The PRIM System: an alternative architecture for emulator development and use*

Joel Goldberg
Alvin Cooperband
Louis Gallenson

University of Southern California Information Sciences Institute
4676 Admiralty Way, Marina Del Rey, California 90291

The architecture of PRIM is unique in coupling a powerful microprogrammable machine (the Standard Computer Corporation MLP-900) to a modern general-purpose computing system (the DEC PDP-10). The TENEX timesharing system running in the PDP-10 is responsible for scheduling use of the MLP-900. Emulator microcode runs in the MLP-900 under the control of a small resident executive that swaps its users and mediates references to PDP-10 services and shared memory. The PRIM system in the PDP-10 (also running under control of TENEX) provides emulators with access to the TENEX file system and peripherals. PRIM also permits on-line user control of an emulation and supports interactive symbolic debugging of both emulator microcode and target code for the various emulated machines. The resulting sharable system, accessible via the ARPANET to users anywhere, supports both emulator developers and users. This architecture has allowed the development of a more powerful and convenient tool with less effort than would have been possible on a stand-alone microprogrammable host. The hardware and software components of the PRIM system are described and the operation of PRIM is outlined from the user's viewpoint. Requirements for creating PRIM-like systems are discussed, and the PRIM approach is compared with other, more conventional approaches.

Background

The PRIM (Programmable Research Instrument) project was initiated in 1972 at USC Information Sciences Institute to provide an additional level of flexibility in computer architecture for programmers and researchers in the ARPA community by making available a fast microprogrammable computer with appropriate software support to facilitate creating and debugging computer environments (emulators) by remote users via the ARPANET. The system was designed to accommodate many users concurrently employing multiple emulators for generalized hardware/software development (simulations near target-system speeds), direct higher-level-language machine architecture research, and the use and development of specialized logic to enhance existing computers in difficult or time consuming tasks. This paper describes the system developed to achieve the first of these design objectives--supporting multi-user concurrent development and use of multiple emulations. The initial PRIM hardware and software were operational in May 1974 with the ability for remote users to compile and debug microcode. The complete PRIM system became operational in March 1976. Since that time only minor improvements and extensions have been needed.

*This research was supported by the Defense Advanced Research Projects Agency under Contract No. DAHC15 72 C0308, ARPA Order No. 2223, Program Code No. 3D30 and 3P10. Views and conclusions contained in this paper are the authors'.

System Description

The PRIM system is built in three levels upon a base consisting of the MLP-900 microprogrammable processor and the TENEX timesharing system (which runs on a DEC PDP-10). First is the operating-system level of PRIM, consisting of the hardware and software that support shared access to the MLP-900 from TENEX processes. Second is the user level, the PRIM system proper, which provides interactive access to a user at a terminal. And third is the tool level, consisting of the set of installed emulation tools, each providing its users with a complete target environment. The user level is for both the emulator developer and the target machine programmer. These levels are illustrated in Figure 1.

