OCTOBER 1980

HEWLETT·PACKARD JOURNAL

HEWLETT·PACKARD JOURNAL
Technical Information from the Laboratories of Hewlett-Packard Company

OCTOBER 1980 Volume 31 • Number 10

Contents:

In this Issue:

We are now in the age of LSI—large-scale integration—and are about to enter the age of VLSI—very large-scale integration. LSI has given us the microcomputer, a complete computer on a tiny chip of silicon smaller than a fingertip, and many other complex integrated circuits with tens of thousands of transistors and logic gates on a chip. In the age of VLSI we'll see circuits with hundreds of thousands or millions of logic elements on a single chip. We'll see them, that is, once we're able to solve the formidable problems of designing such complex devices and writing software for them. Beginning on page 30, Chuck House discusses the problems and the likely solutions. Instead of a single talented designer, we'll have teams of designers working on a chip. These designers will need new tools that automate many of the steps we now do manually. They'll have to be able to call up various computer-aided design tools and different kinds of analyzers at the touch of a button. The system that will give them these advanced analysis and synthesis tools is something Chuck calls the electronic bench. It doesn't exist yet; in fact, we'll need VLSI to make it a reality. Only with VLSI will we be able to make analyzers and other instruments small enough and inexpensive enough to make it practical to build an electronic bench crammed full of them.

That brings us to the subject of this issue, Model 64000 Logic Development System. The 64000 is a tool for developing hardware and software for products based on commercial microcomputers. While it's a long way from the electronic bench, it's a first step towards that goal. It allows up to six designers to share a common data base, and it gives each designer a work station with a dedicated computer and a dedicated logic analyzer built in. Its architecture and capabilities are discussed in the articles on pages 3, 13, 20, and 28.

Our cover photo shows a basic 64000 System consisting of work station, disc drive and printer, along with a close-up of one of the pods that interface the 64000 to the system under development.

-R. P. Dolan

Editor, Richard P. Dolan • Associate Editor, Kenneth A. Shaw • Art Director, Photographer, Arvid A. Danielson
Illustrator, Nancy S. Vanderbloom • Administrative Services, Typography, Anne S. LoPresti • European Production Manager, Dick Leeksma

Logic Development System Accelerates Microcomputer System Design

This expandable, flexible system offers a complete set of facilities for generating and debugging microprocessor-system hardware and software. It's designed with next-generation VLSI circuits in mind.

by Thomas A. Saponas and Brian W. Kerr

MICROPROCESSORS HAVE PROVIDED significant improvements in the performance and flexibility of much of today's electrical and mechanical hardware. One consequence is that our approach to designing products has had to change, and so have the skills of the engineers responsible for these products. The design team of a microprocessor-based product might be more than half software designers. It is not unusual for a product's definition to change in the very late design stages in spite of excellent research and definition at the beginning. Then the flexibility of the software is the vehicle for accommodating such changes.

Because the microprocessor is only one piece of a complete system, it represents a software design problem unlike most computer systems. The processor is usually an integral part of some hardware that has nothing to do with computation. In some cases it is simply being used as a programmable logic element or for control of the human interface with some process. These differences make the conventional tools for generating and debugging hardware and software incomplete for the task facing the microprocessor system designer. The 64000 Logic Development System was meant to provide a complete solution to this task in one package, and to make significant contributions to the efficiency of designers' time.

Architecture

A basic 64000 Logic Development System consists of one Model 64100A Development Station with a Model 64940A Magnetic Tape Cartridge Unit installed, compatible HP hard disc and printer, and software packages to edit, assemble, link, and store program modules. Adding an emulator option and up to 64K bytes of independent emulation memory adds the download function through emulation, which is the standard tool for exercising, debugging, and integrating hardware and software in the early development phases. Further assistance in troubleshooting the target system is gained by adding Model 64300A Logic Analyzer, which monitors activity on the address, data, and control buses of the target microprocessor system. As program modules are completed, they may be mapped into the target system's random-access memory, or with Model 64500A PROM Programming System, they can be downloaded into one of many widely used programmable read-only memories (PROMs). The system may be expanded to accommodate larger design teams or multiple design efforts by adding up to five more development stations (see Fig. 1).

Development Station

The development station keyboard and display (see Fig. 2) provide the interface between the operator and the logic development system. Operating systems, input/output, keyboard, display, and the development station bus are managed by the independent host processor and memory.

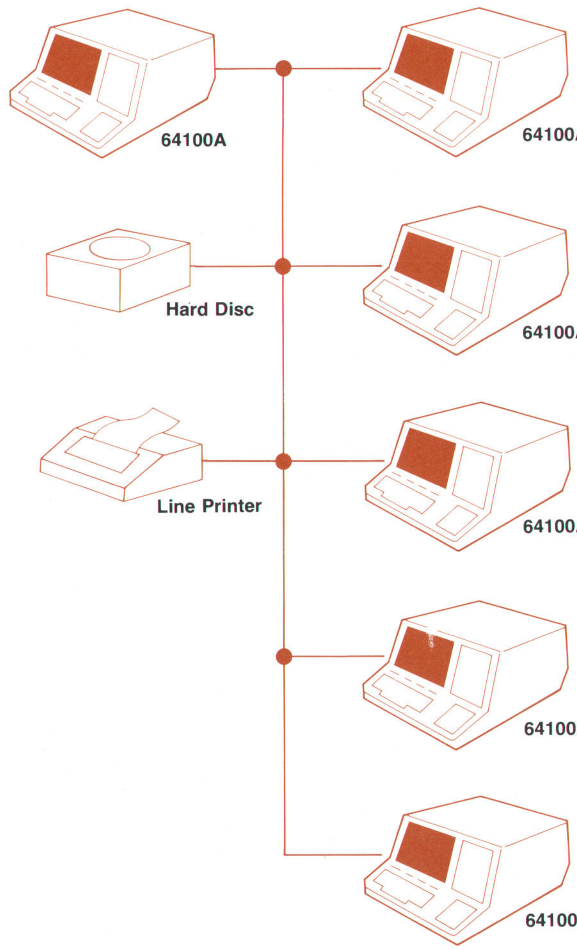


Fig. 1. *The 64000 Logic Development System consists of at least one 64100A Development Station, a hard disc, and a line printer. The system can be expanded to as many as six stations. Each station has its own processor.*

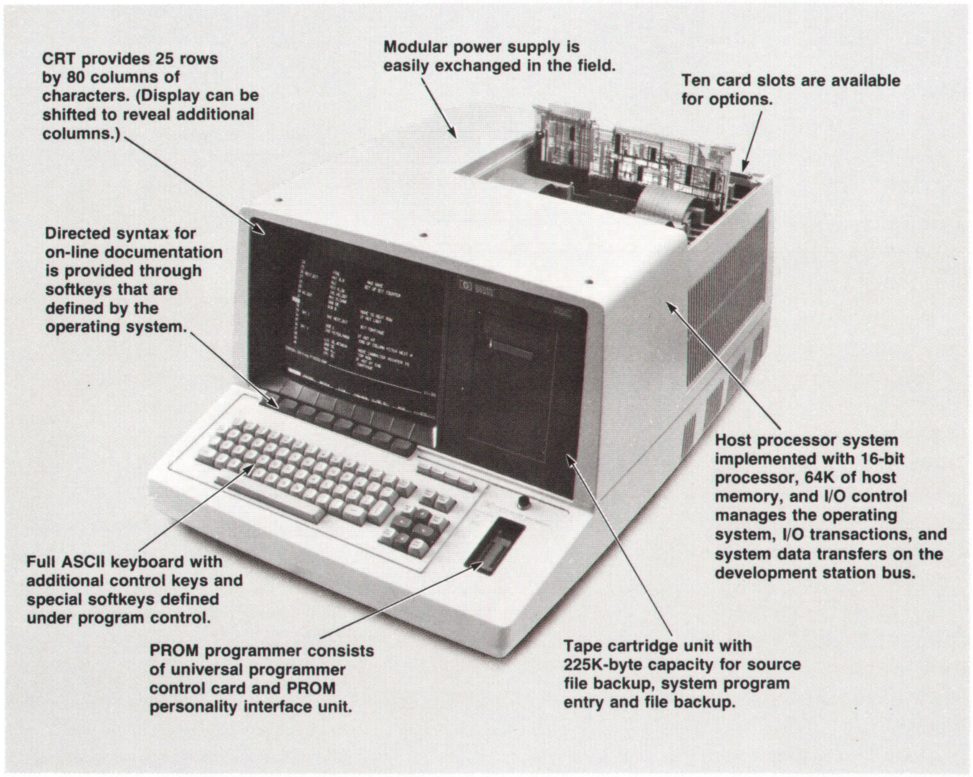

CRT provides 25 rows by 80 columns of characters. (Display can be shifted to reveal additional columns.)
Modular power supply is easily exchanged in the field.
Ten card slots are available for options.
Directed syntax for on-line documentation is provided through softkeys that are defined by the operating system.
Host processor system implemented with 16-bit processor, 64K of host memory, and I/O control manages the operating system, I/O transactions, and system data transfers on the development station bus.
Full ASCII keyboard with additional control keys and special softkeys defined under program control.
PROM programmer consists of universal programmer control card and PROM personality interface unit.
Tape cartridge unit with 225K-byte capacity for source file backup, system program entry and file backup.

Fig. 2. *Model 64100A Development Station includes keyboard, display, and host processor. Options include PROM programmers and emulators for various microprocessors, a logic analyzer, and a tape controller and drive.*

The host processor in each 64100A Development Station is a field-proven 16-bit processor manufactured by HP.[1] Much of the other hardware is adapted from other HP products. However, the emulator option and the PROM programmer are new and are discussed in detail elsewhere in this issue.

The development station's easily accessed card cage has slots to house the circuitry for the various system options. The first three slots of the card cage are occupied by the three cards of the host system, leaving the remaining ten slots available for system options. The development station bus is universal, and options may be placed in any slot. The development station bus carries address, data, and control signals between the host processor system and option card positions.

Each option card can identify itself to the host processor. Thus the option software is self-configuring. Communication with the options is via a 32K-byte memory address space window. When a card is addressed by the host one of three bank switch modes is also set, thereby expanding this window to an effective 96K bytes per option card.

Fig. 3 is a map of the entire 128K-byte address space of the host processor including the 32K-byte window. The display memory is an integral part of the program RAM, making possible the rapid display update required for such things as tracking softkeys and a screen-mode editor. The ROM space in the system is used for the bootstrap programs, for some frequently used utilities, and for the mainframe self-test software. In the current version of the 64000A system, 16K bytes of ROM is unused and reserved for future enhancements. All of the operating software resides in the RAM area and is segmented so that only the current task is in memory.

The emulation system uses a separate emulation bus between emulation control, emulation memory, and analysis cards. A second high-speed bus connects emulation control and emulation memory, and a third bus may be required for input/output in some modules and configurations (see Fig. 4).

Architecture Advantages

The architecture of the 64000 Logic Development System offers several advantages. Each user has a dedicated processor and memory, not just a terminal. Therefore, as stations are added, so is computing power. By contrast, with timesharing systems the user is required to buy sufficient computing power with the very first terminal to support the ultimate size of the system. Philosophically, it is also more

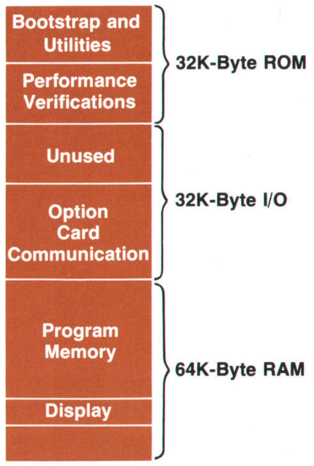
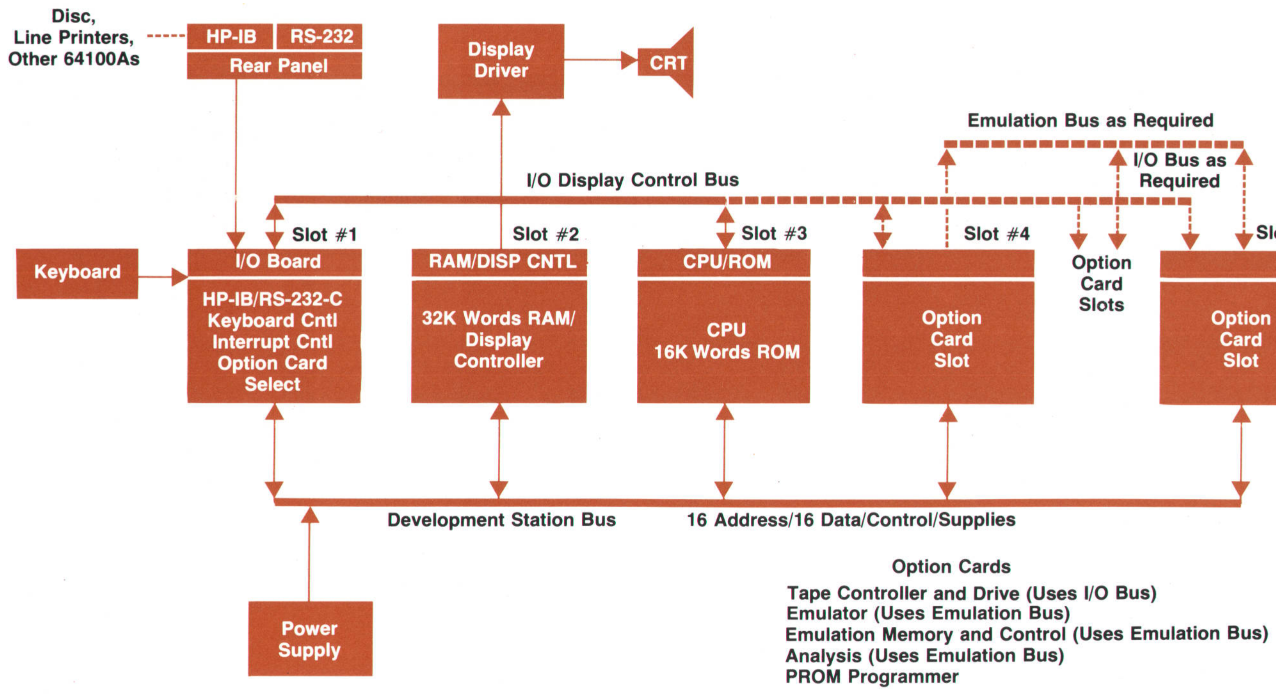


Fig. 3. *Host processor memory map.*



Disc, Line Printers, Other 64100As
HP-IB
RS-232
Rear Panel
Display Driver
CRT
Emulation Bus as Required
I/O Bus as Required
I/O Display Control Bus
Slot #1
Slot #2
Slot #3
Slot #4
Slot #13
Keyboard
I/O Board
HP-IB/RS-232-C Keyboard Cntl Interrupt Cntl Option Card Select
RAM/DISP CNTL
32K Words RAM/ Display Controller
CPU/ROM
CPU 16K Words ROM
Option Card Slot
Option Card Slots
Option Card Slot
Development Station Bus
16 Address/16 Data/Control/Supplies
Power Supply
Option Cards
Tape Controller and Drive (Uses I/O Bus)
Emulator (Uses Emulation Bus)
Emulation Memory and Control (Uses Emulation Bus)
Analysis (Uses Emulation Bus)
PROM Programmer

Fig. 4. *The host processor and the microprocessor being emulated have independent buses and can run simultaneously. Thus software development can be concurrent with emulation.*

reasonable to present to the user a response time that is more a function of the task, which is the case with distributed processing, than to have the response time determined by the total system loading, as in a timesharing system. The 64000 network can also be expanded to include large central data bases or additional 64000 clusters using the RS-232-C port contained in each station.

By sharing peripherals, it is much easier to justify higher-performance units than when each user has a dedicated set. Users get not only higher performance but also the ability to develop software jointly sharing the same data base. Experience has shown that as the software tools improve and the efficiency of programmers increases, the need for disc space rapidly outpaces the original estimates of capacity. Also, with the text editing features of the system providing a convenient way to maintain documentation, a further burden is placed on disc capacity. At HP's Colorado Springs Division, for example, we are now using two to five megabytes of disc space per user per year, compared to approximately one megabyte before these tools were available. The 64000 System expands easily to accommodate such changes.

Operation

At power-up the host processor interrogates a rear-panel switch to determine the ROM program to execute. There are four selectable modes: system bus, local mass storage, ROM, or performance verification. The performance verification mode exercises all of the mainframe hardware, including the memory, tape drive, RS-232-C port, and system bus. The other three modes are bootstrap programs from three sources. The normal mode of operation is to boot from the hard disc, which is on the system bus. The program that is loaded then performs a poll to determine all of the devices on the bus, configures the software I/O drivers based on that poll, and displays a system map. Eight softkey labels are displayed at the bottom of the display indicating the various functions available. The system is now awaiting a command and a status message indicates that state. To perform an assembly of a source file, for example, the softkey labeled assemble is pushed, followed by the name of the file to be assembled. The editor, compiler, and linker all use this same syntax.

Emulation

A challenging aspect of microprocessor system design is the lack of a friendly run time environment for debugging software and hardware. If, for example, the user is developing a microprocessor-controlled meat scale, the product will not have peripherals such as CRT, keyboard, disc, and printer to help the debugging process. Because of the direct interaction of hardware and software, the techniques used in computer development—halting, single-stepping, dumping registers, and software tracing—might so perturb the system that the measurement obtained is meaningless. Because the completed system is usually read-only-memory-based, a convenient software prototyping environment is also essential so that software can be tested and developed before being committed to ROM.

The 64000 emulator option is designed to imitate the microprocessor in the user's system and provide all the necessary debugging facilities. The emulator is used by removing the microprocessor to be emulated from the user's hardware and plugging in the probe from the 64000 System in its place. The user then specifies the memory area to be taken from the user system and that to be provided by the emulator. The answers to these configuration questions are automatically stored in a file so that when the emulator is used later with the same configuration only the file name needs to be specified. The emulator can be used before any user hardware exists by simply specifying the internal clock and all emulation memory. Because the emulator has access to the display, disc, printer, and keyboard, much software development can take place before the user hardware is ready.

In the 64000 System, we have completely separated the emulation processor bus from the host environment (see Fig. 4). This allows passive monitoring of the execution of software without stopping the process. Because of this separation it is also possible to continue emulation while software development is occurring on the same station, thus potentially doubling the use. The two buses are so independent that the prototype containing the emulator probe can be powered down and then up without affecting the host system. Even the data stored in the emulation memory remains unchanged and the processor simply goes through its normal power-up sequence.

Another important benefit of this architecture is the future expandability of emulation. The host processing system puts no restrictions on the speed or word length of the processor being emulated. Future microprocessors will certainly be faster and more powerful, so it is important to allow for this to preserve the capital investment in the development system.

The emulator option for the 64000 Logic Development System is described in the article beginning on page 13.

Directed-Syntax Softkeys Provide Friendly Interface

Since a substantial part of a microprocessor system designer's time is spent at the keyboard of a microprocessor development system, ease of use is very important. By means of directed-syntax softkeys, the 64000 leads the new user through an often bewildering maze of tools. The use of a random-access display further simplifies the operator interface to provide a feeling that the human is in control and not the machine.

Eight unmarked keys immediately below the CRT are labeled by the CRT. These softkeys reflect the complete set of allowable entries and change with each keystroke to reflect the next expected keyword or data in a command. If the user enters only the information prompted by the softkeys the syntax is guaranteed correct. Conversely, any entry not shown in the softkey labels will result in a syntax error. Thus the processor is always telling the user what it expects, avoiding the usual guessing game, "You enter a command and I'll tell you if it's right." In addition to eliminating the guessing game, the softkeys provide exactly the same interface for all operations.

Fig. 5a shows an example using the directory command, which can consist simply of the keyword directory or several options. In Fig. 5b the directory softkey has been pushed and the next allowable alternatives are shown:

<FILE>	user file name
all_files	all disc files
rec_files	all recoverable files
tapefiles	all tape files
listfile	specify an alternate listing file.

