Chapter 31

A Dual-Processor Desk-Top Computer: The HP 9845A

William D. Eads / Jack M. Walden / Edward L. Miller

I. Introduction

What differentiates a desk-top computer, as described in this paper, from a minicomputer? Questions of this type are dangerous and difficult to answer because of the nonspecific usage of the terms and the wide variety of understandings of their meanings on the part of readers. Nevertheless, some useful generalizations can be extracted from common usage, even if they do not apply to all minicomputers or desk-top computers, or to all users.

First, a desk-top computer, unlike a minicomputer, is a complete system that necessarily has a high degree of physical integration of its elements. It has an input device (a keyboard), a display device (a CRT or a single-line display), a mass storage device (mag card, cassette, or floppy disk, for example), a processor, memory, connectors for external I/O devices, and power supplies built into an integrated package which can literally fit on the top of a desk. This high degree of integration is made possible by the large-scale integration of the key components of the computer, including processor, memory, and control logic for internal peripherals.

Second, the typical minicomputer is not ready for operation when it is received by the user, or even when all I/O devices are connected and it is initially powered up. System software, including the operating system, compilers, loaders, interpreters, editors, etc., must first be loaded into memory. The system must be told which I/O devices are (or may be) in the system at each I/O port, and which software module (driver) controls each device; this process is called I/O configuration. Only now is the system available for use. In contrast, the desk-top computer arrives with all system software in ROM already inside the machine, or in packages of optional ROMs that the user can easily install, generally in less than a minute. When I/O devices are attached, the computer can then generally determine for itself the device at each port and which driver is to be used. Users simply connect the external peripherals they plan to use, turn on the equipment, and begin using it. Therefore, desk-top machines incorporate a large degree of *logical* integration.

A third distinction is in the method of use of the two machines. Whereas a mini may have several languages available for the user, and an editor which allows programs to be written in any of these languages, a desk-top machine typically has a single language, with a built-in program editor which understands the syntactic restrictions of that language, and which does not allow a line with syntax errors to be entered into the user's program. Since there is but one language and one user at a time, the operating system for a desk-top machine can accomplish a task with fewer explicit directions from the user. There is no need to use a job control language to specify the language subsystem, any linkage editor, the memory requirements, or what peripherals are to be allocated during program execution. The user simply enters the program or loads it from the built-in mass storage, edits if necessary, and runs it by pressing a single key called RUN.

A similar distinguishing feature is that a desk-top computer can be used as a simple calculator as it stands, at any time during the entry or execution of a program. On most minis, the operating system doesn't understand such constructs as SIN (15) unless the user has entered some interpretive language subsystem, such as BASIC. Even then they don't necessarily have keyboard operation—but may *require* a program to be run.

The desk-top computer which will now be described is Hewlett-Packard System 45, shown in Fig. 1. It contains a typewriter-like keyboard, two cartridge drives for user program and data storage, a 24-line × 80-column CRT, and a built-in 480 line/min 80-column thermal printer, which can make a dot-for-dot copy of any CRT image. The internal thermal printer can also be used as a plotter with 560 by indefinitely many independently addressable dots. This machine has up to 64 Kbyte of user read/write memory (R/W), plus a separate 98-Kbyte operating system including an editor, a BASIC interpreter, and a sophisticated I/O scheduler.

The system is presented in a top-down manner. Section II

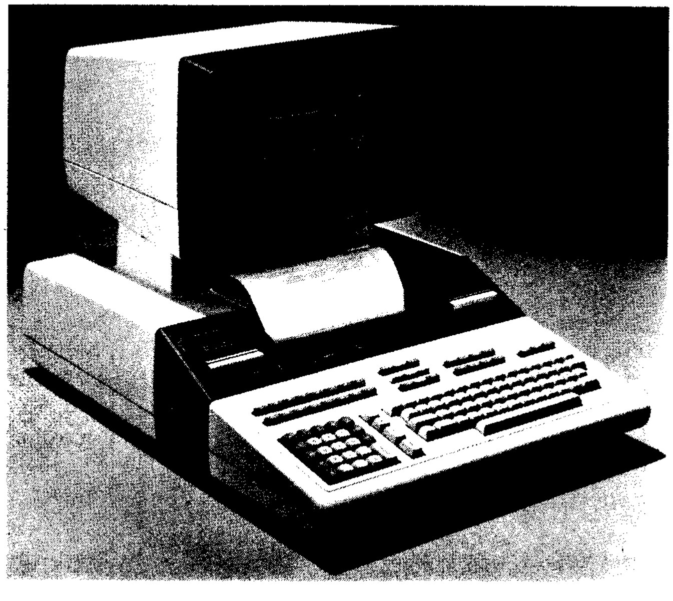


Fig. 1. The 9845A desk-top computer.

discusses the user environment and presents the internal storage format for user programs. The system organization, including process and processor synchronization, control, and communication, is outlined in Sec. III. Some details of each LSI component are provided in Sec. IV. Section V focuses on interprocessor communication and memory address sharing. The paper concludes with some considerations about the primary I/O device, the CRT display.

II. The User Language and Internal Form of Programs

The language of the System 45 is ANSI BASIC, enhanced to include string and matrix operations, subprograms, program linking, tracing, formatted output, mass storage files, and graphics. To aid in readability, variable names have been extended from a single letter or a single letter followed by a single digit to include zero to fourteen lowercase letters, digits, and/or underscores following a single uppercase letter. Major design goals in implementation of the BASIC interpreter were:

1 Expandability, to allow additional language features to be added to the system by use of plug-in ROMs

2 Interactive operation, to allow the user to interrogate and change the values of program variables, even as the program proceeds

3 Maximizing speed of execution within the constraint of interactive operation

4 Allowing program lines to be parsed to a form from which they can be reconstructed, in a form similar to that in which they were originally entered

An understanding of how these goals were achieved may best be found in an illustrative example. As shown in Fig. 2, the first operation in the use of the System 45 involves the keying of a program line into the computer. Completion of the line is signaled by depression of the STORE key. At this time the ASCII characters constituting the program line are placed in the *line buffer*, used for syntax analysis and listing. The system control supervisor calls the syntax supervisor, whose task is to convert the ASCII keystrokes into *internal form*, that is, into the format of program lines as stored in user read/write memory. Beginning at the left side of the source line, the syntax supervisor determines the line number and creates the first word of internal form in the *internal form buffer* (see Fig. 3). Next, the syntax supervisor attempts to match the statement name internal form (IF) with one within a linked list of statement keywords, a segment of which is shown in Fig. 4. In order to allow expandability there are actually as many as three linked lists which the syntax supervisor must scan in searching for a match with the statement name. First, an area of user read/write memory may contain binary programs, the most flexible way of adding language capability. Second, optional ROMs, increasing language capability, may be plugged into an option ROM port. And finally, the fundamental machine capability which exists within every System 45 includes a fixed set of keywords. Tables of keywords associated with the above three mechanisms are searched in the sequence above (allowing an optional capability to override the capability of the basic machine). When a match is found, the character position of the keyword in the source line is placed in the upper half of the second word of the internal form buffer (Fig. 3), and a pointer, associated with the element of the linked list for which a match was found, is placed in the third word. This pointer actually points to the fourth word down from the end of the keyword (see Fig. 4), at which location the ROM code for execution of that statement is located. Two words up from the execution routine is a pointer to the routine which performs syntax analysis of the rest of the statement. The syntax supervisor uses this address to pass control to the next statement syntax routine.

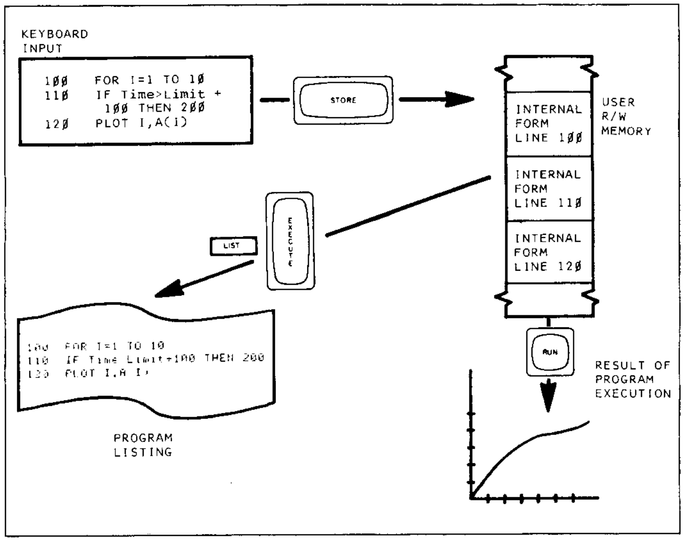


Fig. 2. The programming process.

The statement syntax routine shown in Fig. 4 for the IF statement immediately passes control to the expression syntax routine, which determines the order in which operations will be performed and which operands are used by each operation. The expression syntaxer creates an internal form for expressions which contains a set of operations that will be sequentially executed at run time; the present activity is still part of storing the line. Each operation consists of an operator pointer followed by zero or more operand pointers. For example, the computation of $A * B + C * D$ involves the multiplication of A and B, followed by the multiplication of C and D, followed finally by the sum of these two products, and consists of the three operations, as illustrated in Fig. 5.

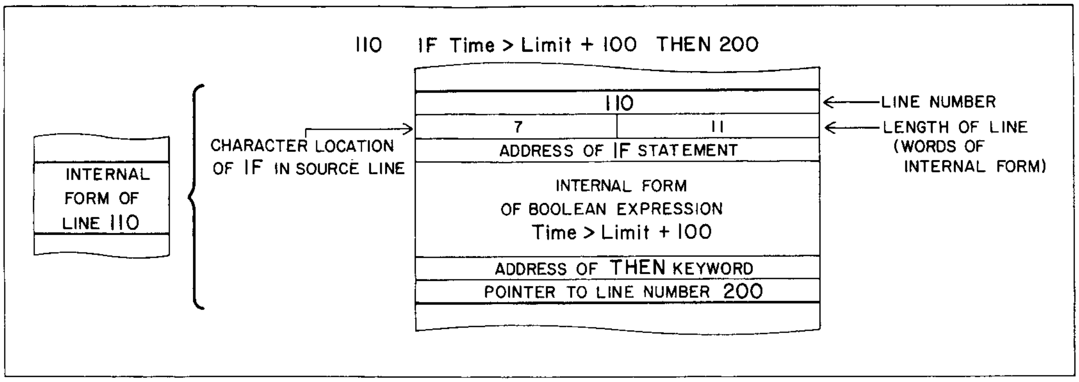

110 IF Time > Limit + 100 THEN 200
INTERNAL FORM OF LINE 110
CHARACTER LOCATION OF IF IN SOURCE LINE
110
7
11
ADDRESS OF IF STATEMENT
INTERNAL FORM OF BOOLEAN EXPRESSION Time > Limit + 100
ADDRESS OF THEN KEYWORD
POINTER TO LINE NUMBER 200
LINE NUMBER
LENGTH OF LINE (WORDS OF INTERNAL FORM)

Fig. 3. The internal form of a typical statement.

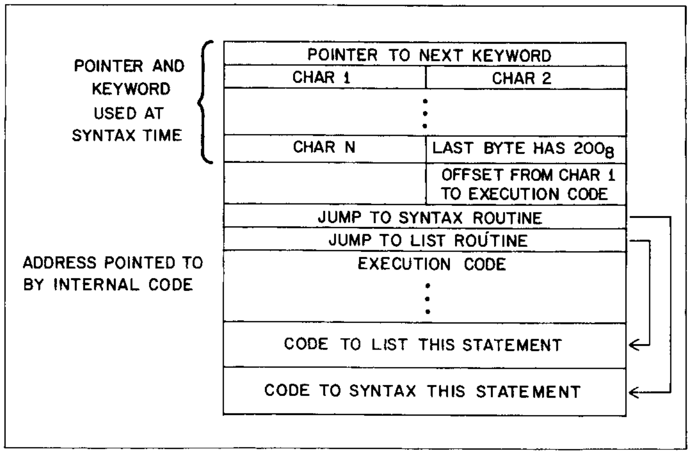

POINTER AND KEYWORD USED AT SYNTAX TIME
ADDRESS POINTED TO BY INTERNAL CODE
POINTER TO NEXT KEYWORD
CHAR 1
CHAR 2
CHAR N
LAST BYTE HAS 200$_8$
OFFSET FROM CHAR 1 TO EXECUTION CODE
JUMP TO SYNTAX ROUTINE
JUMP TO LIST ROUTINE
EXECUTION CODE
CODE TO LIST THIS STATEMENT
CODE TO SYNTAX THIS STATEMENT

Fig. 4. An entry in the linked list of statement keywords.

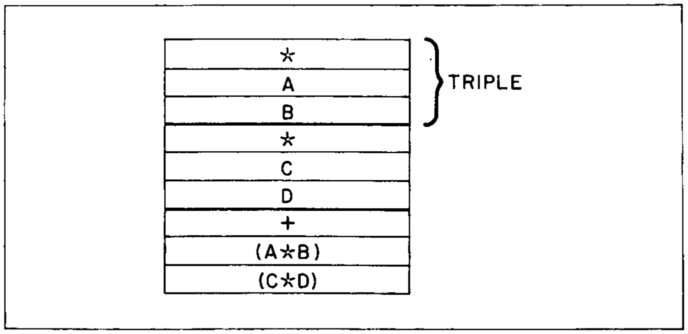


Fig. 5. Internal form for the expression A*B+C*D.

The operator-operand-operand entities in Fig. 5 are called *triples*, as are all other operations involving two operands (dyadic operations). Operations involving a single operand (monadic), such as negation or square root, result in two-word entities called *doubles*. The concept of doubles and triples is extended to *n-tuples*, consisting of an operator pointer followed by $n-1$ generalized operands. Thus the MAX function, which allows N operands, can be written as the $(N+2)$*-tuple* shown in Fig. 6.

The contents of the first triple of Fig. 5 are three sequential 2-byte words: the first is called the operator execution pointer and is a pointer to the first word of an execution routine (in system ROM) which fetches and multiplies two numeric quantities; the second and third entries are pointers into R/W memory to the symbol table entries of variables A and B respectively. The form of symbol table entries for variables is shown in Fig. 7.

Within an expression, the result of each n-tuple is placed in a unique 8-byte scratch-pad register. Forty of these registers are

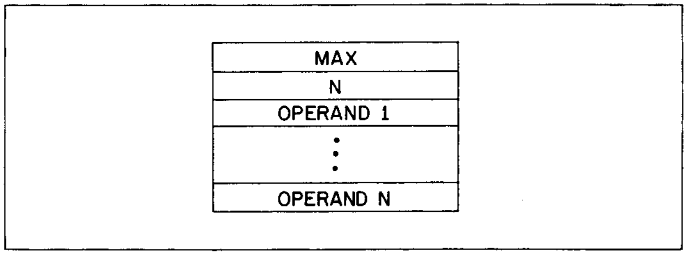


Fig. 6. Internal form for the MAX function.

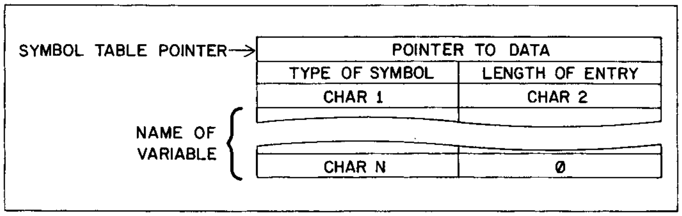


Fig. 7. Format of a symbol table entry.

available in contiguous memory, so that any single expression can have no more than 40 operators. The first operator executed causes the first scratch-pad register, called TEMP 1, to be filled; the second operator fills TEMP 2, and so forth through the expression. Therefore the expression syntax analyzer actually creates the third triple of A * B + C * D (see Fig. 5) as +, TEMP 1, TEMP 2. TEMP 1 is the address of the temporary which will contain A * B, and TEMP 2 is the address of the temporary for C*D. The TEMP pointers are distinguishable from symbol table pointers by the fact that the sign bit (most significant address bit) is 1 for symbol table pointers and 0 for all other machine pointers, including TEMP pointers.

When the expression syntax analyzer recognizes the keyword THEN in the example of Fig. 3, it places a word in the internal form buffer corresponding to the THEN part of the statement. Control is returned to the statement syntax monitor, which recognizes a line number of 200 and places into the internal form buffer a pointer to the symbol table entry for that line; only those lines which are referenced in the program are located in the symbol table. Statement syntaxing is now complete, and control is returned to the syntax monitor. At this time the length of line 110 is known, so that the syntax monitor can place the length of the line, in words of internal form, in the lower half of the second word of the internal form buffer. The final task of the statement syntax analyzer is the placement of the new line in its proper position, ordered by line number, in the user's program area in R/W memory.

The execution of a program as shown in Fig. 2 is most easily understood as a sequence of operations caused by the internal form of the program. The execution of a program in the System 45 may be viewed as sequential execution of each program statement, under control of the operating system. We may therefore use line 110, shown in Fig. 3, as an example. Execution of line 110 within a program would proceed as follows. (The internal code pointer, ICPTR, points initially to the first word of the internal form of the line.)

1 The operating system increments ICPTR by 2.

2 The operating system transfers control to code at the address given by the word pointed to by ICPTR (IF statement code).

3 The IF statement increments ICPTR and transfers control to the expression executor.

4 The expression executor transfers control to the subroutine, which adds two operands.

5 The add subroutine, using ICPTR, fetches Limit and 100 and adds them, placing the result in TEMP 1 and leaving ICPTR pointing to the internal code for the "greater than" operator; it then returns to the expression executor.

