The development of this timesharing system led to insights on microprogrammable organization, instruction sets, reliability, and software and firmware development tools.

The Maxc Systems

Edward R. Fiala
Xerox Palo Alto Research Center

The process of developing a computer system is not only inherently interesting; it also leads to significant organization concepts that the builders are often impelled to share with others. So it was in our development of the Maxc1 and Maxc2 timesharing systems at the Xerox Palo Alto Research Center between 1971 and 1977. From this development came some ideas of system organization that are now seen to have contributed to the success of the effort:

The high availability achieved is attributable to the simple microprogrammable organization of the machines.

Microprogramming organization promotes simplicity by placing much of the complexity in firmware.

This organization of a computer provides the environment for multiple instruction sets.

Causes of failure in integrated circuitry were evenly distributed, but memory error correction was found to be important to overall reliability.

Tools for software and firmware development and design automation are necessary for efficient development.

The Maxc1 and Maxc2 systems

The Maxc1 system was designed and completed during the period from February 1971 to April 1973. Maxc2 was designed during 1973, shelved for two years, then finally built and debugged between June 1975 and April 1977. Despite being a one-of-a-kind system, Maxc1 has been one of the most consistently available systems on the ARPA network since 1974. We attribute this high availability to the system's simple microprogrammable hardware organization and input/output structure.

Maxc is a medium-scale computer designed to run the Tenex timesharing system[1] and Interlisp language[2] developed by Bolt, Beranek, and Newman for ARPA. Tenex was developed on a DEC KA10 processor[3,4] modified by a large paging box designed by BBN. Our principal reasons for choosing Tenex were to acquire Interlisp for artificial intelligence research and to connect to the ARPA network[5,6] so that the PARC staff could be part of the research community sponsored by ARPA.

The overall mainframe organization of Maxc1 is shown in Figure 1 and discussed in the related box. High-speed operation was not a major design objective for the Maxc systems. For significantly higher performance, the slow access time of MOS main storage would have to be bypassed by some cache or instruction prefetching scheme. The methods considered were sufficiently complicated to discourage us from pursuing them.

For PDP-10 emulation Maxc2 achieves an overall power (including swapping delays and storage access time) about 20 percent greater than a KA10, 20 percent less than a KI10, or 40 percent less than a KL10 with similar memory and I/O gear. These comparisons ignore the special advan-