In Fig. 5c, the all-files option is selected and the labels again change to reflect other options. The complete command shown in Fig. 5d calls for all of the files modified after August 28, 1980 to be listed on the line printer.

If the cursor is moved to edit the command, the labels change to reflect the options available at that point in the line. If a softkey is pressed when the cursor is under any character in a keyword, the entire keyword is replaced by the new one and the line is expanded or contracted to accommodate the new entry.

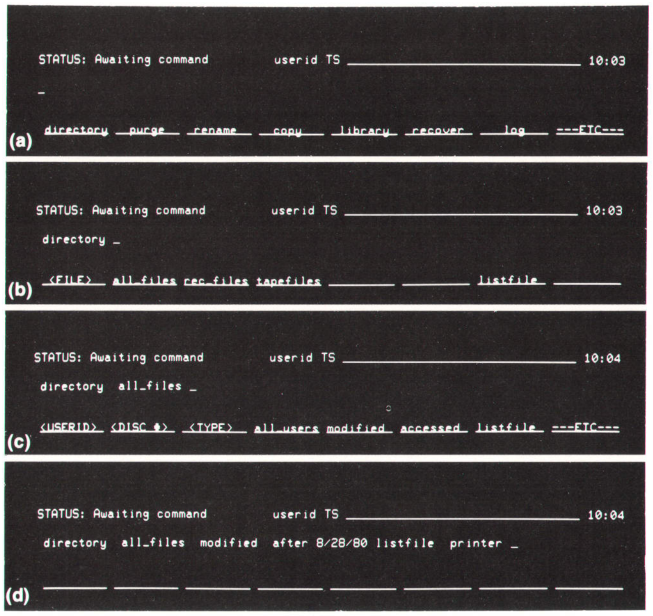

```
(a)
STATUS: Awaiting command          userid TS ______________________ 10:03
_
directory  purge  rename  copy  library  recover  log  ---ETC---

(b)
STATUS: Awaiting command          userid TS ______________________ 10:03
directory _
<FILE>  all_files rec_files tapefiles ______ ______ listfile ______

(c)
STATUS: Awaiting command          userid TS ______________________ 10:04
directory  all_files _
<USERID>  <DISC #>  <TYPE>  all_users modified  accessed  listfile  ---ETC---

(d)
STATUS: Awaiting command          userid TS ______________________ 10:04
directory  all_files  modified  after 8/28/80 listfile  printer _
______ ______ ______ ______ ______ ______ ______ ______
```

Fig. 5. *Constructing a command using the 64000 System's directed syntax softkeys. (a) The user has pressed* ETC *and now sees the softkey labels shown here. (b) The user presses* directory *and sees these new labels. (c) The user continues to construct the command by pressing* all_files. *(d) The complete syntactically correct command calls for a listing of all files modified after August 28, 1980.*

Software

Just as important as the hardware architecture in a complete solution is the software package. 64000 software currently available includes the following modules, some of which come in several versions to accommodate different microprocessors and languages: monitor, multiprocessing operating system, file manager, editor, assembler, compiler, linker, emulator, PROM programmer, and hardware self test.

Since users of the system can range everywhere from the expert digital hardware designer to one with no previous software experience, the 64000 system is designed to provide considerable capability for the experienced software designer, and through the use of the directed-syntax softkeys, to give the new user access to the full capability of the system, not just the subset that is frequently used and remembered. To further enhance the convenience of the system an effort was made to provide a uniform syntax and feature set in all aspects of the development tools. For example, numeric constants can be specified in decimal, hexadecimal, octal, or binary in the assembler, compiler, linker, emulator, PROM programmer, monitor, and editor.

The rules for variable names are the same for the assembler, compiler, linker, and emulator. The feature set for all of the above modules also remains the same for each microprocessor, so that the learning curve for a new processor is much shorter. In some cases the same person has to work with more than one processor type simultaneously, so this approach becomes essential to reduce confusion.

With these features combined with the performance of a 16-bit processor per user and a high-speed hard disc, the turnaround cycle for changes is substantially reduced. As an example, it is possible to edit a file to make corrections, assemble that file, link it to other modules, and then execute the new code on the emulator in one minute. This level of performance encourages proper maintenance of source programs instead of memory patching to fix a problem.

The Editor

Perhaps the most important part of a development system's operator interface is the editor. The functioning of the editor provides the most convincing argument for a random access display. The ability to modify the text by inserting, deleting, or overtyping and see the changes on a key-by-key basis gives the confident feeling of absolute control.

The importance of a symmetric instruction set is just being understood in the microprocessor world, but the

Resource Sharing in the Logic Development System

by Alan J. DeVilbiss

A 64000 Logic Development System is ordered as Model 64001S, with the options wanted listed separately. A 64001S System consists, at a minimum, of one 64100A Development Station, a disc memory, and a magnetic tape cartridge drive. A maximum of six 64100A Development Stations, a printer, and eight disc drives can be connected on a single I/O bus.

The operating system software executing in the host processor of each 64100A is implemented as a single-tasking system, responding to its keyboard inputs independently of any other 64100A stations, except when two or more stations require access to a shared resource simultaneously (e.g., a disc memory or the printer). The use of these shared resources must be coordinated. The sharing protocol is simple, minimizing overhead in the operating system and reducing the number of operations that must be recovered in case of a system fault. Specifically, the shared resources are:

1. Access to a disc memory (excludes directory)
2. Access to read or modify a disc directory
3. Access to the printer.

The mechanical and electrical protocol used on the 64000 I/O bus is compatible with the HP Interface Bus, or HP-IB (IEEE Standard 488-1978). However, in the 64000 System context, messages are restricted to those needed for system operation. For example, I/O drivers and message protocols that would allow direct user control of interface message generation are not available. Therefore, only supported disc memories and printers and other 64100A stations may be connected to a 64100A station.

The HP-IB standard was selected because of the existence of compatible disc memories and printers and a related family of reliable components (integrated interface electronics, connectors, and cables).

Each 64100A station can operate on the HP-IB as an active controller, talker, or listener. The current active controller monitors the state of the network—that is, which 64100A stations are using or are waiting to use a shared resource. The active controller has the exclusive right to use the I/O bus until control is passed to another 64100A. However, a resource reserved by another 64100A may not be used. Disc accesses not involving a disc directory access may be made by the active controller without restriction. Directory and printer accesses are the only two resources that must be reserved. Use of these resources is regulated by queues resident in the active controller for each function. The HP-IB address (from 2 to 7) corresponding to each 64100A is used as a name in the queues, with 0 serving as the null entry. The head of each queue has the exclusive right to use the resource. Addresses within the queue indicate 64100A stations waiting for the resource. Only the active controller can modify the queue by removing its address from the head of the queue. All other entries are moved up by one position when the active controller is finished with the resource. The active controller can also replace the first null entry in the queue with its own address when it requires the resource.

The active controller may modify the queues and make one disc access (a read or write of up to 4096 bytes, typically) and fill the printer buffer if it is at the head of the printer queue. Then control must be passed if any other 64100A has a pending I/O request. The active controller conducts a parallel poll. If no other 64100A responds, the current active controller remains active controller and can continue with its own I/O as required. Affirmative poll response from another 64100A indicates a request to become active controller. If more than one 64100A responds, the address of the responding 64100A next higher (modulo 8) than the current active controller is selected.

The selected 64100A is sent an eight-byte message indicating the current state of the directory and printer queues, and then the Take Control interface message is sent to that 64100A. The selected 64100A becomes active controller and may use the I/O bus and/or modify the queues.

On each 64000 system, one and only one 64100A is designated as master controller. This unit is responsible for initiating system activity by becoming the first active controller when the system is powered-on. Only this unit may assert the Interface Clear message, and therefore it is responsible for restarting a system that has experienced a partial power failure or a disruptive hardware or software fault.

When a 64100A powers on, it must first load its operating system from the system disc at I/O address 0, unit 0. To accomplish this without disturbing a functioning system if this 64100A is entering late, the nonactive controller status is selected at power-up, and I/O bus control is requested by affirmative response to any parallel poll by an active controller. If the unit is not master controller, it must wait until control is passed to it from another 64100A. If the 64100A is designated master controller, it waits for about three seconds (a worst-case delay for a functioning system), asserts Interface Clear and becomes the active controller.

Once a 64100A station has become an active controller and loaded its operating system software from system disc memory, it executes a program to identify all other devices connected to the I/O bus at that time. The results of that procedure are used to control generation of tables in the disc, printer, and network I/O drivers to make proper use of the devices attached to the network.

Each disc memory identified is cataloged by I/O address, disc unit number, type (7905, 7906, 7910, 7920, 7925), directory location and size, and record size. A logical unit number is assigned for each disc. The results of the I/O identification are listed on the 64100 display for reference and to aid in debugging a malfunctioning system.

This architecture makes it easy to change the number of 64100A development stations, the number and/or type of disc drives, and the printer. To effect a change, the system is powered off, reconnected and powered back on. No user-directed change in software is needed.

Fault Recovery

Recovery features have been implemented to lessen the effects of system faults. For example, it would be undesirable if low power on one 64100A station aborted an edit session on another station. All I/O operations have time-outs assigned, with appropriate recovery procedures in the event of malfunction. Disc operations that can't complete are retried. If a pass of control doesn't complete within the allotted time, the process is aborted and the previous active controller resumes control status.

The master controller assumes a system monitor function. Whenever the master controller passes control a three-second timer is started. If this timer expires, control must be requested by affirmative poll response, even if the master controller has no pending I/O request. If another three seconds go by without a response, the active controller is presumed to have crashed or powered off, and the master controller asserts the Interface Clear message and becomes active controller.

Whenever the master controller becomes active controller by Interface Clear, the network queues are initialized to the null state, a restart flag is set and the queues and control are passed around the network one time, independent of I/O requests. The restart flag inhibits normal I/O activity. Each 64100A is given the opportunity to take either the directory or the printer queue head if its internal state indicates it had this position before the restart. This process minimizes the effects of the loss of network state information by a crash of the active controller while another 64100A is modifying the directory or using the printer. When control is returned to the master controller, the restart flag is cleared and normal operation resumes. Time-outs in the printer and network drivers of 64100A stations that were waiting for the directory or the printer cause them to reenter the network queues. The order in the queues may be changed but everyone ultimately is serviced.

Alan J. DeVilbiss

Al DeVilbiss has been a circuit and software designer with HP since 1965. A native of Roanoke, Louisiana, he received his BSEE degree from Louisiana Tech University in 1960 and his MSEE degree from California Institute of Technology in 1961. Before coming to HP he designed flight computers for four years. Two patents, on electronic ignition and vertical amplifier circuits, have resulted from his work. Al is married, has two children, and lives in Colorado Springs, Colorado.

same motivation also exists for symmetry in an editor command set. The first step in the editing process is usually positioning to an area in the file of interest. In the 64000 there are no artificial constraints on file size or workspace use, and positioning can be performed by rolling the text up or down, moving the cursor up or down, paging up or down, randomly by specifying a line number, or searching for a character string in the forward or reverse direction. All operations involving a group of lines, such as deleting, extracting, copying, listing, or performing character replacement are done starting with the line containing the cursor thru or until (inclusive or exclusive) a line number, a character string, the start of the file, the end of the file or the entire file. With directed-syntax softkeys the availability of these symmetrical options is always obvious to the user.

The memory space available to the editor can be viewed as two double-ended queues (Fig. 6). These two queues share the same memory space, so when one contracts the other can expand into available memory. Another way to view this memory is as a single circular buffer with a display window. When an edit session is started two scratch files are created. Since more than one 64100A Development Station may be using copies of the editor at the same time, the names of these files are made unique by appending the bus address of the station. These files serve as temporary storage for text that will not fit in memory.

When the original source file is opened, enough lines to fill the display are read and placed on the CRT screen. More of the source file is read into queue A. The amount of text read is limited to produce a reasonable response time. Many edit sessions do not extend over the entire source program, and a long initial delay can be annoying. Only for very short

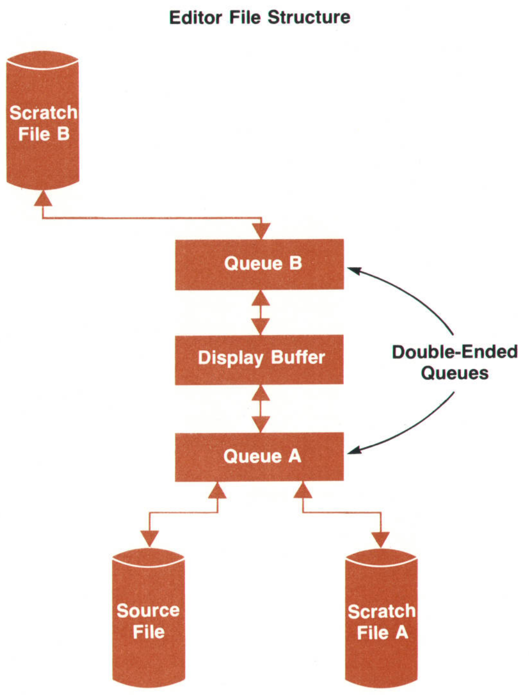


Fig. 6. *The 64000 editor's memory space can be viewed as two double-ended queues that occupy the same memory space, so that when one expands the other contracts. Scratch files are created when an edit session is started.*

64500 PROM Programmer

A universal development system like the 64000 must be able to program a wide variety of PROMs (programmable read-only memories) to store object code for prototypes and limited production runs. The semiconductor industry currently has many memory types available: bipolar ROMs, ultraviolet-light-erasable MOS ROMs, and combination chips containing both a MOS ROM and a microprocessor. Many speed ranges and memory sizes are offered to suit different users' requirements. The goal of the 64500 PROM Programmer design was to create a programming system that would accommodate the widest variety of popular PROMs, be easy to use in the 64000 system, and be low in cost. Low cost means both initial cost and the incremental cost of adding facilities to program other types of PROMs.

A study was initiated to catalog all currently available PROMs. Size, pinouts, power supply requirements, speed, and programming specifications were compared to assess the difficulty of building a truly universal system. From this point, a design strategy emerged. The resulting system consists of a control card occupying one slot in the 64100A mainframe and a socket module that resides in the 64100A panel insert. The control card contains adjustable power supplies and general input/output driver circuits, as well as a 64000 mainframe interface. The individual socket modules match PROM pinouts and tailor the control card's general signals to meet specific PROM programming specifications. Currently, eight socket modules are available.

To further simplify the hardware requirements of the controller and the socket module, all sequence timing and pulse width control are done by software in the PROM driver. Only pulse amplitudes and rise and fall times are set by hardware circuits on the socket module. Software control makes programming the memory chips easier. Each socket module has an identification code that is read by the driver. From this code, the appropriate programming routines and tables for the PROM family are automatically selected. If a single-socket module can program more than one PROM type, the available choices are displayed on softkeys for user selection.

-Roger Cox

files is the entire file read before the user is allowed to issue commands.

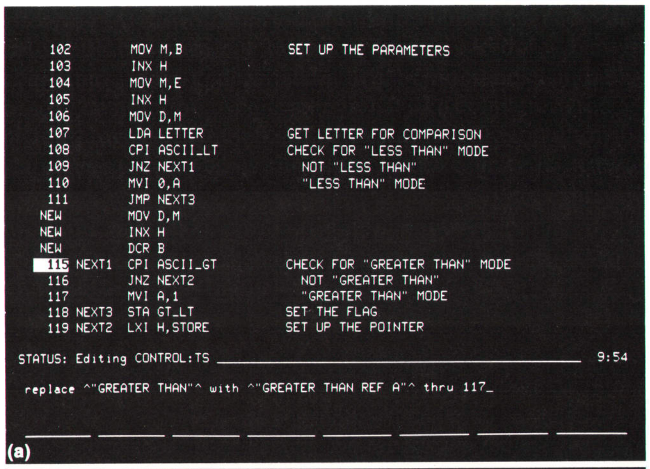
As various commands cause more of the source file to be read the data is brought into memory and shuffled between the two double-ended queues. When the internal memory space is filled records are written to scratch file B in the forward direction. Should a command require moving to an earlier line of text the records are written to scratch file A and read from scratch file B. The original source file is never overwritten.

When the end command is issued a destination file is created. The text is written from scratch file B, the internal buffer space, scratch file A, and the source file into this destination file. The original source file is then purged and the destination file renamed as source. The original file has then been placed in a deleted file list by the 64000 file manager and can be recovered. When the scratch files are closed they are deleted from the disc directory by the file manager.

A particular problem in the microprocessor world is the use of different assemblers and cross assemblers for the same microprocessor, sometimes from the same manufacturer. The text editor is a tool that usually bridges this gap, and in a few cases, dedicated conversion programs are available. To try to accommodate source programs written for a variety of assemblers, the 64000 editor extends the normal string replacement capabilities shown in Fig. 7. By allowing for the recognition of unknown characters or variable length strings of characters terminated by known characters, more generalized editing commands can be issued. The notation used is somewhat like the pattern recognition languge SNOBOL.[2] The example in Fig. 8 shows a statement that reverses the order of the operands in two-operand 8080 instructions. This string replacement capability is further augmented by the ability to specify the columns over which the replacement should apply. The columns are specified in the same manner as the tabset, that is, either by specifying the column numbers or editing a line reflecting the current range specification.