6 The expression executor transfers control to the subroutine, which checks for the "greater than" relation of two operands.

7 The "greater than" subroutine computes Time > (Limit + 100). If true, it returns 1; if false, it returns 0. It then returns to the expression executor.

8 The expression executor transfers control to the THEN subroutine, which returns immediately to the expression executor, which in turn returns to the IF statement executor.

9 If the value returned is nonzero, ICPTR is set, using the symbol table pointer for line 200, to the beginning of that line. If not, ICPTR points to the beginning of the next line following 110.

10 Control is returned to the operating system.

Because of the convenient form of the internal representation of the program line, overhead time for running the above sequence is quite small compared to the run time required to interpret the statement type, determine the sequence of the expression execution, and search through the program for a destination line number.

Listing of program lines, using the internal form of statements (Fig. 3) and the linked list of keywords in ROM (Fig. 4), occurs in a process converse to that of syntax analysis. Using the first word of the internal form, the line number is formed in the *source line buffer*, followed by enough spaces (at least one) to begin the keyword in the column position given by the upper half of the second word of the internal form. From the third word of internal form (the statement execution address) the list routine subtracts 3. From Fig. 4, it can be seen that this is the address of the word whose lower byte is the offset of the statement keyword from the first word of execution code. This offset is then subtracted from the statement execution address to give a pointer to the ACSII representation of the keyword, which is located at the beginning of that section of the linked list of keywords, and follows the pointer to the next keyword. These characters are transferred to the source line buffer one by one until a byte is found whose most significant bit is set, indicating the end of the keyword. Next, the address of the beginning of the execution code is decremented by 1 to determine the location of the routine which lists the rest of the internal form for that statement.

Any statement lister may call the expression lister, which determines the location of the operators and operands associated with that expression. Associated with the ROM execution code is the information necessary to list that operator and its operands—including the ASCII representation of the operator and the number of operands (and how they are arranged syntactically with respect to the operator)—as well as the precedence of the operator. The operator precedence, together with the sequence of operator execution in the internal code, furnishes the expression list monitor with sufficient information to list the expression with the same sequence of operands and operators as was entered originally, along with required parentheses. The only differences between the entered and listed lines involve extraneous or missing spaces and redundant parentheses.

In the example of Fig. 3, control passes from the statement lister to the expression lister after 110 IF is listed, and it produces Time > Limit + 100. Note that no parentheses are listed (even if they were placed around Limit + 100 when it was keyed in). The statement lister then adds THEN from a keyword association with its associated execution address and finally adds 200 from the symbol table reference. Control is then transferred back to the operating system to output or display the now complete line in the source line buffer.

The final feature of the language system to be discussed is that of the user's ability to interact with the program as it is executing, a capability which is called having a *live keyboard*. Since all variables are accessible through a symbol table, since the program execution monitor has control of the processor at the end of the execution of each line, and since the system was built to allow the addition of variables and the addition or deletion of program lines at any time (even between executions of program lines), the capability of interacting with an executing program is extensive. Users can interrogate or change variables as the program runs; they can compute complex expressions; they can even delete, add, or modify program lines as the program executes. While these capabilities may be dangerous for a production program, they are certainly convenient during the development and debugging of new programs, and they can be removed during a program run by the execution of the command SUSPEND INTERACTIVE.

The next section provides an overview of the multiprocessor system used to implement the user program environment just described.

III. System Organization and Control

Examination of Fig. 8 reveals that all communication with the outside word is via the Peripheral Processing Unit (PPU). *All* peripherals—keyboard, CRT, printer, etc.—are tied to the PPU's I/O bus. The Language Processing Unit (LPU) has *no* peripherals attached to it, and it can communicate only with the PPU.

The PPU is responsible for managing all the system resources except block 0 random-access memory (RAM), which is managed by the LPU. The resources managed by the PPU are block 1 RAM, all I/O devices, and the LPU.

Interprocessor Communications

Communication between the processors is solely through the use of shared RAM. There are no dedicated signal lines or interrupts between processors. One mode of communication is via messages stored in buffers. Each processor has a fixed buffer of seven words for sending a message to the other processor. These buffers are guarded and controlled through flags. The second mode of communication is quite diverse. Certain words throughout RAM are allocated as convenient for the processes needing them. They are used as flags, semaphores, tables, etc., to synchronize and control the two processors in ways that are specific to the particular task.

In this control/communication mechanism, there are several cases where a processor must have exclusive access to a table, counter, or buffer area; i.e., while one processor is using this area, the other processor must *not* be allowed access into it. This kind of *exclusive access* can be rigorously controlled by the use of a two-flag exclusion algorithm first proposed by T. Dekker [Shaw, 1974]. This algorithm is implemented (in a somewhat simplified form) in the HP 9845A to control LPU/PPU access to critical constructs. For example, the LPU alone can create buffers; once created, a buffer can be filled by either the PPU or the LPU. Both the LPU and PPU may have occasion to read from or modify a given buffer. Buffers may be destroyed by either the PPU or LPU. Clearly, such cooperative use of buffers requires controlled access.

The simplified two-flag algorithm of Fig. 9, implemented in the HP 9845A, does not include the case of *mutual exclusion*, which, in the general case, could lead to endless *synchronized deadlock* if not accounted for. In the HP 9845A this cannot occur, because the "failure" paths for the LPU and PPU are different; the LPU "waits," whereas the PPU "gives up" and returns to process scheduler.

This exclusive access problem is quite fundamental in all multiprocessor systems—which usually implies large systems. It may surprise some to find it occurring in a desk-top machine.

I/O Process Handling

The PPU establishes and controls the keyboard entry protocol. When the user makes a complete keyboard-record entry (terminated by STORE, EXECUTE, or CONTINUE), the keyboard is disabled until the system interprets the record; i.e., the system examines the line and determines what it should do. As soon as the record is interpreted, the keyboard is reenabled while the actual execution takes place. This sequencing allows concurrent execution of a number of commands but prevents the user from submitting a new record before the system is able to accept it. The PPU allows concurrent execution of keyboard commands, and also execution of keyboard commands concurrent with program execution if there is no resource conflict involved. An example of a conflict would be a GET command to load a program from a tape cartridge, followed immediately by a REWIND of the cartridge

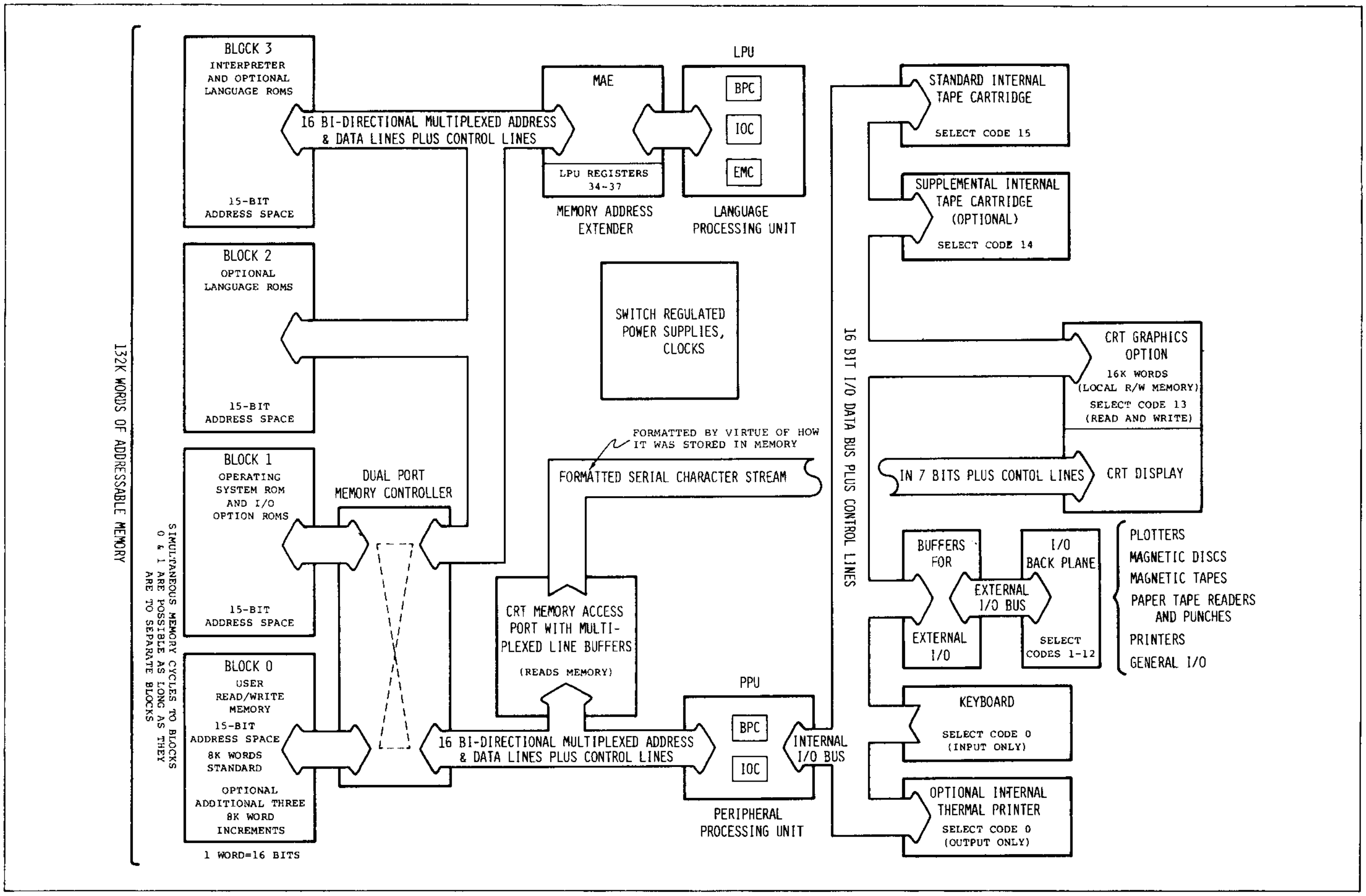

132K WORDS OF ADDRESSABLE MEMORY
SIMULTANEOUS MEMORY CYCLES TO BLOCKS 0 & 1 ARE POSSIBLE AS LONG AS THEY ARE TO SEPARATE BLOCKS
BLOCK 3
INTERPRETER AND OPTIONAL LANGUAGE ROMS
15-BIT ADDRESS SPACE
BLOCK 2
OPTIONAL LANGUAGE ROMS
15-BIT ADDRESS SPACE
BLOCK 1
OPERATING SYSTEM ROM AND I/O OPTION ROMS
15-BIT ADDRESS SPACE
BLOCK 0
USER READ/WRITE MEMORY
15-BIT ADDRESS SPACE
8K WORDS STANDARD
OPTIONAL ADDITIONAL THREE 8K WORD INCREMENTS
1 WORD=16 BITS
16 BI-DIRECTIONAL MULTIPLEXED ADDRESS & DATA LINES PLUS CONTROL LINES
MAE
LPU REGISTERS 34-37
MEMORY ADDRESS EXTENDER
LPU
BPC
IOC
EMC
LANGUAGE PROCESSING UNIT
SWITCH REGULATED POWER SUPPLIES, CLOCKS
DUAL PORT MEMORY CONTROLLER
FORMATTED BY VIRTUE OF HOW IT WAS STORED IN MEMORY
FORMATTED SERIAL CHARACTER STREAM
CRT MEMORY ACCESS PORT WITH MULTI-PLEXED LINE BUFFERS
(READS MEMORY)
16 BI-DIRECTIONAL MULTIPLEXED ADDRESS & DATA LINES PLUS CONTROL LINES
PPU
BPC
IOC
INTERNAL I/O BUS
PERIPHERAL PROCESSING UNIT
16 BIT I/O DATA BUS PLUS CONTROL LINES
STANDARD INTERNAL TAPE CARTRIDGE
SELECT CODE 15
SUPPLEMENTAL INTERNAL TAPE CARTRIDGE (OPTIONAL)
SELECT CODE 14
CRT GRAPHICS OPTION
16K WORDS (LOCAL R/W MEMORY)
SELECT CODE 13 (READ AND WRITE)
IN 7 BITS PLUS CONTOL LINES
CRT DISPLAY
BUFFERS FOR
EXTERNAL I/O
I/O BACK PLANE
EXTERNAL I/O BUS
SELECT CODES 1-12
PLOTTERS
MAGNETIC DISCS
MAGNETIC TAPES
PAPER TAPE READERS AND PUNCHES
PRINTERS
GENERAL I/O
KEYBOARD
SELECT CODE 0 (INPUT ONLY)
OPTIONAL INTERNAL THERMAL PRINTER
SELECT CODE 0 (OUTPUT ONLY)

Fig. 8. System 45 hardware block diagram.

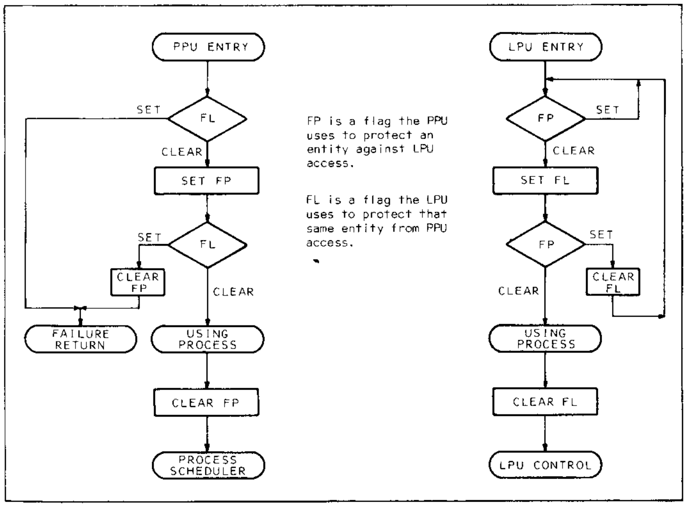


Fig. 9. The two-flag method of exclusive access.

before the GET is completed. When concurrent operations cannot be allowed, a SYSTEM BUSY message is given. Since all peripherals are attached to the PPU, the PPU must perform all transfers of data and programs between the desk-top computer and peripheral devices.

I/O processes can be initiated by the program being executed by the LPU, or by the user via keyboard entry commands. Most such commands can also be stored as a part of a program. The LPU syntaxes, stores, and executes all programs; thus it must be able to interpret and cause execution of most commands. Therefore most commands, although processed by the PPU during keyboard entry, are "handed over" to the LPU for interpretation.

Thus, each I/O activity is *initiated* by the LPU but is turned over to the PPU to be carried out. Each task involves both processors carrying out specific subtasks. These subtasks include communication between processors concerning the state of the subtasks, as well as monitoring, synchronizing, and terminating the overall task. To explain this, each processor and its role will be described.

PPU Process Definition

Except for initialization (power on, SCRATCH ALL) and the Process Scheduler (which is the "idle loop"), all PPU work is carried out by *processes.* When a process is needed it is invoked by "creating" it. A user process and a keyboard process are created during initialization. All other processes are created dynamically at the beginnings of the various individual tasks and are destroyed upon their individual completions.

A process is represented by at least one Process Control Block (PCB). The PCB is a 10-word R/W memory entity used to contain (either directly or indirectly) all the information necessary for the PPU to execute the associated process. Figure 10 shows the structure of a PCB.

PCBs are taken from block 1 RAM by the PPU memory allocator, which maintains a PCB Free List. They are linked to the Process Tree during their active life, and are linked back to the Free List when the process is completed. The Free List is linked through the first word of each PCB.

Some processes need more temporary *process control* storage than the 10 words of a PCB. Those 10 words are strictly allocated in use as per Fig. 10. Additional 10-word entities called *data blocks* may be obtained from the Free List; they are linked to the PCB via the ninth word, Data Block Link (DBL).

Active PCBs are linked together in various ways through the Brother Link (BL), Father Link (FL), and Son Link (SL), labeled in Fig. 10. All processes invoked by the user through execution of a program are represented and controlled by a tree of PCBs linked to the user process (which was created at initialization and is never destroyed). The hierarchy of processes is implemented via the SL BL, and FL links, to create an orderly control structure. In general, the creation of a process, communication between processes, and the removal of a process take place between processes no more than one level apart in this control structure. In this hierarchical structure, the SL points to a process at a lower level, the FL points back up to the higher-level process, and the BL points to associated processes at the same level.

A process tree which might arise during the execution of a program is illustrated in Fig. 11. The Brother Links (BLs) represent the existence of more than one *incomplete* I/O operation invoked by the execution of the program. This can only occur when the system is running in the OVERLAP mode, which allows concurrent, overlapped I/O operations (discussed later).

In addition to the process tree linking with SL, FL, and BL illustrated in Fig. 11, the PCBs are linked together into other important lists through the tenth word, i.e., the Queue Link.

Each peripheral is attached to the machine via an interface which has a peripheral address (select code) in the range 0 to 15. Each I/O operation invoked by a program statement specifies (explicitly, or implicity by system default) the peripheral address of the device to which it is directed. When the LPU passes to the PPU the I/O process to be handled, the PPU creates a PCB to represent the process and links it into the Process Tree. In addition to this process control mechanism (which is independent of particular devices or select codes) it must also maintain knowledge of the specific device. And if other operations to that device exist (in the Process Tree), it must also see that the chronological sequence is preserved. This is accomplished by also linking the PCBs into queues—one for each peripheral address. These queues are headed (pointed to) by a table with an entry for each peripheral address. In addition to the actual hardware peripheral addresses 0 to 15, there are pseudoaddresses 16, 17, and 18, which represent various areas of the CRT: those for PRINT, DISP ("display" command) and implied DISP.

