.1) shows the overall structure of the L* processor and its connection to the remainder of C.ai. In this section we will give only a short description of the function of each component. Later sections will describe them in more detail.

At the heart of the system is the part we call the L* central processor (Pc.L*). The single L* user sees Pc.L* as the processor on which he is running.

Between Pc.L* and the large C.ai memory are a simple address translation control and two cache memories of about 2-4k words each, containing images of parts of the large C.ai memory. One of the caches (the code cache) is used essentially as a read-only memory to hold machine code instructions. The second cache (the data cache) is explicitly accessed by the machine code instructions to read and write L* data types. L* program lists appear as data to the program list interpreter executing in machine code. Students of microprogramming may choose to think of the machine code part as microcode -- in essence it is, because it is fairly inefficient, unencoded, and operates directly on the remaining hardware parts of the processor (e.g., registers). The address translation control of P.L* uses a single segment relocation register and a segment protection register to map the 64k segments of a single L* user's virtual address space into a particular subset of the 64k segments allocated to P.L* by C.ai. The operation of the caches and the address translation control is transparent to the L* user, who sees a uniform virtual memory containing both instructions and data.

The overall control for running multiple users on the L* processor is with the control computer, Ck. Ck has direct access to all the internal working of Pc.L*, the two cache controllers, and the address translation mechanism of P.L*. This enables it to act as a time-sharing monitor for Pc.L*.

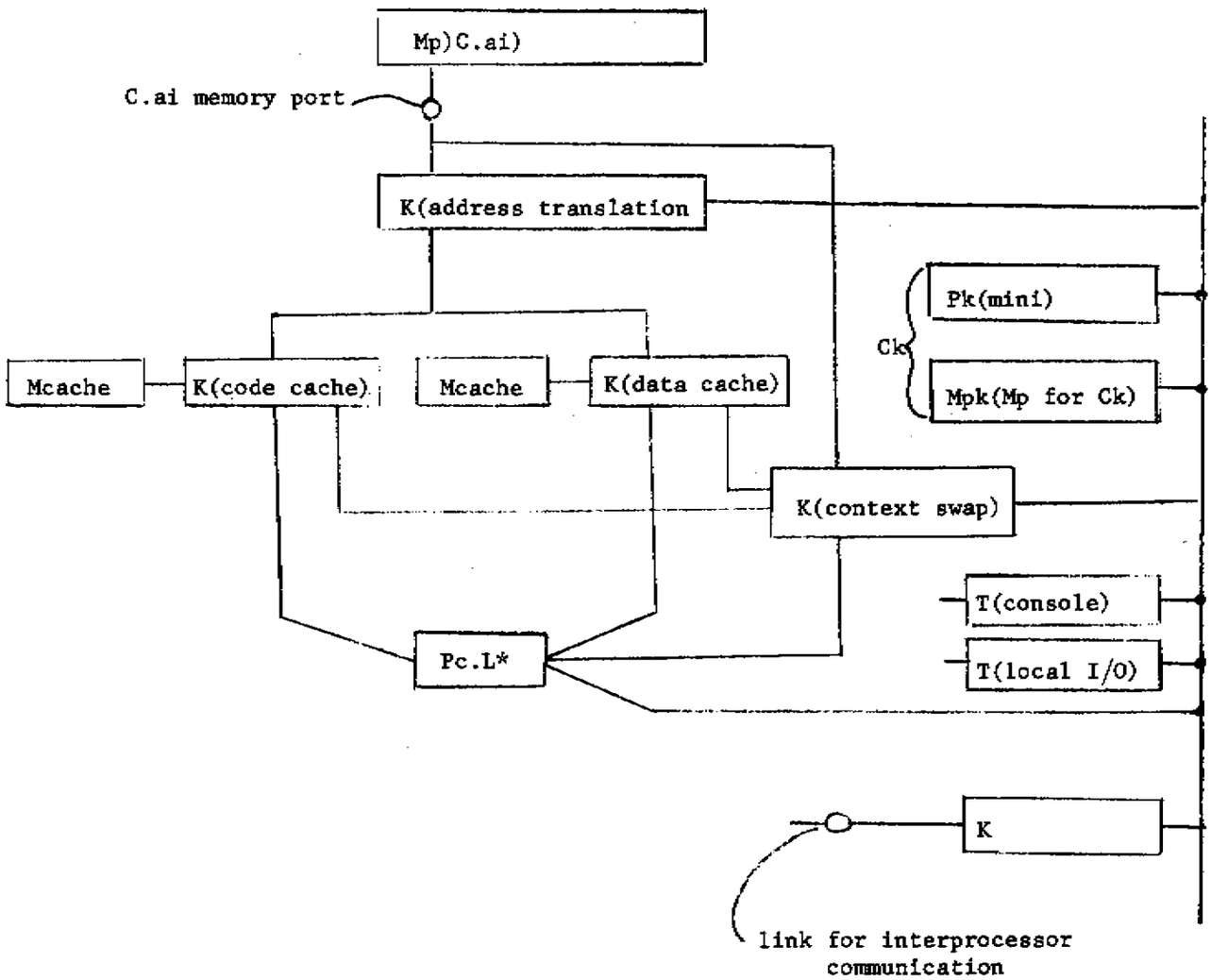


Figure 3.1: PMS of P.L* Overall Structure

4. Pc.L* HARDWARE

Figure 4.1 shows a PMS diagram of Pc.L*. At this level of detail, we see that Pc.L* consists of three parts: the local registers, the instruction control (which handles the main flow of instruction interpretation and execution), and the stack control (an adjunct for machine code subroutine linkage which maintains a pushdown stack in parallel with instruction execution).

Supplementary descriptions of Pc.L* are provided in Appendices 1 and 2: The ISP description in Appendix 1 is an attempt to describe the operation of Pc.L* in considerable detail, and as such is the real heart of this paper. The description's principal failing is the difficulty of representing the interaction of parallel activities in a transparent way (e.g., how the stack controller interacts with the control of instruction execution). To display clearly the parallelism of control flow, in Appendix 2 we have adopted a two-dimensional notation borrowed from Register Transfer Module descriptions.

