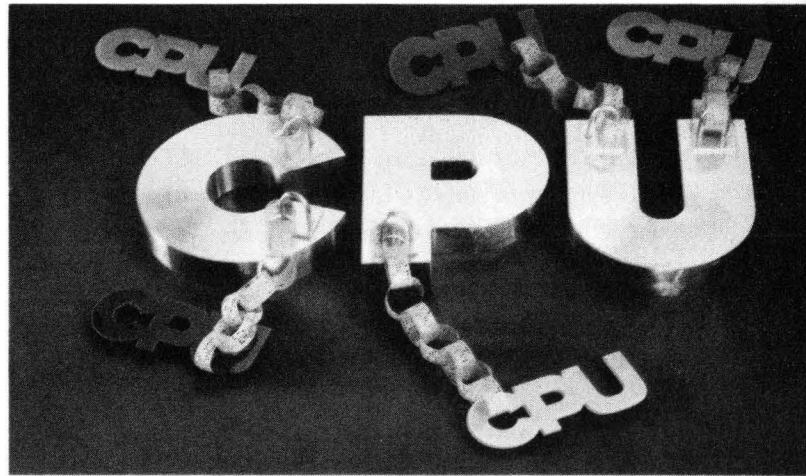


Adaptability to various microprocessors comes from separating prototype- and system-related tasks; in-circuit emulation and new high-level language are bonuses



# 'Universal' development system is aim of master-slave processors

by Robert D. Catterton and Gerald S. Casilli, *Millennium Information Systems Inc., Santa Clara, Calif.*

□ In the ever-changing world of the microprocessor, one element is fixed: heavy investments in personnel training, software, and development aids can lock designers into a particular processor for their systems. Each recently introduced hardware and software development system, for example, is based on a particular family of devices and isn't easily adaptable to other families. What is needed to free the designer from design compromises that reduce performance or cost effectiveness is a "universal" development system that can accommodate many different microprocessors.

A new system, called the Universal-One, achieves universality by a division into two functional areas. Those tasks that are related to the development system are assigned to a master central processing unit, and those that are prototype-related are assigned to a second,

or slave, CPU. As many as four different slaves may be installed simultaneously and individually used through operator commands. This multiple architecture enables the hardware to support new microprocessors with the addition of a pc card containing the new slave CPU.

Since the master processor need not be changed to accommodate new slave units, all of the operating system software remains the same. Presently, the system supports the 8080A and the 2650 central processors as slaves, with in-circuit emulation capability. It's easy to add other 8-bit processors to the system, and 16-bit devices may be added with only relatively little reconfiguration.

Although universality is the basic objective, there are four other major requirements that today's development systems should satisfy. Use of a disk-based storage

system will achieve high throughput for maximum software-development productivity. A disk-based operating system should be specifically tailored for microprocessor development. The user's interface with the system should be simple and remain unchanged regardless of the processor under development. The test and debug capabilities should support development of hardware and software and their integration into an operating prototype system.

## Functions

The master CPU is responsible for all of those system services that are not prototype-dependent, such as:

- File management—the storage and retrieval of data and programs.
- Text editor—maintains text files contained on the disk.
- System input/output—the normal I/O activities between the standard system peripherals, such as flexible disk, printer, and terminal.
- System utilities, including programming of read-only memories for the final version of the prototype.
- Debug functions—the master executes the debug software and controls the slave through a separate debugging hardware module.

The slave CPU's functions include:

- Program assembly—each slave may be used as a resident assembler of prototype programs.
- Prototype-program execution—the prototype program is loaded into the slave memory and executed by the slave.
- Prototype I/O—any special input/output required in the prototype is performed by the slave.

- In-circuit emulation—a cable extends from the slave to the CPU socket in the prototype.