0018-9162/78/0500-0057$00.75

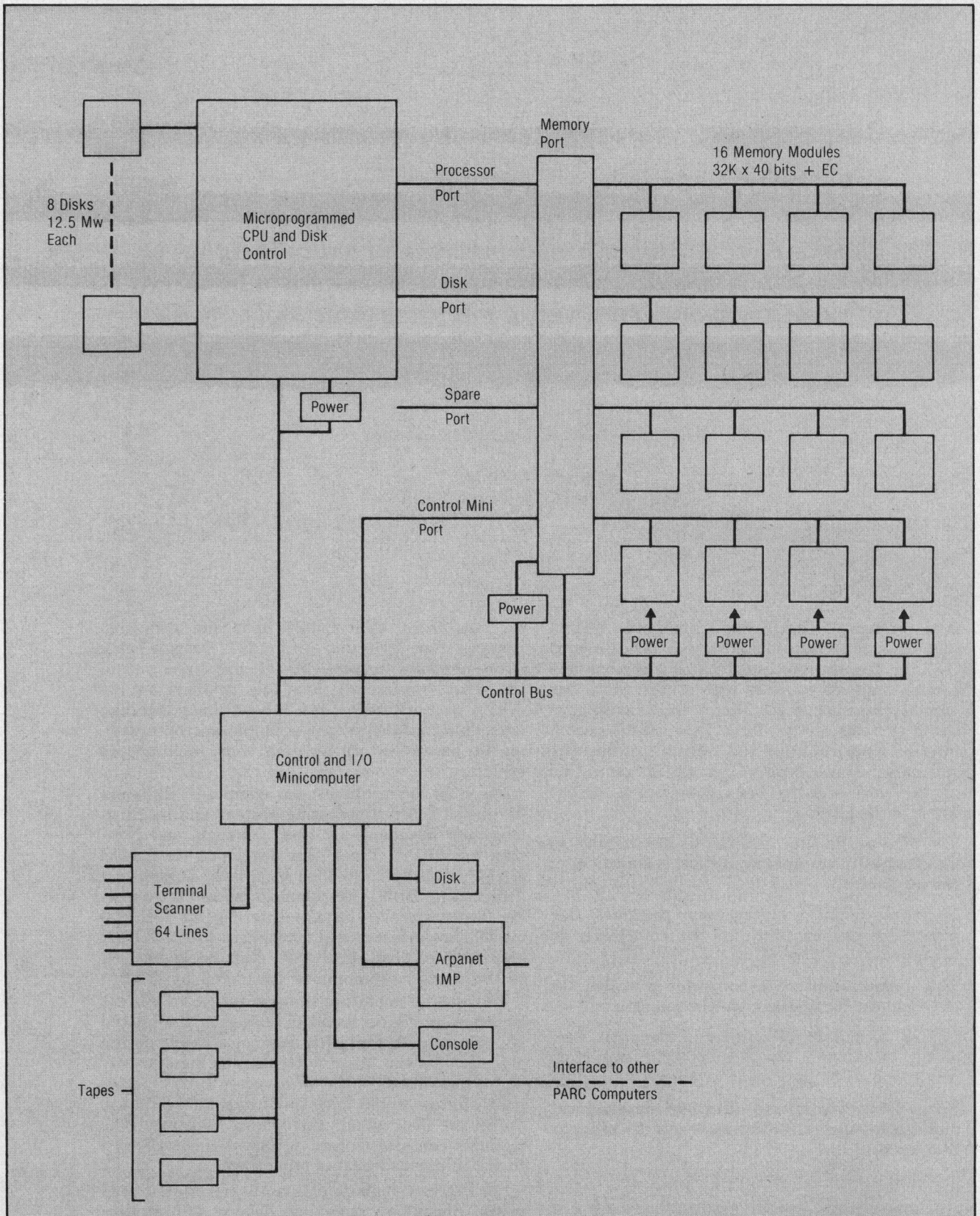


Figure 1. The Maxc system consists of a microprogrammable central processor, disks and disk controller, and main memory, with a separate minicomputer for control and I/O functions (see box).

tage of Maxc for running the Interlisp system (discussed later) and are based upon the performance reported by Bell et al.[4] A comparison based only upon CPU speed would show a factor of about four difference between the KA10 and KL10.

A number of other computer systems[7-14] are based on microprogrammable processors similar in various ways to Maxc. Peuto and Shustek,[15] Hollaar,[16] and Smith[17] discuss ideas that are also of interest. The BCC-500 system, developed by the Berkeley Computer Corporation and now at the University of Hawaii (nothing published), is also similar. Another system that emulates the PDP-10 instruction set is the Superfoonly system at the Stanford Artificial Intelligence Laboratory (no published references on Superfoonly architecture).

Microprogrammable processor characteristics

The Maxc microprogrammable processor layout, shown in Figure 2, is largely general purpose with light specialization for the Tenex virtual memory and PDP-10 instruction decoding. PDP-10 specialization consists of (1) a function for setting the overflow, carry0, and carry1 bits in the flag register for arithmetic; (2) two special bus destinations that route instructions and indirect words into registers; and (3) a bus destination and bus source to manipulate byte pointers. Tenex virtual memory specialization consists of (1) the 1024 x 18 MAP memory, which contains execute, read, and write permission bits and the absolute address

Hardware Parameters of the Maxc Systems

Main storage	393,216-word x 48-bit, 800-nsec access
Secondary storage	Eight Century Data Systems 2314-equivalent disk drives with 3M disk packs
Input/Output	Controlled by Data General Nova 800 minicomputer
Technology	The microprocessor is composed of TTL IC's with a sprinkling of Schottky in time-critical places. PC boards are used for the instruction memory, main storage, and ALU sections. Wirewrap is used elsewhere.
Cycle time	200 nsec/microinstruction
Word size	36 bits (memory words also have four tag bits)
Microstore	2048 words x 72 bits RAM
Instruction rate	9 to 10 cycles/simple PDP-10 instruction

Maxc2 is similar, but has a cycle time of 150 nsec, twice as much microstore, and one of our Alto minicomputers to control peripherals. The memory and disk control sections are identical. The Maxc2 microprocessor, about 25-percent different from Maxc1, uses denser storage IC's to provide twice as much microstore; the bus structure was significantly modified to increase speed. Maxc2 is not connected to the ARPA network and is accessed from Alto minicomputers within PARC.

MOS main-memory words contain 36 data bits and 7 bits for error correction/detection (5 other bits unused). The memory routes requests from four independent ports into independent memory quadrants. Both the microprocessor and control computer treat memory as an input/output device. The microprocessor uses one port exclusively for disk traffic and another for interpreting instructions.

The control computer (Nova on Maxc1, Alto on Maxc2) appears in two roles. It has complete control over the processor and memory, including powering up and down and changing memory interleaving and error correction. It is used to load, start, and debug microprograms and to troubleshoot hardware problems. Under Tenex, it runs input/output drivers for all system peripherals (except the disks) as a slave to Tenex; communication takes place through main memory using an ancillary hardware signal in each direction to initiate communication. When a fatal error occurs, such as a double memory error or microprocessor halt, it again becomes the master and takes appropriate action.

To give some idea of machine size, here is the approximate IC count for various sections of the main frame:

Section	Maxc1	Maxc2
Microstore	832	480
Internal memories	312	224
ALU	300	300
Disk control	260 + 60/unit	260 + 60/unit
Other microprocessor	850	850
Memory port	720	720
393,216x48 storage	18,500	18,500
Control computer	Nova	Alto
I/O	—	191
processor	—	274
memory	—	1,408

It is clear that the major part of the system and its construction costs was in the memory. However, an equivalent system designed today would benefit from 16K x 1 MOS RAMs for main storage and other new parts provided by the semiconductor manufacturers.