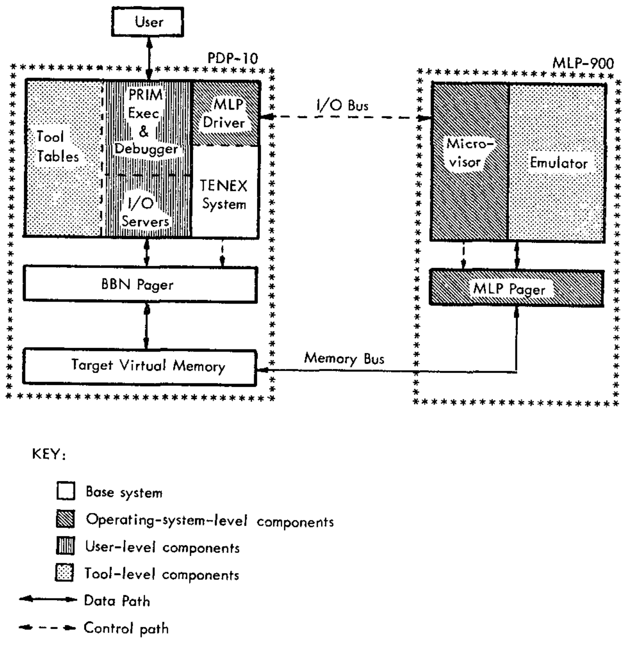

Figure 1. Architecture of PRIM

Base System Components

The MLP-900 (Multi Lingual Processor) at ISI is the prototype microprogrammable processor built by Standard Computer Corporation for a new line intended to follow the IC-7000. It is a large, fast, vertical-word processor with a writable control memory. The processor consists of an Operating Engine and a Control Engine. The Operating Engine is a 36-bit arithmetic/shift unit with 32 general registers, 16 mask registers, and a 1K internal memory. The Control Engine is a control unit with interrupt and branch logic, a subroutine call stack, 128 programmable flip/flops, and 4K of writable control memory. Cycle time is 250 nanoseconds, during which either or both engines can execute a 32-bit instruction.

The DEC PDP-10 is a large, general purpose computer. It is a fairly straightforward machine for connecting new devices, since the I/O bus is extensible and the multiported memory is external to the processor.

TENEX is a versatile timesharing operating system developed by BBN for the PDP-10. It is strongly oriented toward the support of interactive computing and serves both local users and remote users connected via the ARPANET. The operating system does not respond directly to the user (at a terminal of some kind); rather, it allocates resources, manages the file system, and supports the execution of TENEX processes, each process running in its own paged virtual memory and interacting with its own user in an appropriate manner. The creation and control of subordinate processes (by an existing one) is a common method in TENEX for supporting asynchronous but interrelated functions.

PRIM Operating System Components

The operating-system level of PRIM consists of the hardware connection between the MLP-900 and TENEX, the software necessary to allow TENEX processes access to the MLP-900, and the hardware and software required to guarantee the integrity of TENEX--even against errant microcode. Hardware modifications were required only in the MLP-900; they consist of interfaces to the PDP-10 and added protection mechanisms. The software at this level consists of a small operating system resident in the MLP-900, known as the microvisor, and a driver added to the TENEX operating system to shake hands with the microvisor and govern access by users (processes) to the MLP-900.

The MLP-900 is interfaced to PDP-10 main memory and to the PDP-10 I/O bus so that the MLP appears as a PDP-10 I/O device with direct memory access. The memory interface connects the MLP-900 to an existing port (one of four) on the PDP-10 memory. (The MLP-900 design had provided for an external memory and necessary instructions in the Operating Engine, but no specific external memory interface was ever built. There are no peripherals interfaced to the MLP-900.) The I/O bus interface allows the exchange of control information between the MLP-900 and TENEX. Via this interface, either processor can interrupt the other.

The most important modification to the MLP-900 created a privileged supervisor state at the microcode level. The microvisor (and also the hardware diagnostics) run in the privileged state, which allows access to all resources. User microcode (emulators) runs in a user state that protects all the critical resources from modification. The protected resources include the I/O bus interface, the MLP pager (a memory address translator), most of the MLP-900 interrupt system, and control memory itself.

Memory protection is accomplished by the MLP pager incorporated into the MLP-900 memory interface. PDP-10 memory is not directly addressed by MLP microcode; rather, memory references are to virtual addresses in a virtual memory identical to that of a TENEX process. These virtual addresses are translated to real addresses by the MLP pager, which is controlled (via the microvisor) by the MLP driver in TENEX. A reference to a page not in memory results in a page-fault interrupt into the microvisor, followed by a retry of the memory operation after the page is fetched by the TENEX system.

The microvisor is a small resident operating system for the MLP-900; it occupies 512 words of control memory. Most of the microvisor is devoted to swapping user contexts--control memory and MLP registers--as control of the MLP-900 passes from one emulation process to another. The rest of the microvisor responds to emulator requests for service, manages the MLP pager, and performs tasks required by the MLP driver in TENEX.

The MLP driver in TENEX is the sole operating system module added to manage the MLP-900 "I/O device" and to provide access to that device. It allows TENEX processes to create, run, and control subordinate MLP processes in much the same way they can for subordinate TENEX processes. It schedules use of the MLP-900 among contending processes and supervises the microvisor.

The net effect of these operating system components is a shared emulation facility in which each emulator runs in its own context (read-only control memory and registers) independent of all others, accessing its own target (virtual) memory. Each such emulation process is completely controlled by the TENEX process that created it. The TENEX process has potential access to all of the context and target memory and may inspect and/or modify them. As there are no peripherals on the MLP-900, I/O must be performed for an emulator by its controlling TENEX process using TENEX devices or the TENEX file system.