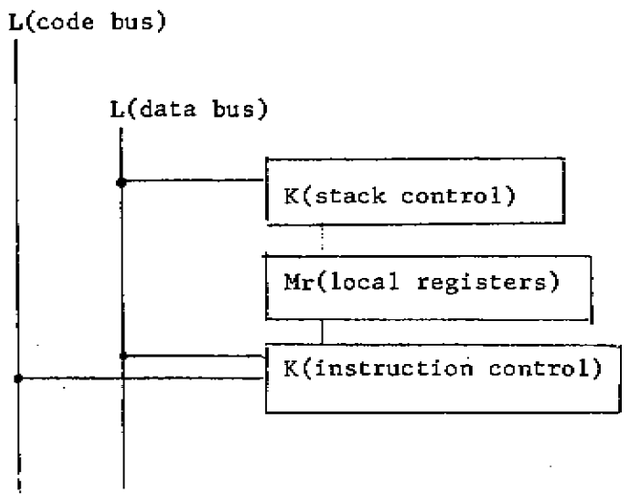


Figure 4.1: PMS of Pc.L*

INSTRUCTION CONTROL

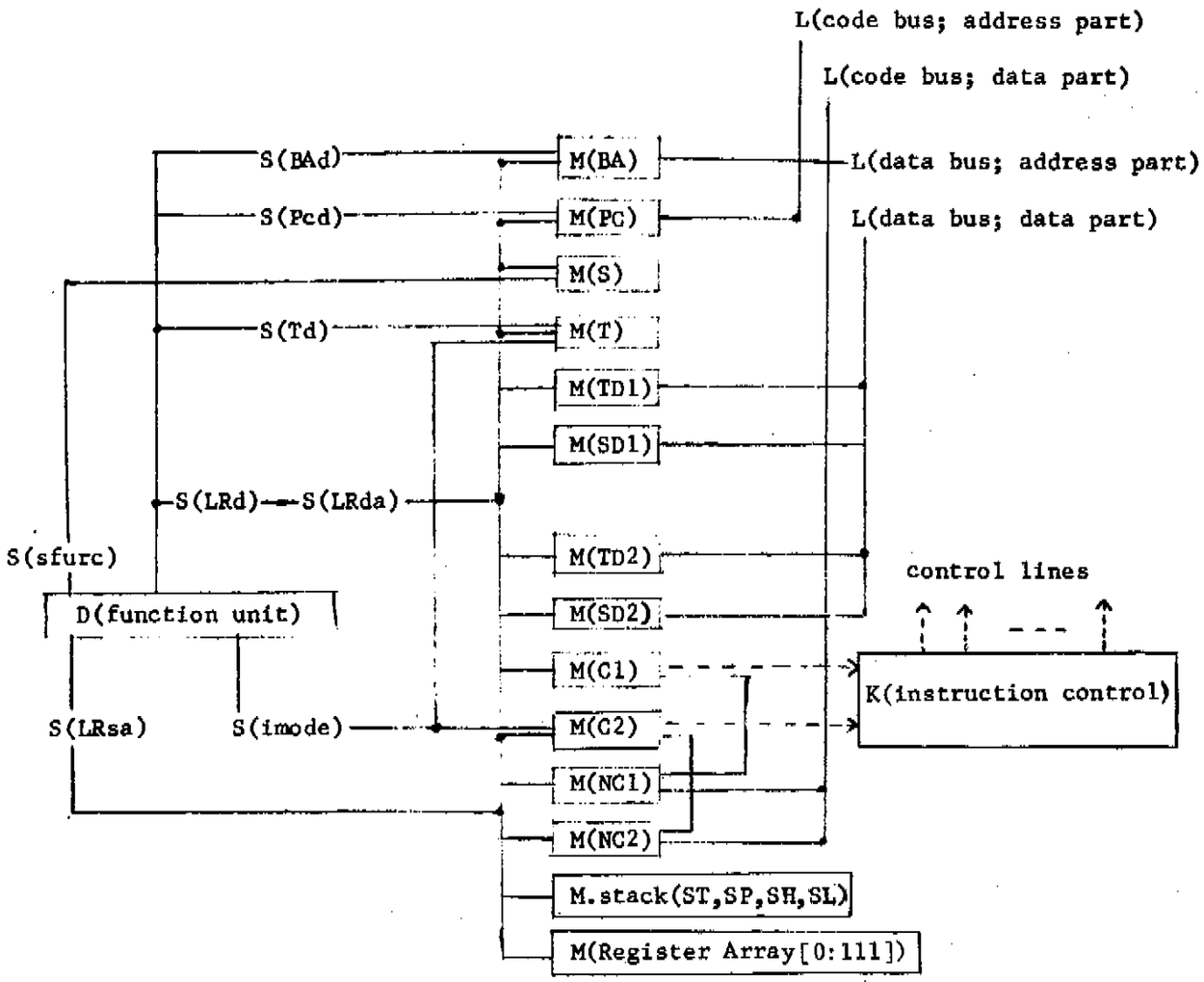
Figure 4.2 shows the local registers in their separate identities and their interconnections via the function unit, along with the various control connections providing for instruction interpretation and execution control.

The local registers contain all processor status information pertinent to a single L* user, which means that a swap of the local register contents is sufficient to change the context of Pc.L* to a different L* user job. The local registers appear as the first 128 words of the L* user's address space. The remainder, up to 2^{24} words, comes via the cache and address translation control from a part of the large C.ai memory.

The details of instruction execution are controlled fairly directly from the fields of a 48-bit doubleword instruction. The wide instruction provides direct control over the various substages of instruction execution at a very low level. This makes the instruction set look like a microcode instruction set, and in fact, one way to view the L* processor is as a flexible microcoded processor. However, we will continue to view it as a very fast machine with a simple, low level machine code.

The choice of the particular instruction set is based on some sample coding of small parts of the L* kernel. It is to be expected that numerous minor additions and alterations (and possibly some major ones) would take place before final freezing of a design.

It is crucial (for reasons of accessibility of machine code by the L* user) that machine code have the same general format as all other words in the L* user's address space. Thus, machine code instructions



Note: Switches (S's) are labeled by the instruction fields that control them.

Note: The terms used in this figure are explained in Appendix 1.

Figure 4.2: PMS for Instruction Control

are pairs of words, and the address of each word has a type associated with it just as does any other address in the user's address space. (The type system is explained on p. 16).

Using a one instruction look-ahead scheme (also operating in parallel with execution), an instruction is fetched from the code cache according to the address in local register PC/Program Counter and read into local registers NC1 and NC2 (the Next Command registers). From there (except for the special case of a control branch) the instruction is transferred into local registers C1 and C2 where it is executed. See Appendix 2 for the control flow of the instruction interpretation process.

The most basic part of instruction execution is the register transfer process via the function unit. There are two inputs to the function unit, plus the specification of which function of its two inputs it is to perform. One of the inputs can be any one of the local registers (selected by the LRsa/Local-Register-source-address field). The second input is the local register T for normal mode instructions; in an immediate mode instruction the second input is local register C2 (i.e., the second word of the current instruction). Output from the function unit consists of a result with result condition bits. The condition bits reside in the local (status) register S and can be set according to the current function unit result. The result itself can be sent to any or all of the local registers PC/Program Counter, BA/Bus Address and T/Temporary. In instructions which are not immediate-mode, the result can also be sent to the local register selected by the LRda/Local-Register-destination-address field.

Next in the instruction execution process come the conditional special actions. The condition bits in an instruction specify a certain function of selected status bits in the status (S) register. If the function value is true, all the special actions specified by the bits in the special action field are performed. Examples of special actions are: interrupt Ck, and skip next instruction, etc. All of the condition bits but one are used to select particular status bits in the S register. The remaining bit specifies whether one of the selected status bits = 1 is sufficient to trigger the special actions, or all of the selected status bits must be 1 before the special actions are taken.

A third part of the instruction execution consists of the external function control; e.g. read/write/pause functions for memory. Read or write operations use local register BA as the bus address register, and local registers TD1, SD1, TD2 and SD2 as the data registers. These operations, resulting in main memory accesses, are initiated after the register transfer for the current instruction has been completed. The pause bit causes execution of the current instruction to be delayed until an active read or Write operation, started in some previous instruction, has been completed.

There is one last thing that happens during instruction execution if the local register ST (stack top) was selected as the source or destination of the register transfer: the stack controller is initiated. Once initiated, the stack controller proceeds, in parallel with continued instruction interpretation, to initiate the memory read or write operation and do the stack pointer manipulation necessary to complete the push or pop of the stack. The operation of the stack controller will be discussed in more detail below.

STACK CONTROL

Figure 4.3 is a PMS diagram of the stack controller and its related local registers and bus connections.

The stack controller is started into action by the appearance of the local register ST as the source (indicating a pop) or the destination (indicating a push) of a register transfer.

The particular format chosen for the stack is such that local register ST holds the top element on the stack, but the top element also appears at the top of the array forming the stack in main memory. This means that the instruction control need not wait for a main memory operation for either a push or a pop, and may continue with subsequent instructions while the stack control takes charge of completing the stack operation. Of course, the instruction control will have to wait if the stack control is still completing the previous stack operation.