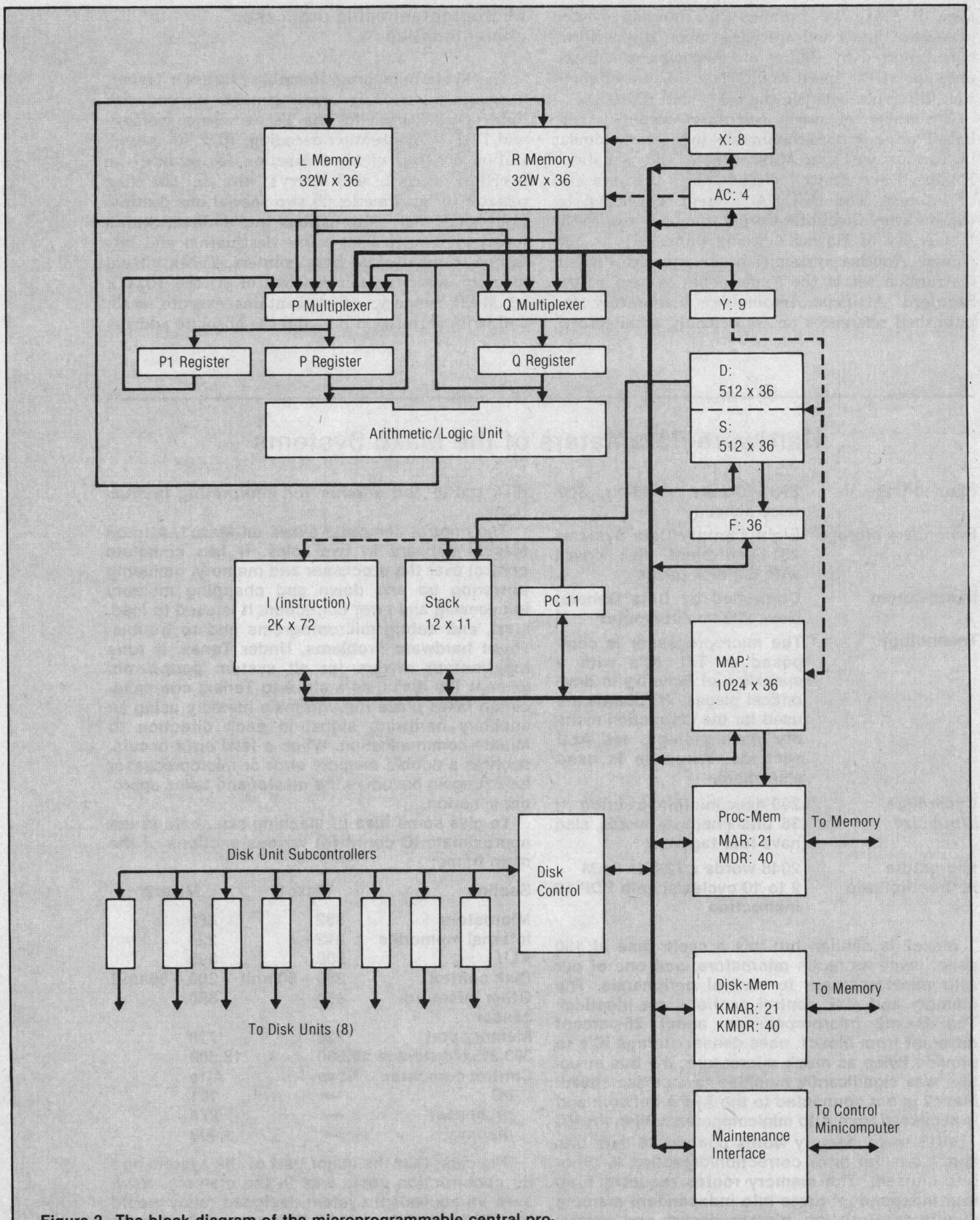


Figure 2. The block diagram of the microprogrammable central processor illustrates several features of general interest, input multiplexing and instruction decoding.

for each 512-word page in the virtual memory; (2) a collection of functions that read, write, or read-modify-write the virtual memory, making various access checks in the MAP.

Several aspects of the microprogrammable processor design in Figure 2 may be of general interest. First, the input multiplexing to the P-register allows cycling the 72-bit value in P and Q by any amount between −3 and +39. The 36-bit cycler output can then be masked by a right-justified mask before loading this result into the P-register. The cycler/masker was not present in early design iterations, and most tightly microprogrammed loops disappeared when it was added. The cycler/masker is used widely throughout system microcode.

The second aspect of general interest is the way instruction decoding is carried out. A PDP-10 instruction in the MDR (memory data register) is first split so that the opcode, index field, AC field, and indirect bit wind up in registers. Then the D (dispatch) memory is addressed by the opcode and three 12-bit microaddresses read from D are pushed onto the stack ('simulated calls'). The main loop of the emulator then "returns" to each of the microaddresses on the stack to emulate the instruction.

This three-microaddress technique is particularly appropriate for PDP-10 emulation, because the PDP-10 instruction set is divided into families whose members are typically characterized in several different ways. These include ways of obtaining arguments (from memory or immediate), common actions performed by all members (e.g., move, compare, test, add, subtract, or multiply), and several ways of disposing of the result (e.g., to AC or memory, or both AC and memory). Thus, the three microaddresses indicate one routine for setting up arguments, one for doing the work, and one for disposing of the result. Consequently, families require very little microcode. For example, the family of 64 test-and-set opcodes is implemented by only 8 microinstructions.

Our Maxc machines use fairly wide microinstructions (72 bits), as shown in Figure 3, so that many sections of the processor can be controlled in parallel by microprograms. Several observations about microinstruction decoding are worth making.

First, it is frequently worthwhile to encode microinstructions tightly. On Maxc1 about 27 percent of the processor IC's are in the microstore; on Maxc2 about 18 percent. There is a tradeoff between microinstruction bits and decoding logic, and designers should make this tradeoff carefully. Maxc microinstructions are not very tightly coded—a paper study showed that all of the power of the machine could have been obtained with 54-bit microinstructions and a little more decoding logic. Frieder and Miller[18] also discuss encoding in the setting of two-level interpretation.