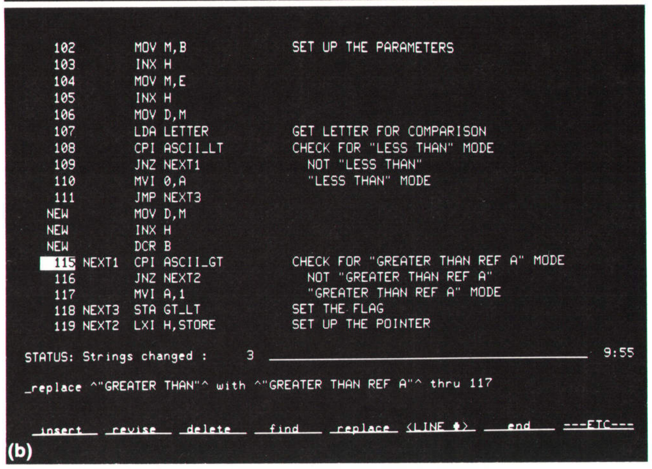
File Management

The heart of all modern software development tools is the file management system. While automatic space allocation is a part of almost all systems, in the 64000 system this facility is significantly extended to include the ability to recover accidently purged files or previous copies of edited



(a)

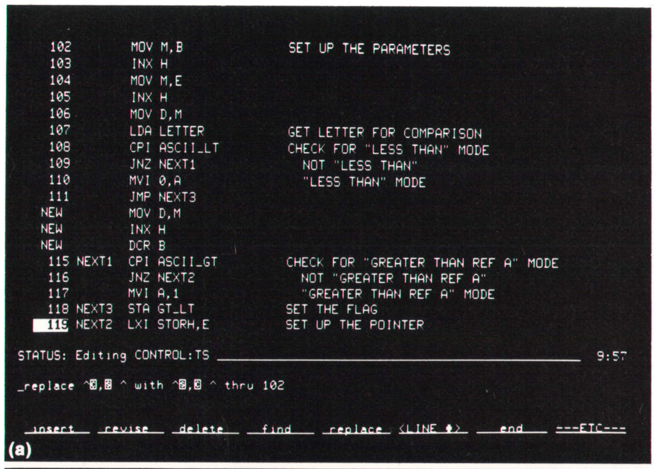
```
  102         MOV M,B              SET UP THE PARAMETERS
  103         INX H
  104         MOV M,E
  105         INX H
  106         MOV D,M
  107         LDA LETTER           GET LETTER FOR COMPARISON
  108         CPI ASCII_LT         CHECK FOR "LESS THAN" MODE
  109         JNZ NEXT1              NOT "LESS THAN"
  110         MVI 0,A                "LESS THAN" MODE
  111         JMP NEXT3
 NEW          MOV D,M
 NEW          INX H
 NEW          DCR B
  115 NEXT1   CPI ASCII_GT         CHECK FOR "GREATER THAN" MODE
  116         JNZ NEXT2              NOT "GREATER THAN"
  117         MVI A,1                "GREATER THAN" MODE
  118 NEXT3   STA GT_LT            SET THE FLAG
  119 NEXT2   LXI H,STORE          SET UP THE POINTER

STATUS: Editing CONTROL:TS ________________________________________ 9:54

 replace ^"GREATER THAN"^ with ^"GREATER THAN REF A"^ thru 117_
```



(b)

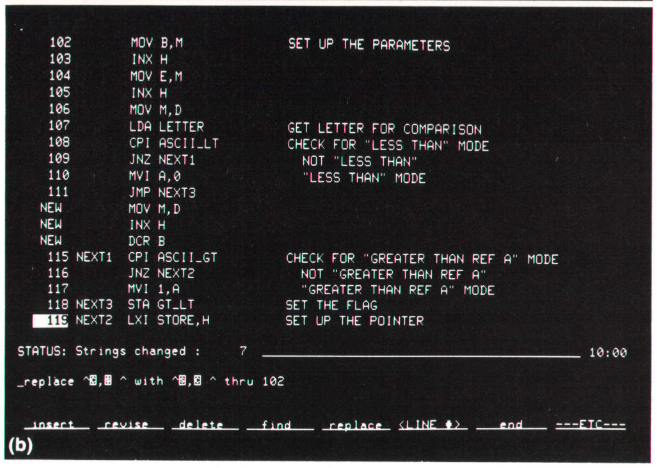
```
  102         MOV M,B              SET UP THE PARAMETERS
  103         INX H
  104         MOV M,E
  105         INX H
  106         MOV D,M
  107         LDA LETTER           GET LETTER FOR COMPARISON
  108         CPI ASCII_LT         CHECK FOR "LESS THAN" MODE
  109         JNZ NEXT1              NOT "LESS THAN"
  110         MVI 0,A                "LESS THAN" MODE
  111         JMP NEXT3
 NEW          MOV D,M
 NEW          INX H
 NEW          DCR B
  115 NEXT1   CPI ASCII_GT         CHECK FOR "GREATER THAN REF A" MODE
  116         JNZ NEXT2              NOT "GREATER THAN REF A"
  117         MVI A,1                "GREATER THAN REF A" MODE
  118 NEXT3   STA GT_LT            SET THE FLAG
  119 NEXT2   LXI H,STORE          SET UP THE POINTER

STATUS: Strings changed :     3 ______________________________ 9:55

_replace ^"GREATER THAN"^ with ^"GREATER THAN REF A"^ thru 117
```

Fig. 7. *Using simple character string replacement. (a) The command executes from the current position (indicated by the line number in inverse video) to the position specified in the command. (b) The status line reports the replacement performed.*



```
 102         MOV M,B                SET UP THE PARAMETERS
 103         INX H
 104         MOV M,E
 105         INX H
 106         MOV D,M
 107         LDA LETTER             GET LETTER FOR COMPARISON
 108         CPI ASCII_LT           CHECK FOR "LESS THAN" MODE
 109         JNZ NEXT1                NOT "LESS THAN"
 110         MVI 0,A                  "LESS THAN" MODE
 111         JMP NEXT3
NEW          MOV D,M
NEW          INX H
NEW          DCR B
 115 NEXT1   CPI ASCII_GT           CHECK FOR "GREATER THAN REF A" MODE
 116         JNZ NEXT2                NOT "GREATER THAN REF A"
 117         MVI A,1                  "GREATER THAN REF A" MODE
 118 NEXT3   STA GT_LT              SET THE FLAG
 119 NEXT2   LXI STORH,E            SET UP THE POINTER

STATUS: Editing CONTROL:TS ______________________________________ 9:57

_replace ^🄲,🅂 ^ with ^🅂,🄲 ^ thru 102

 insert   revise   delete   find   replace  <LINE #>   end   ---ETC---
```

(a)



```
 102         MOV B,M                SET UP THE PARAMETERS
 103         INX H
 104         MOV E,M
 105         INX H
 106         MOV M,D
 107         LDA LETTER             GET LETTER FOR COMPARISON
 108         CPI ASCII_LT           CHECK FOR "LESS THAN" MODE
 109         JNZ NEXT1                NOT "LESS THAN"
 110         MVI A,0                  "LESS THAN" MODE
 111         JMP NEXT3
NEW          MOV M,D
NEW          INX H
NEW          DCR B
 115 NEXT1   CPI ASCII_GT           CHECK FOR "GREATER THAN REF A" MODE
 116         JNZ NEXT2                NOT "GREATER THAN REF A"
 117         MVI 1,A                  "GREATER THAN REF A" MODE
 118 NEXT3   STA GT_LT              SET THE FLAG
 119 NEXT2   LXI STORE,H            SET UP THE POINTER

STATUS: Strings changed :    7 ______________________________ 10:00

_replace ^🄲,🅂 ^ with ^🅂,🄲 ^ thru 102

 insert   revise   delete   find   replace  <LINE #>   end   ---ETC---
```

(b)

Fig. 8. *Powerful text modification using SNOBOL-like features. (a) By using the special characters "anystring" ($\boxed{S}$) and "anycharacter" ($\boxed{C}$) the operand field of this 8080 code can be reversed. (b) The text changes virtually instantaneously and the status line reports seven replacements were performed.*

files up to the time when the space is needed for new files. A further enhancement aimed at managing the increased number of files being used is the user identification added to files names. By entering a user ID at the beginning of a session all operations will be carried out on files under that name. The directory list defaults to listing only the files under that ID.

Further enhancements offered by the 64000 file manager come in the directory, including a listing of space available and comprehensive data on file use. Monitoring revisions to programs is made easy since the date and time of last access and modification of each file are automatically maintained and shown in the directory list. The linking loader also specifies in the load map the date and time of the last update of each relocatable module loaded. The significance of this record keeping in a multiple-design project where program modules are independently maintained cannot be overstated.

Another important function for the file system is the ability to submit a stream of system commands contained in a file. This capability, available on many systems, makes performing a long series of tasks almost foolproof. An extension to this function in the 64000 allows parameters to be passed to one of these command files in a manner similar to assembly language macros. Then more generalized command files can be created, thus reducing the number of files created and used. For example, a command file could be created that automatically sequences through the operations of assembly, linking, loading, and emulating, and only the source file(s) need be specified at the time the command file is invoked. Also, by including a learn mode for building command files the full aid of the directed-syntax softkeys is made available in constructing command files.

Page Structure

The 64000 file management system has a linked list structure. Each of the files consists of blocks of sectors called pages. The number of sectors per page is constant for a given disc but may vary for different discs to optimize certain file management operations. The pages of a file are linked in both forward and backward directions (see Fig. 9). This symmetry is used to its greatest advantage in the 64000 editor. Editor operations such as rolling, paging, and string searching can be done with equal efficiency either forward or backward through the text.

When a file is being updated the same sectors on the disc are used. If the size of the file is increased the file manager allocates another page to the file, linking it to the end of the last page. The list of available pages is kept in much the same way as a file. It is a doubly linked list of pages. Free pages are taken from the front of the list when they are allocated to files. This approach allows files to grow easily without bound and precludes the need for a user-invoked disc packing program. The disc remains continuously packed by the nature of the file structure.

Directory Format

As with most file management systems the keys to locating a file on the disc are kept in a separate area called the directory. The 64000 directory is organized as a hash coded list. Hash coding minimizes the amount of searching required to locate the directory entry for a given file. The hashed value of the file name indicates the directory sector on which the file information is most likely to reside. The

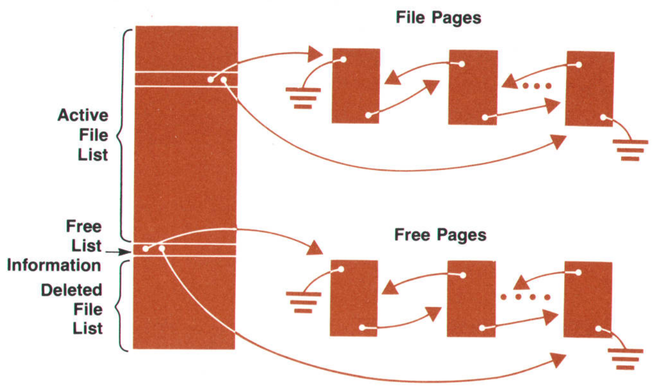


Fig. 9. *64000 file structure. The linked list organization allows for flexible file size.*

64000 Command Parsing

Commands are interpreted in the 64000 System using an LALR (look-ahead, left-to-right) parsing technique. The syntax of the commands of an application module such as the monitor, editor, or PROM programmer is described in a concise and readable format by a grammar. An example of this is the editor's delete command shown in Fig. 1. The complete grammar is given as input to a parser generator program, and the result is a table that is used by the 64000 parser to parse the text that the user types on the command line.

$$\textbf{a) delete} \left[\begin{array}{c} \left\{ \begin{array}{c} \textbf{thru} \\ \textbf{until} \end{array} \right\} \left\{ \begin{array}{c} <\textbf{line\#}> \\ <\textbf{string}> \\ \textbf{start} \\ \textbf{end} \end{array} \right\} \\ \\ \textbf{all} \end{array} \right]$$

```
b) <DELETE_COMMAND>    ::=    <delete><RANGE_SPEC>

   <RANGE_SPEC>        ::=    <EMPTY>
                              <thru><LIMIT>
                              <until><LIMIT>
                                all

   <LIMIT>             ::=    <STRING>
                              <NUMBER>
                                end
                                start

   <delete>            ::=      delete
   <thru>              ::=      thru
   <until>             ::=      until
```

Fig. 1. *Syntax of the editor's* delete *command. (a) Concise syntax. (b) BNF-like grammar used to drive semantic and softkey routines.*

LALR parsing provides a convenient structure for 64000 application programs. When a command is parsed it is decomposed in exactly the same manner as the grammar used to create the parsing tables. Each line of the grammar is an opportunity to perform a semantic function. Thus the 64000 parser acts as a driver for the various functions a program performs.

The same features of LALR parsing that drive the executing functions of 64000 programs are used to drive the softkeys. As a command is typed into the command line the characters are continuously scanned by the 64000 parser. As the various statements of the grammar are applied to the character string the corresponding level of softkeys is selected. This parse continues up to the present position of the cursor in the command line. At the end of the parse the softkeys corresponding to the cursor position are displayed. In this way the user is shown all of the available choices at that time.

Since the command line is scanned almost continuously the softkeys are always consistent with the cursor position. Because of this the cursor can be moved to any position in the line and the softkeys will track the syntax. Also, the correct softkey level is dependent only on the characters contained in the command and not on a sequence of user actions. For users who choose to type instead of using the softkeys and for commands that are recalled into the command line the softkey tracking still works.

LALR parsing is deterministic in the detection of syntax errors. When a string of characters does not correspond to a permissible sequence as defined by the grammar it is detected as an error. At that

```
STATUS: Editing FILEX
  merge _
  <FILE>  from   thru   ____  ____  ____  ____  ____


STATUS: Editing FILEX
  merge FILEZ from 2$ thru 45 _
  ____  ____  ____  ____  ____  ____  ____  ____


ERROR: Invalid line number
  merge FILEZ from 2$ thru 45 _
                    -
  ____  ____  thru  ____  ____  ____  ____  ____
```

Fig. 2. *When a syntax error is detected an instructive message is displayed and the cursor is placed under the error. The softkeys are consistent with the cursor position.*

time the position of the error in the command and the set of correct syntax elements are known. The 64000 convention is to place the cursor at the position of the error and report the error in a manner that specifies what was expected. The softkey parsing is reinitiated as well, so that the softkeys are again labeled with the available choices for the current cursor position (see Fig. 2).

Flexibility is a bonus of the LALR parsing technique. When a change or addition to the syntax of a program is desired, it can be made quickly with a minimum of impact on other features. Tables for the new grammar are generated and, if required, a softkey level template is added or changed. A new message may be added to the table of error messages. The general structure of the softkey parsing is shown in Fig. 3.

-Brian Kerr

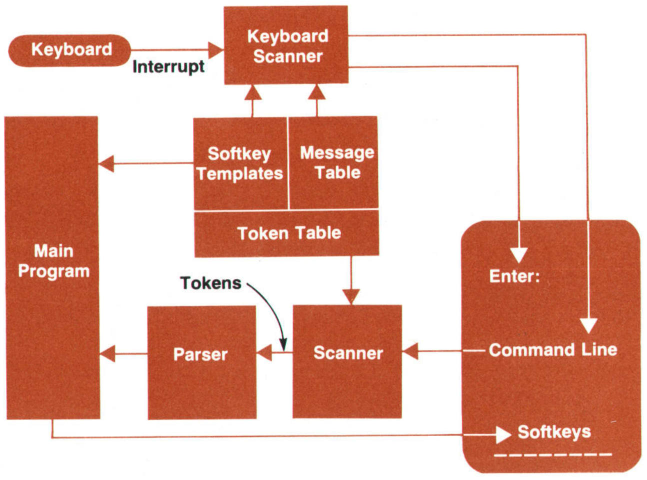


Fig. 3. *Softkey operation. Interactions between the main program and the softkeys are well-defined and suitable for many applications.*

data on that sector will indicate if the file exists or if another directory sector should be searched. As long as the directory is only partially full the file should either be found or proved nonexistent with only one disc read. Directory size has been chosen to correspond to the size of the disc. This guarantees that the directory will not be too full for efficient file lookup.

Each directory entry gives the name, user identification, and type of the file. Each entry contains pointers to the first and last pages of the file. This is the necessary information for accessing and deleting the file. In addition, two dates and times are kept for each file. One is the date and time that the file was last accessed. This is modified with the system date and time whenever the file is opened. The other is the date and time that the file was last modified. It is updated when the file is closed after records have been added or rewritten. These dates provide the user with convenient records of file use. The directory list and cassette backup commands use the dates as qualifiers for operations. For example, the user can store all recently changed files with the command `store all_files modified after 5/31/80`.

Recovering Deleted Files

The linked list file structure allows for a special feature of the 64000 file management system. Since deleted files are added to the end of the free list they are still intact until the entire free list has been allocated to other files. When a file is deleted its directory information is transferred to a special section of the directory. This is a circular list of files that have been deleted. A user who has made a mistake and deleted the wrong file can issue a recover command. This routine searches the recoverable file list for the file and if the file is found checks to insure that its pages have not been allocated to another file. If they have not, the file is restored to the directory of active files. Since the 64000 editor always purges the original file and creates a new copy, the user can recover previous versions.

File Format

All user-accessible files have a similar data format. The data is stored in variable-length records. The number of words of data in a record is placed in the bytes immediately preceding and following the data. Again, this symmetry allows for bidirectional access. It also provides a means for insuring the integrity of the file data. If the two lengths of a record are not the same then a data read or write error can be assumed.

Program modules such as the editor, assembler, and linker are called by the 64000 monitor using a system of overlays. When a module has been selected by the user or the currently running module an operating system routine is called to bring the correct file from the disc. Files of this sort are kept in a special non-record format. They are stored as memory images that can be read directly into the location in memory where they will be executed. It is desirable that this operation be performed as quickly as possible so as to be transparent to the user. To accomplish this the disc is organized in a special way. Normally sectors that are logically adjacent in a file management system are also physically adjacent on the disc. In the 64000 this is not the case. Logically adjacent sectors are spaced some distance apart depending on the particular type of disc. When a sector is read the disc continues to rotate while the data is being transmitted over the system bus and placed in the 64000 memory. By the time the next sector is requested the disc has rotated so that the physical sector is in the correct position to be read. In this way many disc rotations are eliminated.

Acknowledgments

The first impression of a new product is often based on its appearance, but long-term satisfaction is usually based on reliability and ease of maintenance. Bill Fischer and Don Miller were responsible for meeting both of these objectives by designing an attractive yet functional package that is easy to manufacture and service and has good thermal design to improve reliability. Steve Shepard directed all of the hardware design and was responsible for the difficult task of coordinating the many pieces going into a large complex system. Paul Sherwood and Pete Rawson were responsible for the design of a flexible high-performance mainframe to support the broad spectrum of accessories needed in this measurement area.

If one person were to be credited for the whole product there would be little doubt by anyone on the design team that it should be George Haag. He was a major contributor to the system architecture and the primary source of the human interface design. He served as section manager for much of the project and provided the basis for the introduction of a major new product from the division.

Brian W. Kerr

Brian Kerr attended Rice University in Houston, Texas, receiving his BSEE and MEE degrees in 1976 and 1977. With HP since 1977, his responsibilities have included the 64000 editor and file manager and option performance verification. Brian was born in Salisbury, Maryland. He's single, lives in Colorado Springs, Colorado, and enjoys softball, bridge, volleyball, bowling, and cross-country skiing.

Thomas A. Saponas

Tom Saponas received BSEE/CS and MSEE degrees from the University of Colorado in 1972 and joined HP the same year. He's done software design for data acquisition systems and circuit and software design for logic analyzers (1607A and 1611A). Initially software project leader for the 64000 System, he later became section manager. He's a member of IEEE and has authored articles in IEEE and Audio Engineering Society publications. Tom was born in Watertown, South Dakota. He's married, has two children, and has lived in Colorado Springs for most of his life. He's a director of the regional science fair and is interested in car repair, photography, bridge, and cross-country skiing.

References
1. W. D. Eads and D. S. Maitland, "High-Performance NMOS LSI Processor," Hewlett-Packard Journal, June 1976.
2. R.E. Griswold, J.F. Ponge, and I.P. Polonsky, "The SNOBOL 4 Programming Language," Prentice-Hall, 1971.

ORDERING INFORMATION
HP Model 64000 Logic Development System

An HP Logic Development System order should be entered as 64001S followed by the system options that are to be configured, systemized and tested at the factory. A basic system must include at least one HP disc, one 64100A Development Station, and one 64940A Tape Cartridge Unit. All appropriate software and documentation is provided with the selected options. Operating system software is supplied with the disc options. If no disc is ordered, operating system software must be ordered as Option 800 to 64001S or as product number 64800A.

Additional stations may be added to form a cluster by submitting a 64001S order with desired options for each new station. A total of six stations may be configured as a cluster, with each station containing any allowable combination of options.

Most of the options may be added to the system on-site after initial installation. These products may be ordered by subsystem product number and are shipped with appropriate software documentation and cables.