When the stack control is initiated, it first increments (for a push) or decrements (for a pop) the main memory stack pointer (local register SP). It then borrows control of the data bus from the instruction control (which hopefully wasn't needing it anyway just then) to write from ST into main

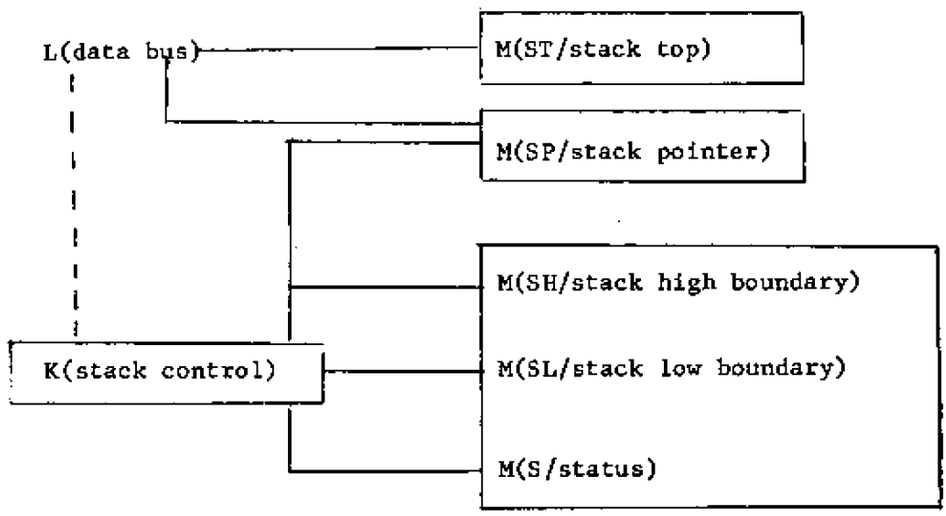


Figure 4.3: PMS for Stack Control

memory (push), or read into ST from main memory (pop).

Overflow and underflow detection are done one step ahead of the actual error condition by comparisons of SP with the stack low boundary register (SL) or the stack high boundary register (SH). The two conditions posted in the status register (S) are: (1) overflow will occur on next push and (2) underflow will occur on next pop. Thus, a stack operation can always immediately proceed if the appropriate condition bit in S is not on, and rechecking of the boundaries for the benefit of the next stack operation can proceed in parallel with the current one.

An alternate stack scheme was considered briefly which had several stack top registers, and which allowed these to exist in an "empty" state (similiar to the Burroughs B5500). Although it is a more complicated scheme to control, it would do better for a push-pop mixture which stays within several levels since no main memory operations would be required. However, we felt that more than just two or three stack top registers would be required to have significant benefits for the L* kernel. Also, the scheme does not take advantage of idle time of the data bus and the fact that the stack has a high probability of residing in the cache. Even with these considerations, we would not want to make a final decision on a stack algorithm without an actual simulation of the system running typical L* programs.

POSSIBILITIES FOR OTHER SPECIAL HARDWARE

One of the bottlenecks in the system, as currently conceived, is the high frequency of allocating and returning available space for the W cells, mainly W, WHS and WHN. In fact, many of the kernel processes would reduce to one or two instructions if it were not for the necessity of obtaining inputs from and returning outputs to the operand stack W (operations which often require allocating and deallocating cells from available space). To help here, we might add special control which essentially buffers the unlinking and linking of available space cells, allowing instruction interpretation to proceed in parallel. We might also consider a mechanism which anticipates a space-exhausted condition and allocates additional bulk available space in parallel with program execution. (This latter would be difficult if we allow, as in conventional L* implementations, the space exhausted condition to be handled by an arbitrary L* program).

Another possibility for further specialization of hardware would be to transfer some of the machine-coded L* kernel into hardware. This we have avoided in order not to bind the processor to conventions that a particular L* user might want to modify to suit his own needs.

5. Ck AND CONTROL OF P.L*

The purpose of the control computer is to allow each L* user to gain complete access to Pc.L*. To accomplish this, Ck provides the functions of typical time-sharing monitors. It controls memory (both primary and secondary), schedules, swaps user contexts, communicates with other special processors and with AMOS (the operating system for C.ai), and handles local I/O devices (if any). The PMS diagram showing the control computer as part of the whole P.L* system was presented in Figure 3.1. This section describes how the components of Ck provide the specified functions.

Primary memory management is accomplished by Ck communication with AMOS and by Ck control of K(address translation). The address translation is done by single segment relocation and segment protection registers which can be set by Ck. A segment is a number of contiguous 64k blocks, obtained from AMOS. The function of shuffling which is normally provided by a segment-oriented time-sharing monitor is accomplished by Ck by requesting AMOS to rearrange or shuffle P.L*'s memory mapping registers. To provide more than 128 64k blocks, Ck will make use of AMOS's swapping mechanisms. A single user is, of course, limited to 128 blocks. A single user can increase or decrease his allocation by requests to Ck. These requests are honored by Ck requests to AMOS for new allocations. Thus all primary memory management (allocation, swapping, shuffling, and segment relocation) is accomplished by either communication with AMOS or by control of K(address translation).

Secondary memory management (for file storage) is handled by Ck which in turn communicates with AMOS to have the transfers actually performed.

Scheduling of users for P.L* is not difficult because all I/O is done through Ck (i.e., Ck does all interrupt handling). There are only two requirements:

(1) Ck must have a clock; and (2) communication from Pc.L* to Ck must have the side-effect of turning PC.L*'s run flag off. Existing scheduling algorithms should work nicely. The PDP-10 DEC monitor has an adequate algorithm for scheduling and could be used by Ck with only minor modifications.

Swapping user contexts is accomplished by Ck control of K(context swap). The context swap controller will transfer the current local register array to the primary memory of the current user. It will then mark the two caches (data and code) as empty. This has the side-effect of causing the data cache to write out any changed words not previously written. Ck can now change the segment relocation and protection registers. Ck now causes K(context swap) to read in the new user's copy of the local register array. When K(data cache) and K(context swap) have completed their work, the swap can be considered complete and Ck can turn Pc.L*'s run flag on.

Communication with other special processors and with AMOS is provided for by connecting Ck's bus to the C.ai inter-processor trunk bus. Protocols for this communication have not been established; but they should be simple.

Local I/O device handling presents no real problems. Local devices can be attached to Ck if needed. I/O operations through Ck can be handled in much the same way as UUO's on the PDP-10.

Communication between Pc.L* and Ck is accomplished by dedicating a portion (~10 words) of Mr(local registers) for a communications area. Pc.L* will have the ability to interrupt Ck with the side-effect of turning Pc.L*'s run flag off. Ck can interrupt Pc.L* at any time because it can set and reset Pc.L*'s run flag.

Considering the functions Ck must provide, we feel that a mini-computer with a good interrupt structure such as the DEC PDP-11 would be adequate.

The hardware we would add to the PDP-11 -- K(context swap), K(address translation), a clock, local I/O, C.ai bus, Mr(local registers) -- could almost all be added directly through the Unibus. Some hardware modification might be desirable. For instance, the trap vector for communications between Pc.L* and Ck should probably be augmented with a control that causes a trap through a branch table with the contents of the first word of the communications area as an index. More hardware to speed up critical sections can probably be shown to be worthwhile.

6. THE L* KERNEL FOR P.L*

The basic approach in our design of an L* machine has been to take an L* kernel like the ones that currently exist on the PDP-10 and PDP-11 and implement it on a much faster, simpler processor of our own design. There was no radical redesign of the L* kernel itself because its structure is largely independent of the machine on which it is to run. A principal reason for this independence is the fact that the kernel supplies initial data types and operations which are so basic that they very likely already exist on any given computer, or can be very simply composed from existing facilities. That is, almost all computers of interest to us ("general purpose computers") have add instructions, logical operations, move instructions to manipulate simple list structures, etc.