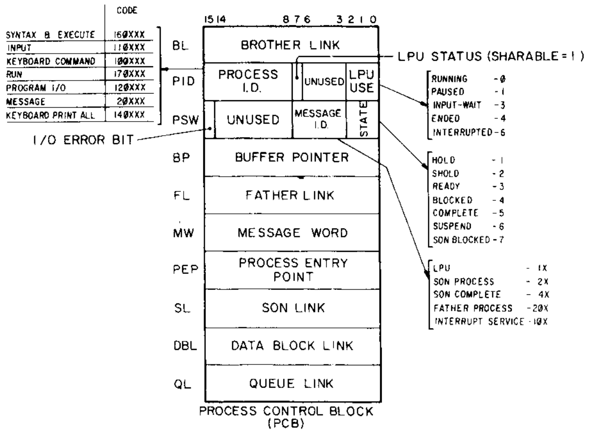


Fig. 10. Format of a process control block (PCB).

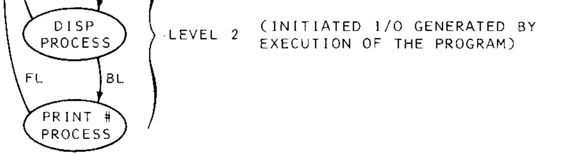

USER PROCESS
LEVEL Ø
(REPRESENTS THE OVERALL OPERATING SYSTEM)
FL
SL
RUN PROCESS
LEVEL 1
(INVOKED BY USER PRESSING RUN KEY TO EXECUTE PROGRAM)
FL
SL
PRINT PROCESS
FL
BL
DISP PROCESS
LEVEL 2
(INITIATED I/O GENERATED BY EXECUTION OF THE PROGRAM)
FL
BL
PRINT # PROCESS

Fig. 11. A typical process tree linking several PCBs.

In addition to the queue of operations for the peripheral address, there is always an associated device buffer. The same table which heads the peripheral address queues also contains pointers to those buffers. This total construct—pointers to PCBs in device queues, and pointers to device buffers—is called the QTABLE.

QTABLE plays an important role in the overall process scheduling. It was mentioned earlier that the Process Scheduler is the PPU idle loop. What the Process Scheduler does to find processes which can be "worked on" is to scan QTABLE for peripheral addresses with active queues attached. If such a queue exists, the top PCB on the queue is examined to see if that process is in a state where anything can be done. If not, the scan continues to the next peripheral address. If something can be done, depending on the state of the process, it is done.

In the System 45, the normal mode of I/O transfers is "interrupt-by-the-character," with all transfers to the peripheral carried out in an Interrupt Service Routine (ISR). The PPU has vectored interrupt as part of its structure (implemented in the Input/Output Chip, IOC in Fig. 8). The overall process of carrying out such tranfers occurs in three stages:

1 Queueing up of the process, obtaining the resources required (buffers, etc.), and activation of the ISR (setting interrupt vector table entry, etc.), *followed by* return of the PPU to the idle loop or other tasks

2 Character-by-character transfer as interrupts occur and watching for the last transfer, when interrupt transfers are terminated

3 Final termination of the process—release of buffers, dequeuing of PCBs, etc.

As indicated in Fig. 10, the third word of the PCB is a Process Status Word (PSW) in which the state of the process is recorded. During its lifetime, a process may go through a number of states to accomplish the three stages of I/O transfer activity previously mentioned. Figure 12 shows the state transitions possible in the life of a PCB. The device transfers in the ISR occur during the BLOCKED state.

Formatting Output

Formatting from internal stored form to external form (such as ASCII character streams) is carried out in the act of transfer from the block 0 (of memory) data buffer to the block 1 (of memory) device buffer. This is performed by the PPU, and so it is interesting to see how this is done within the control structure that has been described.

To see the process involved, refer to Fig. 13. Suppose that the LPU, in executing a program, has encountered a PRINT statement with *n* expressions (items) in its list whose output is to be directed to an external printer at peripheral address 8. The LPU obtains a data buffer adequate to hold the *n* items (the size needed is determined when the PRINT statement is syntaxed and stored) from the block 0 memory manager. It sends a Start I/O message to the PPU with three items of information—the peripheral address, the block 0 data buffer address, and the starting address for the PPU PRINT routine.

The LPU now begins (without further concern for the PPU) to

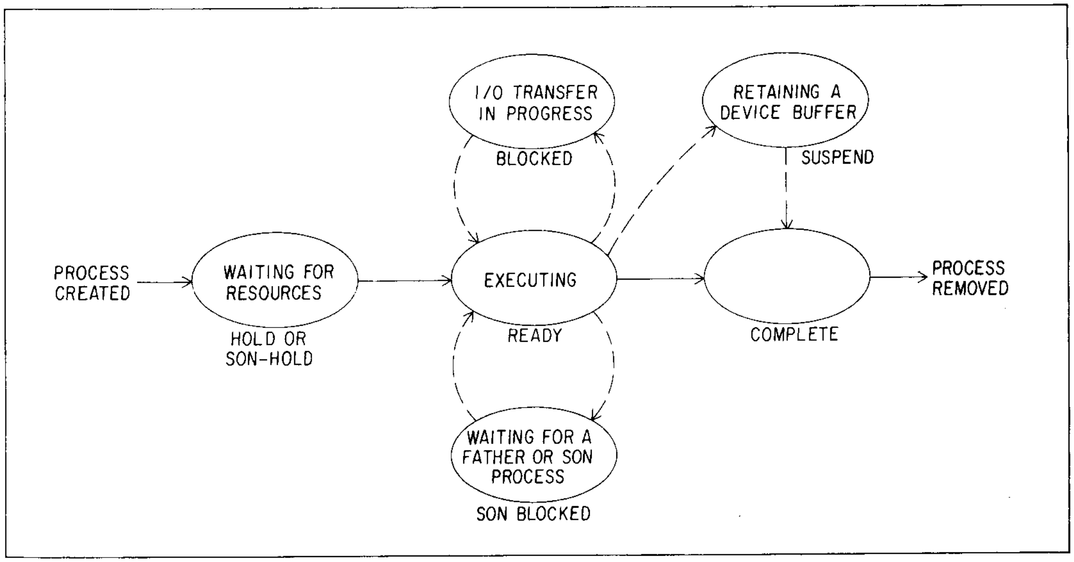


Fig. 12. Possible state transitions in the life of a PCB.

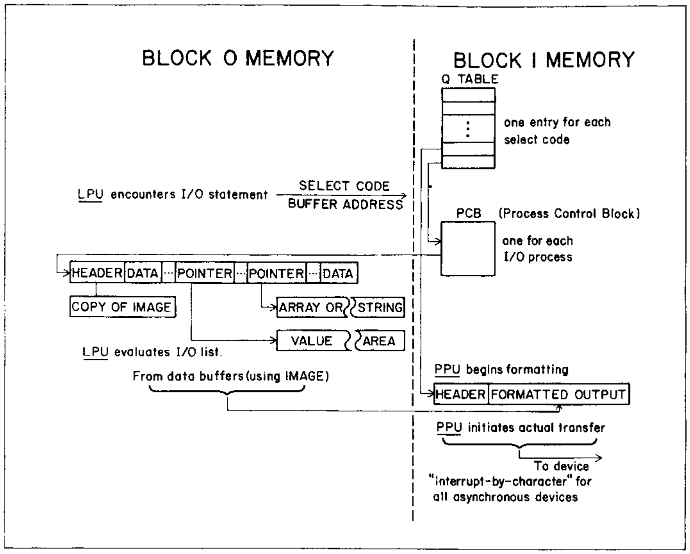


Fig. 13. LPU/PPU interaction during output.

evaluate the output list expressions. As each is evaluated, it is put in the next storage cell of the data buffer, followed by a WAIT item. Simultaneously, the PPU responds to the Start I/O message by obtaining a PCB, filling it with the buffer pointer (BP) and starting address (PEP), and setting it up in the Process Tree, and, we will assume, getting it queued at the head of the appropriate peripheral address queue. The process is in the HOLD state, but the PPU immediately attempts to allocate resources and activate the process. Assuming that a device buffer is available, the PPU will immediately allocate it and set the PCB READY. The Process Scheduler will see the READY and begin execution through the PEP.

The routine at PEP begins the formatting. It will obtain items from the data buffer, formatting each into the device buffer. Three obvious possibilities exist:

1 The formatting catches up with the LPU, by encountering the WAIT item in the buffer. The PPU will change PEP to a "continue formatting" routine, leave the PCB READY, and return to the Process Scheduler. This allows the PPU to work on other processes.

2 The formatting has finished all items when it finds a "done" item in the buffer. The PPU will set the interrupt return vector, initiate the interrupt output, set PEP to a "clear up" routine, BLOCK the PCB, and return to the Process Scheduler. The PPU works on other I/O and on this I/O by interrupt until transfer is done, then marks PCB complete.

3 The formatting generates enough data to fill the device buffer, and so the PPU initiates I/O, sets PEP to a "record gone, resume formatting" routine, sets PCB BLOCKED, and returns to the Process Scheduler. When the final interrupt occurs, the process is made READY, and formatting resumes.

In case 1 above, the Process Scheduler, finding the PCB READY, will execute the routine at PEP. This routine will check whether the WAIT item is still next, or whether it has been replaced by the LPU with data. If WAIT is there, it just returns; thus each scan of the queues causes a recheck. Notice that there is concurrency present in this process. The LPU is evaluating expressions and filling the buffer while the PPU follows it (as processor time is available) item by item in the formatting.

Device Conflicts

One additional task that is extremely important in the correct handling of I/O is the management of possible device conflicts and the allocation of resources. These conflicts are handled in the Process Scheduler routines which switch a process in the HOLD state to READY.

Some obvious things are involved in resource allocation; for example, device buffers from the pool at block 1 R/W memory. One item not so obvious is the DMA channel. There is only one DMA channel available. However, DMA transfers may be desired for several processes on different peripheral addresses. Thus, the resource allocator must keep track of DMA channel utilization and sequentially allocate it to processes needing it.

Another area of device conflict is the relationship between synchronous and asynchronous devices. Synchronous devices, such as tape transports, require service at intervals dictated by the device. If service is not rendered when required, data are lost or erroneous data written. A synchronous device on a low-priority interrupt might have the processor taken away by a higher-priority interrupt, causing it to miss an essential transfer.

If a synchronous device is protected from this by being allowed only on high-priority interrupt levels, DMA transfers may still cause trouble. The DMA, if activated, may "steal" so many memory cycles that the interrupt service routine response may be slowed to a fraction of its normal speed. Again, an essential transfer may be missed.

These conflicts can be resolved by delaying the transfer from HOLD to READY for processes that would create these conditions.

Overlapped and Serial I/O Processing

The Process Tree and PCB linkage shown in Fig. 11 show the existence of PCBs (and therefore active I/O processes) at the head of three device queues: the printer for the PRINT process; the CRT for the DISP process; and a mass storage device for the PRINT # process. Since I/O transfers are, in general, interrupt-by-character (or DMA for the mass storage device), a number of processes at the heads of different queues could have the I/O transfers initiated and be in the BLOCKED state. Transfers would occur randomly from one process to another as interrupts occurred for the various devices. This is *buffered* and *overlapped* I/O. It is the mode for which the System 45 I/O Process Handling was designed. The LPU is allowed to "forge ahead," sending new Start I/O messages and filling new data buffers as long as memory is available for data buffers, PCBs, and device buffers.

However, there are times when all of this overlapped activity is not desired. For example, it disconnects the LPU execution of a PRINT statement from the PPU outputting of the data. This can be very confusing, particularly during program testing and debugging.

At the end of each program line, the LPU examines a flag which serves to control this overlapping of I/O. If the flag is in the SERIAL mode, the LPU waits for the PPU to send it a message that the output associated with that line is finished. It will then start the next line. If the flag is in the OVERLAP mode, the LPU does not wait for the message, but proceeds on to the next line.

The PPU does not normally send a message to the LPU upon the completion of every I/O operation, so how does it know to do so when the mode is SERIAL? In the discussion of formatting, it was mentioned that the PPU knew it was at the last item of a PRINT list when it encountered a "done" item. This item is placed there by the LPU when it has evaluated the last item on the list—if the mode is OVERLAP. If the mode is SERIAL, it places a "reply" item instead of "done." The PPU knows, when it sees "reply," that this is the end of the list and that it should send a message to the LPU that the I/O process is done.

IV. The Hardware Architecture of the 9845A

The internal architecture of the 9845A hardware is illustrated in the block diagram of Fig. 8. The major elements of the diagram are the two processors called the Language Processor Unit (LPU) and the Peripheral Processor Unit (PPU), and the Memory Address Extender (MAE) with its associated four 32-kiloword blocks of memory (block 0 through block 3). Associated with the memory are the Dual Port Memory Controller and the CRT Memory Access Port.

The main purpose of the LPU is to execute the user's program. To do this it executes a BASIC interpreter encoded in ROM located in blocks 2 and 3 of memory. The user's program is stored in block 0 of R/W memory. The main function of the PPU is to perform I/O and certain other activities. A communications protocol involving shared memory is the basis of LPU/PPU communication.

The LPU and PPU are both processors that, in isolation, can command 16-bit memory address spaces. The PPU does in fact have access to such a 64-kiloword portion of memory, i.e., block 0 and block 1. Assembly language coding for the PPU can, in fact, ignore the memory address extension scheme altogether and simply consider the designations of block 0 and block 1 as an artificial distinction between the two halves of its address space. For the LPU, however, the 64-kiloword address space is split into parts of equal size (32 kilowords) and logically distributed among blocks of memory, the sum of whose memory space is far in excess of the address space of the processor. In the scheme embodied by the MAE the LPU can also access the same memory that the PPU does. This gives rise to the need for the Dual Port Memory Controller, whose function is to resolve conflicts arising when the LPU and PPU try simultaneously to access the same block of memory.

The CRT Memory Access Port accesses memory on behalf of the CRT to provide ongoing access to the information stored in the system-managed CRT buffer in block 0. The alphanumeric (i.e., nongraphic-mode) display is formed on the basis of that information, which must be reread each time the CRT screen is to be refreshed.

Neither the LPU nor the PPU is a homogeneous, monolithic entity. Each is composed of smaller functional units which are LSI chips. Among these units are a Binary Processor Chip (BPC), Input-Output Controller (IOC), and, for LPU only, an Extended Math Chip (EMC). The BPCs used in the LPU and PPU are of identical design, as are the IOCs. The main functions of a BPC are to fetch instructions from memory, execute most instructions that reference memory, execute various instructions that perform bit manipulation, and accomplish program branching. Thus, the BPCs are relatively general-purpose devices, and each serves more or less the same general function in the LPU and PPU. The main functions of the IOC are to provide I/O and instructions for manipulating firmware stacks. The reason the PPU has an IOC is to obtain both those capabilities. The LPU, however, does not do I/O; it contains an IOC merely to obtain the use of the stack instructions. The main function of the EMC is to perform BCD arithmetic. This is strictly an LPU activity; therefore the PPU is not equipped with an EMC.

Also shown in Fig. 8 is the PPU-managed I/O Data Bus and the various peripherals that are normally permanently connected to it. The manner in which I/O is accomplished is discussed in conjunction with the IOC. The notion of a select code as the address of a peripheral will be fully explained at that time. At this point, however, it is appropriate to point out that, in general, two peripherals cannot have the same select code. But the keyboard and the internal thermal printer both have select code 0. This is a special case that doesn't cause any problems, because the keyboard is strictly an input device and the printer is strictly an output device.

There now follows a description of the LPU hardware. Since the hardware description of the PPU is a subset of the LPU hardware description, the PPU will not be described separately.

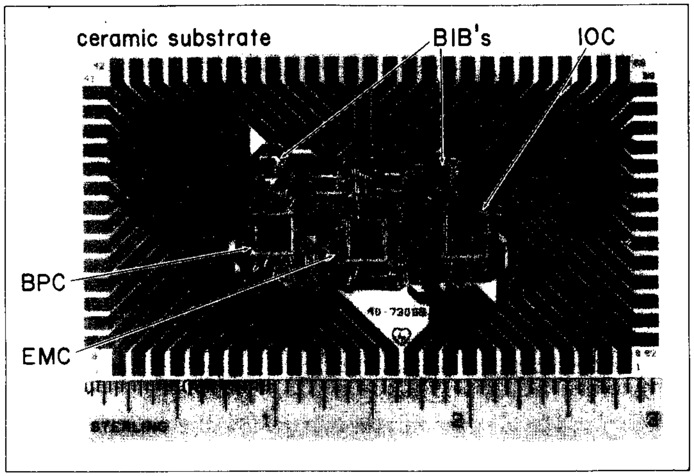


Fig. 14. The processor on its substrate.

Hardware Description of the LPU

The LPU consists of seven integrated circuits mounted on a ceramic substrate (see Fig. 14). Of these, the BPC, IOC, and EMC are N-channel MOS LSI chips. The remaining four chips (Bi-Directional Interface Buffers, or BIBs) are entirely bipolar and serve as buffers to connect the LSI circuitry to circuitry external to the substrate.