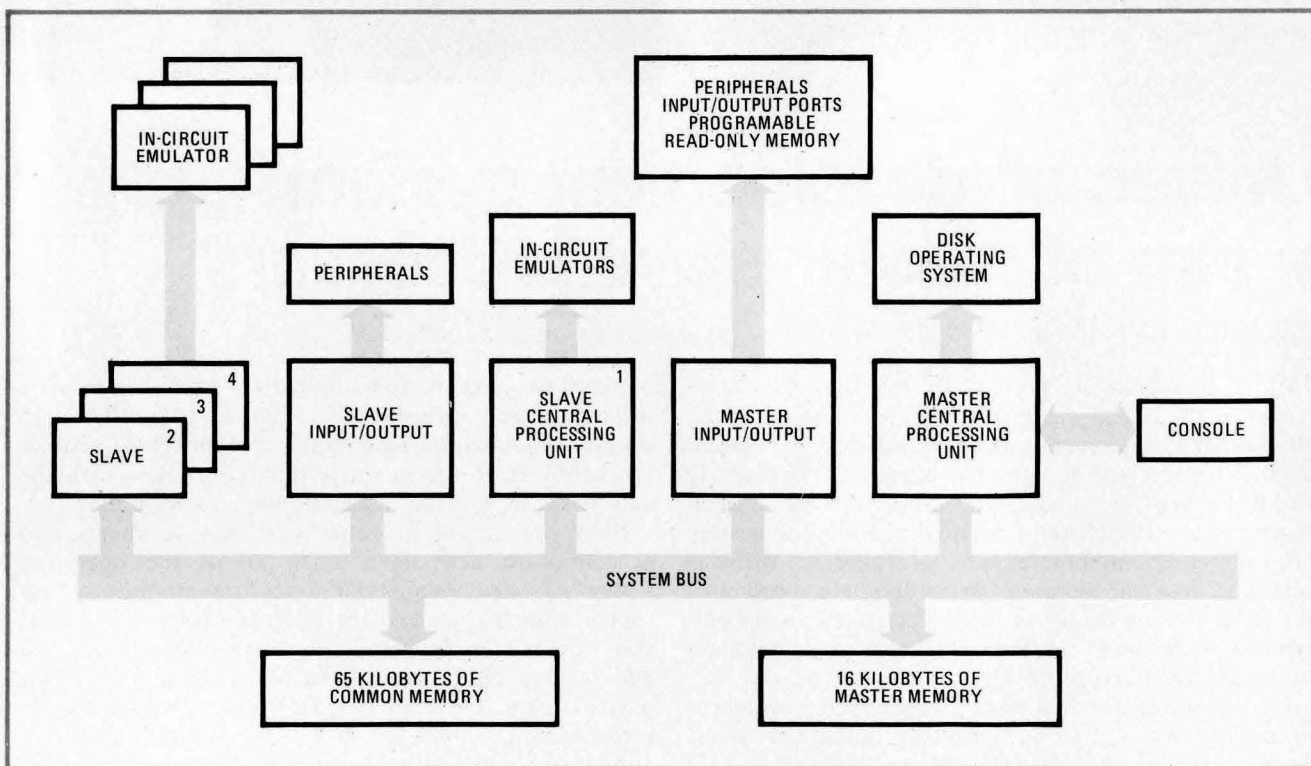
The system architecture (Fig. 1) includes a bus structure to tie the components together and to permit the exchange of data and control signals. The basic bus design was governed primarily by the dual-memory and the multiple-CPU architectures. Other design considerations for the bus were that the memory portion had to be able to handle 8- and 16-bit data words, and that the overall structure had to accommodate future higher-speed microprocessors.

The system services the peripheral I/O devices and debug logic with interrupts rather than with polling. With an interrupt-driven system, the peripherals can get service when they need it, without waiting for their turn in the polling sequence. It also allows an efficient software structure that is relieved of the overhead inherent to polling. In this way, maximum throughput is achieved.

## Memory structure

The random-access memory of the system is organized as 65,536 bytes of common memory and a 16,384-byte master memory. The logic on the master CPU module allows appending any one of four 16-kilobyte segments of common memory (Fig. 2) to the master memory space. This allows master-slave communication for transfer of data during I/O service requests and gives the master access to program-trace information developed by the debug logic discussed later.

Master-memory protection is accomplished by a special bus-control signal, which is sensed on the memory cards. Only the master CPU contains the



**1. Two CPUs.** The Universal-One system uses two central processing units—master and slave. In-circuit emulation is performed through the slave CPU, which duplicates the type of microprocessor used in the prototype. The master CPU handles system-related functions.