The simple, low-level nature of the facilities in our L* processor (with a very few exceptions, such as a stack mechanism) are a result of the fact that L* is not a single specific language system, but a base from which it should be possible to grow many different systems. Thus, we have nothing on which to base an a priori selection of more powerful facilities to be built into the hardware. Instead, we are willing to grow more advanced facilities as needed, from within the system, in the form of sequences of the given low-level facilities. That is, we will add new "instructions" to our machine by writing "microprograms" for them.

The L* kernel for our L* machine is not exactly like any of the L* systems on the PDP-10 or PDP-11, since we were able to remove some constraints forced by those machines. Thus, for example, we are able to have a unique changeable type for each symbol. This type scheme was used in L*(F), but was abandoned on going to L*(G) in favor of a more rigid but far less space-costly scheme.

We will proceed by enumerating and briefly describing a few of the more important ways the L* kernel was adapted to run on the above L* processor.

TYPE SYSTEM

The L* user sees a uniform virtual address space of up to 2^{24} 24 bit words. Each address has a separate type associated with it, which can be changed at will. The types are represented by small integers from the set {1,3,5,...,511}, giving a maximum of 256 types. These small integers are called type indexes because they are used to index into a type table which contains a Uoubleword entry (head of a list) for each type currently in use. The type index is actually stored (shifted right one binary position) in the high order 8 bits of a physical 32-bit word, although this fact is transparent to the user. To the L* user, the types appear to be "abstract" entities since they are not stored anywhere in the memory space he sees. The limit on the number of types imposed by the 8-bit type field may eventually be a problem, for example, if we go from a simple type system to a hierarchical one. Whereas the simple type scheme allows 256 different types, a four-level hierarchical scheme might allow only, say, four alternatives at each of four levels.

OPERAND COMMUNICATION

Kernel processes are written to deal directly with the L* operand stack (list) W. In the PDP-10 and PDP-11 versions of L*, W was used to communicate operands only in the context of the interpretation of a program list. For execution of kernel processes from machine code (e.g., other kernel processes or compiled code), operand communication through W was too slow, so general registers were used instead. This was implemented by kernel

machine code routines called prefixes which transferred process inputs from W to general registers and outputs from the registers back to W when in the context of program list interpretation. In the C.ai L* kernel we are committing ourselves to the belief that we can now afford to use W for operand communication not only in program list context, but also in the low-level machine code context.

This decision provides a considerable reduction in complexity since it removes the logical need for process prefixes. A disadvantage of the decision is that some special kernel processes which for one reason or another cannot use W for operand communication must have special conventions, effectively making them non-accessible from program list context. Two prime examples are C/L and E/L which are used for allocating and returning available space for the working lists (including W itself).

PROCESS PREFIXES

In the section above on types we explained why process prefixes are no longer logically required. Nevertheless, we do have process prefixes because many of the kernel processes do such a small amount of processing (e.g., "add two numbers") that a very large percentage of the machine code for the processes is used for the manipulation of W to obtain inputs and store outputs. By defining several prefixes, we have subroutinized the operand communication. We have not, however, gone all the way to a scheme where all the inputs are transferred to registers, because that loses enough efficiency to outweigh itsbenefits (we think). The definition of the prefix routines is as follows:

> The prefix routines receive a non-standard input (in some register) which is the address of the main part of the process to be executed (i.e., the part divorced from manipulation of W).

P01: Prefix routine for no inputs and 1 output.

Operation: Push W, then branch to main part of process (process stem).

P10: For 1 input and no outputs.

Operation: Pop input W(0) into local register R1, return working cell to available space, then branch to process them.

P11: For 1 input and 1 output.

Operation: Nothing. (Possibly P11 will be non-existent).

P12: For 1 input and 2 outputs.

Operation: Same as P01.

P20: For 2 inputs and no output.

Operation: Pop W(0) into R1, W(1) into R2, return both working cells to available space, then branch to stem.

P21: For 2 inputs and 1 output.

Operation: Pop W(1) into R2 (leaving W(0) in W), return cell to available space, then branch to stem.

P22: For 2 inputs and 2 outputs.

Operation: Nothing. (Possibly P22 will be non-existent).

P31: For 3 inputs, 1 output

Operation: Pop W(1) into R2, W(2) into R3 (leaving W(0) in W), return two cells to available space, then branch to stem.

SPECIAL WORKING CELLS

Some selected W cells plus some temporary working cells have very special status by virtue of residing in the local register array. These cells are the ones that can be directly addressed in the register transfer operations of the machine code. However, in order not to let this fact limit accessibility to these cells, we map the 128 local registers into the first 128 locations in the main address space. This allows the L* user to access them via the data bus in the same way as all the non-special cells residing in main memory.

THE CODE AND DATA CACHES

The speed of cache operation is so critical that we are virtually forced to hardwire the cache algorithm, thus depriving the L* user control over its operation. However, the L* user must be aware of the caches since their performance can drastically affect execution speed.

The code cache is the more critical of the two caches since accesses are made every instruction cycle. We would hope to choose a size for the code cache that would virtually ensure that all active code can reside in the cache at once. We are tacitly assuming (without real justification as yet) that it will not be necessary for L* users to compile many high-level programs into machine code, since such a strategy would be heavily penalized. The code cache size should be large enough to hold the entire L* kernel (~1 K of 48 bit words), plus a reasonable amount of extra space (like a factor of 4) for additional primitives coded by the L* user.

Since the two independent caches both hold images from the same address space, there is the commonly known problem of double images. That is, a user may have altered in the data cache a section of code whose old version is still held in the code cache. This is not actually a serious problem since it should happen relatively infrequently, and in any case any inconsistency will last only to the end of the user's current time slice. We have decided against a solution at the hardware level, so it will be a case of "user beware."

INCREASED COMPLEXITY OF KERNEL CODE

In our quest for increased speed we have been forced to design an instruction set processor which operates at a lower level and has more direct control over the memory than a machine like the PDP-10. We also have been forced to include in the design operations which proceed in parallel with instruction interpretation, such as the stack control and main memory read/write operations. A result of all this is that in comparison with conventional L* systems, machine code instructions are larger and more complex, and a great deal of thought must be given to synchronizing the parallel operations and optimizing the degree of overlap. Thus, we will probably end up with a kernel which is not nearly as simple and easily understandable as conventional versions, and this runs counter to the L* design philosophy. It remains to be seen just how serious the consequences of this will be.

PERFORMANCE

In order to get a rough estimate of speed and code density for our L* processor, we selected six interesting sections from the L* kernel. We compared the coding for these with the equivalent PDP-10 code taken from version 21 of the L*(G) kernel. The details of these comparisons are presented in Appendix 3.

To summarize the results of the comparisons, we found (somewhat surprisingly) that code density for the Pc.L* is roughly comparable to that for L*(G) on the PDP-10. Code density on a PDP-11 is <u>twice</u> that of a PDP-10. Execution speed for Pc.L* is between 40 and 75 times faster

than L*(G) under ideal cache conditions. Under worst conditions (i.e., no hits in either cache), execution speed for Pc.L* degrades to around 10 times faster than L*(G). We believe that, with good organization of data and code, close to ideal cache conditions can be maintained.

REFERENCES

1. Bell, C. G., et al., "C.ai: A Computing Environment for AI Research." Carnegie-Mellon University, Computer Science Department, April, 1971.

2. Newell, A., P. Freeman, D. McCracken, and G. Robertson, "The Kernel Approach to Building Software Systems," to appear in the Computer Science Research Review, Carnegie-Mellon University, 1971.

3. Newell, A., D. McCracken, G. Robertson, and L. DeBenedetti, "L*(F) Reference Manual," Carnegie-Mellon University, Computer Science Department, Jan. 1971.

4. Bell, C.G. and A. Newell, Computer Structures, McGraw-Hill, 1971.

Appendix 1 - ISP DESCRIPTION OF Pc.L*

The operation of the two caches is not described in the ISP. A reference to main memory using PC (e.g. M[PC]) is to be understood as a reference to the code cache, and a reference using BA or SP is actually a reference *t6* the data cache.