Another valuable technique is having several ways to encode common actions. On Maxc there are four microinstruction fields, bus source (BS), bus destination (BD), function 1 (F1) and function 2 (F2), that can loosely be called "function" fields; these control activities outside the ALU section of the microprocessor. The most common bus sources are encoded in BS, most common bus destinations in BD, and practically everything else in F1. F1 also duplicates common BD and BS encodings to allow several sources to be OR'ed on the (open-collector, low true) bus or several desinations to be loaded; F2 duplicates common F1 codes and implements several other functions used in conjunction with F1. In this way, if one wants to do two things at once in an instruction, it is nearly always possible to encode them.

A few opinions about machine features

Hardware capabilities can be divided into several categories:

1. Input/output operations absolutely required to carry out some task.
2. Functions generally useful in a wide variety of ways, not aimed at any particular application.
3. Special-purpose capabilities provided to shorten particular time-critical tasks for specific applications.
4. Features of marginal usefulness that "fell out" of the hardware design.
5. Facilities for hardware checkout or maintenance.

Category 4 features should be avoided, since they create compatibility problems on later versions of the same machine. For example, the PDP-10 has over 50 absolutely redundant opcodes and 50 more that are rarely used, probably fallout from an early PDP-6 implementation. However, every subsequent PDP-10 has had to stay compatible, and diagnostics have to check these opcodes. IBM has also had trouble with not useful features and strange, seemingly unimportant side effects in its 1401 and 360 designs. As a result,

Branch Address (12)	Branch Type (2)	Branch Condition (5)	L Address (5)	R Address (5)	ALU Function (5)	Bus Source (5)	P Select (6)	Q Select (3)	Bus Destination (5)	Function 1 (6)	Function 2 (4)	S Address (8)	Brk (1)

Figure 3. The Maxc microinstruction is fairly wide (72 bits), permitting many sections of the processor to be controlled in parallel. Different opcode fields are related to different processor functions.

different machines in the same family and subsequent machines that are compatible through microprogramming are needlessly complicated and slow.

The proper treatment of category 4 features is to purge them from software and documentation as early as possible and from the hardware, if it does not cost much.

There is a corollary to this argument also. Operations useful and inexpensive in the current hardware environment, but expensive in likely future hardware environments should be avoided. Designers should weigh current utility against compatibility problems in future implementations.

Many areas with complicated logic inside the Maxc processor have caused few difficulties in either initial checkout or subsequent maintenance because a few lines of diagnostic program realistically check them out and pinpoint failures. However, two time-consuming checkout problems were disk transfers (including interrupt system) and multiport memory competition. The long logic paths in these sections mean that diagnostics cannot pinpoint failures. Hardware features to help checkout these areas would have been useful.

With regard to properties of a microprogrammed processor that make it perform quickly on some range of tasks, the following generalizations are offered: Most tasks have one or several "main loops," the speed of which largely dominates the overall system speed. Special-purpose features (category 3) are frequently required to make these critical routines go quickly. My opinion is that a general-purpose machine of the same complexity cannot approach the efficiency of a machine with a few such special features. On the other hand, general-purpose features such as the cycler/masker discussed above are also useful. Thus, it seems that a practical machine should have both special and general capabilities.

System availability

System availability is summarized in Table 1 for the period from April 3, 1977 (when Maxc2 was made available to users), to January 10, 1978. Failures of noncritical input/output devices such as tape drives and IMP were not logged—these do not crash the timesharing system, and most users are never aware of such failures.

We run each system until it crashes, then repair or bypass whatever failed and carry out or schedule preventive maintenance, if this seems required.

Each system has one extra disk drive normally used for file system backup. When a disk drive fails, we switch it out of the file system and reconfigure. In some cases this results only in a short service interruption and the timesharing system continues in the new configuration. Repairs to disk drives switched out of the file system and to other peripherals are normally carried out while running the timesharing system.

Table 1. Reasons for failure are widely distributed (12 of the 13 port failures on Maxc2 were caused by a single intermittent failure).

Reason for Downtime	No. of occurrences Maxc1	Maxc2	
Memory failures	4	2	
Port/connector failures	7	13	
Processor failures	7	1	
Disk failures	3	2	
Power supply failures	0	1	
Power/air conditioning failures	3	2	
I/O and control computer failures	4	5	
Software/firmware bugs	6	2	
Operator errors	2	4	
Undiagnosed failures	2	3	
Preventive maintenance	5	1	
Software/firmware development	3	5	
System reconfiguration	1	2	
Total	47	43	in 283 days

The fact that all secondary-storage devices are identical and interchangeable has undoubtedly contributed to high system availability. It is advisable that other sites seek a simple storage arrangement also. Some sites use low access time drum devices for swapping, but our experience suggests that additional disk drives or main storage modules provide a better enhancement of system power because maintenance complications that result from introducing an additional kind of peripheral are avoided.

The most significant contributor to reliability has been main-memory error correction. During the first six months of operation, we replaced about 12 failing 1Kx1 MOS RAMs per month; this has gradually declined to about three failures a month during the last three years. However, because of error correction, a negligible number of these failures has caused crashes. Even if the failure rate had been lower, protection against intermittent and pattern-sensitive failures would fully justify error correction.

When Tenex restarts, it runs a short memory-diagnostic program that records areas affected by bad storage IC's; Tenex does not use these bad areas. Only when the amount of storage affected becomes significant do we schedule downtime and replace bad IC's.

The majority of processor IC failures are solid. This is fortunate since it means that a problem can be fixed and will not result in more crashes later. However, when an intermittent failure does occur, it may cause reliability problems for a considerable time. In Table 1, for example, 12 of the 13 port failures on Maxc2 were caused by a single intermittent failure.