It is worth noting that the operating system level of PRIM was designed, developed, and tested using a TENEX system that, except for normal scheduled maintenance, was in production use almost continously. The hardware interfaces were checked out using a single 16K memory module borrowed from TENEX and a specially built I/O bus simulator that was connected to the PDP-10 via a high-speed EIA terminal line. After the microvisor was developed and the swapping and paging algorithms validated, the memory module began to be shared among concurrent MLP-900 users; on the TENEX side, however, it was treated as a special region private to the MLP-900 checkout process. The driver was part of the checkout process that controlled the simulated terminal. At this point the PRIM system was first available to users. With this validation of the hardware and software interfaces between the MLP-900 and the PDP-10, the I/O bus simulator was removed and the MLP-900 was connected to the PDP-10 I/O bus. An MLP-900 device driver was then added to TENEX, and the MLP-900 was connected to the full PDP-10 memory. Only the checkout of the MLP driver required stand-alone use of TENEX; since the driver logic remained essentially unchanged, this resulted in only a modest extension of maintenance periods. The driver change had no effect on the microvisor.

User Level PRIM Components

The user level of PRIM, i.e., the level of PRIM with which the user interacts, consists of an executable TENEX process that defines and implements the PRIM command language. Both the emulator developer and the emulator user (target machine programmer) are presented with basically the same command structure. All target-machine-specific data are maintained in tables that are referenced throughout PRIM; there is also a table for the MLP-900 itself for use by the (beginning) emulator developer. The command aspect of PRIM is more fully discussed in the section entitled "Using PRIM."

The PRIM process is at all times either at the command level (interpreting and performing user commands) or at the execution level (running the emulator on the MLP-900 and servicing its I/O requests). When the emulator halts or is interrupted by user intervention, PRIM returns to command level. pecific commands are available to start (or resume) execution.

In addition to command interpretation and execution, the PRIM process performs two other functions. First, it contains a module that creates an emulation process for the MLP-900 and controls its execution at the user's behest, thus bridging the gap between the operating system level and the user (at the terminal). Second, it provides an I/O server for that emulation process. There are no peripheral devices attached to the MLP-900; instead, emulated devices are mapped onto TENEX disk files and assignable PDP-10 devices (terminals, mag tapes, and network connections). The user, via PRIM's MOUNT command, associates actual files or PDP-10 devices with the various emulated devices. The emulator developer implements each device by translating its operations into calls to a PRIM I/O server, which then performs the equivalent operations on the mounted file or device. Those I/O server functions that may involve indeterminate or long delays (such as waiting for the next character from a terminal) are handled by subordinate TENEX processes that operate asynchronously.

Microcode is written in GPM, a higher-order machine-oriented language developed at ISI for the MLP-900. A GPM compiler is available on TENEX as a batch process; it is also used in an interactive mode via PRIM to offer the emulator writer line-at-a-time display and/or modification of control memory.

Tool Level PRIM Components

The tool level of PRIM consists of the set of completed emulation tools that are available to target machine programmers plus an MLP-900 tool available to emulator developers. A completed emulation tool consists of an emulator (pure MLP-900 microcode) and a table file describing the target machine. The MLP-900 tool consists of the GPM compiler and a table file describing the MLP-900.

An emulator that is to be a tool must observe PRIM protocols that are not required of a private emulator, which is treated simply as a target program on the MLP-900 tool. These protocols concern emulated timing, the monitoring of potential break conditions, the reporting of conditions to PRIM, and the clean interruption of emulation when it is necessary to return control to the PRIM command level. (These requirements are detailed in ***PRIM System: Tool Builder's Manual.***) All target machine timing is done in emulated (virtual) time. The emulator itself is responsible for counting emulated cycles and scheduling emulated events (I/O and clock) at the appropriate intervals. Break conditions are monitored by the emulator rather than the target system. Break events are logged for use by the PRIM user-level software. Break conditions and status are reported to the PRIM user-level software by interrupting execution of the target system at appropriate places as necessary.