OPTION	DESCRIPTION	PRICE in U.S.A.	SPECIAL INSTRUCTIONS
006	7906M/102 20M Byte Cartridge Disc	$16000	Minimum—one per cluster Includes operating system software (64800A)
010	7910H 12M Byte Winchester Type Disc	8350	
020	7920M/102 50M Byte MAC Disc	19000	
025	7925M/102 120M Byte MAC Disc	23000	
100	64100A Development Station	8600	Only one per 64001S order Six per cluster
040	64940A Tape Cartridge Drive	1800	Minimum one per cluster
202	64200A/202 8080 Emulator Subsystem	2800	One emulator subsystem only per 64001S order. Includes emulator control cards, emulator pod & appropriate software & documentation.
203	64200A/203 8085 Emulator Subsystem	2800	
212	64210A/212 6800 Emulator Subsystem	2800	
252	64250A/252 Z80 Emulator Subsystem	2800	
151	64150A/151 64K Byte Emulator Memory	6400	One emulator memory only per 64001S. Order for use with Option 2XX. Emulator subsystem includes 64151A memory control & appropriate memory cards.
152	64150A/152 32K Byte Emulator Memory	3400	
153	64150A/153 16K Byte Emulator Memory	2300	
154	64150A/154 8K Byte Emulator Memory	1600	
300	64300A Real-Time Transparent Logic Analyzer	1400	One only per 64001S order. Order for use with option 2XX.
502	64500A/502 PROM Prog for 2716	700	Self test built-in
503	64500A/503 PROM Prog for TIS 470-473	900	Includes 64502A Module for 2716 with self test
504	64500A/504 PROM Prog for Harris 4X-8X-08 Series	900	Includes 64502A Module for 2716 with self test
505	64500A/505 PROM Prog for Siliconix 4X-8X-9X Series	1000	Includes 64502A Module for 2716 with self test
507	64500A/507 PROM Prog for 2708/2704	900	Includes 64502A Module for 2716 with self test
508	64500A/508 PROM Prog for TIS 287/387	900	Includes 64502A Module for 2716 with self test
509	64500A/509 PROM Prog for 2732	900	Includes 64502A Module for 2716 with self test
510	64500A/510 PROM Prog for 8748	Quote	Includes 64502A Module for 2716 with self test
513	64500A/513 PROM Prog for 8755	Quote	Includes 64502A Module for 2716 with self test
800	64800A Operating System Software	550	Included with disc option
840	64840B Relocating Macro Assembler RMA/Linker & Emulator Software for 8080/85	550	
841	64841B RMA/Linker & Emulator Software for 6800	550	
842	64842B RMA/Linker & Emulator Software for Z80	550	
843	64843A RMA/Linker for 6502	550	
844	64844A RMA/Linker for 6805/09	550	
846	64846A RMA/Linker for MCS-48	550	
847	64847A RMA/Linker 9900	550	
848	64848A RMA/Linker 1802	550	
849	64849A RMA/Linker F8/3890	550	
850	64850A RMA/Linker Z8	Quote	
810	PASCAL Compiler 8080/85	2000	
812	PASCAL Compiler Z80	Quote	
008	2608A/046/110 400 lpm Printer	10375	Includes paper catch
031	2631B/046 180 cps Printer	4090	Includes paper catch casters, sound cover
090	64030A Development Station Cart	300	
930	64930A Service Kit	7900	Contains both FSI and PSP

MANUFACTURING DIVISION: COLORADO SPRINGS DIVISION
P.O. Box 2197
Colorado Springs, Colorado 80901 U.S.A.

Emulators for Microprocessor System Development

by James B. Donnelly, Gordon A. Greenley, and Milo E. Muterspaugh

SIM·U·LATE *vt*: to pretend, feign. EM·U·LATE *vt*: to equal. Until recently, the development and debugging of software for new processor-based systems was frequently done with the aid of simulators, which are programs running on a large host computer and having the property of simulating the instruction set and the programming model of the new or target processor. After the software was initially debugged using the simulator, further debugging of the software-hardware system was done with the aid of debug programs and various hardware and software facilities that provided breakpoints, single-stepping, and other capabilities. More recently, logic analyzers have also aided in the process.

With the introduction of microprocessor development systems, a new tool has been made available to the designer in the form of the microprocessor emulator. Today's emulators combine many powerful software and hardware development tools into one convenient, easy-to-use system and greatly facilitate the process of integrating the hardware and software components of newly developed microprocessor-based systems. At the user interface, the hardware portion of the emulator replaces the microprocessor, and in keeping with the definition of emulation, attempts to be as much like the actual microprocessor as possible, both functionally and electrically.

The advantages of using an emulator include the ability to develop software on the actual processor to be used, the ability to load the newly developed programs into emulation memory and execute those programs in the development hardware in real time without having to use PROMs, thus speeding the development cycle, and the ability to debug hardware and software under very controlled conditions by being able to run, halt, and step the processor at will and to examine and modify registers and memory. An additional advantage is the ease with which the emulator is connected to the user system: it simply plugs into the socket where the microprocessor would normally go.

Design Objectives

In developing the emulators for the 64000 Logic De-

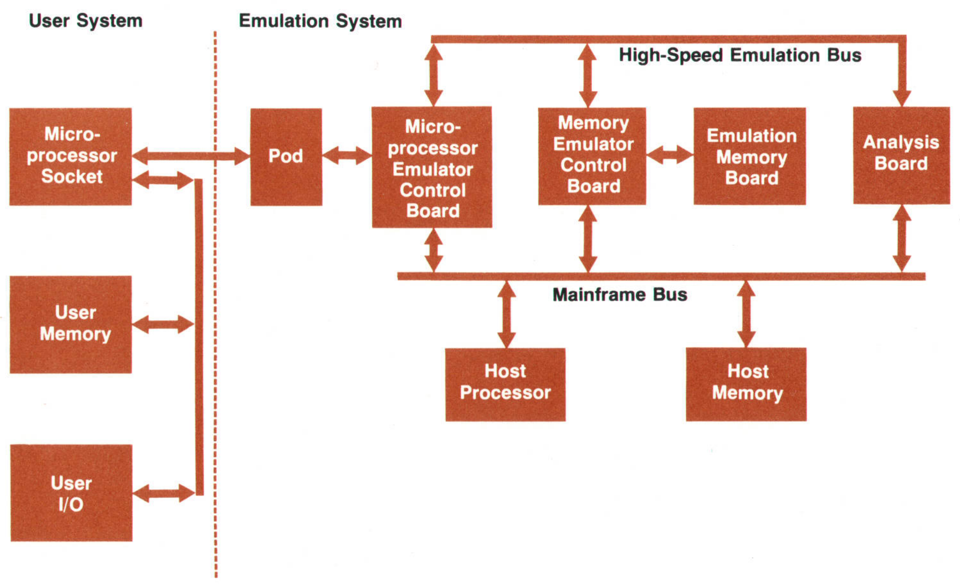


Fig. 1. *The 64000 emulator subsystem consists of a microprocessor emulator, a memory emulator, a logic analyzer, and a software support package.*

velopment System, the principal objective was to maximize transparency to the user and the user's system. This objective was applied to both the functional and the electrical aspects of the emulator.

Functionally, transparency was defined to mean that the user must not be deprived of or restricted in the use of any address space, instructions, interrupt systems, or other features normally available in the microprocessor being emulated.

Electrically, transparency means that the design of the emulator must minimize degradation in timing and electrical loading, so that the emulator will operate in the user's system as much like the emulated processor as possible.

System Description

In the 64000 System, a complete emulation system consists of the microprocessor emulator, the memory emulator, a logic analyzer, and a software support package that integrates the hardware components into a powerful, easy-to-use development tool (see Fig. 1).

The emulator system is partitioned into three interfaces: 1) the user interface, which is defined by the specifications of the processor being emulated, 2) the emulation bus, a high-speed bus that connects the processor emulator, the memory emulator, and the logic analyzer, and 3) the 64100A mainframe bus, which provides for control and communication between the mainframe host processor and the emulation system.

This architecture provides complete separation of the host processor and memory from the emulation system. This allows the host processor to run the emulation support software independently of the emulator, thus relieving the emulation processor of the burden of that overhead and helping to meet the design goal of functional transparency.

The Microprocessor Emulator

The microprocessor component of the emulation system is divided into two subassemblies, a pod external to the 64100A mainframe and a control board contained in the 64100A card cage (see Fig. 2).

The emulator pod contains a high-speed version of the emulated microprocessor, interface buffers, buffer control circuitry, and an internal clock source. A fully buffered architecture is used. Some of the advantages of this configuration are the minimization of potential damage from the user's breadboard and the ability of the 64000 system to gain control of the emulation processor and continue to function even though an electrical fault may exist in the user system. The combination of less than maximum capacitive loading on the processor provided by the isolation of the buffers and the use of high-speed versions of the processors gives the emulator the ability to operate with little or no degradation of timing specifications in most cases. The pod is connected to the user's microprocessor socket by a 30-cm dual flat cable and a 40-pin plug. Each signal wire in the cable is isolated from adjacent signals by alternating ground wires with the signal wires to minimize coupling. The pod connects to the emulator control board by two 1.5-m twisted-pair flat cables. This cable is driven by Schottky TTL buffers and is terminated in its characteristic impedance with one wire of each pair grounded to insure good high-speed signal quality.

The emulator control board consists of a timing section, which converts the timing signals of a given microprocessor into the standardized timing requirements of the 64000 emulation bus, various status and control registers, a 256-byte memory referred to as the background memory, background memory access control circuitry, a state machine called the background controller, and an illegal opcode detector. The function of the control board is to provide timing signals for the emulation memory and logic analyzer units and to provide the status and control interface between the emulation processor and the 64000 host processor.

The Universal Approach

Early in the emulator design phase, it appeared that it might be possible to identify certain functions of the control board that could be considered independent of microprocessor type and that these functions could be designed

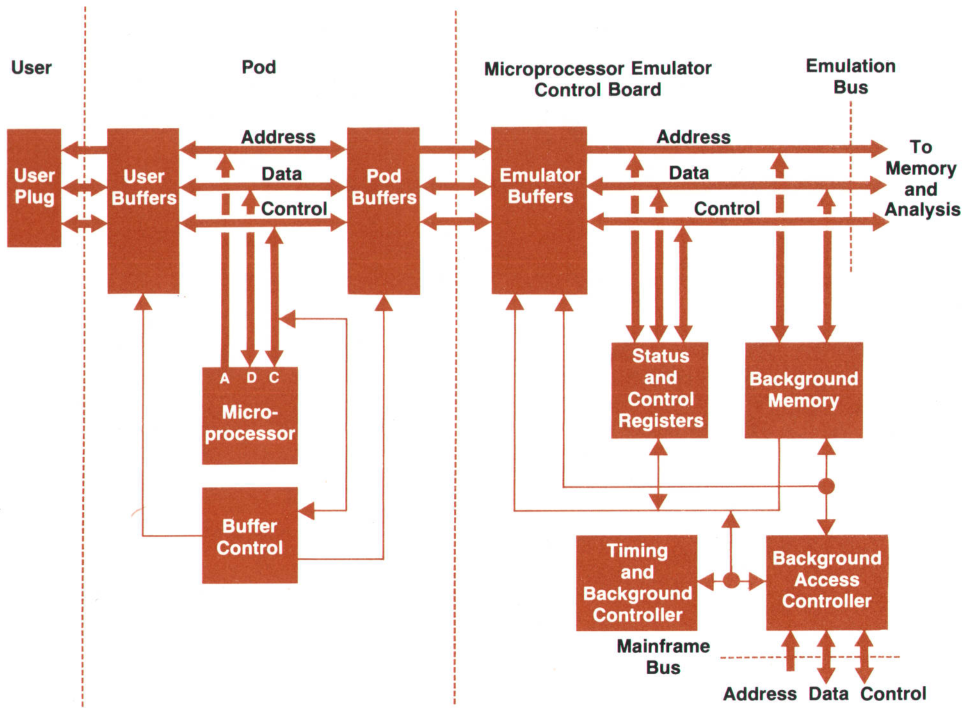


Fig. 2. *The 64000 emulator and host processor have separate buses so the host processor can run the emulation software independently of the emulator, thus helping to make the emulator functionally transparent to the user and the user's system.*

into a universal architecture, which could then become the core of several emulators. The result of this effort became known as the breeder board. It consists of a printed ciruit board containing the interface buffers, status and control registers, background memory and access control, background controller, and illegal opcode detector, plus an undefined wirewrap section to be used by the designer in breadboarding the timing section, which is the principal difference between the various microprocessors. To date, the breeder board has been the basis for three control boards that serve a total of five distinct microprocessors depending on the pod selected.

For HP, this approach has had the obvious advantage of more efficient use of engineering resources and shortened design cycles. The customer has also benefited by virtue of the fact that a common architecture results in a degree of consistency and continuity in the operating characteristics of the various emulators, thus reducing learning time. In addition, this approach has made it possible for some control boards to serve more than one microprocessor by just changing the pod.

Functional Description

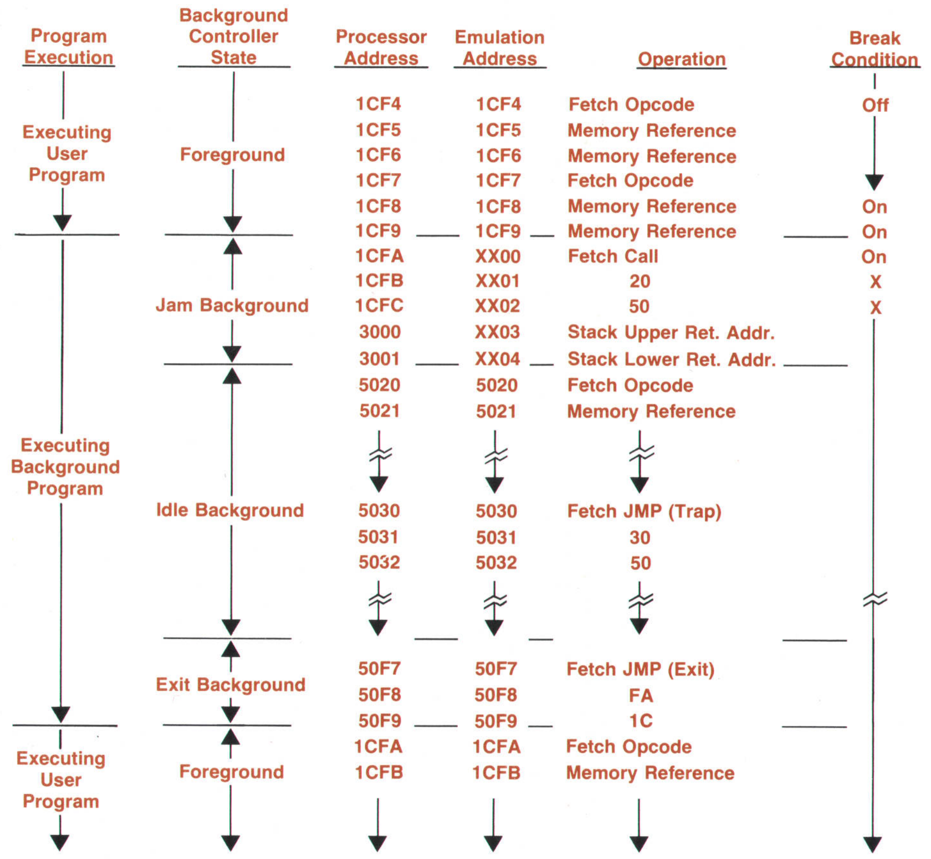
In operation, the emulator exists in one of two states, foreground or background. In the foreground state, the emulator appears to the user system as a standard microprocessor and executes user-written code, which may be physically resident in either user memory or emulation memory or a mixture of both, depending on how the user's memory space has been mapped. It is worthwhile to note that even though physical memory such as ROM may exist at a given address space in a user's system, it is possible to overlay that memory with 64000 emulation memory for code patching and debugging purposes.

In the background state, execution in the user system is suspended and the processor appears halted to the user system. The apparent halted state at the user interface is synthesized by manipulation of the pod buffers while the processor is actually running under 64000 system control in background memory. While in background, all inputs from the user system are inhibited to prevent possible user system interference with the execution of emulator background tasks.

Two important features of the 64000 emulators are key to the achievement of the functional transparency design objective. The first is the concept of background memory and the second is the means by which control is transferred between the user system and the 64000 system, that is, between foreground and background.

Background memory is a 256-byte RAM resident on the emulator control board. This memory is physically distinct from any memory either in the user system or on the emulation memory board (see "Emulation Memory" below), and does not occupy any of the user's address space. The background memory is accessible to both the emulation processor and the 64000 host processor and serves as the primary communication link between the two. The 64000 host processor loads various register unloading and register and memory read/modify routines into background memory and these routines are then executed by the emulation processor when it is transferred from foreground to background.

Transfer of the emulation processor from foreground to background is initiated by the occurrence of a break condition. A break may originate in any one of four sources. It may come from the logic analyzer unit after a specified condition has been met, from the emulation memory unit because of an illegal memory reference or write to ROM, from the processor emulator control board as a result of an illegal opcode fetch, or from the host processor, for example when the user enters a keyboard command for the emulator to stop.



Program Execution
Background Controller State
Processor Address
Emulation Address
Operation
Break Condition
1CF4 1CF4 Fetch Opcode Off
Executing User Program
Foreground
1CF5 1CF5 Memory Reference
1CF6 1CF6 Memory Reference
1CF7 1CF7 Fetch Opcode
1CF8 1CF8 Memory Reference On
1CF9 1CF9 Memory Reference On
1CFA XX00 Fetch Call On
1CFB XX01 20 X
Jam Background
1CFC XX02 50 X
3000 XX03 Stack Upper Ret. Addr.
3001 XX04 Stack Lower Ret. Addr.
5020 5020 Fetch Opcode
5021 5021 Memory Reference
Executing Background Program
Idle Background
5030 5030 Fetch JMP (Trap)
5031 5031 30
5032 5032 50
Exit Background
50F7 50F7 Fetch JMP (Exit)
50F8 50F8 FA
50F9 50F9 1C
1CFA 1CFA Fetch Opcode
Executing User Program
Foreground
1CFB 1CFB Memory Reference

Fig. 3. *The emulator exists in one of two states, foreground or background. The background controller, a four-state state machine, controls the transfer of the emulator processor from foreground to background and vice versa. This chart shows details of the background entry/exit process.*

A prime consideration in choosing the means for transferring control of the processor was the need to have some method that is independent of processor type, since the universal architecture of the control board was intended to work with a variety of processors. For example, a nonmaskable interrupt (NMI) might be a reasonable way to seize control of a processor, but some, such as the 8080, have no NMI. This need led to the use of a technique of jamming addresses independent of the addresses being generated by the processor onto the emulation background memory address bus at the appropriate time in the processor instruction cycle. This causes the opcode fetch to be returned to the processor from background memory.

The jamming process is synchronized by the background controller to the first opcode fetch cycle following the occurence of a break condition. This process simultaneously inhibits the user interface buffers and the address buffers from the processor to the background memory while enabling the jam address counter onto the bus. The jam address counter generates consecutive addresses starting at 00H for the length of one full instruction cycle. The length of the jam count is elastic, since state transitions of the controller occur on opcode fetch cycles and so the count length is a function of the instruction loaded into address 00H. Typically, a call instruction is used in the background code as the first instruction. The use of this type of instruction serves two purposes. First, the processor responds by placing the program counter on the stack. The stack is always in the same two locations in background memory regardless of where the processor stack pointer is set because the address bus is being jammed by the jam counter. This information is later used to determine where to send the processor when the emulator is returned to the foreground state. Second, the program counter is changed to the starting address of the background program, which results in transferring program control to the background memory when the jam cycle is terminated on the next opcode fetch. Functionally, this process may be viewed as a hardware implementation of a nonmaskable interrupt that is independent of processor type (Fig. 3).

The background controller is a state machine having four states: jam background, idle background, exit background, and foreground (see Fig. 4). State transitions occur at the beginning of opcode fetch cycles that are coincident with other qualifying events.

The background controller enters the idle background state on the next fetch following the beginning of the jam cycle previously described. This returns control of the address bus to the emulator processor which begins executing the background entry program. During this time, registers are unloaded, return addresses are computed, and so on. Following the completion of these tasks, the processor enters a jump self loop called TRAP where it awaits further direction from the host processor.

The host processor communicates with and controls the emulator processor indirectly through the medium of the background memory. This is possible because the memory is designed so that the host processor can read or modify background memory at the same time the emulator processor is executing code in that memory. The method of control

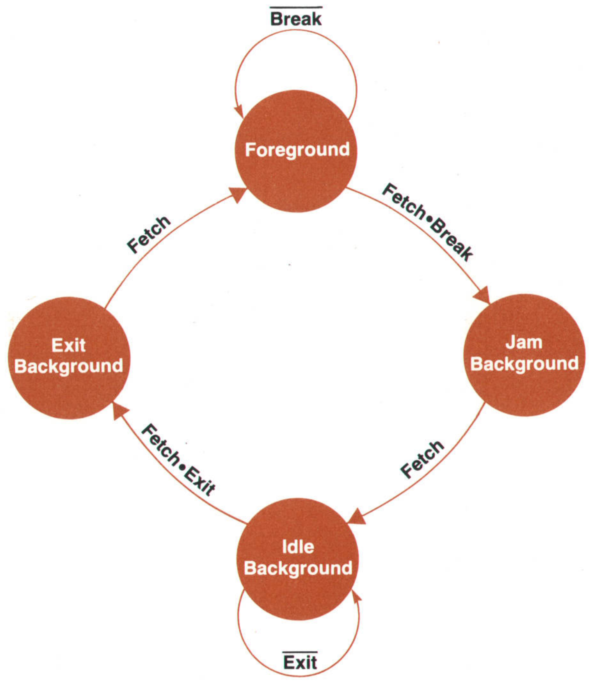


Fig. 4. *Background controller transition diagram.*

involves the host processor loading a program or programs into background memory and then changing the jump address of TRAP on the fly to coincide with the starting address of the desired background program. The emulator processor reads the new jump address and transfers to that point.