## A new compiler

To go along with the development system, Millennium has developed  $\mu$ Basic, a high-level language compiler designed for microprocessor applications. Although it was tailored to meet the needs of engineers, it also provides a useful tool for the professional programmer.

The new compiler offers the advantages of a high-level language—greater programming productivity, easier program maintenance, and portability from one microprocessor to another. In the Millennium development system, it also provides a “universal” programming capability, since the same  $\mu$ Basic statements can produce object programs for the different microprocessors.

As shown in the figure,  $\mu$ Basic statements are first brought into the “statement-analyzer” software package, where they are converted for input to the code emitter. Then, depending on the microprocessor and resident assembler being used, the code emitter generates the assembly-language statements, which are subsequently passed through the assembler to produce object code for the selected microprocessor. This two-step compilation process gives the programmer more flexibility when working out the program for the prototype.

A major criticism of high-level languages in microprocessor applications is that more memory is used than with assembly languages, and execution is slower. However,  $\mu$ Basic allows the programmer to intermix assembly language. In situations where a programmer thinks it necessary, this intermixed assembly language may use the same labels and variables as does the  $\mu$ Basic program.

A debug-optimize report produced by the compiler helps avoid software error conditions that the two-step compilation process might cause. The report shows the  $\mu$ Basic statement followed by the assembly-language listing that was generated to perform the original statement.

Typically, a programmer would first code and debug the program without regard to memory or performance constraints. Then, when the program is functioning correctly, the debug-optimization report can be used to show those areas that may require assembly coding to optimize memory usage. Since memory comes in fixed increments, the most important optimization is usually done when the program size exceeds that specified increment. If the program generated by  $\mu$ Basic does not exceed the memory increment available, then assembly-language optimization may not be needed.

Performance optimization also can be in assembly language. Usually, some small portion of the code is used most of the time—for example, 10 to 15% of the code might be used 80 to 90% of the time. Consequently, a concentration on those heavily used portions will produce the greatest increase in performance.

In its data and statement types,  $\mu$ Basic is generally equivalent to PL/M. The length of the data element may

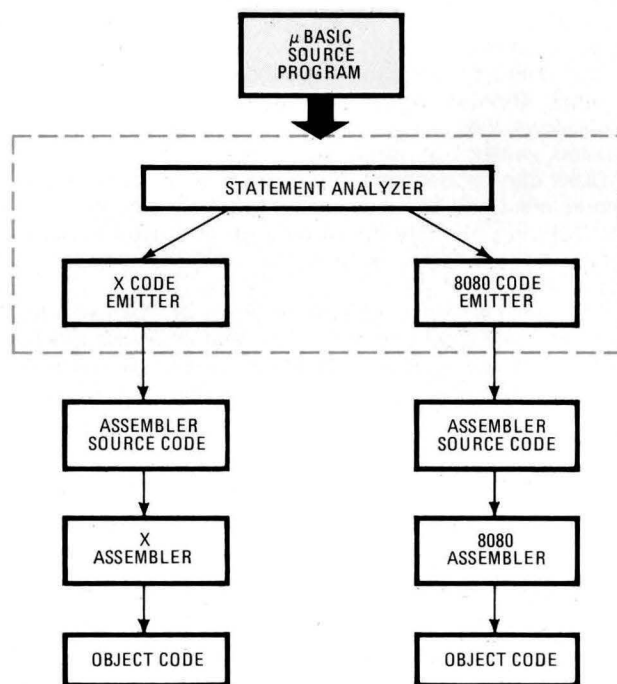
be either 8 or 16 bits, and both 8 and 16-bit elements are supported at the same time.

Examples of statement types are:

- LET—the assignment statement.
- FOR . . . NEXT—used for loop construction.
- IF—the test statement.
- GOTO, GOSUB, RETURN—control transfer statement.
- ON—for a computed GOTO or GOSUB.

The  $\mu$ Basic compiler features an ability to specify memory locations for arrays. This is quite important in connecting a peripheral device to the system. Many peripheral devices operate out of a dedicated-space memory. To conveniently interface a program written in a higher-level language to that device, the programmer must be able to position the array in the same location in memory that the device is using. This is also very important in microprocessor systems where there is a RAM/ROM trade-off. The programmer can control the origin of the portions of the program to be put in ROM and RAM.