Mp and Rc State

```
Mp[0:8388607]<0:63>                          large primary memory (ARPA)
RA[0:111]<0:31>                              local register array
LR[0:127]<0:31>                              effective local register space
M[0:16777215]<0:31> := LR ¤ Mp[64:8388607]   effective primary memory space

rBA<0:31>                                    Bus Address register
rPC<0:31>                                    microcode Program Counter
rS<0:31>                                     Status register
rT<0:31>                                     Temporary (ALU input)
rTD1<0:31>                                   Type Data register 1
rSD1<0:31>                                   Symbol Data register 1
rTD2<0:31>                                   Type Data register 2
rSD2<0:31>                                   Symbol Data register 2
rC1<0:31>                                    Command register 1
rC2<0:31>                                    Command register 2
rNC1<0:31>                                   Next Command register 1
rNC2<0:31>                                   Next Command register 2
rST<0:31>                                    Stack Top
rSP<0:31>                                    Stack Pointer
rSH<0:31>                                    Stack High boundary
rSL<0:31>                                    Stack Low boundary

LR[0]          :=          rBA               Local Register 0
LR[1]          :=          rPC               Local Register 1
LR[2]          :=          rS
LR[3]          :=          rT
LR[4]          :=          rTD1
LR[5]          :=          rSD1
LR[6]          :=          rTD2
LR[7]          :=          rSD2
LR[8]          :=          rC1
LR[9]          :=          rC2
LR[10]         :=          rNC1
LR[11]         :=          rNC2
LR[12]         :=          rST
LR[13]         :=          rSP
LR[14]         :=          rSH
LR[15]         :=          rSL

LR[16:127]     :=          RA[0:111]         Local Registers 16 to 127
```

```
BA<0:23>        :=      rBA<8:31>
PC<0:23>        :=      rPC<8:31>
S<0:23>         :=      rS<8:31>
T<0:23>         :=      rT<8:31>
TD1<0:23>       :=      rTD1<8:31>
SD1<0:23>       :=      rSD1<8:31>
TD2<0:23>       :=      rTD2<8:31>
SD2<0:23>       :=      rSD2<8:31>
C1<0:23>        :=      rC1<8:31>
C2<0:23>        :=      rC2<8:31>
NC1<0:23>       :=      rNC1<8:31>
NC2<0:23>       :=      rNC2<8:31>
ST<0:23>        :=      rST<8:31>
SP<0:23>        :=      rSP<8:31>
SH<0:23>        :=      rSH<8:31>
SL<0:23>        :=      rSL<8:31>

R1<0:23>        :=      LR[16]<8:31>  argument
R2<0:23>        :=      LR[17]<8:31>    registers
R3<0:23>        :=      LR[18]<8:31>
T1<0:23>        :=      LR[19]<8:31>  temporary
T2<0:23>        :=      LR[20]<8:31>    registers
T3<0:23>        :=      LR[21]<8:31>
W.N<0:23>       :=      LR[22]<8:31>  next of L* data stack
W.S<0:23>       :=      LR[23]<8:31>  symbol of L* data stack
WXS.N<0:23>     :=      LR[24]<8:31>
WXS.S<0:23>     :=      LR[25]<8:31>  current routine symbol
WXN.N<0:23>     :=      LR[26]<8:31>
WXN.S<0:23>     :=      LR[27]<8:31>  current routine next
WHS.N<0:23>     :=      LR[28]<8:31>
WHS.S<0:23>     :=      LR[29]<8:31>  higher routine symbol
WHN.N<0:23>     :=      LR[30]<8:31>
WHN.S<0:23>     :=      LR[31]<8:31>  higher routine next
WSPTT.N<0:23>   :=      LR[32]<8:31>
WSPTT.S<0:23>   :=      LR[33]<8:31>  current av.sp. type table
WITT.N<0:23>    :=      LR[34]<8:31>
WITT.S<0:23>    :=      LR[35]<8:31>  current interpreter type table
WIPTT.N<0:23>   :=      LR[36]<8:31>
WIPTT.S<0:23>   :=      LR[37]<8:31>  current T/P context int. ttable
WS.N<0:23>      :=      LR[38]<8:31>  next of signal cell
WS.S<0:23>      :=      LR[39]<8:31>  signal
etc.
```

Status Register Format

```
run                                  :=  S<0> processor running/halted
data_read_in_progress/drip           :=  S<1> data read operation in progress
data_write_in_progress/dwip          :=  S<2> data write operation in progress
stack_overflow/sov                   :=  S<3> stack overflow on next push
stack_underflow/sun                  :=  S<4> stack underflow on next pop
function_unit_result_zero/furz       :=  S<5> conditions on
function_unit_result_positive/furp   :=  S<6>   function unit result
function_unit_result_negative/furn   :=  S<7>     for current instruction
function_unit_result_overflow/furo   :=  S<8>
interrupt_control_computer/int       :=  S<9>
stack_control_busy/scb               :=  S<10>
stack_control_bus_request/scbr       :=  S<11>
```

Instruction Format

```
instruction_word_1/i1<0:23>          :=  C1
instruction_word_2/i2<0:23>          :=  C2

    mode/m                           :=  i1<0>      normal or immediate mode
        immediate_mode/imode         :=  (mode=1)

    read_write_pause/rwp<0:4>        :=  i1<3:7>
        pause_bit/p                  :=  rwp<0>
        read_bit/rd                  :=  rwp<1>
        write_symbol_bit/wrs         :=  rwp<2>
        write_type_bit/wrt           :=  rwp<3>
        read_write_single_double_bit/rwsd  :=  rwp<4>

    function_unit_function/fuf<0:3> :=  i1<8:11>

    parallel_destination/pd<0:3>     :=  i1<12:15>
        PC_destination/PCd           :=  pd<0>
        BA_destination/BAd           :=  pd<1>
        T_destination/Td             :=  pd<2>
        Local_Register_destination/LRd  :=  pd<3>

    set_function_unit_result_conditions/sfurc     := i1<16>

    Local_Register_source_address/LRsa<0:6>       := i1<17:23>


    Local_Register_destination_address/LRda<0:6>  := i2<0:6>

    condition_bits/c<0:8>            :=  i2<7:15>
        conditions_mode/cmode        :=  c<0>
        condition_zero/cz            :=  c<1>
        condition_positive/cp        :=  c<2>
        condition_negative/cn        :=  c<3>
        condition_overflow/co        :=  c<4>
    special_action_bits/sa<0:7>      :=  i2<16:23>
        special_action_skip/sas      :=  sa<0>
        special_action_run_off/saro  :=  sa<1>
        special_action_interrupt_control_computer/saint  :=  sa<2>

    immediate_data/id<0:23>          :=  i2<0:23>
```

Special Action Conditions

```
special_action_condition_0/sac0  :=  (~cmode ∧ ((cz∧furz)∨(cp∧furp)∨(cn∧furn)∨(cc∧furo)))
special_action_condition_1/sac1  :=  (cmode ∧ (cz⊃furz) ∧ (cp⊃furp) ∧ (cn⊃furn) ∧ (co⊃furo))
special_action_condition/sac     :=  (sac0 ∨ sac1)
```

Function Unit Function Definition

```
x1 fu x2  := (
                (fuf=0)  →  0 ;
                (fuf=1)  →  x1 ;
                (fuf=2)  →  x2 ;
                (fuf=3)  →  x2 * 2 {logical} ;
                (fuf=4)  →  x2 / 2 {logical} ;
                (fuf=5)  →  ~ x2 ;
                (fuf=6)  →  x2 + 1 ;
                (fuf=7)  →  x1 + x2 ;
                (fuf=8)  →  x1 ∧ x2 ;
                (fuf=9)  →  x1 ∨ x2 ;
                (fuf=10) →  x2 * 256 {logical} ;
                (fuf=11) →  x2 / 256 {logical} ;
                (fuf=12) →  x2 + 2 ;
                (fuf=13) →  x2 + 4 ;
                )
```