The exit background state is initiated when the host processor causes the emulator processor to make an opcode fetch from a dedicated background address called EXIT. The background controller recognizes the fetch from EXIT and makes the state transition. The opcode loaded into location EXIT is a jump instruction and the following bytes contain the address of the desired foreground entry point.

The transition from the exit background state to foreground immediately follows on the next opcode fetch cycle. At this point, the program counter of the emulator processor has been transferred to the foreground entry address by virtue of the previous jump instruction. The background controller hardware simultaneously enables the user interface buffers and switches the program source from background memory to foreground memory, which may be either user memory or emulation memory as determined by the memory mapper.

The process of entering and exiting background described here is employed in all cases where it is necessary for the host system to control the emulator processor. An example of this is single-stepping, where the emulator is returned to foreground for a single instruction cycle and then immediately jammed into background. Continuous stepping and non-real-time analysis are done in a similar manner.

Emulation Memory

The emulation memory consists of the memory emulator control board and from one to four emulation memory boards. Each fully loaded memory board contains 32K bytes of static memory.

The memory controller interfaces the emulation memory to the mainframe and the emulator system. The emulator has the full bandwidth of the emulation memory. If the mainframe wants to access the emulation memory, the mainframe cycles are held off until the emulator completes its memory cycle. A mainframe cycle is then attempted and a flag is set if there was sufficient time to complete the mainframe memory read. (Only mainframe read cycles are allowed while the emulator is accessing the emulation memory, since write cycles may not be interrupted.) This feature lets the user dynamically watch the memory while the emulation processor is running, provided that sufficient dead time is available.

The memory controller provides mapping of the target processor's address space into 64 blocks of equal size. This is accomplished by placing a mapper RAM in series with the six highest-order address lines from the emulator. Each block can contain from 256 bytes to 32,768 bytes depending on the address bus size and whether the data bus is 8 or 16 bits wide for the processor being emulated. The mapping feature allows the available memory (as little as 8K bytes) to be placed anywhere in the emulated processor's address space. For an 8-bit processor, such as the 8080, each available block of memory can be placed anywhere from 0 to 64K in 1K increments. The mapper also provides status bits for each block of memory. The status bits tell the emulator whether that block of memory is RAM, ROM or undefined.

The memory controller sends a break to the emulator if an illegal memory operation is performed, such as a write to ROM.

Emulator Software

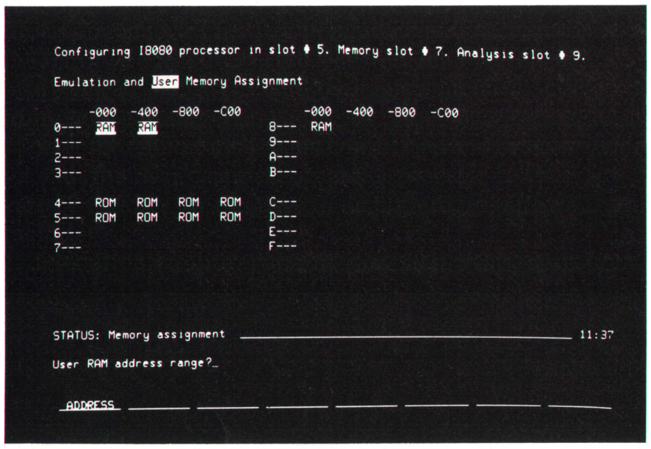
The purpose of the emulator software is to provide a friendly interface for the user to verify program code in a hardware configuration that emulates the end product, a microprocessor-based system. Hardware resources used by the 64000 System emulator software include the processor emulator, the memory emulator (up to 64K bytes), and the logic analyzer unit, which provides 256 states of address, data, status, and count data.

The first task for the user is configuration assignment, that is, specifying the configuration of the hardware (see Fig. 5). This includes

1. Processor clock (internal or external)
2. Illegal opcode detection (enable or disable)
3. Real-time run control (enable or disable)
4. Memory assignment for 64 equal address ranges. Each range can be assigned as emulation memory, user memory, or illegal, and as RAM or ROM.
5. Simulated I/O control addresses for display, printer, keyboard, RS-232-C interface, and disc file(s).

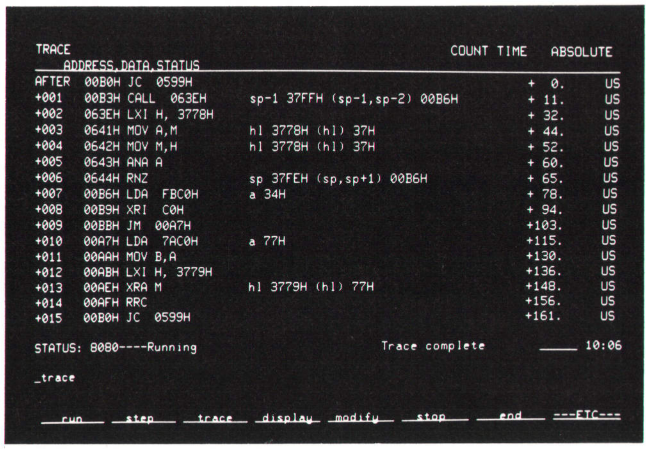
Once the hardware configuration has been set up, the information can be stored in a user-specified file so that repeated emulate sessions can be initialized without repeating the configuration assignment task.

The next user task is loading program code. This is accomplished by specifying the file name of the user program code file. The configuration and/or load-memory file names may be specified when the emulate command is initiated. For example, the following command may be given:



```
Configuring I8080 processor in slot # 5. Memory slot # 7. Analysis slot # 9.

Emulation and User Memory Assignment

     -000  -400  -800  -C00          -000  -400  -800  -C00
0---  RAM   RAM                 8---  RAM
1---                            9---
2---                            A---
3---                            B---

4---  ROM   ROM   ROM   ROM     C---
5---  ROM   ROM   ROM   ROM     D---
6---                            E---
7---                            F---



STATUS: Memory assignment  ________________________________________ 11:37

User RAM address range?_
```

Fig. 5. *To use the emulator, the user must first specify the hardware and memory configuration.*



```
TRACE                                                   COUNT TIME   ABSOLUTE
    ADDRESS,DATA,STATUS
AFTER  00B0H JC  0599H                                        +  0.     US
+001   00B3H CALL  063EH     sp-1 37FFH (sp-1,sp-2) 00B6H     + 11.     US
+002   063EH LXI H, 3778H                                     + 32.     US
+003   0641H MOV A,M         hl 3778H (hl) 37H                + 44.     US
+004   0642H MOV M,H         hl 3778H (hl) 37H                + 52.     US
+005   0643H ANA A                                            + 60.     US
+006   0644H RNZ             sp 37FEH (sp,sp+1) 00B6H         + 65.     US
+007   00B6H LDA  FBC0H      a 34H                            + 78.     US
+008   00B9H XRI  C0H                                         + 94.     US
+009   00BBH JM  00A7H                                        +103.     US
+010   00A7H LDA  7AC0H      a 77H                            +115.     US
+011   00AAH MOV B,A                                          +130.     US
+012   00ABH LXI H, 3779H                                     +136.     US
+013   00AEH XRA M           hl 3779H (hl) 77H                +148.     US
+014   00AFH RRC                                              +156.     US
+015   00B0H JC  0599H                                        +161.     US

STATUS: 8080----Running                    Trace complete     _____ 10:06

_trace
```

Fig. 6. *A typical trace display showing program flow in mnemonic form.*

emulate CONFIG load memory PROGNAME

This command brings in the emulate software, initializes the hardware resources (processor, memory, etc.) as previously stored in CONFIG, and then loads memory with code from PROGNAME.

After the emulator has been configured and program code loaded, the user can start an emulate session. There are a variety of ways for the user to debug program flow. These include:

1. Execution control, such as run, step, stop, trace commands
2. Display options, such as registers, memory, trace
3. Modify options, such as registers or memory
4. Simulated I/O control.

Execution Control

Upon entry to the emulate module, the status of the processor emulator is "ready" and the module is waiting for the next command. Commands that may be used include run, step and stop. These commands have the following syntax:

run [from address] [until term]	run processor at current program counter or specified address. A stop term may be specified.
step [number instructions]	step processor one instruction or specified number of instructions
stop processor	stop processor

The processor may be stopped by an illegal opcode (if enabled), an illegal memory reference, completion of the analysis, or a user command.

Real-Time Trace Command

The trace command allows the user to view program flow. The command is simply:

trace

The resultant trace display shows program flow in menomic form and may look as shown in Fig. 6.

When the program code is loaded, the symbol file is also provided for user convenience. This means that any expression may contain symbolic references. For example, the following trace specification may be given:

trace after SYMBOL

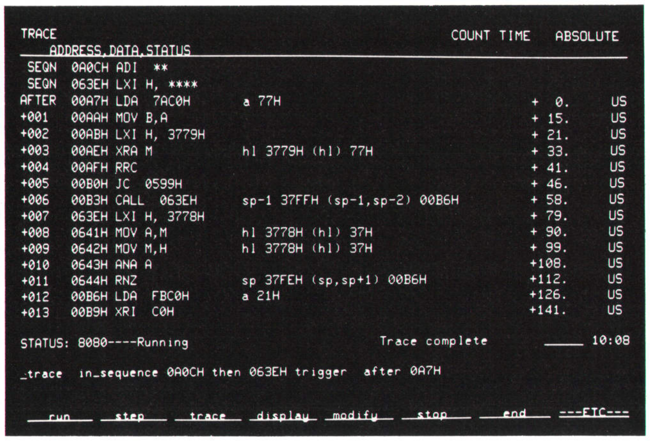
The user may also make the following type of trace specification:

trace after register c = 3

This causes the system to single-cycle the emulator processor and perform the specified trace. The emulator software tries to do the specified task in real time, but if the user makes a specification beyond the real-time analysis capabilities of the system, then the emulator processor is cycled to perform the specification. The trace command can be a complex specification. For example consider the following trace commands:

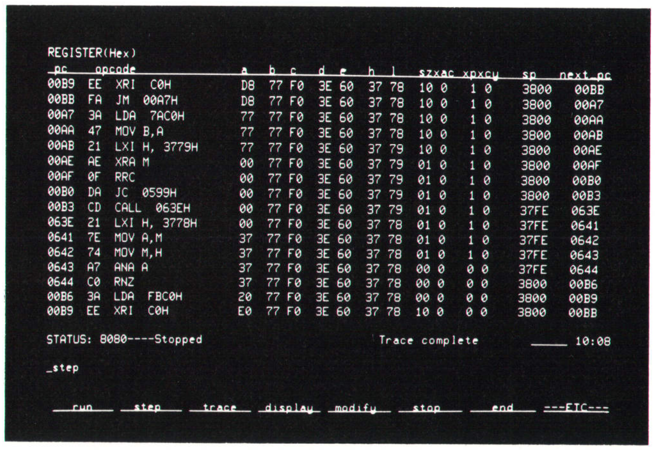
trace in sequence 0A0CH then 063EH
trigger after 00A7H

This specification can be accomplished in a pseudo-run



```
TRACE                                                   COUNT TIME   ABSOLUTE
    ADDRESS,DATA,STATUS
 SEQN  0A0CH ADI  **
 SEQN  063EH LXI H, ****
AFTER  00A7H LDA  7AC0H      a 77H                            +  0.     US
+001   00AAH MOV B,A                                          + 15.     US
+002   00ABH LXI H, 3779H                                     + 21.     US
+003   00AEH XRA M           hl 3779H (hl) 77H                + 33.     US
+004   00AFH RRC                                              + 41.     US
+005   00B0H JC  0599H                                        + 46.     US
+006   00B3H CALL  063EH     sp-1 37FFH (sp-1,sp-2) 00B6H     + 58.     US
+007   063EH LXI H, 3778H                                     + 79.     US
+008   0641H MOV A,M         hl 3778H (hl) 37H                + 90.     US
+009   0642H MOV M,H         hl 3778H (hl) 37H                + 99.     US
+010   0643H ANA A                                            +108.     US
+011   0644H RNZ             sp 37FEH (sp,sp+1) 00B6H         +112.     US
+012   00B6H LDA  FBC0H      a 21H                            +126.     US
+013   00B9H XRI  C0H                                         +141.     US

STATUS: 8080----Running                    Trace complete     _____ 10:08

_trace  in_sequence 0A0CH then 063EH trigger  after 0A7H
```

Fig. 7. *A trace display for a complex trace specification.*



```
REGISTER(Hex)
 pc    opcode                   a   b  c    d  e   h  l   szxac xpxcy   sp    next pc
00B9  EE  XRI  C0H              D8  77 F0  3E 60  37 78  10 0    1 0    3800   00BB
00BB  FA  JM  00A7H             D8  77 F0  3E 60  37 78  10 0    1 0    3800   00A7
00A7  3A  LDA  7AC0H            77  77 F0  3E 60  37 78  10 0    1 0    3800   00AA
00AA  47  MOV B,A               77  77 F0  3E 60  37 78  10 0    1 0    3800   00AB
00AB  21  LXI H, 3779H          77  77 F0  3E 60  37 79  10 0    1 0    3800   00AE
00AE  AE  XRA M                 00  77 F0  3E 60  37 79  01 0    1 0    3800   00AF
00AF  0F  RRC                   00  77 F0  3E 60  37 79  01 0    1 0    3800   00B0
00B0  DA  JC  0599H             00  77 F0  3E 60  37 79  01 0    1 0    3800   00B3
00B3  CD  CALL  063EH           00  77 F0  3E 60  37 79  01 0    1 0    37FE   063E
063E  21  LXI H, 3778H          00  77 F0  3E 60  37 78  01 0    1 0    37FE   0641
0641  7E  MOV A,M               37  77 F0  3E 60  37 78  01 0    1 0    37FE   0642
0642  74  MOV M,H               37  77 F0  3E 60  37 78  01 0    1 0    37FE   0643
0643  A7  ANA A                 37  77 F0  3E 60  37 78  00 0    0 0    37FE   0644
0644  C0  RNZ                   37  77 F0  3E 60  37 78  00 0    0 0    3800   00B6
00B6  3A  LDA  FBC0H            20  77 F0  3E 60  37 78  00 0    0 0    3800   00B9
00B9  EE  XRI  C0H              E0  77 F0  3E 60  37 78  10 0    0 0    3800   00BB

STATUS: 8080----Stopped                               Trace complete        _____ 10:08

_step
```

Fig. 8. *A display of the emulator processor registers.*

mode, that is, the processor can run in real time to 0A0CH and stop, then run in real time to 063EH, and so on. The displayed trace might be as shown in Fig. 7.

Display Options

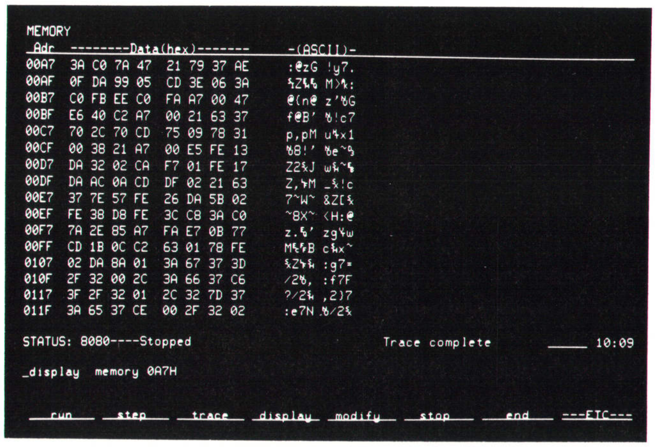
The display options include registers, memory, and trace. An example of a display of the processor registers is as shown in Fig. 8.

Memory displays can be of any assigned memory. Modes of display include absolute, mnemonic, offset, and dynamic. The absolute mode displays memory in hexadecimal and ASCII, as shown in Fig. 9. The mnemonic mode displays memory as opcodes, mnemonically, as shown in Fig. 10.

In the offset mode, displayed addresses are offset by a specified value. The dynamic mode displays memory using a sampled mode (not real time).

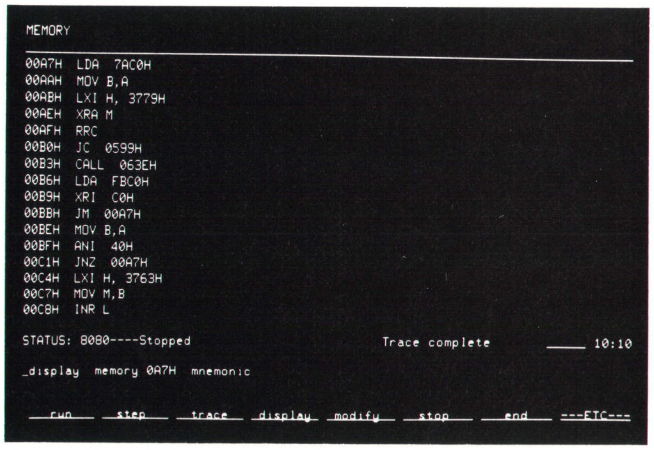
Trace displays show the results of analysis data. Modes of display include:

1. Mnemonic, to display opcodes mnemonically
2. Absolute, to display all data in hexadecimal
3. Packed, to group data by opcode
4. Unpacked, to display all data without grouping
5. Address offset, to display addresses offset by a specified value. This feature allows the user to view program code with addresses as they are on the assembler listing.



```
MEMORY
 Adr  --------Data(hex)-------       -(ASCII)-
00A7  3A C0 7A 47  21 79 37 AE       :@zG !y7.
00AF  0F DA 99 05  CD 3E 06 3A       ␏Z␙␅ M>␆:
00B7  C0 FB EE C0  FA A7 00 47       @{n@ z'␀G
00BF  E6 40 C2 A7  00 21 63 37       f@B' ␀!c7
00C7  70 2C 70 CD  75 09 78 31       p,pM u␉x1
00CF  00 38 21 A7  00 E5 FE 13       ␀8!' ␀e~␓
00D7  DA 32 02 CA  F7 01 FE 17       Z2␂J w␁~␗
00DF  DA AC 0A CD  DF 02 21 63       Z,␊M _␂!c
00E7  37 7E 57 FE  26 DA 5B 02       7~W~ &Z[␂
00EF  FE 38 D8 FE  3C C8 3A C0       ~8X~ <H:@
00F7  7A 2E 85 A7  FA E7 0B 77       z.␅' zg␋w
00FF  CD 1B 0C C2  63 01 78 FE       M␛␌B c␁x~
0107  02 DA 8A 01  3A 67 37 3D       ␂Z␊␁ :g7=
010F  2F 32 00 2C  3A 66 37 C6       /2␀, :f7F
0117  3F 2F 32 01  2C 32 7D 37       ?/2␁ ,2}7
011F  3A 65 37 CE  00 2F 32 02       :e7N ␀/2␂

STATUS: 8080----Stopped                         Trace complete      _____ 10:09

_display  memory 0A7H
```

Fig. 9. *An absolute-mode memory display, showing memory in hexadecimal and ASCII.*



```
MEMORY

00A7H  LDA  7AC0H
00AAH  MOV B,A
00ABH  LXI H, 3779H
00AEH  XRA M
00AFH  RRC
00B0H  JC  0599H
00B3H  CALL  063EH
00B6H  LDA  FBC0H
00B9H  XRI  C0H
00BBH  JM  00A7H
00BEH  MOV B,A
00BFH  ANI  40H
00C1H  JNZ  00A7H
00C4H  LXI H, 3763H
00C7H  MOV M,B
00C8H  INR L

STATUS: 8080----Stopped                         Trace complete      _____ 10:10

_display  memory 0A7H  mnemonic
```

Fig. 10. *A mnemonic-mode memory display, showing memory as opcodes.*

Modify Options

The modify commands include:

1. modify register, to modify any specified register
2. modify memory, to modify any specified memory to a specified value.