The table file that is required for each tool contains all the tool-specific data necessary for PRIM. It lists all the cells of interest, giving their target names and locations in the emulator's context. It defines the syntax of numbers and expressions for the user, typically taken from (but not limited to) the target assembly language. It lists all the implemented I/O devices by name and details the parameters necessary to "install" them. Finally, it describes the formats of the machine instructions.

The syntax of the assembly language is contained in a table of parsing rules that drive the PRIM instruction assembler/disassembler to provide debugging to the user at the assembly-language level. Without the assembly rules, the generation of the table information is a straightforward task, adding negligible overhead to the cost of an emulation tool.

Using PRIM

From the user's point of view, the PRIM system can be in one of three states: the PRIM exec, the PRIM debugger, or the emulated target machine has control. The user's initial interface to the PRIM system is with the PRIM exec. From there he can pass control either to the PRIM debugger or directly to the target machine. From the PRIM debugger, he can pass control either back to the PRIM exec or directly to the target machine. Once control is passed to the target machine it remains there until target execution stops or is stopped by user intervention.

The PRIM exec and debugger have different command languages and provide different services. The PRIM exec has a large number of commands, most of which are used infrequently. These commands all have fixed forms, with only file names and numerical parameter values having a variable content. The command structure adopted for the exec is basically one of keyword recognition. Because of the large number of keys and the relative infrequency of their use, highly descriptive keywords have been adopted that are easy to remember. The PRIM debugger, on the other hand, has a smaller number of commands, most of which are used frequently. Many of these commands have variable forms and content. The command structure adopted for the debugger is one involving single-key-character recognition with complete specification of variables (names or values).

Certain characters have been assigned editing and intervention functions when input by the user. The editing characters apply only to the PRIM exec or debugger, whereas the intervention characters apply also to the execution state where the target machine has control. The editing functions are backspace, backup, delete, and retype; the intervention functions are status and abort. Whenever the user inputs a part of an exec or debugger command that is unacceptable for any reason, PRIM permits the user to correct and redisplay the command by entering the appropriate editing characters. The status intervention causes PRIM to respond with the current status of the emulation tool. The abort intervention causes any operation in process to be terminated and returns control to the top level of either the PRIM exec or debugger, depending on which one last had control. The abort function is used either to abort a command that has been partially input or is in process or to stop a running emulation. Interventions generate interrupts and are recognized immediately, regardless of the state of the PRIM system.

PRIM Exec State

In the exec state, the user interacts with the PRIM exec. The PRIM exec has two major responsibilities: (1) configuring or reconfiguring a target system and (2) saving target environments for later restoration and use and restoring previously saved target environments. In addition the exec performs a miscellaneous set of other services. The various exec commands are listed in Table 1. When a user first calls on the PRIM system, he is supplied with a target system whose configuration is determined by tables created by the emulator developer. Even for emulations supporting a complete range of peripherals, an initial configuration may have no peripherals "installed," so as to permit the user to configure a system with only those peripherals he needs. A PRIM exec command exists that permits a user to install any peripheral device that is supported by the emulator. As part of the installation process the user is asked to supply characteristics of the device that are parameterized in PRIM or in the emulator, such as the number of lines connected to a communications controller. Other target system parameters, such as the size of memory, may also be set with PRIM exec commands.

Table 1: PRIM Exec Commands

Configuration	Save/Restore	Miscellaneous
FILESTATUS	LOAD*	NO*
INSTALL	TABLES*	CHANGE
MOUNT	RESTORE	CLOSE
PERIPHERALS	SAVE	COMMANDS
REASSIGN	SYMBOLS	DEBUG
REWIND		ENABLE
SET		GO
SHOW		NEWS
UNMOUNT		QUIT
		TIME
		TRANSCRIPT

* Commands for emulator developers only.

After target peripherals have been installed, the user can associate them with real peripherals on the PDP-10 (such as a mag tape unit), with files in the TENEX file system, or with his on-line terminal. This process is called "mounting." When a file is mounted on a target peripheral, the user is asked to supply any characteristics of the file that are parameterized, such as the packing structure (byte size), whether bytes contain ASCII or binary data, and whether the file is to be accessed for input, output, read/write, or append only. If mounting involves ASCII data, there is an implied translation between ASCII and the target peripheral's character set; the emulator developer provides character-set translation tables for each such target system peripheral. As data moves in or out between the emulated peripheral and the PRIM user-level software, all necessary changes in packing and translations of character sets are performed.