In comparing  $\mu$ Basic with PL/M (the most widely used high-level language), it can be seen that the latter is a “richer” language. A professional programmer is comfortable using PL/M and can take advantage of its greater complexity. However, the logic designer or other nonprofessional programmer probably will have to expend some effort to learn enough about PL/M to be able to write programs using it. In contrast,  $\mu$ Basic is easy to learn and use, while being quite effective.



circuitry to activate this control line. Thus, the slave processor cannot gain access to the master memory and destroy its contents or (through damage to the file manager or part of its data structure) the files themselves, out on the disk.

The slave can address the common memory as a 65-kilobyte or as a 32,768-word, 16-bit memory. This allows

the 8-bit master to address a 16-bit slave memory as sequential bytes.

There are also commands that permit the operator to display and alter common memory. He may inspect and change the contents of the memory, and he may display and alter the contents of the registers. He may interact with his program and change variables—change register



## Using the software

The Millennium development system has many software features related to its use of a floppy disk for mass storage and the UDOS operating system for the disks. The system can have up to four floppy-disk drives all in use at the same time. A file name in use on one disk can be the same as one on another. The user can specify the file he wants by appending the floppy-disk drive number to the file name; i.e., TESTPROG/1 or TESTPROG/2.

Through use of the VERIFY command, a user can check the floppy disks to determine if any of the tracks are bad. The bad tracks are recorded in the disk's directory and thereafter are not allocated to a file.

The user need not create a file or otherwise establish it before writing data on it. When he issues a UDOS command with a file name as an output device, the file will automatically be created, and the name will be placed in the directory for the floppy disk.

The user need not allocate space for a file before using it, for disk space is dynamically allocated by UDOS as it is needed. When the file is closed, the space allocated is recorded in the directory. When the file is deleted, the space allocated is freed up and made available for allocation to other files.

A file name may contain as many as eight alphanumeric characters and special characters. This allows the user to use names that are more indicative of the file content; i.e., PROGLIST rather than PRGLST, or, worse yet, PGLS. A disk file may contain anywhere from 1 to 311,296 data bytes. The user need not concern himself with extraneous data or otherwise keep track of the number of "real" data bytes in his file.

The entire contents of a disk can be duplicated in another. This feature allows back-up of important disks and allows the user to recover if a file is inadvertently deleted, written over, or otherwise destroyed.

Disks can be identified with a string of up to 44 ASCII characters. Users can thus briefly describe the contents of the disk and the date it was created, and need not rely totally on the label, which could become marred or destroyed.

The user can string together a group of files into one with a single UDOS command. This feature allows development of the source program in small, manageable pieces. Subsequently, all of the pieces can be combined

and placed on a single file, which can be assembled. If an error shows up in the assembly, only that piece of the source program which contains the error need be edited. All of the pieces can then be combined again and the assembly repeated.

All I/O operations can be assigned to channels by software. The user can assign any device attached to the system to any one of up to eight I/O channels and need not concern himself with the characteristics of the device. This feature allows the user to prepare programs whose input and output sources can be determined at run time. Channels can be assigned for a program externally through the console or internally by the program itself.

A sequence of UDOS commands can be executed one at a time from a command file. The user can thus invoke any number of commands simply by issuing the name of the command file. The individual command can be filled with parameters that are given at the time the command file is invoked. Thus frequently used command sequences can be invoked simply. Command files can also be chained—the last UDOS command in a file can be the name of another file, allowing a series of jobs to be run in a batch mode, perhaps overnight, unattended.

The text editor is line-oriented and has a command repertoire similar to those available on large time-sharing systems. The user can create a file of assembly-language statements or a data file by entering lines of text through the system console. Subsequently, he can insert lines anywhere in the file, delete lines, replace them, or modify part of the text on a line.

During a text-editing session, the user can get lines of text from any file and merge them into the file being edited or put lines of text from the file being edited to any other file. This feature provides the capability of manipulating lines of text from several files and merging them into one file quickly and easily. With the text editor, the user can combine several text-editing commands into one complex command and then cause it to be executed several times.