Our experience operating the Maxc systems has not suggested any low-cost method for signifi-

cantly improving availability or reducing operating costs. The table shows that reasons for downtime are distributed fairly evenly over many causes, so no single improvement would be particularly significant. Perhaps applying memory error correction end to end, so that port as well as storage components were guarded, would be better. Beyond that, the operating system could be developed so that more disk and memory failures do not cause system crashes.

Firmware characteristics

The original Maxc system emulated the PDP-10 user mode instruction set. For supervisor mode, we implemented additional instructions to control disks, priority interrupt system, and virtual memory map, and to signal the control computer. In 1976 we added an additional instruction set specially designed to support Interlisp. Table 2 shows roughly how the microcode is divided among different uses.

Table 2. Some 2000 microinstructions implement the Maxc system, although 5000 more are used in the diagnostics.

Use	Instruction Count
PDP-10 instruction set	744
Interlisp instruction set	472
Tenex Map	157
Disk	228
I/O instructions	237
Priority interrupt system	107
Other	93
Total	2038
Diagnostics	~5000

The microprogrammed hardware organization greatly simplified implementation of complicated parts of the system. For example, the very complicated Tenex virtual-memory scheme was implemented efficiently with little special hardware—a non-microprogrammable organization, such as the BBN pager, would have required several hundred more IC's. Also, disk control hardware was simplified because main memory transfers, header checking, preambles, postambles, and checksums are all handled by firmware rather than hardware.

For floating-point instructions, the main advantage of microprogrammed organization is flexibility. Correct implementation of floating point is sufficiently obscure that it is hard to design a hardware floating-point unit correctly, but in our microprogrammed machine this complexity is confined to the firmware, where it is much easier to change.

Diagnostic firmware on a microprogrammable machine is simple and can generally pinpoint problems more precisely than on a non-microprogrammable machine. On Maxc a very small kernel of working hardware must be debugged with test sequences from the control computer. Thereafter, firmware diagnostics are used for testing and generally pinpoint failures to three or four IC's.

The Maxc firmware contains redundant checks that cause a halt when inconsistencies are detected. These checks are inserted in places where no execution time or space penalty is incurred. Also, a diagnostic instruction executed occasionally by Tenex verifies by checksum that the microstore and constants in other internal memories of the processor are correct and does a few other simple tests that checkout most of the processor hardware.

Finally, microprogrammed organization has made possible an additional instruction set called Byte Lisp into which Interlisp programs are compiled. Concurrent use of multiple instruction sets is a powerful concept elaborated more fully below.

Systems with multiple instruction sets

The PDP-10 instruction set is a well-developed "classical" set with a huge repertoire of simple instructions for doing logic and arithmetic and moving data from place to place. In addition, it has a few more complicated instructions for dealing with special data types (floating-point numbers and pushdown stacks). However, if one compiles a high-level language program into the PDP-10 instruction set, the result falls far short of what is possible.

Classical instruction sets result in compiled programs that are slower and larger than is easily achieved by a special-purpose instruction set. To illustrate, in the course of enhancing the Maxc Interlisp implementation, we identified five or six key places where programs were spending considerable time (function call and return, type testing, garbage collection, etc.). We added special PDP-10 instructions to reduce the time spent in these places. This resulted in about a 25-percent speed improvement over the unmodified instruction set. Special-purpose firmware of this sort has frequently been used on microprogrammable systems[19-21].

The Byte Lisp instruction set, based upon Deutsch,[22] uses the same virtual memory and registers as the Tenex instruction set, but instructions are 9-bit bytes rather than 36-bit words, and opcodes are specialized to operations statically frequent in Interlisp programs.[23-25] Byte Lisp compiled functions average 72 percent smaller and 15 percent faster (on top of the 25 percent mentioned above) than PDP-10 compiled functions.

Code size is more significant in Interlisp than in typical application programs. In a typical application program, program storage, if significant, is usually dominated by size of data objects being manipulated. In this kind of application, a classical instruction set works fine (perhaps modified by special microcode in some time-critical places). However, Interlisp is a large system into which features have been added continuously for about ten years. The standard system has about 200,000 words of PDP-10 compiled code (some systems are much larger), and the typical working set is about one-half PDP-10 compiled code. When compiled instead into Byte Lisp, program disk storage is reduced by about 50 percent and working-set size of programs being executed by about 30 percent.

Naturally, the performance comparison is different on a machine like the PDP-10 with large 36-bit words from what it would be on a machine that used, say, 16-bit words. One would expect greater speed improvement and less size improvement on machines with smaller word size. Byte Lisp is not an independent instruction set—Interlisp uses many PDP-10-coded procedures, and some Byte Lisp instructions trap to PDP-10 coded subroutines for execution. Because the other instruction set remains available, Byte Lisp can concentrate on representing compiled programs compactly without also having to provide infrequently-used opcodes for unusual types of data manipulation.

In other words, Byte Lisp *opcodes* were chosen for common operations in Interlisp compiled code—the goal was to represent compiled programs compactly. But the decision to execute an operation in microcode was governed by how much this would improve performance. Thus, the inner loops of the garbage collector were microprogrammed, since these consumed considerable CPU time, even though they were statically infrequent in Interlisp. Conversely, CONS, one of the three most common operations in Interlisp, was made an opcode because it occurred frequently in Interlisp compiled code; but CONS trapped to a PDP-10 procedure for execution since microprogramming would not have improved performance much.