Simulated I/O

Simulated I/O control allows the user to use 64000 input/output facilities until the real I/O system can be interfaced to the processor. Since this is done in a sampled mode, not in real time, it is called simulated I/O. The general procedure is to give the control address for the I/O device desired, followed by a status byte specifying the type of request. Any additional parameters are placed after the control address.

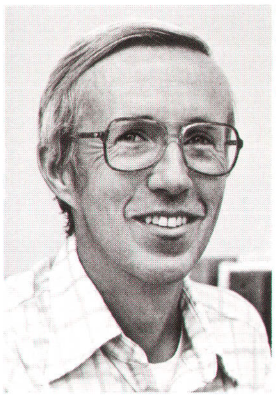
The standard I/O devices are display, printer, RS-232-C interface, keyboard, and disc files. Display requests are open, close, roll lines 1-18 up and write to line 18, set row (1-18) and column (1-80), and write to row/column. Printer requests are open, close, and write line. RS-232-C requests are set controls/modes, read status, read/write single byte, and read/write buffer data. Keyboard requests are open, close, set mode, read line, and read special keystrokes. Disc file requests are create (up to 6 files), open, close, position to record, read/write record, and change file name.

Conclusion

The 64000 emulation system, with wholly separate host and emulation processor architecture, buffered pod for isolation and protection from the user system, the background memory concept, and a novel method of host and emulation system interaction, provides a new level of transparency to the user system and offers unrestricted use of the full address space, interrupt systems, and all other functions of the microprocessor being emulated. This, coupled with flexible memory mapping, real-time analysis unit and an integrated software support package, provides a powerful emulation tool in a new microprocessor development system.

James B. Donnelly

Jim Donnelly came to HP in 1967 with six years experience as an electronic design engineer. He's contributed to the design of several oscilloscopes and the 1610A Logic Analyzer, and designed the 8080, 8085, and 6800 emulators for the 64000 System. He's now a group leader in the 64000 lab. Jim was born in Pueblo, Colorado. He served in the U.S. Air Force from 1953 to 1957, then attended Colorado State University, graduating in 1961 with a BSEE degree. He's a member of IEEE and a resident of Colorado Springs, Colorado. He's married, has a daughter, and enjoys bicycling and running.



Gordon A. Greenley

Gordon Greenley came to HP in 1964 after receiving his MSEE degree from the University of Colorado. He's done circuit design for sampling oscilloscopes, software development for the 1610A Logic Analyzer, and software development for the 64000 System. A native of Moline, Illinois, he received his BSEE degree in 1957 from the University of Colorado and worked several years as a service engineer before coming to HP. Gordon is a photographer who does all of his own developing and printing, both color and black-and-white. He also enjoys swimming and camping. He's married, has two daughters, and lives in Colorado Springs.

Milo E. Muterspaugh

Milo Muterspaugh is a native of Cleveland, Ohio. He received the BSEE and MSEE degrees from the University of Arizona in 1966 and 1968, joined HP in 1968, and designed the emulation memory and tape cassette for the 64000 system. Milo spent four years in the U.S. Air Force from 1959 to 1963. He and his wife live in Colorado Springs, Colorado, and his interests include tennis, bicycling and amateur sports car racing.

Acknowledgments

Thanks go to Tom Saponas and Roger Cox, who contributed their ideas in the early definition phase. Paul Sherwood was responsible for the electrical design of the Z-80 emulator. Kip Stewart and Brad Yackle wrote the software packages for the 6800 and Z-80 emulators, respectively. Mechanical design of the emulator pods was done by Carl Glitzke, Bobby Self and Bill Fisher. Special thanks go to the production engineers and technicians and all the support people without whom this project would not have been possible.

The Pascal/64000 Compiler

by Izagma I. Alonso-Velez and Jacques Gregori Bourque

PASCAL IS A STRUCTURED computer programming language rich in control and data structures that make programming natural, that is, the Pascal strucstures are close to the way one would express the same concepts in English. The block structure of Pascal encourages the programmer to write modular and well-structured programs, and features such as type checking force the programmer to understand the program logic in detail before and during program development. The fact that the program is well structured and written in a way that is natural to the programmer makes understanding of the program easier, both at the time it is being developed (for debugging purposes) and later when it needs to be changed (for maintenance purposes). In summary, Pascal makes program development easier and more enjoyable all the way from the moment of conceptualization, through writing and debugging the program, to maintaining it at a later time.

Pascal/64000

A compiler is a program that translates a high-level computer programming language into low-level machine language instructions. Effectively, the compiler simulates a high-level language machine.

The Pascal/64000 compiler is designed to translate programs written in Pascal into code for microprocessors. It is implemented as a subset of the language definition given by Jensen and Wirth,[1] but several options and extensions have been added to the language to make it more appropriate for microprocessor programming.

Extensions include type-changing capabilities, an OTHERWISE clause for the CASE statement, the BYTE standard type (for microprocessors with byte addressing capabilities), some standard procedures such as SHIFT and SHIFTC for manipulating data and ADDR for getting at the address of a variable, separate compilation of modules (in standard Pascal the whole program has to be compiled in a single module), constant expressions, and HEX, OCTAL and BINARY bases.

One of the options available in the compiler allows the user to declare variables and procedures as GLOBAL or EXTERNAL for separate compilation. This also permits the use of routines not written in Pascal. These routines can be declared as EXTERNAL in the Pascal program and, as long as the parameter passing is compatible with the Pascal calling sequence, they can then be called and used from the Pascal source program. The Pascal compiler subroutine calling sequence is fully documented to allow the programmer to use non-Pascal routines.

Other important options include the capability of separating data from program code (for example, data can be allocated to RAM and program code to ROM) and the accessing of absolute addresses (can be used to implement memory mapped I/O).

The following is a list of compiler options and a short description of each. It is important to note that the programmer who prefers standard Pascal can ignore all the options and extensions and write portable standard Pascal programs.

$ANSI ON$, $ANSI OFF$
ON causes a warning message to be issued for any feature of Pascal/64000 that is not part of standard Pascal. Default: OFF.

ASM_FILE
This option causes the compiler to create a source file containing the equivalent assembler source information of the program being compiled. This source file (named ASM8085) is acceptable to the assembler for the 8085 microprocessor. If the LIST_CODE option is ON the ASM8085 file also contains intermixed Pascal source lines as assembler comments. Default: OFF.

$DEBUG ON$, $DEBUG OFF$
ON causes all arithmetic operations with bytes and integers to call external library routines, which insure that no overflow, underflow, or divide-by-zero operations occur. Default: OFF.

$EMIT_CODE ON$, $EMIT_CODE OFF$
ON specifies that executable code is to be emitted to the relocatable code file. Default: ON.

END_ORG
Used after the ORG option to return the variable allocation to the previous mode.

$EXTENSIONS ON$, EXTENSIONS OFF$
ON allows the programmer to use the microprocessor-oriented extensions to the Pascal language. OFF disallows the use of these language extensions. The extensions include functional type changing, the address function, the BYTE data type, built-in functions, SHIFT and SHIFTC, and nondecimal constant representations. EXTENSIONS ON turns RECURSIVE OFF and vice versa. Default: OFF.

$EXTVAR ON$, $EXTVAR OFF$
ON causes all variables defined until the subsequent EXTVAR OFF is encountered to be declared EXTERNAL. No local storage is allocated in this module for such variables. Default: OFF.

$GLOBPROC ON$, $GLOBPROC OFF$
ON causes all main-block procedures defined until the subsequent GLOBPROC OFF is encountered to be declared GLOBAL so they may be accessed by other modules. Default: OFF.

$GLOBVAR ON$, $GLOBVAR OFF$
ON causes all main-block variables defined until the subsequent GLOBVAR OFF is encountered to be declared GLOBAL so they may be accessed by other modules. Default: OFF.

$LIST ON$, $LIST OFF$
ON causes the source file to be copied to the list file. OFF suppresses the listing except for lines that contain errors. Default: ON.

$LIST_CODE ON$, $LIST_CODE OFF$
ON specifies that the program list file will contain the symbolic form (assembly language) of the code produced intermixed with the source lines. Default: OFF.

$OPTIMIZE ON$, $OPTIMIZE OFF$
ON causes certain run time checks to be ignored, such as prechecking the range values of a CASE statement. This mode will typically produce somewhat smaller and faster modules that are susceptible to bad (out of range) data at run time. This option should only be used for well-structured programs that have been thoroughly debugged. Default: OFF.

$ORG number$
All variables declared until END_ORG is encountered will be allocated sequential absolute addresses starting from the number specified.

$PAGE$
Causes a form feed to be output to the listing file. Default: NULL.

$RECURSIVE ON$, $RECURSIVE OFF$
ON causes all procedures declared until the subsequent RECURSIVE OFF is encountered to be compiled to allow recursive or reentrant calling sequences. OFF causes procedures to be compiled in a static mode which does not allow for recursive or reentrant calling sequences. Default: ON.

$SEPARATE ON$, $SEPARATE OFF$
ON enables the separation of program, constants, and data, such that program code and constants are put in the PROG relocatable area and data is put in the DATA relocatable area. OFF puts all program code, constants, and data into the PROG relocatable area. Default: OFF.

$TITLE "string"$
The first 50 characters of the string are moved into the header line printed at the top of each subsequent page. Default: NULL.

Program Debugging with Pascal/64000

by P. Alan McDonley

High-level languages allow a programmer to create algorithms logically without concern for processor-dependent steps. During the debug phase of program development using the target machine emulator, a programmer must trace the program in machine code, a language different from the source code language, such as Pascal, that was used to design the algorithm.

Pascal/64000 generates relocatable symbolic information during the code generation pass (pass 2) to help the user debug programs. In particular, the user can request an expanded listing (see Fig. 1). This listing contains the assembly language source statements corresponding to the machine code placed in the relocatable file, intermixed with the original Pascal source lines. All of the symbols and labels used in the compiler-generated assembly language source lines are available during emulation to ease the user's translation from the original Pascal to the machine code seen when tracing execution.

In Fig. 1, the leftmost number is the source line number. Next is a relocatable offset, and next a level number. Below each line are the relocatable offset, opcode and mnemonic equivalent of the code put in the relocatable file.

```
 1 0000   1    "8085"
 2 0000   1    PROGRAM DRIVER;
                 0000                 NAME   "DRIVER Pascal"

 3 0000   1    VAR
 4 0000   1    $ORG 3400H$
 5 0000   1      DISPLAY:ARRAY[1..80] OF BYTE;
 6 0000   1    $END_ORG$
 7 0000   1    $EXTVAR ON$
 8 0000   1      ANSWER:BYTE;
                 0000                 EXT    ANSWER

 9 0000   1    $EXTVAR OFF$
10 0000   1      DISPLAY_INDEX:BYTE;
11 0001   1    $GLOBPROC$
12 0001   1    PROCEDURE DISPLAY_ANSWER;

13 0000   2      BEGIN
                 0000             DISPLAY_ANSWER:

14 0000   2        DISPLAY[DISPLAY_INDEX]:=ANSWER;
                 0000  3A  ????       LDA    DRIVER_D
                 0003  CD  ????       CALL   Zbtoint
                 0006  11  0100       LXI    D,1
                 0009  EB             XCHG
                 000A  CD  ????       CALL   Zintsub
                 000D  11  0034       LXI    D,3400H
                 0010  19             DAD    D
                 0011  3A  ????       LDA    ANSWER
                 0014  77             MOV    M,A

15 0015   2        IF DISPLAY_INDEX<80 THEN DISPLAY_INDEX:=DISPLAY_INDEX+1
                 0015  3A  ????       LDA    DRIVER_D
                 0018  2E  50         MVI    L,80
                 001A  CD  ????       CALL   Zbyteles
                 001D  CA  ????       JZ     DISPLAY_ANSW_L1

16 0020   2        ELSE DISPLAY_INDEX:=1;
                 0020  3A  ????       LDA    DRIVER_D
                 0023  C6  01         ADI    1
                 0025  32  ????       STA    DRIVER_D
                 0028  C3  ????       JMP    DISPLAY_ANSW_L2
                 002B             DISPLAY_ANSW_L1:
                 002B  21  ????       LXI    H,DRIVER_D
                 002E  36  01         MVI    M,1
                 0030             DISPLAY_ANSW_L2:

17 0030   2      END;
                 0030  C9             RET
                 0031             DISPLAY_ANSWE_C:
                 0031             DISPLAY_ANSWE_E:
                 0031             DISPLAY_ANSWE_D:

18 0000   1
19 0000   1
                 0031             DRIVER:
                 0031             DRIVER_C:
                 0031             DRIVER_E:
                 0031             DRIVER_D:
                 0031                 DS     1
                 0032                 GLB    DISPLAY_ANSWER
                 0032                 GLB    DRIVER
                 0032                 EXT    Zbyteles
                 0032                 EXT    Zbtoint
                 0032                 EXT    Zintsub
                 0032                 END

End of compilation, number of errors=      0
```

Fig. 1. *Expanded listing of relocatable code produced by the Pascal/64000 compiler contains assembly language statements intermixed with the original Pascal source lines. This makes program debugging easier.*

The user interacts with the emulator using statements such as:

run from DISPLAY_ANSWER until LINE_17

or

display memory ANSWER

where DISPLAY_ANSWER is the name of a global procedure in the listing above, LINE_17 is a local symbol that the compiler generated for line 17 of the source, and ANSWER is the name of a global variable. Using this listing, the programmer can modify variables and execute segments of a procedure or program separately, so that each part may be proved correct and the interactions more closely followed.

Global and external variables may be accessed by name during emulation. Local variables are renamed by the compiler and may be inspected and modified using the new name found in the expanded listing. In the listing above DRIVER_D is the local name of DISPLAY_INDEX. To use specific variables for debugging purposes, the user may declare them to be GLOBAL. This option causes the symbol name (up to 15 characters) to be sent to the linker as a global symbol in the relocatable file.

Traditionally, when errors are detected during execution, intermediate results are printed at run time and errors are narrowed to a few lines of source code, which can then be proved incorrect by hand execution. Much time can be spent with this type of program development.

Run time library routines may have features to aid the user in debugging programs or may be designed for final product use, where errors are not expected. A DIVISION BY 0 error message would mean little to the grocery store clerk attempting to weigh tomatoes.

The Pascal/64000 debug library provides the user with range checking for arithmetic operations, protection against misuse of dynamic memory space, and detection of some other types of nonfatal errors. When an error occurs, program execution is suspended to allow the input parameters and program flow at the error to be examined. By listing local symbols in a file called Derrors, the value of each register and the address of the calling routine are displayed. Fig. 2 shows a sample listing of the local symbols in Derrors. When an error is detected, the program counter address at which program execution stops is displayed. Matching this address with the upper addresses in the middle column of the Derrors listing reveals the type of error that caused execution to stop. The lower entries in the rightmost column of the listing show the values of the registers passed to the

```
Z_END_PROGRAM         0EBFH        C3H
Z_ERR_CASE            0EA7H        08H
Z_ERR_DIV_BY_0        0E9CH        08H
Z_ERR_FUTURE          0EE6H        BDH
Z_ERR_HEAP            0EB7H        08H
Z_ERR_OVERFLOW        0E94H        08H
Z_ERR_SET             0EA0H        08H
Z_ERR_UNDERFLOW       0E98H        08H
Z_PSW_FLAGS           0EE8H        7FH
Z_REG_A               0EE9H        6EH
Z_REG_B               0EEBH        A4H
Z_REG_C               0EEAH        A1H
Z_REG_D               0EEDH        46H
Z_REG_E               0EECH        DDH
Z_REG_H               0EEFH        B7H
Z_REG_L               0EEEH        20H
Z_ZCALLER_H           0EE7H        D3H
Z_ZCALLER_L           0EE6H        BDH
```

Fig. 2. *A typical listing of local symbols, program counter addresses, and register contents at the point where an error is detected. Knowing the address at which program execution stops, the user can determine the type of error from this listing.*

routine that detected the error.

By viewing the stack, the current state of each recursion into procedures and functions can also be determined. In all, with the aid of the 64000 emulators and Pascal, the productivity of microprocessor software designers is raised substantially. Pascal/64000 has been designed to support the user without knowing the user's configuration, providing the tools needed to code efficiently for microprocessors in a high-level language.

P. Alan McDonley

Al McDonley received his BSME degree from New Mexico State University in 1976. He joined HP the same year as a product designer for the 64000 System, and he's now a software designer, working on run time libraries for the Pascal/64000 compiler. He's taught courses in computer programming in APL, BASIC, FORTRAN, and Pascal. Born in Oakland, California, Al is single and lives in Colorado Springs, Colorado. His interests include sailing, archery, photography, pipe organs, country dancing, stereo systems, electric vehicles, electronics, and computers.

$WARN OFF$, $WARN OFF$

ON specifies that the warning messages will be displayed and written to the listing file. OFF specifies that only error messages will be displayed and listed. Default: ON.

$WIDTH number$

The number determines the number of significant characters in the source line. Additional characters are ignored. Default: 120.

In accordance with the 64000 design philosophy, the Pascal compiler is designed to be easy to use and have capabilities that, combined with emulation, provide powerful debugging tools. Any global procedure or variable can be addressed by name from emulation, and program statements can be accessed by their Pascal program source line numbers.

The compiler is evoked by pressing the softkey labeled compile. The softkeys then guide the user to the available options. The first line of the source program is a special compiler directive that indicates to the compiler which microprocessor it is to compile for. The microprocessor name appears embedded in quotes: "8085", "Z80", and so on. During compilation the status line of the 64100A displays the compiler status at each point.

Implementation

Pascal/64000 is implemented in two passes (Fig. 1). The first pass reads the Pascal source program and checks for errors. If no errors are found the compiler generates data for the second pass or code generator. This data consists of an intermediate language (IL), which contains the information from the source program that the second pass needs to generate code for the given microprocessor. The code generator then reads the IL and from it produces the relocatable code to perform the operations described by the programmer in the original Pascal source program.

If errors are found during the first pass, the compiler writes the errors to the display. At the end of compilation the display also makes available to the programmer a summary of the meaning of each error found in the program. If a list file has been indicated, the compiler includes information about errors in the list file as well. Errors are listed even if the NOLIST option is on. In the event of errors the compiler does not generate relocatable code; the code generator is not evoked and only the listing second pass is executed.

Intermediate Languages

Intermediate languages have been implemented as zero-address, one-address, two-address, and three-address forms. Only the three-address form can explicitly describe each of the source and result operands of a binary operation. Each of the other methods has some implicitly specified operands.

The zero-address form uses a data stack, where all source and result operands are implicitly found. Loads and stores are equivalent to stack push and pop operations. Binary operations assume that both source operands are on the stack before the instruction. They are popped after the operation and the result is pushed onto the stack. This form of IL is generally well suited to top-down or recursive-descent compilers, since it allows for the generation of an IL for a particular language construct at the first possible moment after semantic recognition. It is the IL used in the popular P-code versions of the portable Pascal compiler.

The one-address form uses a single implicit register as part of each IL instruction. All operations may operate on this single register or on this register and memory.