The PRIM exec permits the user to save target environment description tables on TENEX files so that the results of a session can be used at a later session. The user can save the total target context, the target system configuration, the contents of target memory, or target system user symbols. The emulator developer has the further ability to save various parts of the emulator and its context, the PRIM debugger context, or part of the context of the PRIM exec itself. Analogous commands permit the user to restore the various saved contexts. Appropriate interlocks exist to guarantee that changes resulting from a restoration of part of the total context do not violate assumptions inherent in the remainder of the context.

PRIM Debug State

In the debug state, the user interacts with the PRIM debugger. The PRIM debugger is a target-system-independent, interactive program for debugging a PRIM emulator or a target program running on such an emulator. It is tailored to a specific target system by tables prepared as part of an emulation. Basically, it permits a user to examine, search, and modify target system locations and set and clear breakpoints in the target system. Target system assembly language and symbolic names are recognized and generated, and arithmetic is performed according to the conventions of the target machine. Four major classes of debugger commands exist: debug control, target system control, display, and storage. The various Debugger commands are listed in Table 2.

Table 2: PRIM Debugger Commands

Debug Control	Target Control	Display	Storage
COMMENT	MLP-BREAK*	MLP-TYPE*	MLP-CHANGE*
FORMAT	MLP-STEP*	EQUALS	CLEAR
IF	BREAK	EVALUATE	SET
KILL-SYMBOL	DEBREAK	JUMP-HISTORY	
MODE	GO	LOCATE	
NEW-SYMBOL	SINGLE-STEP	NEXT	
OPEN-SYMBOL		PRIOR	
QUIT		SAME	
		TYPE	

* Commands for emulator developers only.

Debug control commands allow the user to execute commands conditionally, to transfer control to the PRIM exec, and to designate the form and notation of the debugger's representation of data. Conditional execution is particularly important where a command is to be executed at a later time in an indeterminate context, as when the command in question is in a command file or is a subcommand of a breakpoint command. Target system data may be represented as text or instructions or as formatted or unformatted numeric values; numbers may be represented as symbols or as signed or unsigned integers. The user may specify multi-field data formats, select user symbol tables loaded earlier via a PRIM exec command, and add new symbols to a symbol table or delete existing ones.

Target system control commands allow the user to continue execution of a target system (either at the system's program counter or at a supplied address), to single-step a target program, and to set and clear breakpoints in the target system. Breakpoints can be set on references (read, write, or execute) to target machine locations or on events that have been predefined by the emulator developer (events such as I/O activity, interrupts, clock ticks, and various anomaly states). Breakpointing in the PRIM system is performed as a meta-operation by the emulator, thus using no target system resources. When a break condition is detected by the emulator, it is logged for the debugger and, at some stable state in the target CPU execution, control is passed to PRIM (to suspend any I/O in process) and then to the debugger (to respond to the break condition). The most powerful feature of PRIM breakpointing is the ability for a user to associate with each breakpoint a break-time debugger "program" that is to be executed when the break condition is detected and control is passed to the debugger. Coupled with conditional execution of debugger commands and the ability to continue target system execution, breaktime programs permit the user to trace execution, monitor events, and perform other complex exploratory functions automatically and repeatedly.

Display commands allow the user to examine the contents of designated target locations or to search extended areas within the target system according to content. Displayed locations may be modified. Storage commands allow the user to change the contents of target system locations without having to display them first.

Execution State

In the execution state, the emulator has control. If the user has "mounted" his interactive terminal on one of the emulated devices, then when control is passed from either the PRIM exec or debugger to the emulator, the user's terminal is also switched onto the emulated device. Consequently, while in the execution state, outputs for that device are sent to the user's terminal, and inputs from his terminal are treated as coming from that device. The user's terminal can be "mounted" on any emulated device for which the emulator developer has defined a mapping between ASCII characters and target system bytes. Although the user's terminal can be mounted for input on only one emulated device at a time, it can be mounted for output on several emulated devices concurrently. Even when in the execution state, with the user's terminal mounted on an emulated device, the terminal's status and abort intervention characters are treated as PRIM (rather than target-system) inputs; where the conventional PRIM intervention characters have been preempted by the target system, the user can assign the intervention functions to other ASCII characters.