The user can set tabs dynamically and designate any console key as the tab character at any time during a text editing session. He can also issue UDOS commands and cause other system functions to be initiated during a text-editing session.

contents or change the data elements being used in the debug process.

### The disk operating system

A universal disk operating system called UDOS was developed for the multiple-CPU architecture. This software is executed by the master in its own totally protected master memory. The UDOS feature is floppy-disk-oriented, taking into account the characteristics and peculiarities of such disks. Many file-management functions usually performed by the user are performed automatically. The user need only direct that certain data be stored on a file or taken from a file.

The operating system allows the user to develop microcomputer programs with a high-level language (see "A new compiler"), a symbolic assembler, or both. The user can prepare a program with a text editor, correct

and modify it quickly and easily, assemble it, load the resulting object code into common memory (or into the prototype memory), and cause it to be executed under debug control.

During execution, the program steps can be traced, breakpoints can be set, and memory can be inspected and altered as required. Subsequently, the program can be corrected or modified at the source level, using the text editor, then reassembled, loaded, and executed again for the next round of debugging. (see "Using the software").

### In-circuit emulation

Each slave contains circuitry to support in-circuit emulation. When the prototype becomes ready for test, all of the development-system resources become available to it once the emulator cable is plugged into the

microprocessor socket of the prototype. The operator can then use the system's debugging software to debug the prototype hardware and software and then to integrate them.

The system supports two operating modes for emulation. In one, the user can substitute the memory of the development system for that of the prototype. In the other mode, when the prototype's memory becomes available and its I/O functions have been thoroughly tested, the operator can execute programs from the prototype memory while maintaining full control through the development system.

When operating with the prototype memory, most of the system debugging features are still available. The user can use the address breakpoint and do a full trace. If this mode requires the programmable ROM of the final prototype, the master can directly program the assembled instruction into the PROM chips. If the object resides on paper tape, it can be loaded into the system and transferred to the PROMs.

The user can switch emulation modes at any time by a console command, with no hardware changes. The cable may be left attached to the slave even when the emulation feature is not in use.

The development system's memory is comparable to the memory speed of most prototype systems, and thus it nearly simulates real-time operation when programs are executed from the system. When programs are executed from the prototype memory, the slave can operate at the the prototype's clock and memory speeds. Timing differences resulting from the use of the umbilical cord are minimal.

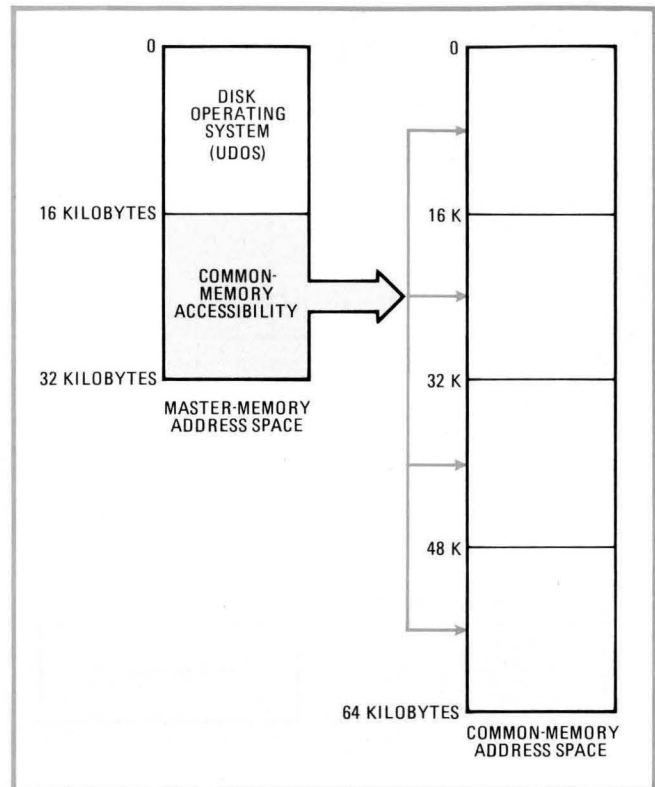
### Master-slave interaction

When input/output from a master-controlled peripheral is required by a slave program, the slave CPU executes a service-request instruction, which causes the slave to pause temporarily while the master obtains the necessary data for the slave program. When the I/O requirements are completed, the master releases the slave so that it may continue the process of program execution.