The two-address form uses a fixed number of registers and allows an IL instruction to operate explicitly on a pair of

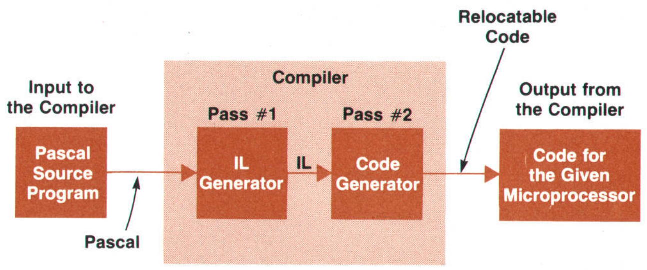


Fig. 1. *Pascal/64000 is a two-pass compiler. The first pass reads the Pascal source program, checks for errors, and produces an intermediate language (IL). The second pass generates code for a specified microprocessor.*

registers or on a register and memory. A pair of operands may be specified for each instruction and the result of an operation goes into one of the specified operands (usually one of the explicit registers).

By allowing each source and result operand to be explicitly described, the three-address form permits the IL description of a program to be more suitable for translation to target processors with any type of stack or register architecture. The other three forms with their implicit result operands are more conveniently translated to target machines with a stack architecture (zero-address IL), single-register architecture (one-address IL), or multiple-register architecture (two-address IL).

Pascal/64000 Intermediate Language

The Pascal/64000 compiler generates relocatable object code for microprocessors from an intermediate language (IL) temporary file created by the compiler during pass 1. This IL file is logically equivalent to the original source program. The code generator module (pass 2) creates the machine-specific object code relocatable file from this IL file.

The Pascal/64000 compiler uses a three-address (or quadruples) IL. The four parts of a quadruple are the instruction or operation, the leftmost source item, the rightmost source item, and the result. For example, the Pascal expression:

```
A: = B-C;
```

would cause generation of the intermediate language quadruple:

```
SUB B,C,A     Subtract C from B, store result in A.
```

For comparison, the equivalent code using a zero-address IL (the P-code portable Pascal compilers use this form) would generate the following IL instructions:

```
LOAD B        Push value of B onto stack
LOAD C        Push value of C onto stack
SUB           Subtract first stack item from second, pop
              both, push result onto stack
STORE A       Pop stack into A.
```

For a one-address IL the following instructions are equivalent:

```
LOAD B        Load accumulator with B
SUB C         Subtract C from accumulator
STORE A       Store accumulator into A.
```

For a two-address IL the following instructions are equivalent:

```
LOAD r,B      Load register r with B
SUB r,C       Subtract C from register
STORE r,A     Store register into A.
```

For this example the number of IL instructions for each form of IL is in the ratio of 4:3:3:1 for zero-address, one-address, two-address and three-address forms, respectively. Some important results for optimization can be inferred from the compactness of quadruple IL representations. It is time-consuming for a code generator to analyze multiple IL instructions to detect patterns for optimization. Since the quadruple form of IL packs more information in a single instruction, it simplifies the effort to generate reasonably efficient object code for a specific target microprocessor.

Each operand of a Pascal/64000 intermediate language quadruple has an explicit operand type, which specifies its addressing mode as a memory location (absolute, relocatable or external) or as an implied address (immediate constant or temporary pseudo-address). The mapping of these operand types to a specific microprocessor instruction set is left to the code generator. Some processors with limited memory accessing modes use a purely static (but relocatable) form for all explicit memory references. For these processors recursion is supported by additional run time routines to permit safe recursive calling sequences. For other processors with more sophisticated memory accessing modes (particularly if register and stack relative addressing is available) data and parameters are allocated to the stack in a more traditional dynamic local memory allocation scheme.

Most optimizations implemented by the Pascal/64000 compiler are local optimizations performed by the pass 2 code generator specific to the target processor. However, some optimization of expression evaluation is done during pass 1. Expressions are built into trees as they are being parsed. The IL generator traverses these trees before generating the IL instructions and attempts to minimize the number of temporary results needed to evaluate the expression. These expression trees are also used to discover constant expressions, which are folded into a single constant before any IL is generated. It is possible to perform some global optimizations during pass 1, and this may allow for a reduction in the size of the IL file.

Code Generation

The intermediate language representation of Pascal/64000 contains all the information needed to create processor-specific code equivalent to the source program. The translation of the intermediate language to relocatable code for a specific target microprocessor is guided by the limitations of the target processor's instruction set.

All programs must eventually fit into a system that has been implemented in a specific hardware configuration, usually with some fixed memory size. Generally, if more memory is required in a specific implementation, it will cost more to design and produce that system. The speed of program execution is generally less important, in the sense that specific program modules that consume a large percentage of program execution time can almost always be reprogrammed to execute faster. With these observations concerning the relative importance of memory use and execution time, code generation patterns have been chosen to minimize memory use rather than execution time where obvious tradeoffs can be made.

Two areas where the memory minimization objective can have a significant impact on the structural form of the code generation patterns are the use of static versus dynamic allocation of memory for parameters and local variables and

The 64000 Linker

by James B. Stewart

The 64000 linker takes relocatable object files generated by the assembler or Pascal compiler and combines them to produce an executable absolute file. The linker resolves symbolic references between relocatable files (linking). It also assigns relocatable code to an absolute location in the target processor's logical address space and changes memory references to refer to the absolute memory locations (relocation). The linker was designed with three major goals: to support a wide variety of microprocessors, to be easy to use, and to provide the user with a complete set of features to facilitate linking relocatable modules for complex microprocessor systems.

The designer of a microprocessor system needs to control the locations of code and data in memory. Before the widespread use of linkers, this was done by coding the entire system in one assembly language program with fixed absolute addresses. A small change in the code required that the entire system be reassembled. Besides being time-consuming, this made it difficult for multiple designers to work concurrently on the same software.

A relocating linker overcomes these problems. Each program segment may be developed and assembled independently. The designer specifies to the assembler that the code is relocatable. At link time, the relocatable code from multiple files is concatenated into one continuous piece of memory.

The 64000 assembler and linker provide the user with several relocatable areas. The assembly language statements ORG, PROG, DATA, and COMN define the relocatability of code. ORG defines code to be absolute or nonrelocatable. PROG and DATA are general-purpose relocation counters that allow the user to partition code to be loaded at different memory locations, for example all program in ROM and all data in RAM. COMN specifies that the data be relocated to the same starting address as the COMN from all other relocatable modules. This is similar to unnamed COMMON in FORTAN. When the relocatable modules are linked, the user provides the starting addresses for the PROG, DATA, and COMN relocatable code. To provide greater flexibility, the user may define several PROG, DATA, and COMN areas. For example the PROG, DATA, and COMN areas for files A and B may start at memory locations 1000H, 2000H, and 3000H respectively, and for files C and D at locations 8000H, E000H, and 3000H.

A load map and a cross-reference table may be generated for each link. The load map (Fig. 1) describes the final memory locations of all relocatable files. The linker also keeps track of memory use and warns the user if any conflicts exist. A "memory overlap" error message is given for any memory that has been allocated more than once.

A feature of the 64000 linker known as no-load allows the user to design overlays into the system. Any subset of the relocatable files may be declared to be no-loaded. This subset is linked and relocated with the files that are not no-loaded. The only difference is that the absolute file generated by the linker contains no code from the no-loaded relocatable files. For example, suppose the user has 6000 bytes of code and data, but only 4000 bytes of physical memory. It may be possible to use overlays to partition the program into pieces that will fit in 4000 bytes. This is done by creating two separate absolute files. The first contains one set of relocatable routines plus the shared routines and data. The second contains the remaining relocatable routines, also linked to the shared routines and data. The shared routines and data would be no-loaded in the case of the second absolute file.

All 64000 emulators allow the user to debug programs using the symbols from the source code. This is particularly useful when dealing with the relocated code, since the user doesn't have to know

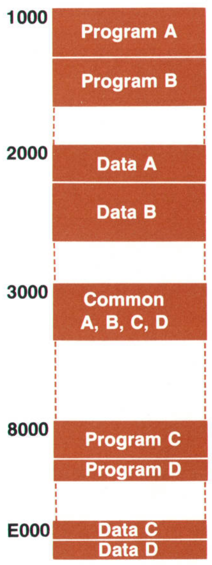


Fig. 1. *A load map may be generated each time the 64000 linker is used. The map shows the final memory locations of all relocatable files.*

where in memory the linker put the code. Any location in memory may be referred to by its symbolic name or its absolute address. To accomplish this, the assembler outputs the entire symbol table for each source program. When the relocatable code is linked, its relocation addresses are saved so they may be used during emulation to find the absolute values of symbols. The linker also generates a symbol file of global symbols. This file has two uses. It is used by the emulator, along with assembler symbol tables, to provide symbolic debugging. It may also be used in subsequent links to preload the linker's symbol table. This feature has uses in overlays and in reducing linking and download time in large systems.

A table-driven architecture allows the linker to support a variety of target processors. Information in each relocatable file defines the intended target processor. Each supported processor corresponds to a system disc file. This file is used by the linker to configure itself for the particular processor.

The configuration files contain two basic types of information: general information such as word width and addressing space, and tables or sequences of instructions for the linker. The different instruction types and addressing modes allowed in the target processor correspond to entry points in the linker table.

Within the assembler-generated relocatable files, each operand address is tagged as either absolute (no relocation), PROG relocatable, DATA relocatable, COMN relocatable, or EXTernal reference. Relocatable and external tags contain a reference to an entry point in the processor-dependent linker table. Knowing the relocatability of the operand, the linker first computes its absolute address, independent of the target processor. It then follows the instructions in the linker table to generate the actual operand. The table allows operations such as shifts, masks and compares, which may be performed on various operands such as the absolute address, the current program counter, or constants. In the 6800 microprocessor, for example, the direct addressing mode requires that an instruction's operand address be in the range $0 \leq \text{address} \leq 255$. The linker table for handling the direct addressing mode performs the following operations:

```
LOADWORD = ABSOLUTE_ADDRESS
TEMP = 0FFH
IF LOADWORD > TEMP THEN "Address out of range"
OUTPUT = LOBYTE (LOADWORD)
PROGRAM_COUNTER = PROGRAM_COUNTER + 1
RETURN
```

The various instruction formats and addressing modes for all supported microprocessors are implemented using similar sequences of simple instructions. The obvious advantages are the speed and ease with which the linker can be configured to support additional processors. Typical linker tables are generated with 20 to 50 lines of processor-specific code.

James B. Stewart

Software designer Kip Stewart came to HP in 1977. He was born in Binghamton, New York and attended the University of Colorado, receiving a BA degree in mathematics in 1976. Currently on leave from HP, he's serving as a visiting instructor in electrical engineering at North Carolina Agricultural and Technical State University. Kip is married, has one child, and makes his permanent home in Colorado Springs, Colorado. Outside of work he spends time with a church youth group and enjoys volleyball, swimming and skiing.

the use of run time library subroutines to perform many relatively simple operations. The 8085 microprocessor, for example, is able to access memory directly as bytes or words with immediate two-byte absolute (relocatable) addresses, and it may access bytes of memory indirectly through register pairs. Dynamic allocation of local data using stack relative addressing must be performed by in-line code or through subroutine calls using the stack offset value as a parameter. A static allocation scheme permits access to local variables or parameters with an arbitrary offset from some (relocatable) label with a direct access instruction which requires only three bytes. This permits access to both byte and word simple variables. Since Pascal programs must access many variables, this reduction of code size by 40 to 50% for each variable access can save a significant amount of memory in a large program. This static allocation of local variables does add additional code and run time overhead for the user requiring recursive calling sequences. These additional memory and time considerations are a reminder to use recursion only where absolutely necessary.

The 8085 instruction set does not support arithmetic for 16-bit signed numbers. IF I, J, and K are type INTEGER, the statement:

K:=I−J−K

generates the following 8085 code, calling library routine Zintsub to perform the subtraction operation:

```
LHLD TEST1_D          put I in register HL
XCHG                  move I to register DE
LHLD TEST1_D+2        put J in HL
CALL Zintsub          subtract J from I
XCHG                  put result in DE
LHLD TEST1_D+4        get K
CALL Zintsub          subtract K from (I-J)
SHLD TEST1_D+4        store the result to K.
```

The 16-bit subtraction routine from the non-debug library is a relatively short program:

```
Zintsub  PUSH PSW  SAVE ACCUMULATOR
         DCX H     TWO'S COMPLEMENT REG HL (Y)
         MOV A,H   COMPLEMENT HIGH BYTE
         CMA
         MOV H,A
         MOV A,L
         CMA       COMPLEMENT LOW BYTE
         MOV L,A
         POP PSW   GET BACK ACCUMULATOR
                   AND FLAGS
         DAD D     X+(-Y) ADD DE AND HL
         RET
```

Using in-line code it would take eight bytes of code to perform the integer subtraction operation each time it is needed. Using the library approach above, it takes eleven bytes for the library routine and only three additional bytes for the subroutine call each time a subtraction is required. After only three integer subtractions the program is already four bytes smaller. For ten subtractions in-line code generation would have added 80 bytes of code to the program, while library calls add only 41 bytes.

This comparison of in-line code versus library subroutines for even simple operations accounts for a significant memory savings when applied to the most commonly used operations that cannot be accomplished in a few bytes of instructions on the target machine.

When the linker creates an absolute file, it tries to find any unsatisfied symbols or routines in a specified library file. It only needs to append run time library routines that have been specifically requested. The actual code size added to an absolute file from the run time library is typically much smaller then the 4K bytes required for loading the entire library.

If a user feels the need for a run time library routine that performs some special operations or is otherwise tailored to the specific application, the user can write another version of any run time library routine using the same name as that used in the library. The new relocatable file is then loaded with the linker in a specific location and the linker will not load the library module of the same name. Thus the run time library serves as a basis for the user's program environment and may be used or improved as the program requirements evolve.

Performance

A certain amount of overhead is expected whenever a high-level language is used. One can hardly claim that it is possible to write all programs in Pascal in such a way that the code generated by the compiler will be as efficient as the code that would have been obtained by direct assembly coding. However, as described above, some optimization has been implemented to generate efficient code: the con-

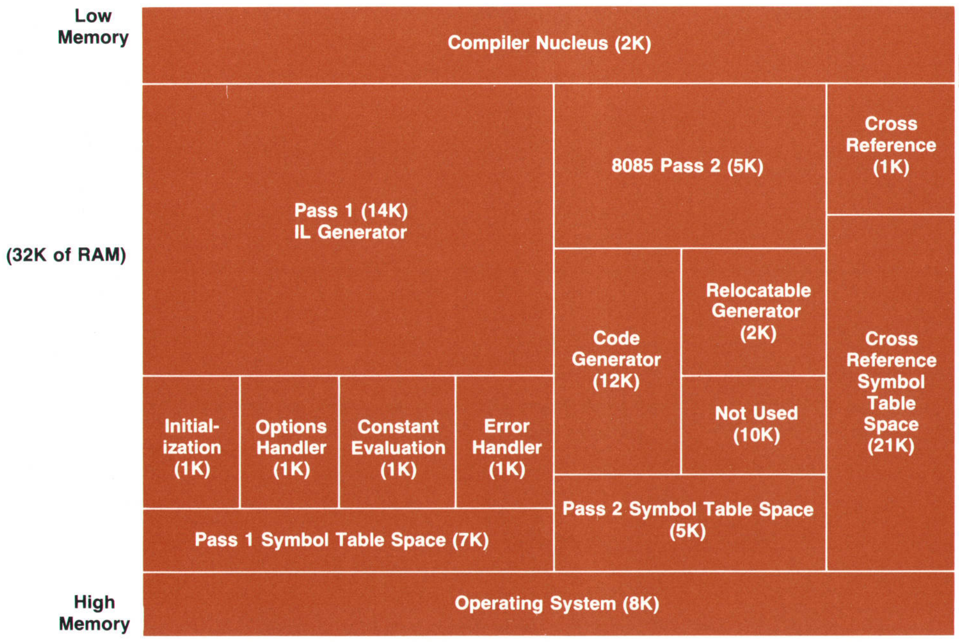


Fig. 2. *The Pascal/64000 compiler uses only 24K words of memory. Parts of the compiler are overlaid by other parts as shown by this diagram. The compiler nucleus is not overlaid. (The 8K operating system memory is not part of the compiler area.)*

tents of registers are remembered over operations, short jumps are implemented for predefined labels that are within range, the overhead for parameter passing is in the receiving routine, and so on. In short, the Pascal/64000 compiler generates good space-efficient code.

The speed of the compiler is 400-600 lines per minute, depending on the way the programmer writes the program and what kind of program is being written. The compiler speed may also vary from microprocessor to microprocessor, since it depends on the level of difficulty and the amount of work required to generate code for the given microprocessor.

By overlaying different parts of the compiler, it was made to fit in 24K words of storage without degrading its performance. A diagram of the compiler overlay structure is given in Fig. 2.

Conclusion

Because of the inherent inefficiencies involved in using a high-level language, users of small computers have in the past written their programs almost totally in assembly language. Pascal/64000 is an alternative. It has all the well-known advantages of a high-level language in addition to space-efficient code generation.

The Pascal/64000 compiler is implemented as a subset of the basic definition of standard Pascal with extensions and options that make it possible for microprocessor programmers to use a high-level language efficiently. The programmer can ignore the extensions and options and write standard Pascal, if desired.

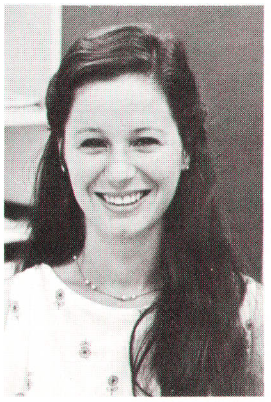
Currently the 8080/8085 and Z80 microprocessors are supported and others will be supported in the future.

Acknowledgments

We wish to thank Martin Smith for major contributions to the development of the Pascal/64000 compiler. He is the principal designer and implementer of the intermediate language and the first pass of the compiler. Early efforts in

Jacques Gregori Bourque

Greg Bourque has been a principal developer of the Pascal/64000 compiler since he joined HP in 1977. A native of Amityville, New York, he received a BS degree in astronomy from California Institute of Technology in 1968 and an MA degree in astronomy from the University of California at Los Angeles in 1970. Before joining HP he worked as a nuclear astrophysicist and as a scientific programmer and computer consultant. He's a member of AAAS and the HP Pascal standard task force, and has lectured in FORTRAN at Colorado State University. Greg is married, has two children, and lives in Colorado Springs, Colorado. He fills his spare time with raising his family, gardening, landscaping, astronomy, and working on his PhD thesis, which he expects to complete next year.



Izagma I. Alonso-Velez

Izagma Alonso-Velez was born in Columbus, Georgia and grew up in San Juan, Puerto Rico. She graduated from the University of Puerto Rico in 1974 with a BS degree in mathematics and worked as a scientific programmer for the next three years. In 1978 she received an MS degree in computer science from Georgia Institute of Technology and joined HP's Colorado Springs Division as a software development engineer. She's been responsible for major portions of the Pascal/64000 compiler. Izagma lives in Colorado Springs, but has traveled all over the world. She's fluent in Spanish and English and has some facility in four other languages. Her interests include current events, crafts (leather, macrame, drawing, sewing), dancing, and playing cards.

the compiler design phase were greatly aided by an HP 3000-based Pascal compiler obtained from Roger Ison of Desktop Computer Division. Co-op students Cheryl Brown and Cheryl Fallander were helpful in testing the compiler and in implementing the run time libraries.

We wish to recognize the HP Pascal Standard Task Force for its efforts in defining a standard Pascal language for applications programming. This language standard will assure the continued use of Pascal on HP products within HP and by its customers.

Reference

1. K. Jensen and N. Wirth, "Pascal User Manual and Report," Springer-Verlag, 1974.

An Assembler for All Microprocessors

by Brad E. Yackle

THE FIRST PRACTICAL PROGRAMMING TOOL offered to the software designer was the assembler. It is a very basic level of programming, since each instruction usually controls a single function of the processor. Then higher-level languages were introduced, allowing programmers to generate software faster and easier, and making code more readable and transportable. However, assemblers will always be part of a computer system, especially a microprocessor system. Assembly-level programming is very close to the machine language of the processor and is therefore good for interacting with hardware and I/O devices. Since assembler code allows complete control of the processor, the assembly language programmer can generate the most efficient code possible. Assembly-level programming is the only practical programming tool for custom or bit-slice processors.