Function Unit Result Calculation

```
fu_result/fur<>:23>  := ( ~ imode → (T fu LR[LRsa]<8:31>);
                          imode → (id fu LR[LRsa]<8:31>) )
```

Function to indicate synchronization of parallel activity

```
Pause_until(b) := (~ b → Pause_until(b))
```

Read/Write Functions

```
rwpause                 :=   Pause_until(~(drip ∨ dwip))
bus_free_pause          :=   Pause_until(~(drip ∨ dwip ∨ scbr))

read_single             :=   ( TD1<15:22> ← M[BA]<0:7> ;
                               TD1<0:14> ← 0 ; TD1<23> ← 1 ;
                               SD1 ← M[BA]<8:31> )

read_double             :=   ( read_single ;
                               TD2<15:22> ← M[BA+1]<0:7> ;
                               TD2<0:14> ← 0 ; TD2<23> ← 1 ;
                               SD2 ← M[BA+1]<8:31> )

write_symbol_single     :=   ( M[BA]<8:31> ← SD1 )

write_symbol_double     :=   ( write_symbol_single ;
                               M[BA+1]<8:31> ← SD2 )

write_both_single       :=   ( write_symbol_single ;
                               M[BA]<0:7> ← TD1<15:22> )

write_both_double       :=   ( write_both_single ;
                               M[BA+1]<8:31> ← SD2 ;
                               M[BA+1]<0:7> ← TD2<15:22> )
```

Instruction Interpretation Process

```
run ← ( (C1∘C2 ← NC1∘NC2 ; PC<0:22> ← PC<0:22>+1 ; Next
          p → rwpause; Next instruction_execution );
        NC1∘NC2 ← M[PC]<8:31>∘M[PC+1]<8:31> ; Next
        (PCdv(sac^sas)) → (NC1∘NC2 ← M[PC]<8:31>∘M[PC+1]<8:31> ;
                           PC<0:22> ← PC<0:22> + 1 ; Next )
      )
```

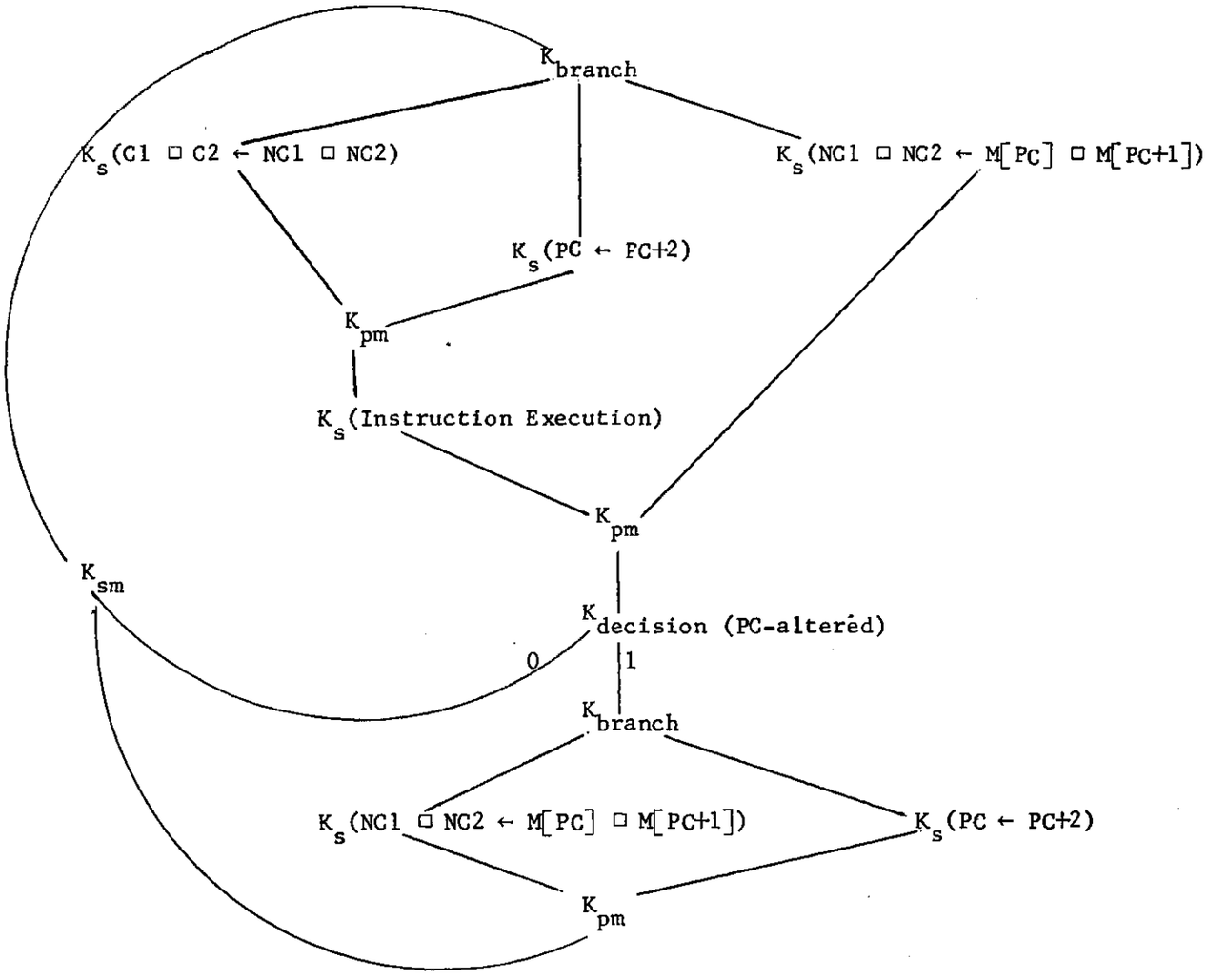
Instruction Execution Process

```
instruction_execution := (

    PCd → PC ← fur ;
    BAd → BA ← fur ;
    Td  → T  ← fur ;
    ~ imode → ( LRd → LR[LRda]<8:31> ← fur );

    (LRsa = 12) → (          Pause_until(~scb); Next      pop stack operation
                             scb ← 1 ;
                             sun → trap(?); Next          underflow detection
                             sov ← 0 ;
                             pop_stack ; Next
                             scb ← 0 ) ;
    ~imode ∧ (LRda = 12) → ( Pause_until(~scb); Next      push stack operation
                             scb ← 1 ;
                             sov → trap(?); Next          overflow detection
                             sun ← 0 ;
                             push_stack ; Next
                             scb ← 0 )

    sfurc ← ( (fur=0) → (furz ← 1 ; furp ← 0 ; furn ← 0);
              (fur≠0 ∧ ~fur<0>) → (furz ← 0 ; furp ← 1 ; furn ← 0);
              (fur≠0 ∧ fur<0>) → (furz ← 0 ; furp ← 0 ; furn ← 1)
            ); Next

    ~imode → (sac →
                  (saro → run ← 0 ;
                   saint → (run ← 0 ; int ← 1) )
            ); Next

    rd ∧ ~rwsd          →   (bus_free_pause; Next drip ← 1; read_single; Next drip ← 0);
    rd ∧ rwsd           →   (bus_free_pause; Next drip ← 1; read_double; Next drip ← 0);
    wrs ∧ ~wrt ∧ ~rwsd  →   (bus_free_pause; Next dwip ← 1; write_symbol_single; Next dwip ← 0);
    wrs ∧ ~wrt ∧ rwsd   →   (bus_free_pause; Next dwip ← 1; write_symbol_double; Next dwip ← 0);
    wrs ∧ wrt ∧ ~rwsd   →   (bus_free_pause; Next dwip ← 1; write_both_single; Next dwip ← 0);
    wrs ∧ wrt ∧ rwsd    →   (bus_free_pause; Next dwip ← 1; write_both_double; Next dwip ← 0);
)
```

```
push_stack   :=   ( SP - SP • 1 ;
                    scbr • 1 ;
                    rwpause ; Next
                    M [SPK8 : 31> «- ST ;
                    {SP > SH) - sov <- 1 ; Next
                    r.cbr <- C )

pop_stack    :=   ( SP - SP - 1 ;
                    scbr • 1 ;
                    rwpause ; Next
                    5? - H[SP]<8:31> ;
                    (SP < SL) - sun • 1 ; Next
                    scbr • 0 )
```