Extra instruction sets such as Byte Lisp do not require much additional microstore because they share support machinery with the classical instruction set (e.g., the divide and multiply microcode is shared), and because the map, disk, priority interrupt, and I/O microcode do not have to be replicated for additional instruction sets. Because Byte Lisp and PDP-10 instruction sets use a common virtual memory and registers, the Tenex operating system is almost unaffected by the existence of the extra instruction set.

The use of microprogramming to implement alternative instruction sets has been used on a number of systems (e.g., 1401 compatibility on IBM 360), and the idea of instruction set specialization for a high-level language is also popular (e.g., Algol specialization on Burroughs machines). However, the idea of cooperating, compatible instruction sets is not generally appreciated, and this is the idea that has been successful on Maxc.

As a result of these experiences, it appears that on a microprogrammed machine, the desired architecture is a family of instruction sets sharing common word size, common registers, and a common virtual-memory organization (which should be very large—262,144 words on Tenex is much too small). One of the instruction sets should incorporate operations for input/output and other classical bit-manipulation operations. Other instruction sets should be provided for principal compilers. These instruction sets need not be self-sufficient, but can trap to the general instruction set in situations where microprogramming does not offer much advantage.

Software tools for system development

Overall time required to design and build the Maxc1 system and to get Tenex running on it satisfactorily for users was about 26 months—12 months to design the system, 8 months to debug the hardware, and 3 (non-overlapping) months each for firmware and Tenex software; an additional 3 months of software debugging were required after system release before it was reliable. The software and firmware debugging actually took longer, but most time overlapped hardware debugging. It might be more accurate to say that most firmware debugging time was spent debugging hardware and most software debugging time was spent debugging the firmware. During the course of the Maxc1/2 development and other projects we have gradually developed a number of software tools that speed up the development cycle of systems such as Maxc.

To speed up hardware design, we have developed design automation software to mechanize aspects of creating and revising logic drawings, wire lists, etc. Our current software tools would substantially reduce the 12 months required for Maxc1 hardware design, if that were to be repeated today. Two other design automation systems, SUDS and the S-1 design system, both at the Stanford Artificial Intelligence Laboratory, are similar to the PARC system. Unfortunately, there are no published references on any of these systems.

In addition, we developed a machine-independent microassembler called Micro and a hardware/firmware debugging program called Midas. Other efforts along this line are numerous.[26-40] These software tools run on the control computer (Nova on Maxc1, Alto on Maxc2). The nature of the hardware interface used by the control computer is discussed in the hardware box.

In bringing up a new hardware system quickly, it is essential to have substantial diagnostic software/firmware support as soon as possible. It is impractical to launch into hardware debugging without this support. There is a brief period between

the point when hardware design is complete and the time when the first hardware assemblage is ready for checkout. This period has been getting shorter as design and construction automation has progressed. Debugging delays are avoided by providing a large complement of debugging firmware/software during this small time period.

Micro is a cross-assembler that runs on an Alto minicomputer. It is possible to define a natural microassembly language for a wide range of machines by means of memory, field, macro, and neutral definitions in the Micro language. We have successfully used Micro to define assembly languages for four different microprogrammable machines.

Micro differs from conventional macroassemblers in its unusual parsing algorithm and use of neutral symbols and left-arrow clauses. Roughly speaking, the definition file for a particular machine consists of memory definitions for each of the target memories (S, D, R, L, and IM on Maxc), definitions of various fields within memory words, and macro and neutral definitions that transform symbolic expressions into field stores in memory words.

Neutral definitions are used to represent classes of objects. For example, on Maxc the ALU operations (defined by the P+Q, P−Q, etc. macros) are in the class defined by the neutral "ALU;" bus destinations (X←, Y←, etc.) are defined by the neutral "B←;" legal data paths are represented by macros such as "B←ALU."

The following example shows how macros and neutrals are used to assemble data for an instruction memory word:

```
X←P+Q, P←777777R, GOTO[XAC,G=1]; *Typical
  microinstruction
```

"X←P+Q" and "P←777777R" above are left-arrow clauses and GOTO is a macro. In expanding the "X←P+Q" clause, Micro first parses "P+Q;" this is a single symbol because "+", "−", etc. are ordinary symbol constituents without special meaning in Micro. Evaluating the "P+Q" macro stores the code for "add" into the ALUF field of the microinstruction and leaves behind the neutral "ALU." Then evaluating "X←" stores the code for load-X into the BD (bus destination) field of the microinstruction and leaves the neutral "B←." Finally, evaluating the macro "B←ALU" stores the code for bus←ALU into the BS (bus source) field of the microinstruction.

Micro also contains an assortment of operations for manipulating integers, controlling conditional assembly, handling literals, and dealing with memory addresses. These are similar to features in conventional macroassemblers and are not discussed here.

Midas is a largely machine-independent cross-debugger used for hardware and firmware checkout. Because most of Midas is machine-independent, it can be adapted to a new hardware system quickly. On the Maxc systems, Midas runs on the control computer; versions of it exist for other microprogrammable systems as well.

The machine-independent part of Midas contains software for displaying registers and memory words, for loading microprograms assembled by Micro, and for driving register and memory tests with various data patterns. Most software for controlling stop, go, breakpoints, etc. is also machine-independent. An elaborate command file facility (really a programming language) is used to run diagnostic microprograms and report failures. Most of the documentation for Midas is also machine-independent.

The machine-dependent part of Midas consists of procedures to read-write registers and memories, to start and stop the hardware, and to symbolically print registers, memories, and other signals read from the hardware.

Midas displays the names and values of registers and memories on the large Alto display, and uses the keyboard and a pointing device called the "mouse" to enter new commands. The displayed values are automatically updated at breakpoints and other times when the state of the microprocessor is modified.