The number of microprocessors on the market and being developed by industry is very large. Each processor has a set of instructions that control its functions. Unfortunately, each processor is different; it has different instructions, registers, speed, memory size, and so on. One assembler cannot possibly be general enough to understand the assembly languages of all processors, so typically a new assembler must be generated for each.

The prospect of generating a new assembler for each processor's assembly language is highly undesirable. First there is the problem of writing the basic assembler to handle the syntax of assembly language programming. The assembler must handle I/O operations as well as parse the operand fields. It must be able to handle expressions, generate object code, and give error messages when necessary. All assemblers have the same basic syntax for instructions. In general, assemblers expect an optional label field followed by an opcode and then some type of operand. However, each assembler must recognize a different set of instructions along with register and/or address-type operands. Therefore, code must be added and/or modified to handle each new processor. Each time this is done, there is a possibility of generating new errors in the common assembler functions. Later, if modifications or changes are necessary, all of the assemblers may have to be modified.

Thus, a new assembler for each new processor language introduces two software problems, arising from the duplication of code. One is the introduction of new errors when translating code from the basic assembler to each new one, and the second is the problem of software update which is multiplied with each duplication of code.

64000 Assembler

The assembler for the 64000 Logic Development System is designed to be flexible enough to understand the instruction set of any processor's assembly language. This means that the 64000 assembler contains some processor-dependent code to handle the variety of instruction sets. However, the problem of software duplication is minimized by making the majority of code processor-independent and putting the dependent code in tables that the assembler reads to understand the instructions. An assembler like this is known as a table-driven assembler. Its main functions are the same for all languages, and it contains additional information in the form of tables to understand processor-dependent instructions.

The common functions of the assembler cover the interaction with the host computer system. This includes reading and parsing the source file. The assembler handles all of the input and output file operations dealing not only with the source file but the relocatable and list files as well. It parses the source lines and identifies the instructions for the particular language. It keeps a symbol table containing symbols along with associated values and symbol types. It checks operand fields and flags errors if syntax and/or address rules are violated. The assembler is designed to be as general as possible to allow for the minor differences in the syntaxes of different processors' assembly languages.

The part of the 64000 assembler that interprets table code to understand each processor's instruction set consists of a set of routines that use standard assembler functions but read the table code to decide which functions to perform. Thus the assembler can be redefined simply by reading different table code.

Assembler Operation

The 64000 assembler reads the first line of the source file and expects to find a key that tells it which type of processor language is in the file. It then reads another file that contains the table code for the language. The table code can be broken into two parts, the opcode set and the set of rules governing the operand field.

Each processor has a set of instructions, which are given names by the designers. These names are commonly called the opcode or mnemonic set of the processor, and are generally abbreviations of the functions performed. For example, let us suppose we have a processor that has an accumulator and an instruction to load data into it. An assembly language statement to do this might look like the following:

```
LDA        DATA
```

where the opcode is LDA, which means load (LD) the accumulator (A) with data found at the address pointed to by the symbol DATA. The opcode set of the processor is composed of all of its opcodes, including a set of standard opcodes that control program listing, external and global symbols, the macro facility, and other functions.

Once an opcode is identified the assembler checks to see whether it is an instruction that requires table code to understand the operand. If so, control is transferred to the special routines that use the table code to control the assembler. The tables instruct the assembler how to parse the operand field, what values to expect, how to generate the object code, and what error messages to generate, if any.

Since a set of tables is the only requirement necessary for the assembler to recognize different languages, we decided to make this capability available to the user. A user can generate an assembler for a custom chip or bit-slice processor, or enhance existing assemblers with custom instructions. To generate a custom assembler the user must describe the syntax of each instruction and how to generate the object code. The 64000 assembler will take care of all system overhead. It will generate relocatable files that can be handled by the system linker and will produce list files like any of the other system assemblers.

Table Processor

The part of the assembler that handles the table code is really a type of simple processor itself. It takes the specially coded table information and decodes it into instructions for the assembler. These instructions call assembler functions, such as expression handlers and object code generators. They also allow for arithmetic operations and testing of the results.

The best way to show how the process works is to give a simple example. Let us suppose that we have a processor that has two instructions that have the same type operand and addressing modes. We will call them LDA and STA, for load accumulator and store accumulator. The object code forms of these instructions are both 8-bit opcodes and require one register as their operand. The value of the register is combined with the eight bits of opcode and resides in the third and fourth bit positions as follows:

```
00rr0000
```

The user will predefine to the assembler the registers that are legal for the instructions, and will give these registers a value and a type. Let us assume that the user makes the obvious choice and defines the registers as type "register."

```
REGISTERS
  A = 00
  B = 01
  C = 10
  D = 11
```

The object code that the assembler is expected to produce is also defined:

```
LDA = 10000000
STA = 11000000
```

The assembler will now recognize these mnemonics on source lines and pass the defined object code to the next set of table instructions for processing. The table instructions process the code as foilows.

```
        EXPRESSION          General-purpose expression
                              parser
        IF TYPE <> REGISTER THEN GOTO OPERAND_
          ERROR
        LOAD VALUE          Get the register number
        SHIFT_LEFT 4        Move to proper position
        OR OBJECT_CODE      Combines with opcode value
        GEN_CODE ABS 8,     Generate the code
          ACCUMULATOR
        DONE                Signal to return to assembler
OPERAND_ERROR
        ERROR IO_ERR        Invalid operand found
        DONE                Return
```

This routine first calls a general-purpose expression handler designed to parse expressions and return a value and a type. Next it checks the type returned to make sure it is one of the predefined registers. If the operand is legal the value of the register is shifted left four bits and combined with the object code passed by the main assembler. Line 6 generates eight bits of absolute data to the relocatable file which is the desired result of the instruction. If an error is found then an error message is generated from the instruction in the ninth line.

Brad E. Yackle

Brad Yackle joined HP in 1977 after receiving the MS degree in computer science from the University of California at Santa Barbara. He also has a BS degree in computer science from California Polytechnic University in San Luis Obispo. Brad designs software at HP and enjoys skiing, swimming and racquetball as outside activities. A native of Abbington, Pennsylvania, he's married and lives in Colorado Springs, Colorado.

Conclusion

In conclusion, the 64000 assembler is a very general table-driven assembler. It is easy to maintain and expand to handle new processors. This increases its reliability, since the majority of its code is processor-independent and well tested. This also aids in software update, since we are not faced with duplication of code. Assembler tables can be changed without affecting the main assembler, and the user has the ability to enhance existing assemblers or generate others for new languages.

Viewpoints

Chuck House on the Electronic Bench

THE ELECTRONICS INDUSTRY is entering the age of VLSI (very large-scale integration). The potential of VLSI is staggering. For example, we'll have extremely powerful 32-bit parallel computers with one-megabyte instruction rates on a single chip for a few hundred dollars within a very few years. We'll go from 16K to 64K to 256K to 1M RAM chips in the same time frame. We'll also be facing some great design challenges because of these silicon advances. The software crisis is already said to be upon us, since the cost of developing correct code for ROM-based designs far outweighs the cost of the silicon for even relatively high-volume products. The 64000 Logic Development System described in this issue was created to address these problems.

The 64000 System and the needs of VLSI portend a dramatic shift in emphasis in the types of tools available for designers. For years, instrumentation has provided analysis capability for use after the initial design was realized. We are now starting to create synthesis tools, which aid the designer in realizing products faster, more accurately, and more productively. This shift from analysis tools to synthesis tools is fundamental to our ability to take advantage of the "macro" power of VLSI. It is conceptually impossible to realize effective designs with millions of gates and millions to billions of coded instructions in software without new automated techniques to replace the "brute force" techniques employed in our industry so far.

A quick example might be the familiar rectangle layouts for emitter, base, and collector of a transistor. They are replicated many times, and relocated in tedious fashion by a designer or draftsman as a function of the desired electrical circuit. True, this process has been automated in recent years, primarily with computer-aided artwork generators that include checking algorithms to assure that the process design rules are followed. This has eliminated some of the drafting and spatial relations tedium, but it has had little impact upon the creative design process. A more useful step might be the macro-cell approach: a series of functional cells is preprocessed in silicon, and a simple design algorithm for interconnecting cells creates the mask set to realize the equivalent custom gate array required.

At a much higher synthesis level, it's conceivable that the mathematical transfer function of the desired IC could be entered into a computer-aided design tool, which would generate the mask sets to create the IC. This is the goal of the California Institute of Technology "Bristleblocs" project, which has both industrial and academic sponsors. The premise of these attempts to work at a macro level is that the view of the forest allows a better perspective for the designer than a consistent and unremitting examination of each tree in the forest, or as some frustrated designers express it, "Chewing on the tree bark incessantly, trying to find the forest."

Adoption of the premise that such high-level design is desirable *and* practical is necessarily rooted in two major assumptions. First, tools must exist that translate the designer's high-level constructs into correct, effective, low-level realizations. These are the synthesis tools mentioned above. Second, analysis tools must be adapted to this environment, which means that they must provide analysis functions at every hierarchical level from high to low, much as a microscope or TV camera has pan or zoom capability.

One additional requirement is imposed by the magnitude of the task, since many projects are designed, produced, and maintained by increasingly large teams of people. Thus, synthesis and analysis tools are increasingly obliged to link to each other simultaneously, across large distances, across cultural and educational barriers, and even across time.

These are stiff requirements, but then so are the challenges facing designers if these requirements are not satisfied. How might they be met? I think that we can see the day, not too distant, when engineers will have an *electronic bench*, much as we discuss *electronic mail* and *electronic offices* and *electronic homes*. Such an *electronic bench* will satisfy the three requirements of synthesis, analysis, and linking.

To illustrate this concept, Fig. 1 portrays a typical product life cycle for a digital product, along with the classical design aids and analysis tools used by most companies today. There are several points worth noting. First, virtually all design aids and analysis tools in use today are not linked in any data base or even measurement-interactive manner. Second, the level of synthesis capability in the design aids is extremely primitive. Third, the level of zoom from high-level analysis to low-level is likewise primitive. Fourth, the operator interface is variable, and quite formidable, from one piece of equipment to another. Examining the needs that VLSI design imposes, these conditions are clearly unacceptable.

There are some current examples of the electronic bench concept at such places as automotive design research centers, airframe manufacturers, and the larger computer and semiconductor design

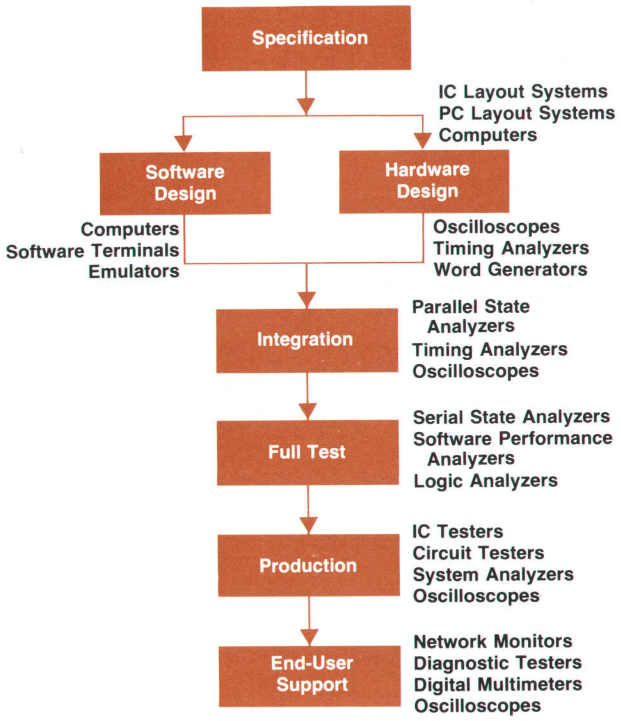


Fig. 1. *Design aids and analysis tools used at various points in the life cycle of a typical digital product.*

centers. These centers, usually built around computer-aided drafting systems, are very expensive, but also very productive and cost-effective. Just as the computer mainframe and minicomputer manufacturers have developed precursors of the type of software development system exemplified by the 64000 System, these CAA and CAD centers point the way toward the electronic bench.

In effect, the solution will embody an intelligent terminal or work station that can provide the capabilities of any required design aid or analysis function. Work performed at this work station will automatically link to a shared data base for the entire program, which includes the R&D functions, production test, service diagnostics, and documentation. Likewise, environmental and life test data will become the beginning of a library of service data that links with lab analysis, production data, and user performance data to promote design improvements and better field-support diagnostic procedures.

It is not hard to postulate such capabilities or their desirability. What has been difficult is a cost-effective and performance-effective realization. There are three major handicaps in this regard when we examine the realities of existing digital analysis tools, to say nothing of the shortage of effective synthesis tools.

1. The user interface of most instruments is very complex, and the commonality of terms, functions, and operations is very low. For example, the specific functions available by name on the front panel of a storage oscilloscope, a logic timing analyzer, and a serial data bus analyzer bear little resemblance to each other. Each front panel takes considerable "getting used to" for a beginning operator, and knowing one of them well can often seem more a handicap than a help when trying the next machine.
2. Today's realizations of this equipment are sophisticated, reasonably expensive, and relatively bulky. The thought of creating an integrated solution has historically been dismissed as not practical in terms of size, heat, weight, and cost.
3. Linking of many measurement hierarchies (the zoom concept) has not been required or practical because of the available instrumentation, and because the problems being tackled could be solved by "brute force" techniques.

The 64000 architectural concept may serve to illustrate how these handicaps might be diminished. The foremost problem, the human interface, is addressed via a standard typewriter keyboard, along with the guided syntax and softkey format. The versatility of screen graphics for menu selections or guided prompting is well established in instrumentation by now. It is a simple extension to provide conversion from one type of equipment to another. The difficulty with such a concept is the reality of its implementation.

Let's consider the manner in which the guided syntax structure operates. The guided syntax softkeys represent another important enhancement of the softkey-with-"help" approach embodied in several of HP's more recent computer systems. Not only do these keys provide prompting of the next correct or allowable entries, but they also allow full flexibility for system reconfiguration as the resident operating system module is swapped from the disc.

Notice the significance of this architecture. The stored program that determines the machine characteristics that appear to the operator is totally resident on disc. Thus, *redefining the instrument is easy*, and the operating system reconfiguration time is about one-third of a second! Moreover, the guided syntax approach removes the need for a different set of keycaps on the front panel, and the user is never faced with relearning the panel functions as the instrument changes.

Thus, the 64000 has a system architecture that links all data files, provides redefinition of effective functions at each work station, and allows easy operator interaction with those significant changes. The major remaining tasks are two-fold: to provide extended operating system enhancements in the guided syntax format, and to provide data acquisition modules for specific functions that may be required.

This flexibility might be employed as an emulating terminal for any computer system, as the following whimsical softkey choices illustrate.

64000S	HP 1000 TERM	HP 3000 TERM	IBM TERM	DEC TERM	APPLE TERM	HP 85A	ETC

When 64000S is pressed the choices would be (the current wakeup mode):

EDIT	COMPILE	ASSEMBLE	LINK	EMULATE	PROM PGM	(CMDFILE)	ETC

When EDIT is pressed, the EDIT module is brought in from the system disc, and these become the key labels:

INSERT	REVISE	DELETE	FIND	REPLACE	<LINE #>	END	ETC

An obvious set of choices under an Analyzer key choice might be:

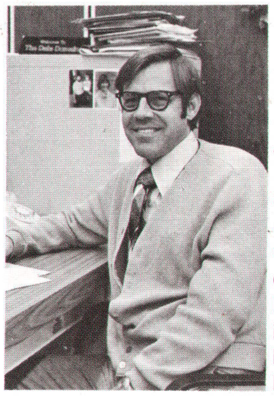
Logic State Analyzer	Logic Timing Analyzer	Serial State Analyzer	Analog O'scope	Digital O'scope	Network Analyzer	Spectrum Analyzer	ETC

The trace point conditions for the state analyzer, the timing analyzer and the scope could be the same, providing the zoom capability mentioned earlier. It becomes practical to consider microprogrammable measurement intelligence, which could modify the degree of zoom or pan according to dynamic decisions about the observed data. Obviously, the data base linkage methods could also admit software control of multiple measurements at multiple stations for simultaneous analysis of major system problems. Perhaps the most productive improvements will come with high-level software analyzers, linked to the greatly improved code generation capabilities described herein. These tools must not only provide code generation, editing, and debug aid, but also validation, verification, optimization, and maintenance functions. The 64000 already provides an important enhancement for these needs. Further extensions are imperative for the effective reduction of the software bottleneck in our industry.

The technology that allows us to consider the true possibility of such a system is based heavily upon the VLSI extensions that the system intends to support. For example, by reducing major equipment such as a sophisticated logic state analyzer to a one or two-card module allows zoom potential, because several different modules can be resident in the card cage of a work station. Also, a cluster network can be composed of different configurations in each work station, and potentially could even include a desktop computer for information graphics or management information systems. A significant problem in terms of computer power—IC cell layout and lead routing, or PC board layouts—could be routed to a major computer network from the cluster as well.

The 64000 described in this issue already takes a significant step in microcomputer software development integration by virtue of its LSI computer support in each work station, guided syntax interaction to allow conversion from one function to another, and four-bus interaction capability, which allows significant data base and measurement networking. The programming effectiveness for designers developing structured code on this system, debugging it in breadboard systems, and moving toward final product is dramatic, and it is a contribution to synthesis, more than to analysis. This shows up most dramatically in larger project teams, where the linked files and the data base management system help to mitigate the classic communication difficulties of large teams. Hardware system synthesis, whether at an IC or PC board level, should be amenable to similar enhancement. The hardest task in my view is the question of effective benchmarking of simulations, which conceptually is possible, but realistically seems relatively difficult to attain.

The next few years should see significant development of tools to enable the electronic bench concept to be realized. This electronic bench will encompass the necessary synthesis, analysis, and linking functions. Clearly the costs of such powerful automated design centers will be dramatically reduced, concurrent with substantially improved combinational performance. With the aid of such instrumentation concepts, we hope to support the design and analysis requirements of the VLSI era.



Charles H. House

Chuck House has been involved with HP logic systems for a decade, first as an R&D project engineer, then as R&D manager, and now as operations manager. He received the fourth annual Award of Achievement from Electronics Magazine for the original HP logic analyzer program. With HP since 1962, and in Colorado Springs since 1964, Chuck is beginning to think he's a Colorado native (his wife and mother are both from Denver). He's been involved in many civic activities—Colorado Air Pollution Control Commission, County Park and Recreation Board, science fair board, engineering advisory committee of the university of Colorado, county arboretum and horticultural group president. Co-author of "Logic Circuits and Microcomputer Systems" (McGraw-Hill 1980), he has contributed to seven books and twenty-five technical papers in the past decade. Chuck has a BS degree in solid-state physics from California Institute of Technology, an MSEE from Stanford University, and an MA in the history of science from the University of Colorado. He's married and has four children.

Hewlett-Packard Company, 1501 Page Mill Road, Palo Alto, California 94304

Bulk Rate
U.S. Postage
Paid
Hewlett-Packard
Company

HEWLETT·PACKARD JOURNAL

OCTOBER 1980 Volume 31 • Number 10

Technical Information from the Laboratories of Hewlett-Packard Company

Hewlett-Packard Company, 1501 Page Mill Road
Palo Alto, California 94304 U.S.A.
Hewlett-Packard Central Mailing Department
Van Heuven Goedhartlaan 121
1180 AM Amstelveen The Netherlands
Yokogawa-Hewlett-Packard Ltd., Suginami-Ku
Tokyo 168 Japan

OCT 17 1980
0200032503&&&HARR&JA00
MR JULIAN A HARRIS
CHAYO ELECTRONICS LTD
P O BOX 2807
PENSACOLA FL 32503