Figure 15 is a block diagram of the LPU and PPU. All of the processing capability of the processor resides in the three LSI chips; except for inversion of the IDA Bus the four BIBs are logically powerless. The three LSI chips communicate among themselves, and also with the outside world, via a collection of control signals and a 16-bit bus called the IDA Bus (IDA stands for *instruction/data/address*). The processor uses 16-bit addressing for memory and implements a single level of indirect addressing.[1]

Memory Conventions

Most of the traffic on the IDA Bus has to do with memory. Both the address of memory locations and the contents of those locations (data and machine instructions) are transmitted over the IDA Bus. Further, memory can be physically distributed along the bus. Each of the three chips in the processor contains registers which are addressable, and of course, addressable memory also exists external to the processor.

[1] Except during interrupt, when a two-level indirect is forced. This is explained in connection with interrupts.

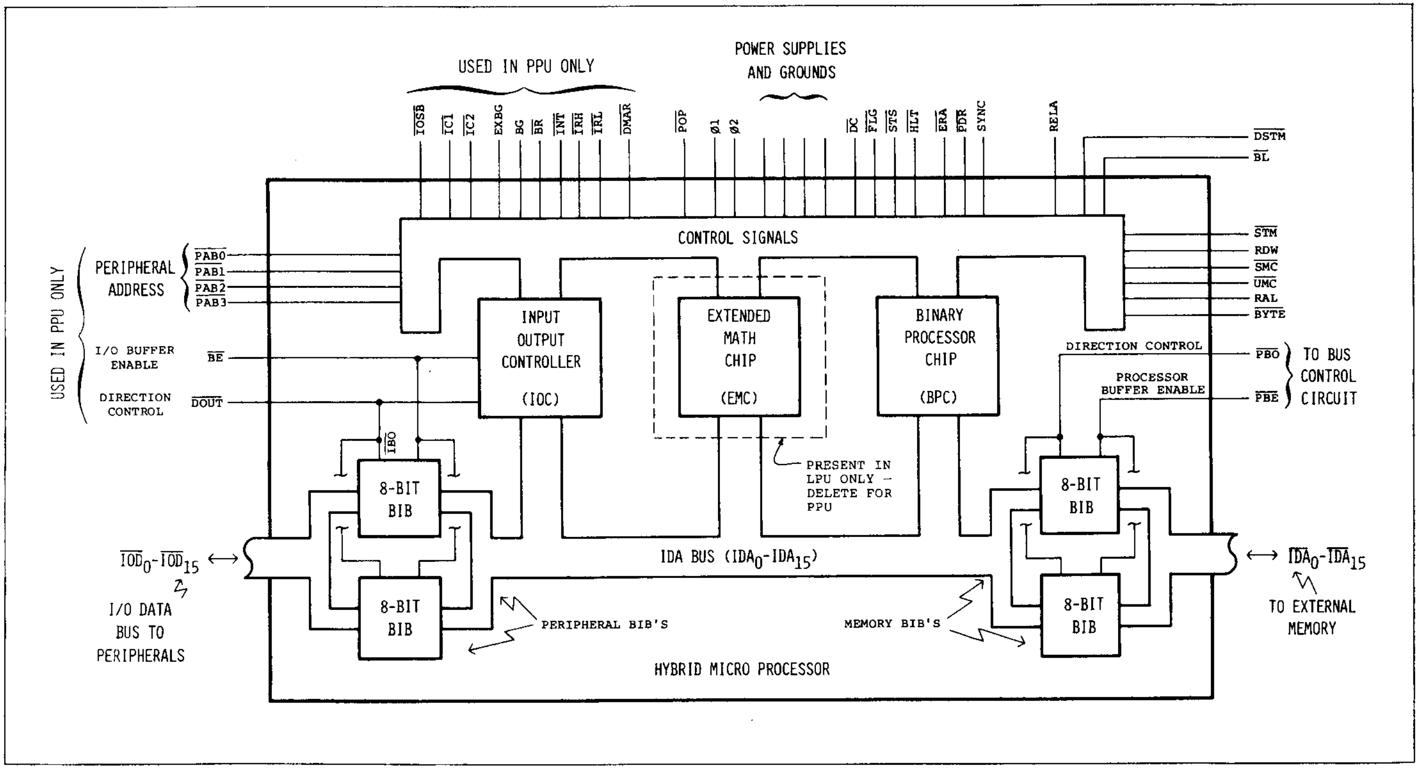


Fig. 15. Processor block diagram.

The first 32 addresses of the address space do not refer to external memory. Instead, these addresses (0–37_8) are reserved to designate addressable registers within the microprocessor. Figure 16 lists these registers. There are also a number of nonaddressable internal-use registers in the processor. Registers range in size from 1 to 16 bits; most are 16-bit registers.

A memory cycle involves some control lines as well as the IDA Bus. Start Memory ($\overline{\text{STM}}$) is used to initiate a memory cycle by identifying the contents of the IDA Bus as an address. Either of two memory complete signals is used to identify the conclusion of a memory cycle. These are Unsynchronized Memory Complete ($\overline{\text{UMC}}$) and Synchronized Memory Complete ($\overline{\text{SMC}}$). A line called Read/Write ($\overline{\text{RWD}}$) specifies the direction of data movement. Each element in the system decodes the addresses for which it contains addressable memory. To initiate a memory cycle, an element in the system puts the address of the desired location on the IDA Bus, sets the Read/Write line, and gives Start Memory. It is part of the system definition that whatever is on the IDA Bus when a Start Memory is given is an address of a memory (or register) location. Then, elsewhere in the system the address is decoded and recognized, and that agency begins to function as memory.

Among the several service functions performed by the BPC, for the IOC and EMC, is the generation of a signal called RAL (Register Access Line). This occurs whenever an address on the IDA Bus is within the range reserved for register designation (0–37_8). RAL is used by the external memory to prevent its response to any memory cycle having such an address.

	Octal Address	Name	Location	Description (# of Bits)
	0	A	BPC	Arithmetic Accumulator (16)
	1	B	BPC	Arithmetic Accumulator (16)
	2	P	BPC	Program Location Counter (16)
	3	R	BPC	Return Stack Pointer (16)
	4	R4	IOC	Peripheral Activity Designator (—)
	5	R5	IOC	Peripheral Activity Designation (—)
	6	R6	IOC	Peripheral Activity Designator (—)
	7	R7	IOC	Peripheral Activity Designator (—)
	10	IV	IOC	Interrupt Vector (upper 12 of 16)
* →	11	PA	IOC	Peripheral Address Register (least 4 of 16)
	12	W	IOC	Working Register (16)
† →	13	DMAPA	IOC	2 MSB = CB & DB; 4 LSB = DMA Periph. Add. Reg.
	14	DMAMA	IOC	DMA Memory Address & Direction Register (16)
	15	DMAC	IOC	DMA Count Register (16)
	16	C	IOC	Stack Pointer (16)
	7	D	IOC	Stack Pointer (16)
	20-23	AR2	EMC	BCD Arithmetic Accumulator (4 × 16)
	24	SE	EMC	Shift Extend Register (least 4 of 16)
* →	25-27	X	EMC	Internal Arithmetic Register (3 X 16)
	30-37	UNASSIGNED		
	77770/ 177770	ARI	R/W	BCD Arithmetic Register (4 × 16)

* Not available for general use. Part of processes internal to a chip.

† Read register 13_8 produces:

CB and DB are actually discrete registers, and while they can only be read by reading R13, storinging into R13 will not alter their values. Use the CBL, CBU, DBL and DBU machine instructions for that purpose.

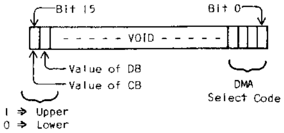


Fig. 16. The processor registers.

General Description of the BPC

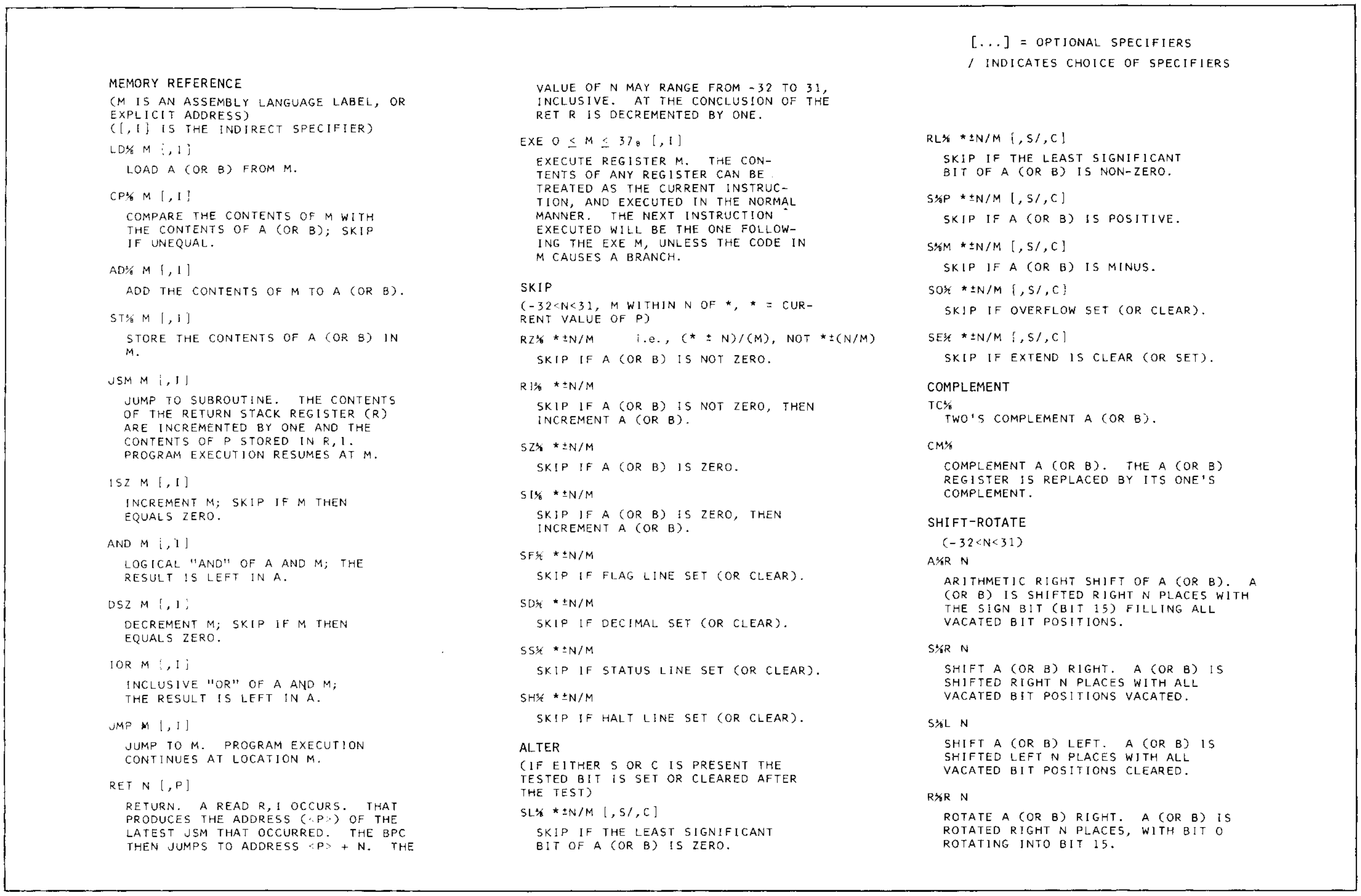
The BPC has two main functions. The first is to fetch machine instructions from memory for itself, the IOC, and the EMC. A fetched instruction may pertain to one or more of those chips. A chip that is not associated with a fetched instruction simply ignores that instruction. The second main function of the BPC is to execute the 56 instructions in its own repertoire. A condensed description of these instructions is shown in their assembly language format in Fig. 17. These instructions include general-purpose register and memory reference instructions, branching instructions, bit manipulation instructions, and some binary arithmetic instructions. Most of the BPC's instructions involve one of the two accumulator registers, A and B.

There are four addressable registers within the BPC, and they have the following functions: the A and B registers are used as accumulator registers for arithmetic operations, and also as source or destination locations for most BPC machine instructions referencing memory. The R register is an indirect pointer into an area of Read/Write memory designated to store return addresses associated with nests of subroutines encountered during program execution. The P register contains the program counter; its value is the address of the memory location from which the next machine instruction will be fetched.

Upon the completion of each instruction the program counter (P register) has been incremented by 1, except for the instructions JMP, JSM, and RET, and for SKIP instructions whose SKIP condition has been met. For those instructions the value of P will depend on the activity of the particular instruction.

Indirect Addressing Indirect addressing is a technique in which an instruction that references memory treats the first one or more references as an intermediate step in referencing the final destination. An intermediate reference yields the address of the next location to be referenced. When an intermediate location can point to yet another intermediate location, such addressing is termed *multilevel* indirect addressing. The BPC implements single-level indirect addressing for all memory references except those of a single special case. That special case involves two levels and occurs during an interrupt. Indirect addressing is *not* a property of the memory; it is a property of the chips that use the memory. Any chip that is to implement instructions employing indirect addressing must contain special "gear works" for that purpose.

To indicate indirect addressing for a memory-reference instruction, bit 15 of that particular instruction will be set. During



```
[...] = OPTIONAL SPECIFIERS
/ INDICATES CHOICE OF SPECIFIERS

MEMORY REFERENCE
(M IS AN ASSEMBLY LANGUAGE LABEL, OR
EXPLICIT ADDRESS)
([,I] IS THE INDIRECT SPECIFIER)
LD% M [,I]
  LOAD A (OR B) FROM M.

CP% M [,I]
  COMPARE THE CONTENTS OF M WITH
  THE CONTENTS OF A (OR B); SKIP
  IF UNEQUAL.

AD% M [,I]
  ADD THE CONTENTS OF M TO A (OR B).

ST% M [,I]
  STORE THE CONTENTS OF A (OR B) IN
  M.

JSM M [,I]
  JUMP TO SUBROUTINE.  THE CONTENTS
  OF THE RETURN STACK REGISTER (R)
  ARE INCREMENTED BY ONE AND THE
  CONTENTS OF P STORED IN R,I.
  PROGRAM EXECUTION RESUMES AT M.

ISZ M [,I]
  INCREMENT M; SKIP IF M THEN
  EQUALS ZERO.

AND M [,I]
  LOGICAL "AND" OF A AND M; THE
  RESULT IS LEFT IN A.

DSZ M [,I]
  DECREMENT M; SKIP IF M THEN
  EQUALS ZERO.

IOR M [,I]
  INCLUSIVE "OR" OF A AND M;
  THE RESULT IS LEFT IN A.

JMP M [,I]
  JUMP TO M.  PROGRAM EXECUTION
  CONTINUES AT LOCATION M.

RET N [,P]
  RETURN.  A READ R,I OCCURS.  THAT
  PRODUCES THE ADDRESS (<P>) OF THE
  LATEST JSM THAT OCCURRED.  THE BPC
  THEN JUMPS TO ADDRESS <P> + N.  THE
  VALUE OF N MAY RANGE FROM -32 TO 31,
  INCLUSIVE.  AT THE CONCLUSION OF THE
  RET R IS DECREMENTED BY ONE.

EXE 0 ≤ M ≤ 37₈ [,I]
  EXECUTE REGISTER M.  THE CON-
  TENTS OF ANY REGISTER CAN BE
  TREATED AS THE CURRENT INSTRUC-
  TION, AND EXECUTED IN THE NORMAL
  MANNER.  THE NEXT INSTRUCTION
  EXECUTED WILL BE THE ONE FOLLOW-
  ING THE EXE M, UNLESS THE CODE IN
  M CAUSES A BRANCH.

SKIP
(-32<N<31, M WITHIN N OF *, * = CUR-
RENT VALUE OF P)
RZ% *±N/M      i.e., (* ± N)/(M), NOT *±(N/M)
  SKIP IF A (OR B) IS NOT ZERO.

RI% *±N/M
  SKIP IF A (OR B) IS NOT ZERO, THEN
  INCREMENT A (OR B).

SZ% *±N/M
  SKIP IF A (OR B) IS ZERO.

SI% *±N/M
  SKIP IF A (OR B) IS ZERO, THEN
  INCREMENT A (OR B).

SF% *±N/M
  SKIP IF FLAG LINE SET (OR CLEAR).

SD% *±N/M
  SKIP IF DECIMAL SET (OR CLEAR).

SS% *±N/M
  SKIP IF STATUS LINE SET (OR CLEAR).

SH% *±N/M
  SKIP IF HALT LINE SET (OR CLEAR).

ALTER
(IF EITHER S OR C IS PRESENT THE
TESTED BIT IS SET OR CLEARED AFTER
THE TEST)
SL% *±N/M [,S/,C]
  SKIP IF THE LEAST SIGNIFICANT
  BIT OF A (OR B) IS ZERO.

RL% *±N/M [,S/,C]
  SKIP IF THE LEAST SIGNIFICANT
  BIT OF A (OR B) IS NON-ZERO.

S%P *±N/M [,S/,C]
  SKIP IF A (OR B) IS POSITIVE.

S%M *±N/M [,S/,C]
  SKIP IF A (OR B) IS MINUS.

SO% *±N/M [,S/,C]
  SKIP IF OVERFLOW SET (OR CLEAR).

SE% *±N/M [,S/,C]
  SKIP IF EXTEND IS CLEAR (OR SET).

COMPLEMENT
TC%
  TWO'S COMPLEMENT A (OR B).

CM%
  COMPLEMENT A (OR B).  THE A (OR B)
  REGISTER IS REPLACED BY ITS ONE'S
  COMPLEMENT.

SHIFT-ROTATE
  (-32<N<31)
A%R N
  ARITHMETIC RIGHT SHIFT OF A (OR B).  A
  (OR B) IS SHIFTED RIGHT N PLACES WITH
  THE SIGN BIT (BIT 15) FILLING ALL
  VACATED BIT POSITIONS.

S%R N
  SHIFT A (OR B) RIGHT.  A (OR B) IS
  SHIFTED RIGHT N PLACES WITH ALL
  VACATED BIT POSITIONS VACATED.

S%L N
  SHIFT A (OR B) LEFT.  A (OR B) IS
  SHIFTED LEFT N PLACES WITH ALL
  VACATED BIT POSITIONS CLEARED.

R%R N
  ROTATE A (OR B) RIGHT.  A (OR B) IS
  ROTATED RIGHT N PLACES, WITH BIT 0
  ROTATING INTO BIT 15.
```

Fig. 17. BPC machine-instructions.

execution, the contents of the referenced location will be read and its entire 16-bit contents treated as the address of the final destination to be read from or written into.