During the earliest stage of hardware debugging, registers and memories in the processor and memory system are read and written from Midas through a hardware interface to the target machine. A basic kernel of processor hardware is debugged by these tests from the control computer. When registers and memories can be read and written successfully, subsequent debugging is carried out by diagnostic microprograms that halt on detected errors. The user interface to diagnostic microprograms is provided by Midas command files that control loading, display of interesting memory words, execution, and failure reporting of the diagnostics.

On Maxc a sequence of basic diagnostic microprograms tests registers, memories, and functions with cycled ones and zeroes. These are supplemented by random-number reliability tests for various sections of the microprogrammed processor. PDP-10 programs are used to test disk and main storage reliability. Main storage failures and about 90 percent of processor failures are fixed without resorting to the use of a scope. Only disk control and port failures are troublesome.

On another machine, to avoid checkout problems with long logic paths, we have made about 2000 internal processor signals available through multiplexers. A simulator incorporated in Midas checks consistency of these signals at each clock while single-stepping through random microinstructions; this has been a helpful checkout aid. The Stanford AI S-1 project is also using multiplexers to help with checkout.

The major problem in using these 2000 signals is one of observation—a human cannot notice very many of them at once, even if they appear on the display. To alleviate this problem, the signals are arranged in symbolically named 16-bit units, and each of the 16-bit units can be exploded into a complete symbolic printout when desired. Command files display the collection of signals in any hardware subsection.

When running the simulator, current readout, readout at the previous clock, and signals detected as being wrong are stored in separate tables inside Midas. The display can window any one of these tables. The simulator reports inconsistent signal names and the three tables can be looked at to discover what isn't working.

Conclusions

The most important conclusion I have drawn from my experience with the Maxc projects is that a hardware system should be simple. Diagnostic firmware/software must be thorough, and a "firmware/software overkill" position should be strived for with respect to hardware debugging and maintenance. Hardware simplicity and diagnostic support must not be considered too narrowly—system peripherals must also be included.

Microprogrammable hardware organization contributes to simplicity by moving complications out of the hardware and into the firmware, where they can be checked out easily and, once checked out, will not be a source of future trouble.

The use of a control computer has also been of great value in developing and maintaining the Maxc systems. It is an invaluable aid in debugging or troubleshooting the microprogrammed processor kernel that must work before diagnostic firmware can start pinpointing failures.

In addition, software tools should be developed by design organizations. Design automation software is probably most important. The Micro and Midas programs for firmware development and hardware/firmware checkout are two other valuable tools we have developed.

Finally, microprogrammable hardware organization allows additional instruction sets and other special-purpose microcode to be added at little extra cost. This potential should be exploited by providing a family of instruction sets, each tailored to the requirements of a commonly used high-level language. The instruction sets should use common word size, common virtual memory, and common registers so that they can reside harmoniously in a single operating system. ■

Acknowledgement

The group that designed, built, and debugged Maxc1/2 consisted of C. Thacker, B. Lampson, E. Fiala, E. McCreight, E. Taft, P. Heckel, L. Deutsch, H. Sturgis, C. Simonyi, R. Shoup, T. Strollo, L. Clark, and M. Overton.

References

1. D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "A Paged Time Sharing System for the PDP-10," *Proc. Third Symposium On Operating Systems Principles*, Stanford Univ., October 18-20, 1971.
2. Warren Teitelman, *Interlisp Reference Manual*, Xerox Palo Alto Research Center, December 1975.
3. The PDP-10 Software Documentation Group, *PDP-10 Reference Handbook*, Digital Equipment Corp., Maynard, Massachusetts, 1971.
4. C. G. Bell, A. Kotok, T. N. Hastings, and R. Hill, "The Evolution of the DECsystem 10," *CACM*, Vol. 21, No. 1, January 1978, pp. 44-63.
5. L. G. Roberts, "The ARPA Net," in *Computer-Communication Networks*, N. Abramson and F. Kuo (eds.), Prentice-Hall, Englewood Cliffs, New Jersey, 1973.
6. L. G. Roberts, "Network Rationale: A 5-Year Reevaluation," *Digest of Papers, COMPCON 1973*, p. 3-6.
7. J. L. Gilgoff, "Microprogramming the IBM System/360 model 60/62," IBM-TR-00.1139, May 1964.
8. C. R. Campbell, D. A. Neilson, et al., "Micro-programming the Spectra 70/35," *Datamation*, Vol. 12, No. 9, September 1966, pp. 64-67.
9. W. T. Wilner, "Design of the Burroughs B1700," *AFIPS Conf. Proc., Vol. 41, 1972 Fall Joint Computer Conference*, pp. 489-497.
10. P. Kornerup and B. D. Shriver, "An Overview of the Mathilda System," *SIGMICRO Newsletter*, Vol. 5, No. 4, January 1975, pp. 25-53.
11. Lewis Gallenson, Alvin Cooperband, and Joel Goldberg, "PRIM System: Overview," ISI/RR 77-58, March 1977.
12. Lewis Gallenson, Joel Goldberg, and Alvin Cooperband: "The PRIM System: An Alternative Architecture For Emulator Development and Use," *Proc. Tenth Annual Workshop on Microprogramming*, October 5-7, 1977, Niagara Falls, New York.
13. M. A. McCormack, T. T. Schansman, and K. K. Womack, "1401 Compatibility Feature on the IBM System/360 Model 30," *CACM*, Vol. 8, No. 12, December 1965, pp. 773-776.
14. Special Issue on Computer Architecture, *CACM*, Vol. 21, No. 1, January 1978.
15. B. L. Peuto and L. J. Shustek, "Current Issues in the Architecture of Microprocessors," *Computer*, Vol. 10, No. 2, February 1977, pp. 20-25.
16. L. A. Hollaar, "A Programmably Loadable Control Store for the Burroughs D-Machine," IEEE Computer Society Repository R74-309.
17. L. W. Smith, "Instruction Sequencing System," *IBM Technical Disclosure Bulletin*, April 11, 1969, pp. 1496-1497.