The debug logic is on a separate module and includes breakpoint registers, address-computation circuitry, two program-counter registers, and single-step and interrupt logic. The functions controlled by this logic are independent of the slave microprocessor and thus support the universal aspects of the system design for application to a variety of target processors.

Part of the master-slave interaction includes control of breakpoint and trace operations. The master loads the breakpoint addresses under command from the user. When the memory address and operation from the slave match the breakpoint value, the program running under the slave pauses, and control is passed to the master. The debug module stores the slave's instruction-fetch address to enable the software to examine the prototype program and to interpret operating codes for the trace printout. Synchronization signals are provided to aid the user in triggering events necessary to debugging of prototype hardware.

The two memory-address breakpoint registers may be



**2. Memory addressing.** The master CPU can address 32 kilobytes of memory. Of this total, 16 kilobytes are used by the disk-operating system, UDOS, while the other half can consist of any of four 16-kilobyte blocks in the common-memory addressing space.

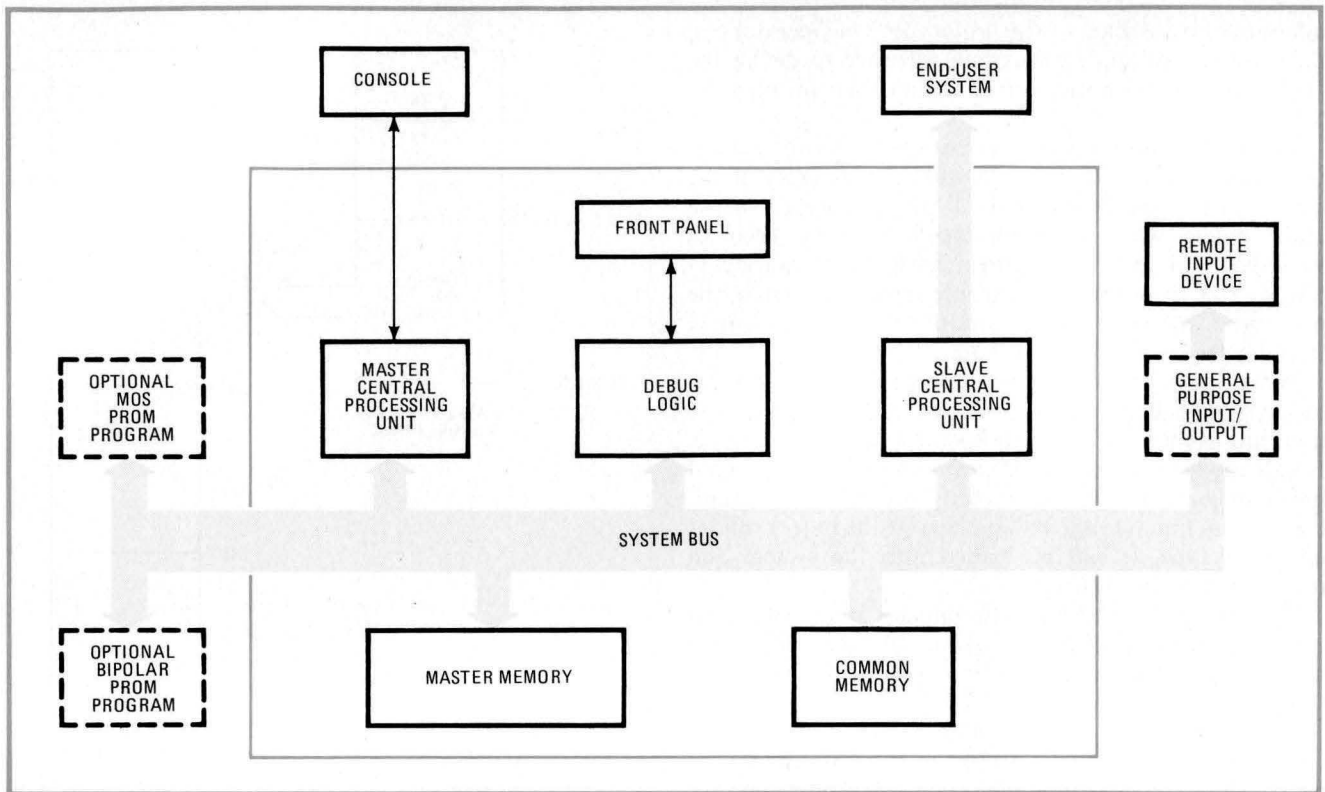
set to break on any of a variety of memory-access conditions. Another capability is a dynamic trace of the user program. On an instruction-by-instruction basis, the user can trace the activity of the program being executed, with a display of the location of the instruction, its mnemonic, the register contents, and the state of the machine (such as the condition of the carry flip-flop).