Appendix 2: CONTROL FLOW DIAGRAMS

I. Control Flow of Instruction Interpretation



$K_s \equiv K_{simple}$

$K_{pm} \equiv K_{parallel\ merge}$

$K_{sm} \equiv K_{serial\ merge}$

PC-altered := PC-destination $\vee$ (special-action-condition $\wedge$ special-action-skip)

Appendix 3 - CODING COMPARISONS WITH L*(G)

On the following pages we display the code for six selected portions of the L* kernel for both Pc.L* and PDP-10 L*(G).

Timing estimates for the code are listed to the left of each instruction (in units of microseconds). For the Pc.L* code, a second number within parentheses indicates how much longer an instruction would be delayed if the previous read or write operation was a cache miss.

The language used for the Pc.L* should be self-explanatory, except perhaps for the use of square brackets. They are used to delimit immediate operands.

The assumptions made in estimating timings for the Pc.L* code are:

(1) All instructions fetched by the instruction control are present in the code cache.

(2) All references to the main memory stack are hits in the data cache.

(3) A simple register transfer takes ~50 ns (e.g., move).

(4) A register transfer with a non-degenerate function-unit-function takes ~100ns.

(5) A register transfer which alters PC, or a special action skip adds ~50 ns.

(6) A reference to the data cache takes ~50 ns if it is present in the cache, and ~600 ns if it must be copied from main memory.

Summary of Comparisons

		(a) Time Estimate on Pc.L* (μsec)	(b) Time on PDP-10 L*(G)21 (μsec)	(a) No. of 48-bit Pc.L* Instructions	(b) No. of 36-bit PDP-10 Instructions
I	Interpret-Advance cycle of program list interpreter	1.05	42	12	17
II	Push W	.65	28	8	10
III	Pop W	.45	21	7	8
IV	S (Get Symbol)	.3	23	4	9
V	N (Get Next)	.35	21	5	8
VI	R (Replace Symbol)	1.15	66	17	25

I(a) Interpret-Advance Cycle of Program List Interpreter (for Pc.L*)

(Heart of L*L Language Interpretation)

Timing Estimates (μsecs)		Pc.L* Instructions	Comments
.05	Interpret:	BA ← WXS.S;Read	Read type index of symbol to be interpreted into TD1.
.05		T ← WIPTT.S	Get base of interpreter type table.
.1 (.55)	Pause;	BA ← T+TD1;Read	Read interpreter into SD1.
.05		R1 ← WXS.S	Symbol to be interpreted to R1 as input to interpreter.
.05		ST ← PC	Save return address on stack.
.1 (.50)	Pause;	PC ← SD1	Branch to interpreter.
.15		← WXN.S-[STOP]; <¬ zero result> → Skip	Skip next instruction if WXN.S ≠ STOP
--		PC ← [Exit]	Go to exit from current context of interpretation if WXN had STOP mark.
.1		BA ← WXN.S;Read Double	Read next program list cell.
.2 (.55)	Pause;	← SD1-[NIL]; <¬ zero result> → Skip	Skip next instruction if link of next cell is not NIL
--		PC ← [Ascend]	Go to ascend if WXN.S.N=NIL.
.05		WXS.S ← SD2	Advance
.05		WXN.S ← SD1	"
.1		PC ← [Interpret]	Branch back to interpret cycle.
1.05 μsec			

Kb). Interpret-Advance Cycle (L*(G)21)

```
2.09    $.I.P1:     MOVE    R1, WXS         ;get symbol to interpret into R1
3.11                PUSHJ   MSTKP,  JiLTI   ;call routine to load type index
                    ----------------------------------------
2.77                LSH     R1,  -8         ;get type map displacement
1.75                MOVEI   R1,  TMAP (R1)  ;locate type map entry
3.18                POPJ    MSTKP,          ;return to interpreter

2.71                HRRZ    R1,  (RD        ;get type index from type map entry
2.75                ADD     R1,  WIPTT      ;add base of interpreter type table
2.71                HRRZ    R5,  (R1)       ;get interpreter from type table
2.09                MOVE    R1,  WXS        ;get symbol to be interpreted to R1
3.39                PUSHJ   MSTKP, (R5)     ;call interpreter

2.09    $.I.P2:     HRRZ    R5, WXN         ;get symbol in WXN to R5
1.79                CAIN    R5, STOP        ;test for end-of-current-execution l
                    JRST    $.I.P4          ;mark found, return to caller
2.71                HLRZ    R5, (WXN)       ;get next in WXN to R5
1.79                CAIN    R5, NIL         ;test if WXN.S.N £ NIL (not end of j
                                                                              list)
                    JRST    $.I.P3          jend of list, ascend
2.85                HRR     WXS, (WXN)      ;advance to
3.20                HLR     WXN, (WXN)      ;next call on program list
1.47                JRST    $.I.P1          jbranch back to interpret
```

42 usees

II(a). Pushing of W (for Pc.L*)

Timing Estimates (μsecs)		Pc.L* Instructions	Comments
.1	PushW:	BA,T ← WSPTT + [<type index for T/L (type list)>]; Read	Address of 1st cell on T/L av.sp. list to SD1
.1 (.55)	Pause;	BA,T1 ← SD1; Read	Get link of 1st av.sp. cell
.2 (.55)	Pause;	← SD1-[NIL];<¬ result zero> → Skip	Skip next instruction if space not exhausted
--		PC ← [<space exhausted code>]	Space exhausted-branch out to handle condition
.05		BA ← T; Write Symbol	Unlink 1st cell
.05		SD1 ← W.N	
.05		SD2 ← W.S	New cell gets copy of head of W
.05 (.45)		BA ← T1; Write Symbol Double	
.05		W.N ← T1	Link new cell to head of W
.65 μsec			

III(a). Popping of W (for Pc.L*)

.05	PopW:	BA,T1 ← W.N; Read Double	Read contents of 2nd cell on W
.1 (.55)	Pause;	W.N ← SD1	Copy contents of 2nd cell into head cell
.05		W.S ← SD2	
.1		BA,T ← WSPTT + [<type index for T/L>]; Read	Address of 1st T/L av.sp. cell to SD1.
.05 (.55)		BA ← T1; Write Symbol	Link previous 1st av.sp. cell to cell to be returned.
.05		BA ← T	
.05 (.50)		SD1 ← T1; Write Symbol	Cell being returned becomes 1st av.sp. cell
.45 μsec			

II(b). Pushing of W (L*(G)21)

```
3.11   PushW:   PUSHJ  MSTKP, %C.L      ;call routine to create T/L symbol
                - - - - - - - - - - - - -
2.43     %C.L:  HRRZ   R5,WSPTT         ;get current av.sp. type table
2.71            HRRZ   R1,$TL(R5)       ;get ptr. to av.sp. list for T/L
2.71            HLRZ   R4,(R1)          ;get link of first av.sp. cell
1.79            CAIN   R4,NIL           ;test if av.sp. not exhausted
 --             JRST   $C.L1            ;jump out if exhausted
2.71            HLRZ   R4,(R1)          ;get link to 2nd cell
3.29            HRRM   R4,$TL(R5)       ;unlink allocated cell from av.sp. list
3.18            POPJ   MSTKP,           ;return to PushW
                - - - - - - - - - - - - -
2.86            MOVEM  W,(R1)           ;copy head of W into new cell
2.58            HRL    W,R1             ;link new cell to head
~28 µsecs.
```