18. Gideon Frieder and Jill Miller, "An Analysis of Code Density for the Two Level Programmable Control of the Nanodata QM-1," *Proc. Tenth Annual Workshop on Microprogramming*, Niagara Falls, New York, October 5-7, 1977, pp. 26-30.

19. R. Belgard, "A Generalized Virtual Memory Package for B1700 Interpreter Writers," *SIGMICRO Newsletter*, Vol. 7, No. 4, December 1976, p. 31.

20. S. Habib, "Microprogrammed Enhancements to Higher Level Languages—An Overview," *Proc. Seventh Annual Workshop on Microprogramming*, Palo Alto, California, October 1974, pp. 80-84 (preprint).

21. F. R. Broca and R. E. Merwin, "Direct Microprogrammed Execution of the Intermediate Text From a High-level Language Compiler," *Proc. ACM 1973 Annual Conference*, pp. 57-62.

22. L. Peter Deutsch, "A Lisp Machine With Very Compact Programs," *Proc. Third International Joint Conf. on Artificial Intelligence*, Stanford University, August 20-23, 1973.

23. R. Greenblatt, "The LISP Machine," MIT Report, 1975.

24. R. Brody, "Tuning the Hardware via a High Level Language (ALGOL)," *AFIPS Conf. Proc., Vol. 42, 1973 National Computer Conference*, p. 518.

25. T. R. Bashkow, A. Sassion, et al., "System Design of a Fortran Machine," *IEEE Trans. on Electronic Computers*, EC Vol. 16, No. 4, September 1967, pp. 485-499.

26. R. C. Calhoun, "Diagnostics at the Microprogramming Level," *Modern Data*, Vol. 2, No. 5, May 1969, pp. 58-60.

27. J. M. Hemphill and S. A. Szygenda, "Deriving Design Guidelines for Diagnosable Computer Systems," *Proc. First Annual Symposium on Computer Architecture*, Gainesville, Florida, December 1973, pp. 131-135.

28. G. W. Karcher and J. Y. Hsu, "A Cross Assembler Implemented on IBM 360 for Variably Defined Microcode," *SIGMICRO Newsletter*, Vol. 5, No. 3, October 1974, p. 89.

29. D. J. Dewitt, M. S. Schlansker, and D. E. Atkins, "A Microprogramming Language for the B-1726," *Preprints, Sixth Annual Workshop on Microprogramming*, College Park, Maryland, September 1973, pp. 21-29.

30. R. H. Evans, L. H. Moffett, and R. E. Merwin, "Design of Assembly Level Language for Horizontal Encoded Microprogrammed Control Unit," *Preprints, Seventh Annual Workshop on Microprogramming*, Palo Alto, California, October 1974, pp. 217-224.

31. G. L. M. Noguez, "Design of a Microprogramming Language," *Preprints, Sixth Annual Workshop on Microprogramming*, College Park, Maryland, September 1973, pp. 145-155.

32. P. W. Mallett and T. G. Lewis, "Approaches to Design of High Level Languages for Microprogramming," *Preprints, Seventh Annual Workshop on Microprogramming*, Palo Alto, California, October 1974, pp. 66-73.

33. J. D. Atkins, "Paradigmatic Universal Microcode Assembler," *Workshop on Microprogramming*, Grenoble, France, June 1970.

34. W. T. Wilner, "Microprogramming Environment on The Burroughs B1700," *Digest of Papers, COMPCON 1972*, San Francisco, September 1972.

35. M. Gasser, "An Interactive Debugger for Software and Firmware," *Preprints, Sixth Annual Workshop on Microprogramming*, College Park, Maryland, September 1973, p. 113-119.

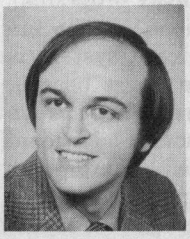
36. W. G. Bouricius, "Procedure for Testing Microprograms," *Preprints, Seventh Annual Workshop on Microprogramming*, Palo Alto, California, October 1974, pp. 235-240.

37. B. C. Hodges and A. J. Edwards, "Support Software for Microprogram Development," *SIGMICRO Newsletter*, Vol. 5, No. 4, January 1975, pp. 17-24.

38. C. Vickery, "Software Aids for Microprogram Development," *Preprints, Seventh Annual Workshop on Microprogramming*, Palo Alto, California, October 1974, pp. 208-211.

39. T. M. Whiteney, "A Test Procedure for Microprogrammed Systems," *Preprints, Third Annual Workshop on Microprogramming*, Buffalo, New York, October1970.

40. R. Gossler, "Development Systems for Microcomputer Programming," *Elektronik* (Germany), Vol. 26, No. 5, May 1977, pp. 36-43.



Edward R. Fiala has been a member of the research staff at Xerox Palo Alto Research Center since 1971, where his main interests have been in operating systems and the development of new computers. From 1968 to 1970 he was employed as a system programmer by Bolt, Beranek and Newman and by Berkeley Computer Corp. A member of ACM, he received his SM in electrical engineering from MIT in 1968.