Dynamics of Interaction

A typical PRIM session starts with a target-system user restoring a context from a file saved at the close of a previous session, thus reestablishing the state of that session. If a complete context is restored, all files that had been mounted on target peripherals are automatically remounted and repositioned to the proper byte. The user then normally transfers control to the debugger, where he can "operate" the emulated console switches, manipulate machine locations, and establish breakpoints. When he is ready to resume testing, the user transfers control to the target machine.

Test inputs typically are supplied either via the on-line terminal or via input files that had been prepared previously (either of which can be mounted on a target peripheral). The target system normally is run with these inputs either until a breakpoint occurs or until the user intervenes to operate console switches or because the system is not behaving as expected. When control gets to the debugger, the user explores the target system to identify the cause of the problem; he can replace bad code or correct data so as to permit the system to continue executing. If he must terminate a session in the middle of some operation that cannot be recreated easily, he can save the necessary context for the next session.

Special facilities are available to the emulator developer: additional commands exist in both the PRIM exec and debugger specifically to support emulator development. The source code for an emulator is created outside of the PRIM system; similarly, the compilation of that code occurs outside of PRIM. But the resulting binary version of an emulator may be loaded into that portion of the PRIM context that maps onto the MLP-900 memory, registers, and flip/flops. Similarly, the file describing the target machine is created and assembled outside of PRIM to produce loadable target-machine description tables; these tables also can be loaded into PRIM by an emulator developer. When he has done this, the emulator developer then can work with a PRIM tool consisting of the MLP-900 within which emulator microcode is an object program, and concurrently he can work with his prototype emulation tool within which target code is the object. He has the ability to single-step or set breakpoints both within emulator microcode and target-system object code, and he can examine and modify both data and code in both the MLP-900 and target system. To the extent that he has created target-system assembly language syntax rules, he can use target system symbolic assembly notation.

The ease with which the user can (1) pass control among the PRIM exec, PRIM debugger, and target system, (2) explore and modify the target system, and (3) recreate or repeat error conditions permits errors to be identified and corrected far more rapidly than in any but the most sophisticated native systems.

Requirements for Reproducing PRIM

Since the MLP-900 is a one-of-a-kind machine (although not originally intended as such), we have naturally given some thought to the building of PRIM-like systems on readily available equipment. The most obvious requirement is for a microprogrammable processor and a general-purpose timesharing system on some computer that, taken together, can provide the base system on which the operating-system level of PRIM can be reproduced. The user level of PRIM can be rewritten for any general purpose system. Although the existing code is directly transferable only to another TENEX system, most of it can be transferred to any PDP-10-based operating system.

There are three principal requirements for a replacement for the MLP-900: an interfaces to the main system must be buildable, a protection mechanism must exist or be implementable for emulators, and context must be swappable. The ability to build the interfaces probably depends more on the choice of main system than on the microprogrammable processor. The most important requirement is that the microprogrammable machine support the mechanisms necessary to protect all the resources of the main system--control interface and memory--and to guarantee main-system control of the microprogrammable host. The swapping requirement is easily taken for granted, but for many microprogrammable processors the ability to interrupt the running microcode, unload it, and restore it exactly at a later time simply does not exist. The only commercially available machine of which the authors are aware that could satisfy all of these requirements without extensive hardware modifications is the Nanodata QM-1. Since the QM-1 is itself nano-programmed, the necessary mechanisms can, at least in principle, be supported at the nano level.

The requirements on the operating system and its general-purpose computer are less severe. Obviously, the hardware must be capable of supporting new, strange devices, and the operating system must be flexible enough to allow new device drivers to be created. In addition, the computer must have an externally accessible memory, and the memory management and address translation must be simple enough to allow the implementation of adequate memory protection in the microprogrammable machine. The issues likely to dictate the choice of operating system and computer, in practice, are not technical. Rather, they are issues of access to the operating system and its code, and maintenance of the hardware and, especially, the operating system. In these respects, TENEX has been a nearly ideal system.

Conclusions

As PRIM represents a new architecture for the support of emulation, it is instructive to compare it with the principal alternatives designed to support similar functions. The two most common existing designs are simulation systems, which run on general purpose computers, and emulation systems, which typically run on stand-alone single-user (usually hands-on) microprogrammable computers. Another design possibility is the creation on a large microprogrammable computer of a complete new operating system (at the microprogram level) to support multi-user multiple emulations.