Memory Reference Instructions and Page Addressing Machine instructions fetched from memory are 16-bit instructions. Some of those bits represent the particular type to which that instruction belongs. Other bits differentiate the instruction from others of the same type. If a BPC machine instruction is one that involves reading from, storing into, or otherwise manipulating the contents of a memory location, it is said to be a memory reference instruction. *Load into A* (LDA) and *store from B* (STB) are examples. Each memory reference instruction contains 10 bits to represent the address of the location that is to be referenced by the instruction. Those 10 bits represent one of $1{,}024_{10}$ locations on either the *base page* or the *current page* of memory; an additional bit (the B/C bit) in the machine instruction indicates which. As far as the processor is concerned, its base page is always a particular, nonchanging range of addresses that is exactly $1{,}024_{10}$ in number. A memory-reference machine instruction fetched from any location in memory (i.e., from any value of the program counter) may reference directly (i.e., without using indirect addressing) any location on the base page. The base-page addresses are 000000_8–000777_8 and 177000_8–177777_8.

The reason the base page was split was to provide a convenient means to ensure that half of it would be in ROM and half in R/W memory, without resorting to special decoding circuits. By separating the base page as described the desired division comes for free, simply by putting the right kind of memory at the right addresses.

What goes in a machine instruction's 10-bit address field is a displacement from some reference location, as an actual complete address has too many bits in it to fit in the instruction. Also, it is the responsibility of the assembler to control the B/C bit at the time the machine instruction is assembled. It does this easily enough by determining whether the address of the operand (or its "value") of an instruction is in the range 177000_8 through 177777_8 or 0 through 777_8.

For base-page references the 10-bit field is sufficient to indicate completely which of the 1,024 locations on the base page is to be referenced. The 32 register addresses are considered a part of the base page.

Current-page addressing refers to memory-reference instructions which reference a location which is not on the base page. Since there are more than 1,024 locations that are not the base page, the 10-bit field by itself is not enough to completely specify the exact location involved. Also, there are two types of current pages. Each type is also $1{,}024_{10}$ consecutive words in length. The value of P determines the particular collection of addresses that are the current page at any given time. This is done in one of two distinct ways, as determined by the signal called RELA. Depending upon RELA, the BPC is said to address memory in the *relative* mode or in the *absolute* mode. Both the BPC in the LPU and the BPC in the PPU operate in the relative addressing mode.

In the absolute mode of addressing the memory address space is divided into a base page and 64 possible current pages. The possible current pages are the consecutive $1{,}024_{10}$ word groups beginning with 000000_8. The possible current pages can be numbered 0 through 63_{10}. Thus, the "zero page" is addresses 000000_8–001777_8. Note that the base page is *not* the same as the zero page; the base page overlaps pages 0 and 63.

In relative addressing there are as many possible current pages as there are values of the program counter. In the relative addressing mode a current page is the 512_{10} consecutive locations prior to (that is, having lower-valued addresses than) the current location (value of P), and the 511_{10} consecutive locations following the current location.

During the execution of each memory-reference machine instruction referencing the current page, the BPC uses the value of the P register to form a full 16-bit address based on the 10 bits of address contained within the instruction. How the supplied 10 bits are manipulated before becoming part of the actual address, and how the remaining 6 bits are supplied, depends upon whether the addressing mode is relative or absolute. Base-page addressing requires different manipulation but is the same in either mode.

Subroutines The processor implements subroutines in the following way. The *Jump Subroutine* (JSM) instruction is used to cause a jump (change in value of P) to the start of the subroutine. The BPC saves the value of P that corresponds to the word of programming that is the JSM. That value is saved in a section of Read/Write memory called the *return stack.*

The return stack is a group of contiguous locations whose starting address less 1 was initially stored in the R register (in the BPC). Thus, R is an indirect pointer. What a JSM does is to increment the value in R and then use that new value as the address at which to store the value of P. Once this activity is complete, P is actually set to the address of the first word of the subroutine and its execution commences.

A subroutine is terminated with a RET n instruction. The essence of this instruction is to read the location that R points to, set P to that value plus n, and then decrement R. The most common return is a RET 1. Different values of n permit different returns corresponding to error or other special conditions. For instance, interrupt service routines are generally terminated with a RET 0.

Subroutines can be nested as deep as the size of the return stack will allow. The subroutines themselves can be in either ROM or Read/Write memory.

Flags The BPC is capable of branching based on the condition of each of four signals externally supplied to the chip. These signals are Decimal Carry (DC), Halt (HLT), Flag (FLG), and Status (STS). In the LPU the EMC acts as a source for Decimal Carry, which represents an overflow condition during certain arithmetic operations performed by the EMC. There is no EMC in the PPU, and the DC signal in the PPU is controlled by the CRT. It is used to indicate the duration of CRT retrace.

Bus Requests and Interrupts

Bus Request and Interrupt are two protocols that involve interchip communication. Bus Request ($\overline{\text{BR}}$) provides a way for a chip in the processor, or even a device external to the processor (such as the CRT), to request unfettered use of the IDA Bus. A signal called the Bus Grant (BG) is generated if all chips and any other interested entities agree to allow it. The requesting agency can use the IDA Bus for whatever purpose it wants (typically to do memory cycles). During the time that Bus Grant is in effect all chips suspend their activity. Bus Grant can be given even in the middle of the execution of an instruction. Because of this, the chips do not grant a Bus Request indiscriminately. Furthermore, a Bus Grant not requested by the IOC is used by the IOC to create Extended Bus Grant (EXBG), which is routed from chip to chip in a definite order; chips or other entities not at the top of the chain can exercise the right not to pass along the signal. This allows a Bus Request from the IOC to have a higher priority than any entity farther down the chain. Even if both are requesting the Bus, the IOC can "steal" EXBG by not passing it along. Farther down the chain from the IOC, BG serves to indicate only that the IDA Bus is being granted to somebody; a particular requesting device must wait until it sees EXBG before it can use the bus.

An entity on the Bus may ground BG as long as BG is not already being given. This allows any entity anywhere on the chain to protect its own access to the Bus against all agencies. Further, the BPC itself refuses to issue a BG as long as any memory cycle is in progress.

During an instruction fetch a line called *interrupt* ($\overline{\text{INT}}$) can signal the other chips to which the IOC has agreed to allow an interrupt requested by a peripheral. The management of this decision is complicated, but once the decision is made, the IOC signals the BPC with $\overline{\text{INT}}$. This has to occur during a certain period of time ending with the end of the instruction fetch. (A signal called SYNC identifies the instruction fetch.)

What the chips in the system must do when an interrupt occurs is to abort the execution of the instruction just fetched (it will be fetched again, later). The $\overline{\text{INT}}$ causes the BPC to execute the instruction JSM 10_8-indirect in place of the fetched instruction. Register address 10_8 is located in the IOC, and is the Interrupt Vector register (IV). That register is a pointer into a stack of indirect addresses for the starting locations for the various interrupt service routines. These routines handle the traffic needed by the interrupting peripheral. A special mechanism in the IOC sets the bottom four bits of IV to correspond to the select code or peripheral address of the particular peripheral that requested the interrupt. Thus IV points to different service routines, according to which peripheral has interrupted.

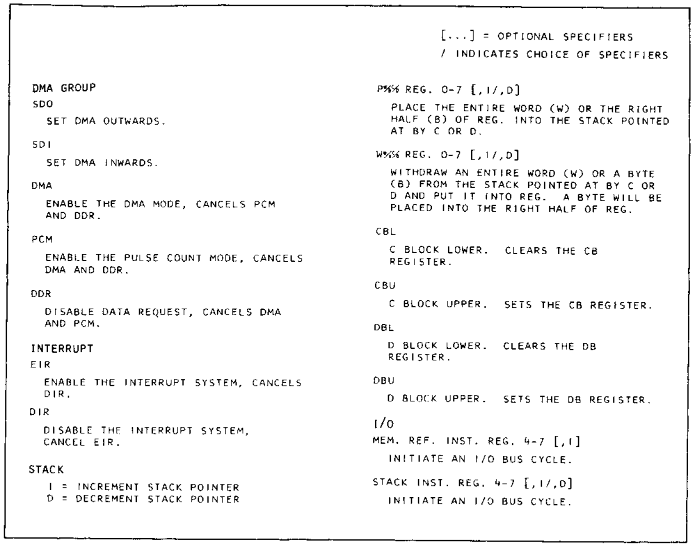
The JSM 10_8-indirect causes the value of P for the aborted instruction to be saved on the return stack. A RET 0 at the end of the service routine results in that very instruction's being fetched over again, at the conclusion of the service routine.

General Description of the IOC

The IOC has two main functions. One is to manage the transfer of information between the processor and peripheral devices. This is done by providing capabilities classified as Standard I/O, Interrupt, and Direct Memory Access (DMA). The second main function is to provide machine instructions allowing software management of stacks in Read/Write memory. Figure 18 is a condensed description of the machine instructions in the IOC's repertoire.

General Information about I/O The IOC allows up to 16 peripheral devices to be present at one time. Each peripheral device is connected to the I/O Data (IOD) Bus, Peripheral Address Bus, and the various control signals necessary for that particular device's operation. Individual I/O operations (exchanges of single words) occur between the processor and one peripheral at a time, although interrupt and DMA modes of operation can cause automatic interleaving of individual operations. A select code transmitted by the Peripheral Address Bus ($\overline{\text{PAB0}}$–$\overline{\text{PAB3}}$) identifies which of the 16 devices is the object of an individual I/O operation.

In addition, the peripheral interface is the source of the Flag and Status bits for the BPC instructions SFS, SFC, SSS, and SSC. Since there can be many interfaces, but only one each of Flag and Status, only the interface addressed by the select code is allowed to ground these lines. Their logic is such that if the addressed peripheral is not present on the I/O Bus, Status and Flag are logically false.



```
DMA GROUP                                      [...] = OPTIONAL SPECIFIERS
SDO                                            / INDICATES CHOICE OF SPECIFIERS
  SET DMA OUTWARDS.
SDI
  SET DMA INWARDS.
DMA
  ENABLE THE DMA MODE, CANCELS PCM
  AND DDR.
PCM
  ENABLE THE PULSE COUNT MODE, CANCELS
  DMA AND DDR.
DDR
  DISABLE DATA REQUEST, CANCELS DMA
  AND PCM.
INTERRUPT
EIR
  ENABLE THE INTERRUPT SYSTEM, CANCELS
  DIR.
DIR
  DISABLE THE INTERRUPT SYSTEM,
  CANCEL EIR.
STACK
  I = INCREMENT STACK POINTER
  D = DECREMENT STACK POINTER

P%% REG. 0-7 [,I/,D]
  PLACE THE ENTIRE WORD (W) OR THE RIGHT
  HALF (B) OF REG. INTO THE STACK POINTED
  AT BY C OR D.
W%% REG. 0-7 [,I/,D]
  WITHDRAW AN ENTIRE WORD (W) OR A BYTE
  (B) FROM THE STACK POINTED AT BY C OR
  D AND PUT IT INTO REG.  A BYTE WILL BE
  PLACED INTO THE RIGHT HALF OF REG.
CBL
  C BLOCK LOWER.  CLEARS THE CB
  REGISTER.
CBU
  C BLOCK UPPER.  SETS THE CB REGISTER.
DBL
  D BLOCK LOWER.  CLEARS THE DB
  REGISTER.
DBU
  D BLOCK UPPER.  SETS THE DB REGISTER.
I/O
MEM. REF. INST. REG. 4-7 [,I]
  INITIATE AN I/O BUS CYCLE.
STACK INST. REG. 4-7 [,I/,D]
  INITIATE AN I/O BUS CYCLE.
```

Fig. 18. IOC machine-instructions.

$\overline{\text{IC1}}$ and $\overline{\text{IC2}}$ are two control lines that are sent to each peripheral interface by the IOC. The state of these two lines during the non-DMA transfer of information can be decoded to mean something by the interface. Just what "something" will be is subject to agreement between the firmware designer and the interface designer; it can be anything they want, and it might not be the same for different interfaces.

I/O Bus Cycles The IOC's repertoire contains no machine instructions dedicated to I/O operations. That is, there is no specific "output instruction," and no specific "input instruction." Existing machine instructions cause I/O by referencing certain register addresses that cause I/O *bus cycles*. An I/O bus cycle is an exchange of a word between the IDA Bus and IOD Bus, via the Peripheral BIBs. The peripheral involved is specified by the contents of the 4-bit PA register, which controls the peripheral address lines. I/O bus cycles are termed read or write I/O bus cycles, depending upon whether information is being read from or written into a peripheral. Each of the three modes of I/O operation (Standard I/O, Interrupt, and DMA) utilizes I/O bus cycles. The explanation of the various modes of I/O amounts to showing different ways to initiate I/O bus cycles.

I/O bus cycles *do not* involve handshake. It is the responsibility of the firmware not to initiate an I/O bus cycle involving a device that is not ready. To do so will result in lost data, and there will be no warning that this has happened.

Standard I/O Standard I/O is I/O that has been explicitly programmed by the system programmer, using explicit assembly language coding. Standard I/O involves three activities:

1 Setting the peripheral address (in the PA register)

2 Investigating the status of the peripheral

3 Initiating an I/O Bus Cycle

During standard I/O operation, an I/O bus cycle is initiated by any machine instruction that incorporates a reference to one of addresses R4 through R7 ("in" the IOC). One way that can be done is with a BPC memory-reference instruction: for instance, STA R4 (for a write cycle), or LDA R4 (for a read cycle). However, there are no addresses R4 through R7. The use of addresses 4–7 is just a signal to the IOC to initiate an I/O bus cycle. Each different address produces a different combination of $\overline{IC1}$ and $\overline{IC2}$.

The Interrupt System When the processor grants an interrupt, the program segment currently being executed is automatically suspended, and there is an automatic JSM to an *interrupt service (sub)routine* that corresponds to the device that has interrupted. The service routine uses Standard I/O to accomplish its task.

The IOC allows two levels of interrupt, and has an accompanying two levels of priority. Priority is determined by select code: select codes $0–7_8$ are the lower level (priority level 1), and select codes $10_8–17_8$ are the higher level (priority level 2). Within a priority level all devices are of "equal" priority, and operation is on a first-come–first-served basis; a level-1 device cannot be interrupted by another level-1 device, but only by a level-2 device. However, priorities are not equal in the case of simultaneous requests by two or more devices on the same level. In such an instance the device with the higher-numbered select code has priority. With no interrupt service routine in progress, any interrupt will be granted.

Devices request an interrupt by grounding one of two interrupt request lines ($\overline{IRL}$ and $\overline{IRH}$—one for each priority level). The IOC determines the requesting select code by means of an interrupt poll. If the IOC grants the interrupt, it saves on an internal stack the existing select code located in PA, puts the interrupting select code in PA, and does a JSM-Indirect through an interrupt table to get to the interrupt service routine. [The top of this stack *is* the Peripheral Address register (PA-11_8).] The stack is deep enough to hold the select code in use prior to any interrupts, plus the select codes for two levels of interrupt.

It is the responsibility of the firmware to maintain an *interrupt table* of 16 consecutive words, starting at some Read/Write Memory address whose four least significant bits are 0s. The words in the interrupt table are set to the starting addresses of the various interrupt service routines for use with the 16 different select codes. When a peripheral is allowed to interrupt, its select code is used to determine which interrupt service routine to jump to. The interrupt service routine then handles the I/O operations needed by the interrupting device.

The firmware must also store the address of the first word of the interrupt table in the IV register (Interrupt Vector register, address 10, located in the IOC). Those contents will merge with the interrupting select code to produce the address of the appropriate table entry. A two-level indirect jump is used to arrive at the interrupt service routine. This happens automatically, because the BPC aborts its instruction fetch and generates a JSM IV, 1 as part of what it does during an interrupt, and because the IOC forces the BPC to do two consecutive "first-level" indirect accesses.

It is difficult to say specific things about interrupt service routines in general; much depends upon the particulars of the host software system. The next few paragraphs examine some generalities relating to interrupt service routines.

The first observation is on the number of service routines. In general, there is not a single service routine for each select code, or even for each type of peripheral. The usual case is collections of routines that perform related functions within the needs of a certain class of peripheral activity; each class of activity has its own collection.

For instance, it is unlikely that there will be a single interrupt service routine for a disk. On the customer's level there are many commands in the disk's operating system. On the firmware level there are a series of routines that perform "fundamental units" of activity, where each fundamental unit involves some amount of I/O. Most commands in the user's disk operating system are made up of a series of these fundamental units of activity. Fundamental units of activity for the disk are things like moving the head to a given track, reading a given sector from a track into such and such a buffer, and writing from such and such a buffer into a given sector.