III(b). Popping of W (L*(G)21)

```
2.09   PopW:    HLRZ   R1,W             ;get address of 2nd cell on W
2.71            MOVE   W,(R1)           ;copy contents of 2nd cell into head cell
3.11            PUSHJ  MSTKP,%E.L       ;call routine to erase old 2nd cell
                - - - - - - - - - - - - -
2.43     %E.L:  HRRZ   R5,WSPTT         ;get current av.sp. type table
2.71            HRLZ   R4,$TL(R5)       ;get ptr. of av.sp. list to LH of R4
2.86            MOVEM  R4,(R1)          ;link av.sp. list to cell being returned
3.29            HRRM   R1,$TL(R5)       ;make returned cell new head of av.sp. list
3.18            POPJ   MSTKP,           ;return from %E.L
                - - - - - - - - - - - - -
~21 µsecs.
```

IV(a). S - Get the symbol of W(0) (Set signal cell) (for Pc.L*)

Timing Estimates (μsecs.)		Pc.L* Instructions	Comments
.05	S:	BA ← W.S; Read Double	Read symbol and next of W(0).
.1	(.55)	Pause; W.S ← SD2	Symbol to W(0)
.05		WS.S ← SD1	Next to signal cell
.1		PC ← ST	Return to caller
.3 μsecs			

V(a). N - Get the next of W(0) (Set signal cell) (for Pc.L*)

.05	N:	BA ← W.S; Read	Read next of W(0). (W(0).N)
.1	(.55)	Pause; BA ← SD1; Read	Read W(0).N.N
.05		W.S ← SD1	W(0) ← W(0).N
.05	(.55)	Pause; WS.S ← SD1	WS.S ← W(0).N.N
.1		PC ← ST	Return to caller
.35 μsecs			

IV(b). S - Get symbol of W(0). (Set signal cell) (L*(G)21)

```
1.47     S:    JSP   R6,P11          ;call prefix routine for 1 input, 1 output
                                                                    processes
2.09       P11: HRRZ  R1,W           ;input W(0) to R1
3.39            PUSHJ MSTKP,(R6)     ;call process stem
2.71       %S:  HLRZ  R2,(R1)        ;R2 ← W(0).N
2.71            HRRZ  R1,(R1)        ;output W(0).S in R1
2.23            HRR   WS,R2          ;set signal cell = W(0).N
3.18            POPJ  MSTKP,         ;return to P11 prefix routine
2.23            HRR   W,R1           ;output from R1 into W
3.18            POPJ  MSTKP,         ;return to caller of process
```

~23 μsecs

V(b). N - Get next of W(0). (Set signal cell) (L*(G)21)

```
1.47     N:    JSP   R6,P11          ;call P11 prefix routine
2.09       P11: HRRZ  R1,W
3.39            PUSHJ MSTKP,(R6)
2.71       %N:  HLRZ  R1,(R1)        ;output W(0).N in R1
3.20            HLR   WS,(R1)        ;set signal cell = W(0).N.N
3.18            POPJ  MSTKP,         ;return to P11
2.23            HRR   W,R1
3.18            POPJ  MSTKP,
```

~21 μsecs

VI(a). R - Replace symbol of W(0) by W(1). (for Pc.L*)

Timing Estimates (μsecs)		Pc.L* Instructions	Comments
.05	R:	T ← [℔R]	
.1		PC ← [P20]	Branch to prefix P20
	P20:		Prefix for routines with 2 inputs, no outputs
.05		BA,T1 ← W.N; Read Double	Read 2nd cell on W
.05		T0 ← T	Save T (process stem addr.)
.05		R1 ← W.S	W(0) input to R1
.05 (.50)	Pause;	BA,T2 ← SD1; Read Double	Read 3rd cell on W
.05		R2 ← SD2	W(1) input to R2
.05 (.55)	Pause;	W.N ← SD1	Copy contents of 3rd W
.05		W.S ← SD2	Call into head cell of W
.1		BA,T ← WSPTT + [<T/L type index>]; Read	Locate T/L av.sp. list
.1 (.55)	Pause;	BA ← T2; Write Symbol	Link av.sp. list to 3rd W cell
.05		BA ← T	
.05 (.50)		SD1 ← T1; Write Symbol	2nd W cell becomes head of av.sp. list
.1		PC ← T0	Branch to process stem
.1	℔R:	BA ← R1+1	Input W(0) is in R1
.05 (.35)		SD1 ← R2; Write Symbol	W(1) is in R2. Do the Replace.
.1		PC ← ST	Return to caller

1.15 μsec

R - Replace symbol of W(0) by W(1) (L*(G)21)

```
1.47    R:          JSP   R6, P2Ø          ;call prefix routine for 2 inputs, no
                                             outputs
        - - - - - - - - - - - - - - - - -

2.09    P2Ø         HRRZ  R2, W            ;W(0) input to R2
~ 21                <Pop W>                ;(This is the code for Popping W
                                             displayed on a previous page)

3.73                PUSH  MSTKP, W         ;save W(1) input on stack
~ 21                <Pop W>                ;(again, the code from previous page)

3.80                POP   MSTKP, R1        ;saved W(1) input to R1
2.09                HRRZ  R1, R1           ;zero link of R1

3.01                EXCH  R1, R2           ;W(0) input to R1, W(1) input to R2
1.75                JRST  (R6)             ;branch to process stem
        - - - - - - - - - - - - - - - - -

3.29    % R:        HRRM  R2, (R1)         ;replace symbol of W(0) by W(1)
3.18                POPJ  MSTKP,           ;return to caller of R
```

~ 66 μsec

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security clarification of title, body of abatract and indexing annotation muat be entered when the overall report ia classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Carnegie-Mellon University Department of Computer Science Pittsburgh, Pennsylvania 15213	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP
3. REPORT TITLE C.ai(P.L*) -- An L* Processor for C.ai	
4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)* Scientific Report	
5. AUTHOR(S) *(First name, middle initial, laat name)* D. McCracken, G. Robertson	
6. REPORT DATE October 11, 1971	7a. TOTAL NO. OF PAGES 49 — 7b. NO. OF REFS 4
8a. CONTRACT OR GRANT NO. F44620-70-C-0107 6. PROJECT NO. c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) CMU-CS-71-106 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned thia report)*
tO. DISTRIBUTION STATEMENT This document has been approved for public release and sale; its distribution is unlimited.	
11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Air Force Office of Scientific Research 1400 Wilson Blvd. (SRMA) Arlington, Va. 22209

13. ABSTRACT

The results of a preliminary design study for a specialized language processor (P.*l*) for L* are presented. The objective of the study is to give an example of a specialized processor for C.ai.

The L* processor is to run 20-30 simultaneous L* users with very large address spaces at a speed improvement of better than 10 times a typical PDP-10 L* system. Its cost should be low relative to the memory resources of C.ai.

The design presented is that of an L* central processor (Pc.L*) with a low-level instruction set (about the level of typical microcode). Pc.L* is time-shared by a mini-computer that sits to the side, so that each L* user sees himself as running on a base L* processor. User contexts are switched by swapping processor status information in Pc.L*.

The results of this preliminary design effort, although still unclear in spots, shows that a specialized processor could run very large L* systems on C.ai at 20-40 times the speed of a PDP-10.

DD ,FO?.81473

Security Classification

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
specialized processor						
microcode						
kernel						
context-swapping						