PRIM looks to the target programmer much like a good interactive simulation system, except that MLP-900 emulated target-cycle times are in the range of 5 to 30 microseconds whereas for simulation systems they are in the millisecond range or worse. This difference is very often the margin between a productive facility and an unproductive one.

A self-hosted operating system could in principle create an emulation facility of power equal to PRIM and with lower run-time overhead than PRIM requires. To achieve similar results would require an operating system in the microprogrammable host that has most of the features of a modern timesharing system such as TENEX. The only drawback to this approach is the cost of producing such an operating system. As far as we know, no one has built one to date.

In comparing the PRIM system to a stand-alone, single-user microprogramming facility--the most common form currently found--three principal differences appear: coupling of processes, time-shared and remote access, and reliance on virtual time. The coupling of the MLP-900 to TENEX provides the PRIM user with a complete facility in which his emulator appears as just another process (subsystem, or task) much like an editor, compiler, or file management function. The user has TENEX, with its full power and all its utilities, as his sole point of interaction. (Note that this applies to the microprogrammer developing an emulator on the MLP-900 tool as well as to the target programmer using an emulation tool.) The use of the TENEX file system for emulated I/O results first in an automatic separation of data areas for different users (thus affording protection) and second in easy application of TENEX utilities to those files, so that a "deck of cards" can be created with a text editor or a "bootstrap tape" with a loader.

Time-sharing of the MLP-900 itself eliminates the need to schedule block use of the emulation host; all PRIM users have access at all times and are entitled to a share of the MLP-900. And with TENEX providing the user interface, all of its facilities for timeshared and remote access pertain to PRIM. Time-sharing also makes better use of the MLP-900 since in PRIM's interactive program development environment the user typically spends far more time talking to the PRIM exec or debugger about his machine than he does actually running it. Only for the case in which several users wish seriously to exercise their emulators or target machines does the overhead of I/O, scheduling, and swapping become apparent. A fringe benefit of time-sharing PRIM and the MLP-900 is the opportunity to produce a multi-machine emulation system without having to create a multi-machine emulator. Rather, one emulation process can be initiated for each machine in the emulated system, with the interconnections made through the PRIM I/O facility.

The use of virtual time in PRIM emulation tools is necessitated both by PRIM's asynchronous I/O and by the time-sharing of the MLP-900. There are decided advantages to virtual time over real time (or "proportional real time") for purposes of timing the target machine. Virtual timing is both accurate and completely reproducible. By freeing the emulator from any dependence on actual device or memory speeds, it permits time to stop whenever the emulator is stopped. As a result, timing-dependent bugs are both reproducible and amenable to treatment by normal debugging aids, which are outside the virtual time framework. Furthermore, virtual timing provides a simple way to alter target timing relationships, either for stressing time-critical code or for experimenting with effects of faster (or slower) devices.

To date three complete emulation tools have been developed to run under PRIM: an AN/UYK-20, a U1050, and an Intel 8080. The first two of these have been implemented with an essentially complete set of emulated peripherals. All three PRIM tools have been in use, both at ISI and by remote users via the ARPANET, since early in 1976. It has been our experience with all of these tools that PRIM has provided a more convenient vehicle (sometimes, dramatically so) for writing and debugging programs than the original target system.

Because of the uniqueness of the MLP-900, we have explored alternatives for duplicating the capabilities of PRIM with current-generation commercial systems. The preliminary results of these investigations suggest that a TENEX operating system running on a DEC KL (1040 or 1090T) hardware system with a Nanodata QM-1 microprogrammable processor interfaced as a direct-memory-access I/O device would duplicate the PRIM architecture and minimize the new development cost.

REFERENCES

1. Bobrow, D. G., J. D. Burch, D. L. Murphy, and R. L. Tomlinson, "TENEX, A Paged Time-Sharing System for the PDP-10," *Communications of the ACM*, Vol. 15, No. 3, March 1972, pp. 135-143.

2. *MLP-900 Multilingual Processor--Principles of Operation*, STANDARD Computer Corporation, Santa Ana, Calif., 1970.

3. *PRIM System: User Reference Manual* (in progress).

4. *PRIM System: Tool Builder's Manual* (in progress).