Assume a fairly involved user's command for a disk is to be performed, one that requires reading the directory on the disk to determine the location of a certain file on the disk and then loading that file into memory. The series of routines here include moving the head to the start of the directory, reading through the information in the directory sector by sector until the information about the desired file is found, moving the head to the file's location, reading its header, reading its first sector, etc.

Each service routine is told or already knows which service routine follows it for the particular high-level task at hand, and if it has a choice based on the way events turn out (error conditions etc.), it knows how to handle that, too. As each new step in the sequence requiring a different interrupt service routine is reached, the concluding routine changes the appropriate entry of the interrupt table to the starting address of the next service routine. In this way a versatile collection of interrupt service routines can serve many purposes.

The computer can be almost anywhere in its internal coding when an interrupt is granted. Since the code is suspended, with JSM, it is obvious that the way to get back to the right spot is with a RET 0,P. (The ,P instructs the IOC to return to the select code in use prior to the interrupt.) But it will do no good to come back if the items in memory related to the routine are not the same. *The interrupt service routine must save and later restore any memory location that will be directly or indirectly disturbed by the activity of the service routine.* This could include the extend and overflow registers of the BPC, decimal carry and shift-extend of the EMC, and possibly CB and DB of the IOC.

The entire interrupt system can be turned off by a DIR machine instruction. After this instruction is given the IOC will refuse to grant any interrupts whatsoever until the interrupt system is turned back on with EIR. While the IOC will not grant any interrupts, the RET 0,P works as usual so that interrupt service routines may be safely terminated, even while the interrupt system is turned off.

Direct Memory Access Direct memory access is a means to exchange entire collections of data between memory and peripherals. Such a collection must be a series of consecutive memory locations. Once started, the process is automatic; it is done under control of hardware in the IOC, and regulated by the interface.

The DMA process can transfer data in two ways: single words are transferred one at a time, on a cycle-steal basis; or strings of words are transferred consecutively in a burst mode. In either instance data are transferred one word at a time. To transfer a word, a peripheral signals the IOC, which then requests control of the IDA Bus with $\overline{\text{BR}}$. That results in an external halt in all other system activity on the bus for the duration of the peripheral's request for DMA service. Herein lies the difference between burst mode and cycle-steal operation: in cycle-steal operation the peripheral ceases to request service after one word is transferred, and requests service again when ready, while in the burst mode the request is held to allow a series of high-speed consecutive transfers to occur.

During a DMA transfer of a collection of data, the IOC knows the next memory location involved, whether to input or output, which select code to use, and (possibly) whether or not the transfer of the entire collection is complete. This information is in registers in the IOC, which are set up by the firmware before the peripheral is told to begin DMA activity. After that, actual transfers are initiated at the request of the interface.

The DMA process is altogether independent of the operation of standard I/O and of the interrupt system and, except for theft of the IDA Bus for memory cycles, does not interfere with them in any way.

The four least significant bits of DMAPA specify the select code which is to be the peripheral side of the DMA activity. During an I/O bus cycle given in response to a DMA data request, the content of the PAB Lines will be determined by the four least significant bits of DMAPA rather than by the PA register.

Name	*Address*	*Meaning*
DMAPA	*(=13)*	*DMA peripheral address*
DMAMA	*(=14)*	*DMA memory address*
DMAC	*(=15)*	*DMA count*
DMAD	....	*DMA direction*

DMAMA is set to the address of the first word in the block to be transferred. This is the lowest-numbered address; after each transfer DMAMA is automatically incremented by the IOC. A separate one-bit register (DMAD) exists to specify the direction of the transfer; DMAD is controlled by its own set and clear machine instructions and is not addressable.

DMAC can, if desired, be set to $n - 1$, where n is the number of words to be transferred. During each transfer the count in DMAC is decremented. During the last transfer DMAC goes negative and the IOC automatically generates signals which the interface can use to recognize the last transfer. In the case of a transfer of unknown size, DMAC should be set to a very large count, to thwart the automatic termination mechanism. In such cases it is up to the peripheral to identify the last transfer.

Once the control registers are set up, a "start DMA" command is given to the interface through standard programmed I/O. The "start DMA" command is an output I/O bus cycle with a particular combination of $\overline{\text{IC1}}$, $\overline{\text{IC2}}$, and (perhaps) a particular bit pattern in the transmitted word. The patterns themselves are subject to agreement between the firmware designer and the interface designer. Sophisticated peripherals using DMA in both directions will have two start commands, one for input and one for output. It is also possible that other information can be encoded in the start command (the number of words to be transferred, for instance).

Stack Operations A stack that is implemented in firmware is simply a series of consecutive memory locations accessed indirectly through a pointer. The entries in the stack do not change their physical locations in the memory during additions and deletions. Instead, the value of the pointer is incremented or decremented.

The IOC implements some firmware stack-manipulation machine instructions. Two registers are provided as stack pointers: C and D. There are eight place and withdraw instructions for putting things into stacks and getting them out. Furthermore, the place and withdraw instructions can handle full 16-bit words, or pack 8-bit bytes in words of a stack. And last, there are provisions for automatic incrementing and decrementing of the stack pointer registers, C and D.

The mnemonics for the place and withdraw instructions are easy to decipher. All place instructions begin with P, and all withdraw instructions begin with W. The next character is a W or

B, for word or byte. The next character is either a C or D, depending upon which stack pointer is to be used. There are eight combinations, and each is a legitimate instruction.

The place and withdraw instructions outwardly resemble the memory reference instructions of the BPC: a mnemonic followed by an operand that is understood as an address, followed by an optional, I or, D. The range of values that the operand may have is restricted, however. The value of the operand must be between 0 and 7, inclusive. Thus, the place and withdraw instructions can place from, or withdraw into; the first eight registers. These are A, B, P, R, *and R4 through R7.* Therefore, the place and withdraw instructions can initiate I/O bus cycles; *they can do I/O.*

Regardless of which of ,I (increment) or ,D (decrement) is specified, a place instruction will do the increment or decrement of the pointer prior to the actual place operation. Withdraw instructions do the increment or decrement after actual withdraw operation. The reason for this is that it always leaves the stack with the pointer pointing at the new "top of the stack," and allows intermixing of place and withdraw instructions without adjustment of the pointer.

Because the stack in memory is composed of words rather than bytes, some means is required to extend the addressing of the pointer registers to include designation of bytes within the addressed word.

Left-right indication of bytes is accomplished with a signal called $\overline{\text{BL}}$. $\overline{\text{BL}}$ (Byte Left Not) is in turn controlled by bit 0 of either the C or D register. Sixteen-bit addressing is maintained by providing an additional 1-bit register for use with each stack pointer register. The nonaddressable registers are called CB (C Block) and DB (D Block). They are designated *block* because, as the most significant bit of the word pointer value, they divide the address space into two halves, or blocks. It is unfortunate that this terminology was chosen (it was done before the MAE was developed). Do not confuse those blocks with block 0 through block 3 of the Memory Address Extension scheme.

During the automatic increment or decrement to the pointer register, CB and DB function as most significant seventeenth bits of their respective registers. An advantage of having the bit that designates the byte be the least significant bit is that it simplifies the process of arithmetic computation upon byte addresses.

The CB and DB registers can be set to their initial values by machine instructions for setting and clearing each register. For instance, DBU (D Block Upper) sets the DB register; CBL (C Block Lower) clears the CB register.

General Description of the EMC

The Extended Math Chip (EMC) provides 15 instructions. Eleven of these operate on BCD-coded 3-word mantissa data. Two operate on blocks of data of from 1 to 16 words. One is a binary multiply and one clears the Decimal Carry (DC) register. A condensed description of these machine instructions is shown in Fig. 19.

Unless specified otherwise, the contents of registers A, B, SE, and DC are not changed by the execution of any of the EMC's instructions.

AR1 is the label of the 4-word arithmetic register located in R/W memory, locations 177770_8 through 177773_8. The assembler predefines the symbol AR1 as address 177770_8.

AR2 is the label of a 4-word arithmetic accumulator register located within the EMC, and occupying register addresses 20_8 through 23_8. The assembler predefines the symbol AR2 as address 20_8.

SE is the label for the 4-bit shift-extended register, located within the EMC. Although SE is addressable and can be read from and stored into, its primary use is as internal intermediate storage during those EMC instructions that read something from, or put something into, A0–A3. The assembler predefines SE as 24_8.

DC is the mnemonic for the 1-bit decimal-carry register located within the EMC. DC is set by the carry output of the decimal adder. Sometimes DC is part of the actual computation, as well as being a repository for overflow. In such cases the initial value of DC affects the result. However, DC will usually be zero at the beginning of such an instruction. The firmware sees to that by various means. DC does not have a register address. Instead, it is the object of the BPC instructions SDS and SDC (Skip if Decimal Carry Set and Skip if Decimal Carry Clear) and the EMC instruction CDC (Clear Decimal Carry).

It takes a special mechanism to handle BCD numbers. Done in firmware alone, such a mechanism would be slow and cumbersome. The EMC supplies some useful operations on portions of BCD floating-point numbers. This trims the mechanism in size and speeds it up significantly.

The EMC can perform operations on 12-digit BCD-encoded floating-point numbers. Such numbers occupy 4 words of memory, and the various parts of a number are put into specific portions of the 4 words, as shown in Fig. 20. The exponent and mantissa signs (E_s and M_s, respectively) are encoded as 0 and 1 for positive and negative, respectively. All the digits D_1 through D_{12} are encoded in BCD, while the exponent is a 10-bit signed 2's complement number. D_1 is the most significant digit, and D_{12} is the least significant digit. A decimal point is assumed to exist between D_1 and D_2.

Except for intermediate results within the individual arithmetic operations, D_1 will never be 0 unless the entire number is 0. Sometimes, after each individual arithmetic operation the answer needs to be normalized; that is, the digits of the answer need to be shifted toward D_1 until D_1 is no longer 0. The exponent then needs to be adjusted to reflect the change.

An important consideration concerning BCD arithmetic, as implemented by the processor, is that mantissas are represented

THE FOUR-WORD GROUP

CLR N $0 \rightarrow \text{LOCATION} < A >$
$0 \rightarrow \text{LOCATION} < A > + 1$
∘
∘
∘
$0 \rightarrow \text{LOCATION} < A > + N - 1$

XFR N $\text{LOCATION} < A > \rightarrow \text{LOCATION} < B >$
$\text{LOCATION} < A > + 1 \rightarrow \text{LOCATION} < B > + 1$
∘
∘
∘
$\text{LOCATION} < A > + N - 1 \rightarrow \text{LOCATION} < B > + N - 1$

THE MANTISSA SHIFT GROUP

MRX MANTISSA RIGHT SHIFT OF AR1 r-TIMES, $r = < B_{0-3} >$, AND $0 \leq r \leq 17_8 = 15_{10}$.

1st SHIFT: $< A_{0-3} > \rightarrow D_1; \ldots . < D_i > \rightarrow D_{i+1}; \ldots . D_{12}$ IS LOST

jth SHIFT: $0 \rightarrow D_1; \ldots . < D_i > \rightarrow D_{i+1}; \ldots . D_{12}$ IS LOST

rth SHIFT: $0 \rightarrow D_1; \ldots . < D_i > \rightarrow D_{i+1}; \ldots . < D_{12} > \rightarrow A_{0-3}; 0 \rightarrow DC; 0 \rightarrow A_{4-15}$

MRY MANTISSA RIGHT SHIFT OF AR2 $< B_{0-3} >$-TIMES. OTHERWISE IDENTICAL TO MRX.

MLY MANTISSA LEFT SHIFT OF AR2 ONE TIME.

$< A_{0-3} > \rightarrow D_{12}; \ldots < D_i > \rightarrow D_{i-1}; \ldots . < D_1 > \rightarrow A_{0-3}; 0 \rightarrow DC; 0 \rightarrow A_{4-15}$

AT THE CONCLUSION OF THE OPERATION SE EQUALS $< A_{0-3} >$.

DRS MANTISSA RIGHT SHIFT OF AR1 ONE TIME.

$0 \rightarrow D_1; \ldots . < D_i > \rightarrow D_{i+1}; \ldots . < D_{12} > \rightarrow A_{0-3}; 0 \rightarrow A_{4-15}$

AT THE CONCLUSION OF THE OPERATION SE EQUALS $< A_{0-3} >$.

NRM NORMALIZE AR2. THE MANTISSA DIGITS OF AR2 ARE SHIFTED LEFT UNTIL $D_1 \neq 0$. IF THE ORIGINAL D_1 IS NON-ZERO, NO SHIFTS OCCUR. IF TWELVE SHIFTS OCCUR, THEN AR2 EQUALS ZERO, AND NO FURTHER SHIFTS ARE DONE. THE NUMBER OF SHIFTS IS STORED AS A BINARY NUMBER IN B.

THE ARITHMETIC GROUP

CMX TEN'S COMPLEMENT OF AR1. THE MANTISSA OF AR1 IS REPLACED WITH ITS TEN'S COMPLEMENT, AND DC IS SET TO ZERO.

CMY TEN'S COMPLEMENT OF AR2. THE MANTISSA OF AR2 IS REPLACED WITH ITS TEN'S COMPLEMENT, AND DC IS SET TO ZERO.

CDC CLEAR DECIMAL CARRY. CLEARS THE DC REGISTER; $0 \rightarrow DC$.

FXA FIXED-POINT ADDITION.

DURING THE ADDITION THE EXPONENTS ARE NOT CONSIDERED, AND ARE LEFT STRICTLY ALONE. THE SIGNS ARE ALSO LEFT COMPLETELY ALONE.

$$\begin{array}{rl}
< AR1 > = & D_1 \quad D_2 \quad D_3\text{--------}D_{12} \\
< AR2 > = & D_1 \quad D_2 \quad D_3\text{--------}D_{12} \\
+ & \qquad\qquad\quad < DC > \leftarrow \text{INITIAL VALUE OF DC} \\
\hline
(\text{OVERFLOW}) \rightarrow \text{"}D_0\text{"} & D_1 \quad D_2 \quad D_3\text{--------}D_{12} \rightarrow AR2 \\
\searrow & DC\ (\text{FINAL VALUE OF DC})
\end{array}$$

<A>, <B>, ETC., DENOTE THE CONTENTS OF THE INDICATED REGISTER

MWA MANTISSA WORD ADD.

DURING THE ADDITION THE EXPONENTS ARE NOT CONSIDERED, AND ARE LEFT STRICTLY ALONE, AS ARE THE SIGNS. MWA IS INTENDED PRIMARILY FOR USE IN ROUNDING ROUTINES.

$$\begin{array}{rl}
 & 0\text{-------}0 \quad \overbrace{D_9 \quad D_{10} \quad D_{11} \quad D_{12}}^{< B >} \\
< AR2 > = & D_1\text{-------} \quad D_9 \quad D_{10} \quad D_{11} \quad D_{12} \\
+ & \qquad\qquad\qquad\quad < DC > \leftarrow \text{INITIAL VALUE OF DC} \\
\hline
(\text{OVERFLOW}) \rightarrow \text{"}D_0\text{"} & D_1\text{--------} D_9 \quad D_{10} \quad D_{11} \quad D_{12} \rightarrow AR2 \\
\searrow & DC\ (\text{FINAL VALUE OF DC})
\end{array}$$

FMP FAST MULTIPLY. THE MANTISSAS OF AR1 AND AR2 ARE ADDED TOGETHER (ALONG WITH DC AS D_{12}) $< B_{0-3} >$-TIMES; THE RESULT ACCUMULATES IN AR2.

THE REPEATED ADDITIONS ARE LIKELY TO CAUSE SOME UNKNOWN NUMBER OF OVERFLOWS TO OCCUR. THE NUMBER OF OVERFLOWS THAT OCCURS IS RETURNED IN A_{0-3}.

FMP IS USED REPEATEDLY TO ACCUMULATE PARTIAL PRODUCTS DURING BCD MULTIPLICATION. FMP OPERATES STRICTLY UPON MANTISSA PORTIONS; SIGNS AND EXPONENTS ARE LEFT STRICTLY ALONE.

$$< AR2 > + ((< AR1 >) \cdot (< B_{0-3} >)) + \overset{\text{REPRESENTS THE INITIAL VALUE OF DC. AFTERWARDS, DC = 0}}{DC} \rightarrow AR2$$

$$0 \rightarrow DC, \qquad 0 \rightarrow A_{4-15} \qquad\qquad \#\ \text{OF OVERFLOWS} \rightarrow A_{0-3}$$

MPY BINARY MULTIPLY USING BOOTH'S ALGORITHM. THE (BINARY) SIGNED TWO'S COMPLEMENT CONTENTS OF THE A AND B REGISTERS ARE MULTIPLIED TOGETHER. THE THIRTY-TWO BIT PRODUCT IS ALSO A SIGNED TWO'S COMPLEMENT NUMBER, AND IS STORED BACK INTO A AND B. B RECEIVES THE SIGN AND MOST-SIGNIFICANT BITS, AND A THE LEAST-SIGNIFICANT BITS:

$$< A > \cdot < B > \rightarrow < \overset{\frown}{B > < A} >$$

FDV FAST DIVIDE. THE MANTISSAS OF AR1 AND AR2 ARE ADDED TOGETHER UNTIL THE FIRST DECIMAL OVERFLOW OCCURS. THE RESULT OF THESE ADDITIONS ACCUMULATES INTO AR2. THE NUMBER OF ADDITIONS WITHOUT OVERFLOW (n) IS PLACED INTO B.

$$< AR2 > + < AR1 > + < DC > \rightarrow AR2 \quad (\text{REPEATEDLY UNTIL OVERFLOW})$$

THEN

$$0 \rightarrow DC, \qquad 0 \rightarrow B_{4-15}, \qquad n \rightarrow B_{0-3}$$

FDV IS USED IN FLOATING-POINT DIVISION TO FIND THE QUOTIENT DIGITS OF A DIVISION. IN GENERAL, MORE THAN ONE APPLICATION OF FDV IS NEEDED TO FIND EACH DIGIT OF THE QUOTIENT.