Dynamic trace may be performed on every instruction, on instructions between two memory limits, or on only the jump instructions. The jump-instruction trace reduces print-out time and runs through the program faster. If the user isolates a problem area, he may go back to the full-trace mode and examine every one of the instructions.

### I/O and interrupts

The functions associated with the master and slave CPUs dictate the need for separate master/slave input/output and interrupt structures. The master has a 256-port I/O address space and a 32-level interrupt structure. Sixteen interrupts are devoted to debug functions and service requests. The other 16 are related to the system I/O.

The master card contains the I/O ports to support such standard peripheral devices as the dual-drive floppy disk, a line printer, and a cathode-ray tube or teletypewriter console. With the addition of a standard general-purpose I/O card, the system-related functions are easily expanded to support other peripherals, such as high-



**3. Smaller system.** For applications in which users have already invested in software development aids, the Universal-One can be pared down to provide only emulation and PROM programming. Memory is much smaller, while the blocks shown in dashed lines are optional.

speed paper-tape or card readers.

The slave has a 256-port I/O address space and an eight-level priority-interrupt structure. It cannot directly address the system I/O. However, through the use of service requests to the master, it has full access to the system peripherals.

The user also has the option of using a general-purpose I/O card as interface between the slave and its special devices, such as the prototype's keyboard or printer. In such a case, the slave will perform its own I/O functions on those devices. The general-purpose card provides a full EIA-RS-232-compatible port and four 8-bit input/output ports.

### Expandable PROM programming

Capability for programming erasable metal-oxide-semiconductor and bipolar-fusible PROMs for the final version of the prototype is integral to the development system. Two card slots in the motherboard and three front-panel sockets are provided with the standard system. Personality cards are available for programming the 1702A MOS PROM and the 82S115 4- and 8-bit bipolar family. New programming cards are easily substituted for other families of PROMs.

As well as eliminating the need for a separate PROM programmer, this feature is more cost-effective, since dual I/O circuitry is unnecessary and operation is controlled by the master CPU rather than by a separate processor. The programming cards are interrupt-driven, freeing the master for other tasks during the programming of each byte.

Even though a PROM verifies correctly, it may lose

charge or "grow back" a fusible link if not programmed properly. Therefore, the cards have many protection and error-checking features such as over-voltage protection, current limiting to prevent overstressing, and power-failure protection against partial programming of the devices.

### The universal emulator

Many companies already have some method of accomplishing the pure software-development function of assembling and editing programs, but they lack means of performing emulation or PROM programming for use in the prototype system. Other companies have a complete microprocessor development system, but they are involved in multi-project situations with one particular project fully occupying their development system. In either situation, companies may find a second version of the Millennium development system useful. With an expanded front panel and a paring-down of the system memory to 12 kilobytes, it becomes a universal emulator and PROM programmer (Fig. 3).

All of the software debug functions for both emulation modes previously discussed will be retained. The basic functions, such as patch, dump, examine, breakpoint, and others will be resident in the PROM. Only the trace program, which will change for each target slave, will be loaded into master memory from the console device. User programs may be entered into common memory either from the console device or remotely from a host computer via an EIA-RS-232 serial interface. Also, PROMs may be used to hold user programs that will be executed in the prototype. □