AS WITH THE OTHER BCD INSTRUCTIONS, THE SIGNS AND EXPONENTS OF AR1 AND AR2 ARE LEFT STRICTLY ALONE.

Fig. 19. EMC machine-instructions.

FLOATING-POINT DATA FORMAT.

ADDRESS	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	E_s	TWO'S COMPLEMENT EXPONENT									EMPTY					M_s
M + 1	D_1				D_2				D_3				D_4			
M + 2	D_5				D_6				D_7				D_8			
M + 3	D_9				D_{10}				D_{11}				D_{12}			

THE INTERNAL FLOATING POINT REPRESENTATION OF .003587219 (= 3.587219 x 10^{-3}).

ADDRESS	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0
M + 1	0011				0101				1000				0111			
M + 2	0010				0001				1001				0000			
M + 3	0000				0000				0000				0000			

Fig. 20. Floating-point data format.

in a sign/magnitude format. Ten's complements are used by the firmware in the computational processes, but only as an intermediate step. Furthermore, it is done in such a way that the automatic generation of the correct sign of a sum does not occur. There is also the frequent need to recomplement an answer.

AR2 frequently functions as an accumulator for EMC operations on BCD numbers, much as the A and B registers are accumulators for the instructions ADA and ADB.

V. Memory Address Extension

General Considerations

The essence of a memory address extention scheme is the concatenation of additional upper address bits to the addresses sent to memory by the processor. A variety of schemes have been devised, and many are not unlike the one to be described. In particular, the use of registers to specify the values of the additional bits is very common. Simple schemes simply always use the contents of such a register to expand the address. More flexibility than this was needed for the 9845A.

It was recognized that certain kinds of memory contents would always be grouped together. That is, the main operating system (whose code is in ROM), the various user's programs (in R/W), operating system data, user's data, and option ROM coding are all occupants of groups of memory disparate but contiguous within themselves. Furthermore, these separate collections frequently need access to each other. The occasions when operating-system code wishes to access the user's program, or when the user's program wishes to access the user's data, are occasions when it would be desirable to have some sort of automatic mechanism for changing the values of the additional address bits. Not only would this save a lot of code (and execution time) otherwise used for manipulating the contents of the address extension registers, but it can also provide an external structure useful in organizing the architecture of the internal software system.

The key features of the memory address extension scheme explained below are these. First, there are several registers used to determine the values of the additional address bits. There is a means to identify the purpose for which a memory cycle is being performed: instruction fetch, indirect reference, base-page reference, etc. Each such purpose can invoke different registers, each providing different and previously determined additional address bits. Note that this is not done simply on a machine instruction–to–machine instruction basis. The process is automatic on a memory cycle–by–memory cycle basis. This is a very important distinction because it allows programmers to let the MAE hardware do the work for them as their program runs, freeing them from constantly giving machine language instructions to a less automatic address extension device.

Second, the MAE hardware is responsive to the most significant bit of the address produced by the processor. By controlling the value of this bit (at programming and assembly time for direct references and at run time via programmer algorithms for indirect accesses), the programmer can signal the MAE hardware whether the additional address bits are to be selected according to the various registers mentioned above, or are to be selected from among fixed and predetermined values. (It could as easily have been from a second collection of additional registers, but this added level of flexibility was deemed unnecessary for the 9845A.) In this way, code executing at addresses in one-half the processor's address space can easily access data in the other half—but the two halves of the processor's address space are represented by a preselected range of memory addresses, on the one hand, and by an arbitrary range of memory address anywhere in memory, on the other. This is of great utility in an operating system whose controlling programming has to be able to quickly access memory anywhere in the system, or in a system where code to be executed can be located anywhere in memory.

As shown in Fig. 8, the computer has a memory with 128 kilowords, yet each processor has the inherent ability to address only 64 kilowords. On the surface it might seem that each processor handles half the memory, but that is not so. Instead, the memory is divided into four 32-kiloword blocks.

The LPU's 64-kiloword address space is split into two 32-kiloword blocks, as shown in Fig. 21. The Memory Address Extender (MAE) embodies a set of conventions to dynamically determine which blocks make up the two halves of the processor's address space. These conventions involve the processor's most significant addresses bit, the type of memory cycle (i.e., for what purpose—instruction fetch, indirect reference, etc.), and the contents of some additional registers in the MAE. Those registers are R34, R35, and R37 (each is named for its octal address). These each have two bits. The size of the registers is related to the number of blocks managed by the MAE; in principle those registers could be 16 bits each, allowing a possible 64K blocks of 32 kilowords each.

System programmers have exclusive control of the contents of R34–R37. In this way they can control what particular blocks are accessed as the MAE implements its conventions.

The memory address extension scheme is performed for the LPU only. The address space for the PPU is exactly 64K. It just so happens that the bottom half of that address space is the same physical memory that the LPU calls block 1, and that the upper half is the same as what the LPU calls block 0. This arrangement is somewhat arbitrary and was chosen for convenience in coordinating LPU and PPU activities. Bear in mind that the PPU has no connection with the MAE. The function of the MAE is, in principle, altogether separate from the notion of having the processors share memory. If the computer had only the LPU, it would still (presumably) have the MAE. Also, the problems

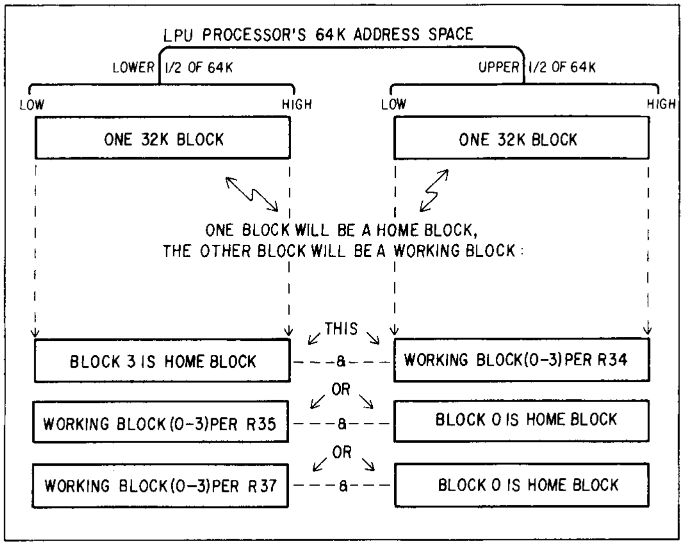

LPU PROCESSOR'S 64K ADDRESS SPACE
LOWER 1/2 OF 64K
UPPER 1/2 OF 64K
LOW
HIGH
LOW
HIGH
ONE 32K BLOCK
ONE 32K BLOCK
ONE BLOCK WILL BE A HOME BLOCK,
THE OTHER BLOCK WILL BE A WORKING BLOCK:
THIS
BLOCK 3 IS HOME BLOCK
&
WORKING BLOCK(0-3)PER R34
OR
WORKING BLOCK (0-3)PER R35
&
BLOCK 0 IS HOME BLOCK
OR
WORKING BLOCK(0-3)PER R37
&
BLOCK 0 IS HOME BLOCK

Fig. 21. Block addressing structure implemented by the memory address extender (MAE).

arising from both processors' trying, at the same time, to access block 0 or block 1, and the subsequent need for a dual-port memory controller, are not related to memory address extension.

Basic Principles

The LPU's processor, in terms of its internal architecture and operation, knows absolutely nothing of the memory address extension scheme. Regardless of how many blocks are implemented by the MAE, the LPU understands only a single 64K address space. Yet it is typical for a memory-reference machine instruction for the BPC (refer to Fig. 17) to be fetched from (i.e., located in) one block while its operand (the location in memory referenced) is in a different block. Such an instance requires automatic block switching by the MAE during the execution of the memory-reference instruction. Figure 22 illustrates the various conditions under which the various blocks are accessed.

An understanding of Fig. 22 requires the notion of home blocks and working blocks. A home block is a block that is always the accessed block whenever some particular condition is met. The various home block designations are fixed and cannot be changed. (The foregoing does not mean that certain blocks are always home blocks. Rather, particular circumstances always access certain blocks as home blocks. But any block can also be accessed as a working block, too.) A working block is one that is designated according to the contents of R34–R37. The circumstances which determine which block is the home block also determines which of R34 through R37 is used to identify the working block.

As an example, block 3 is the home block for instruction fetches, while R34 designates the working block for instruction fetches. In other words, the programmer can execute code in block 3 by accessing it as home block, or, execute code out of any other block by setting R34 to its block number and accessing that code as working-block code.

THESE CONDITIONS PREVAIL: FOR THESE TYPES OF MEMORY CYCLES:	THE MAE LISTENS TO THE NATURE OF THE MEMORY CYCLE TRAFFIC AND IMPLEMENTS THESE BLOCK ALLOCATIONS:		
	HOME BLOCK IS DESIGNATED BY:	HOME BLOCK IS:	WORKING BLOCK IS DESIGNATED BY:
ALL INSTRUCTION FETCHES, ALL LINK-POINTER FETCHES FOR INDIRECT REFERENCES, AND ALL BPC DIRECT REFERENCES	ADDRESS BIT 15=0	3 LOWER 1/2 BASE PAGE IN BLOCK 3	UPPER 1/2 BASE PAGE AUTOMATICALLY IN BLOCK 0 R34
IOC AND EMC MEMORY REFERENCES, AND BPC INDIRECT FINAL DESTINATION FETCHES	ADDRESS BIT 15=1	0	R35
BUS GRANT (TESTER)	ADDRESS BIT 15=1	0	R37

Fig. 22. Table of simplified MAE operation.

Figure 22 shows that there are three different categories of memory cycles: instruction fetches etc.; IOC and EMC memory references etc.; and bus grants. The MAE listens to the nature of the traffic on the IDA Bus and constantly classifies it according to these categories. Each category can result in an access to either the permanently associated home block or the programmer-designated working block. The most significant bit of the address determines which. That address bit was programmer-controllable at the time the code being executed was assembled.

Some Special Considerations

Observe that, by its address, the upper half of the LPU's base page has the form of a working-block reference. It would appear that there could be four different upper halves, one for each setting of R34. However, in this operating system it is inconvenient to have multiple instances of the upper half of the base page. Accordingly, the MAE automatically routes all references to the upper half of the base page (which it recognizes by its very high addresses) onto block 0. The PPU, of course, has its own base page. See Fig. 23.

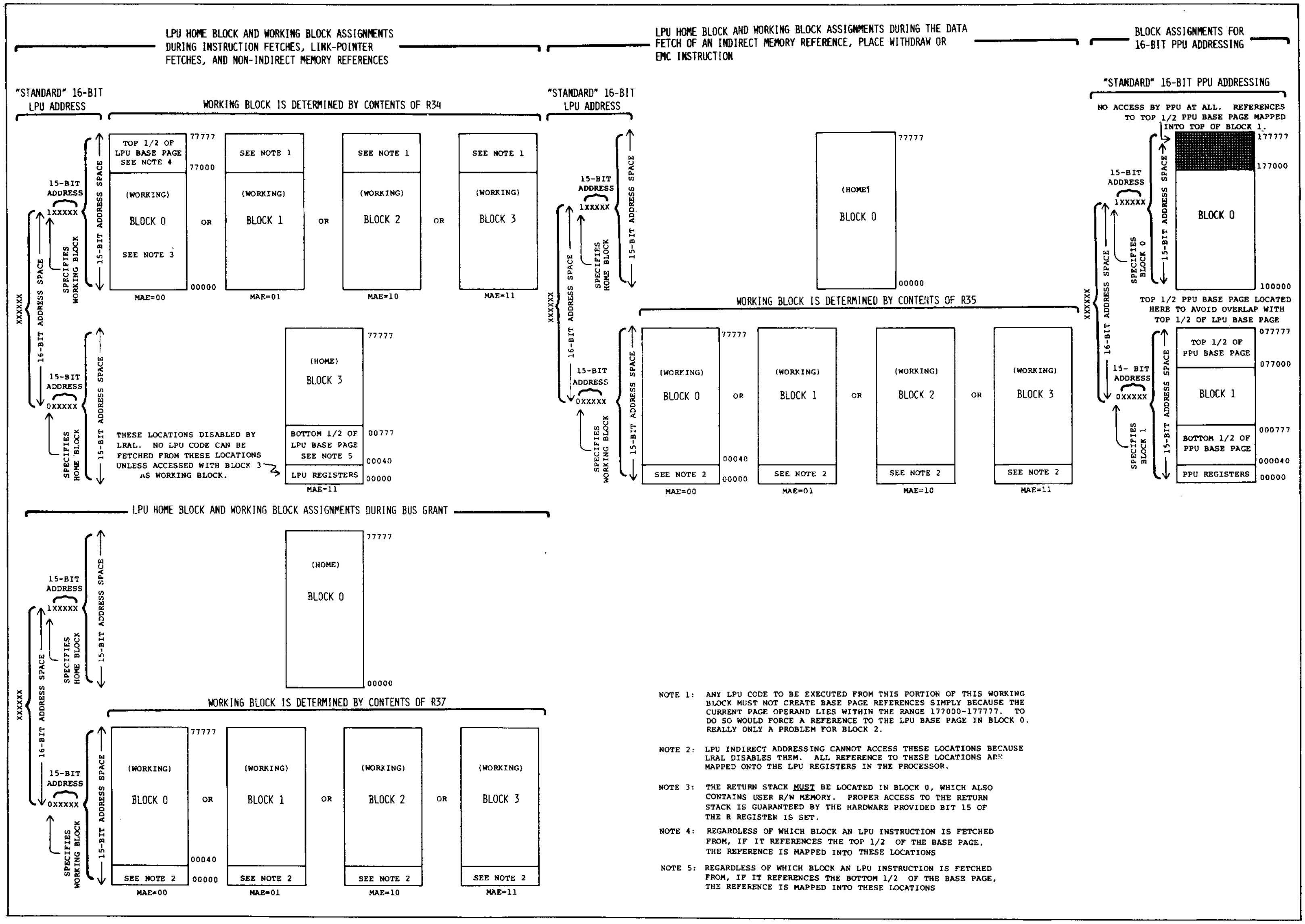
Whenever any part of the system addresses a location whose address falls within the range 0–37_8, inclusive, the BPC generates a signal called RAL. This line is used by the bulk memory to prevent itself from responding; this allows the physical location of those addresses to be distributed throughout the system. This causes no problem with R34-type block allocations, as in these cases the addressing space occupied by the registers maps into the home block; and for any block allocation there is only ever one home block. But for register references via indirect addressing, or by the IOC or EMC, some wasted physical memory locations result because it is the working block that has the address space of the registers. So those locations, *in each block*, cannot be accessed. A similar condition exists for R37-type block allocations. These and other details of MAE operation are shown in Fig. 23.

These facts summarize MAE operation in the absence of a bus grant:

1 The MAE knows which memory cycles are instruction fetches.

2 If an instruction is not a BPC memory-reference instruction, its associated memory accesses are done thus:
   a Home block is block 0.
   b Working block is determined by R35.
   c Bit 15 equal to 1 implies home block; bit 15 equal to 0 implies working block.

3 If an instruction is a BPC memory-reference instruction, its associated memory accesses are done thus:


LPU HOME BLOCK AND WORKING BLOCK ASSIGNMENTS DURING INSTRUCTION FETCHES, LINK-POINTER FETCHES, AND NON-INDIRECT MEMORY REFERENCES
"STANDARD" 16-BIT LPU ADDRESS
WORKING BLOCK IS DETERMINED BY CONTENTS OF R34
15-BIT ADDRESS
1XXXXX
SPECIFIES WORKING BLOCK
XXXXXX
16-BIT ADDRESS SPACE
15-BIT ADDRESS SPACE
TOP 1/2 OF LPU BASE PAGE SEE NOTE 4
(WORKING)
BLOCK 0
SEE NOTE 3
77777
77000
00000
MAE=00
OR
SEE NOTE 1
(WORKING)
BLOCK 1
MAE=01
SEE NOTE 1
(WORKING)
BLOCK 2
MAE=10
SEE NOTE 1
(WORKING)
BLOCK 3
MAE=11
15-BIT ADDRESS
0XXXXX
SPECIFIES HOME BLOCK
(HOME)
BLOCK 3
BOTTOM 1/2 OF LPU BASE PAGE SEE NOTE 5
LPU REGISTERS
77777
00777
00040
00000
MAE=11
THESE LOCATIONS DISABLED BY LRAL. NO LPU CODE CAN BE FETCHED FROM THESE LOCATIONS UNLESS ACCESSED WITH BLOCK 3 AS WORKING BLOCK.
LPU HOME BLOCK AND WORKING BLOCK ASSIGNMENTS DURING BUS GRANT
(HOME)
BLOCK 0
77777
00000
WORKING BLOCK IS DETERMINED BY CONTENTS OF R37
(WORKING)
BLOCK 0
SEE NOTE 2
77777
00040
00000
MAE=00
(WORKING)
BLOCK 1
SEE NOTE 2
MAE=01
(WORKING)
BLOCK 2
SEE NOTE 2
MAE=10
(WORKING)
BLOCK 3
SEE NOTE 2
MAE=11
LPU HOME BLOCK AND WORKING BLOCK ASSIGNMENTS DURING THE DATA FETCH OF AN INDIRECT MEMORY REFERENCE, PLACE WITHDRAW OR EMC INSTRUCTION
"STANDARD" 16-BIT LPU ADDRESS
(HOME)
BLOCK 0
77777
00000
WORKING BLOCK IS DETERMINED BY CONTENTS OF R35
(WORKING)
BLOCK 0
SEE NOTE 2
77777
00040
00000
MAE=00
(WORKING)
BLOCK 1
SEE NOTE 2
MAE=01
(WORKING)
BLOCK 2
SEE NOTE 2
MAE=10
(WORKING)
BLOCK 3
SEE NOTE 2
MAE=11
BLOCK ASSIGNMENTS FOR 16-BIT PPU ADDRESSING
"STANDARD" 16-BIT PPU ADDRESSING
NO ACCESS BY PPU AT ALL. REFERENCES TO TOP 1/2 PPU BASE PAGE MAPPED INTO TOP OF BLOCK 1.
SPECIFIES BLOCK 0
BLOCK 0
177777
177000
100000
TOP 1/2 PPU BASE PAGE LOCATED HERE TO AVOID OVERLAP WITH TOP 1/2 OF LPU BASE PAGE
15- BIT ADDRESS
SPECIFIES BLOCK 1
TOP 1/2 OF PPU BASE PAGE
BLOCK 1
BOTTOM 1/2 OF PPU BASE PAGE
PPU REGISTERS
077777
077000
000777
000040
00000

NOTE 1: ANY LPU CODE TO BE EXECUTED FROM THIS PORTION OF THIS WORKING BLOCK MUST NOT CREATE BASE PAGE REFERENCES SIMPLY BECAUSE THE CURRENT PAGE OPERAND LIES WITHIN THE RANGE 177000-177777. TO DO SO WOULD FORCE A REFERENCE TO THE LPU BASE PAGE IN BLOCK 0. REALLY ONLY A PROBLEM FOR BLOCK 2.

NOTE 2: LPU INDIRECT ADDRESSING CANNOT ACCESS THESE LOCATIONS BECAUSE LRAL DISABLES THEM. ALL REFERENCE TO THESE LOCATIONS ARE MAPPED ONTO THE LPU REGISTERS IN THE PROCESSOR.

NOTE 3: THE RETURN STACK **MUST** BE LOCATED IN BLOCK 0, WHICH ALSO CONTAINS USER R/W MEMORY. PROPER ACCESS TO THE RETURN STACK IS GUARANTEED BY THE HARDWARE PROVIDED BIT 15 OF THE R REGISTER IS SET.

NOTE 4: REGARDLESS OF WHICH BLOCK AN LPU INSTRUCTION IS FETCHED FROM, IF IT REFERENCES THE TOP 1/2 OF THE BASE PAGE, THE REFERENCE IS MAPPED INTO THESE LOCATIONS

NOTE 5: REGARDLESS OF WHICH BLOCK AN LPU INSTRUCTION IS FETCHED FROM, IF IT REFERENCES THE BOTTOM 1/2 OF THE BASE PAGE, THE REFERENCE IS MAPPED INTO THESE LOCATIONS

Fig. 23. The PPU address space compared to the extended LPU address space.

a Current-page nonindirect references are almost always made to the same block the instruction was fetched from.

b Base-page nonindirect references are made to the particular part of the base page specified.

c Block 3 contains the lower half of the base page and block 0 has the upper half, regardless of which working block is specified.

d For indirect references the link pointer is accessed according to whether it is on the current page or on the base page, as described above, but the access to the final destination location is made according to the block allocation rules for IOC and EMC instructions.

These facts summarize memory access during a bus grant:

1 The MAE remembers which block allocation scheme was suspended in order to do the bus grant and will correctly restore the suspended mode when that activity is completed.

2 During a bus grant:
a Home block is block 0.
b Working block is determined by R37.
c Bit 15 equal to 1 implies home block; bit 15 equal to 0 implies working block.

VI. Description of the Display System

General Description

The display is a dual raster-scan CRT display. A 12-in, high-resolution, magnetic-deflection CRT is used to provide adequate viewing area for high-quality alphanumeric and graphic information. In the alphanumeric mode, up to 25 lines of 80 characters can be displayed at one time from a standard 128-character ASCII character set. A foreign character set can be added, as an option, to allow the user to display either French, Spanish, German, or Katakana. Other languages are also possible. Three methods of highlighting information are available to the user: inverse video, underlining, and blinking. Each of these functions can be independently changed on a character-by-character basis. The viewing area for 25 lines of 80 characters, called the alpha raster, is approximately 9.3-in. by 4.8-in. This permits a matrix of 720 × 375 dots to be displayed. Characters are formed from 7 by 9 dot matrices located in 9 by 15 dot fields.

High-resolution raster graphics can be added to the display as an option. In the graphics mode of operation, the viewing area, called the graphics raster, is approximately 7.9 in by 6.5 in. This permits a matrix of 560 × 455 dots to be displayed. The graphics raster is a separate, independent raster that is switched into operation when the display is in the graphics mode. The dual raster-scan capability allows the size and aspect ratio of each raster to be chosen to optimize the quality and capability of the display for the function the user wishes to perform, and to achieve compatibility with the internal thermal printer/plotter.

Display Quality

A considerable emphasis was placed on optimizing the design to achieve a high-quality display. To achieve high quality in a CRT display requires the optimization of many parameters. Some of the most important include character size and legibility, brightness, resolution, contrast, glare, focus, position distortion, and stability. Display quality was one of the major reasons for adding the dual raster-scan capability. The alpha raster is tailored to display 80 adequately spaced characters per line, while using the maximum width possible without introducing excessive distortion due to nonuniformity in the CRT screen. A 7 by 9 character font in a 9 by 15 cell was chosen because this matrix is sufficient to generate aesthetically pleasing characters. The extra rows in the cell are used for spaces, ascenders that are needed for some of the European characters, and descenders that are used in some of the lowercase Roman characters.

The graphics raster displays the same high-quality characters but is limited to 62 per line. The graphics raster increases the resolution in the vertical dimension to maximize the proportion of screen area that can be used.

Uniform character size over the entire screen is difficult to achieve in CRT displays. Nonlinear current drives must be supplied to the yoke because the faceplate is not spherical. To achieve a more accurate current waveform, an active correction technique was employed in the display. The yoke current is compared to a reference current generated by a diode function generator and is corrected when a difference occurs. With this scheme, an improvement factor greater than 2 was achieved in the position distortion.

Since visible motion on a display is quite annoying, it was decided to refresh the display at 60 Hz even when the line frequency is 50 Hz to minimize flicker. Sufficient magnetic shielding has been added to eliminate interference due to internal sources within the mainframe itself, as well as from reasonable external magnetic fields.

In the graphics mode of operation the CRT is treated as a genuine peripheral with a select code and driven via the IOD Bus. This capability is briefly considered at the end of this section.

In contrast, the alphanumeric interface is a dedicated mechanism that automatically generates the CRT's display according to the contents of memory. It is connected to the PPU's IDA Bus and performs its own accesses to memory. Thus, to generate a display, the PPU needs only to format and manage the contents of a CRT display buffer in block 1 memory. The alphanumeric interface uses bus requests to interrogate that buffer, and responds to certain conventions regarding control bytes that are placed into the buffer amid the data by the controlling firmware.

The control bytes and their associated conventions amount to a command set for the alphanumeric display. Their employment allows efficient use of the memory allocated to the CRT display buffer. Rather than structuring the buffer to be a character-for-character image of the display, the buffer contains a compacted version of the data. For instance, the blanks to the right of a line are supplied automatically by the display itself, following an end-of-line (EOL) control character. Other control bytes instruct the alphanumeric interface where in the buffer to begin the display; control the location of the cursor; and specify underlining or blinking.

The size chosen for the display buffer is large enough to contain enough characters to fill an entire display. But because of efficient allocation of memory (e.g., suppression of trailing blanks by EOL control characters) the buffer is rarely full and can be loaded with more lines of information than the CRT can display at one time. The display buffer can hold four pages of average BASIC statements. The controlling firmware can cause the display to scroll through the data in the buffer in response to the operator's pressing various control keys on the keyboard. Scrolling requires only the manipulation of a few control bytes, not the wholesale rearrangement of data in the display buffer.

Alpha Display Control Logic

The Control Logic is the alphanumeric interface between the mainframe and the display. It reads memory via DMA, processes the data, and holds them in a format that the display can use. Each byte of a data word represents either a combination of features to be set or cleared, an ASCII character, or a control code for the display. Figure 24 shows the functions that can be interpreted from each byte.

Data bytes consist of a 7-bit ASCII code and a high-order 0, and they will be interpreted as the corresponding ASCII code unless the foreign character set has previously been chosen. If the high-order bit is set, the five low-order feature bits are latched and held until another feature byte occurs to change the state.

The EOL command fills the remainder of the current line buffer (one of two local buffers within the CRT) with blanks. The next data byte will be the first character of the next line of displayed characters.

During normal operation, the Control Logic will read the data at address 70000_8, complement them, and use that as the address for the first character to be displayed. From that point on the address will be incremented by 1 for each new data word. The NWA command indicates that the contents of the next address are to be interpreted as the complement of the address for the next data word. The address will then be incremented from the new point.

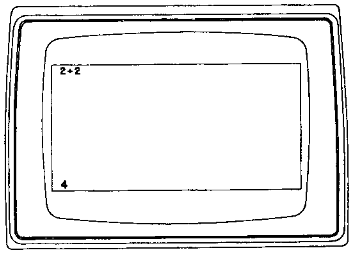
In addition to being the pointer of the first word of a page, 70000_8 is also used to choose between the alpha mode and the graphics mode. If the high-order bit is a 0 and the graphics hardware is installed, then the display will be the graphics mode.

An example of an alpha mode data pattern is shown in Fig. 25.

As each data byte is processed, the data are placed into a 12-bit word. The first 7 bits contain the ASCII code for the displayed character. The last 5 bits indicate whether any ongoing highlighting should be applied to this character. These feature bits were previously specified by a control code, whereupon they were latched by the Control Logic and were applied to every character until the latches were changed or cleared. These 12-bit words are stored in groups of 80 in one of two local line buffers within the

BIT # WITHIN BYTE								
7	6	5	4	3	2	1	0	MEANING
0	X	X	X	X	X	X	X	DATA
1	0	0	0	0	0	0	0	CLEAR ALL FEATURE LATCHES
1	0	X	X	X	X	X	0/1	CLEAR/SET THE CURSOR LATCH
1	0	X	X	X	X	0/1	X	CLEAR/SET THE INVERSE VIDEO LATCH
1	0	X	X	X	0/1	X	X	CLEAR/SET THE BLINKING LATCH
1	0	X	X	0/1	X	X	X	CLEAR/SET THE UNDERLINE LATCH
1	0	X	0/1	X	X	X	X	CLEAR/SET THE FOREIGN CHAR. SET LATCH
1	1	X	X	X	X	X	1	END OF LINE COMMAND (EOL)
1	1	X	X	X	X	X	0	NEW WORD ADDRESS COMMAND (NWA)

Fig. 24. The alphanumeric mode control bytes.



MEMORY ADDRESS	MEMORY DATA	FUNCTION
70000	107776	ALPHA, FIRST ADDRESS=70001
70001	140000	NWA, N_u (IGNORED)
70002	107770	NEXT ADDRESS=70007
70007	100040	CLEAR, BLANK
70010	031053	2, +
70011	031301	2, EOL
70012	140000	NWA, N_u (IGNORED)
70013	107757	NEXT ADDRESS=70020
70020	140701	EOL, EOL
70021	140701	EOL, EOL
•	•	•
•	•	•
•	•	•
•	•	•
•	•	•
•	•	•
70032	140701	EOL, EOL
70033	140440	EOL, BLANK
70034	032301	4, EOL

TOTAL=20 WORDS

Fig. 25. Sample alphanumeric data pattern.

CRT. The purpose of having two line buffers is to provide the Display Logic with one full line of characters to display while the Control Logic is loading the next line of 80 characters into the other buffer. This means that the Control Logic is actually one line ahead of the display. When the Control Logic has entered 80 characters into a line buffer, it waits for the Display Logic to indicate that it is ready for a new line. The Control Logic provides the Display Logic with the newly filled Line Buffer and starts to refill the used Line Buffer with new data. This occurs for each character line on the display. As the Control Logic completes each line it signals the Display Logic that there is a full Line Buffer. The Display Logic cannot wait for a new line once one has been requested, or the data will not be displayed in the correct position on the screen. So if a new full Line Buffer is not available when the Display Logic indicates that it is ready for a new line, the Display Logic will blank the video for the remainder of the page. This is done because the Control Logic and Display Logic will not be synchronized until the beginning of the new page. The Line Buffer must be filled in 637 μs. This figure comes from the time it takes to display the 15 scans that make up the dot matrix of a line of characters. For each scan all 80 words in the buffer are read 15 times before the buffer is refilled.

The Display's Effect on the Mainframe

On account of the nature of the display's mode of data retrieval, there is a definite effect on the performance of the mainframe. Since it is necessary for the display to access memory on a regular basis, it uses memory cycles which might have been used by the PPU for other operations. This will inevitably slow down the PPU. The PPU can execute about 1 million memory cycles per second. The display must read at least one word for every two lines of characters (two blank lines) but doesn't need to read more than 82 words per line of characters (a feature byte and a character byte in every word with a new word address). If a character line is 40 words in length (80 characters or partial lines with features), the display will require 40 memory cycles/line × 25 lines/page × 60 pages/s, or 60,000 memory cycles per second. This would reduce the PPU to the use of 940,000 memory cycles per second, or a 6.0 percent increase in execution time. These memory cycles may also indirectly slow the LPU by temporarily holding the Dual Port Memory Controller in an inconvenient position, but that result is probably negligible.

Over a short term (less than 637 μs) the display will be accessing memory to fill a single line of characters. This rate is 158,000 memory cycles per second, which increases PPU execution time by 23 percent (PPU will be allowed 763,000 memory cycles per second). However, as soon as the line is complete, memory access drops to zero until the next line needs to be refreshed.

A conflict occurs when some peripheral device, such as a disk, attempts a burst-mode DMA and where the efficiency of the device depends upon a data transfer rate close to the maximum. The problem arises when the display requires a sufficient number of memory cycles to complete a character line in less than 637 μs while at the same time a disk requires data at a rate determined by the rotating speed of the disk. If the display is allowed memory cycles in such a DMA burst, a disk location might be past the head when the data finally arrive. Similarly, if the display is deprived of memory cycles during the burst, the analog scanning of the display might have started displaying a line before the digital circuitry has completely acquired and processed the next line from memory. To avoid this and allow for efficient use of disk systems the following convention has been adopted. If the display is deprived of enough memory cycles that it cannot fill a character line by the time that line starts to be scanned on the display, then the remainder of the video output for that page will be blanked. Video will be resumed at the beginning of the next instance of displaying that page. Therefore, it is possible for the display to be blank for about 0.3 s if a DMA occurs which reads 64 Kbyte of memory at once. A longer blanked period can occur if smaller DMAs occur regularly after the start of each refresh cycle.

Graphics Overview

Graphics-mode operations allows the generation of entirely arbitrary patterns on the CRT screen through the use of a separate graphics raster. The screen appears as a field 560 dots wide by 455 dots high. The CRT is equipped with an additional interface (select code 13) and a 16-kiloword cache memory. A correspondence between the bits in the cache memory and the dots on the screen is established. The user's software, with help from extra BASIC language constructs supplied by a GRAPHICS option ROM, can generate an image on the CRT by manipulating the contents of the cache memory.

The graphics mode of operation has its own cursors, including one for digitizing information presented on the screen. Also, the CRT need not be in the graphics mode for manipulation of the graphics memory to occur. The CRT display can be switched between the graphics image and the regular alphanumeric format at will.

An additional feature is the CRT–Thermal Printer dump. This was made possible by providing the ability to use the contents of the 16K cache memory as a source of data to drive the internal thermal printer. That printer has a thermal printhead with 560 uniformly spaced print resistors. The Graphics Dump produces a dot-for-dot image of the CRT's graphics-mode display on the printer.

References

Shaw [1974]

Section 3

Evolution of HP Calculators

Desk-top calculators present a total computation environment to the user. The syntax and semantics of all the keys are predefined. Individual keystrokes vary widely in power from simple addition to complex I/O operations. Further, support functions such as editing, debugging aids, syntax analyzing, incremental execution, and keyboard monitoring are not only completely defined but also locked into hardware. This is to be contrasted with computer systems whose instruction sets are specified and whose computational environment is defined by ever-evolving multiple layers of software.

This section focuses on the architectures of the Hewlett-Packard series of desk-top calculators, starting with the HP 9100A (c. 1968); its first-generation descendants, the HP 9810/20/30 (c. 1972); and its second-generation descendants, the HP 9815/35/45 (c. 1976). The series span the technology range from discrete components through MSI to LSI in the latest generation. The advances in technology have allowed costs to decrease while allowing functionality to increase. Performance has increased by a factor of 8, operating-system ROM by a factor of 25, and user RAM by a factor of 240. These advances are graphically displayed in Figs. 1 to 4.

These computers represent an unplanned family with no

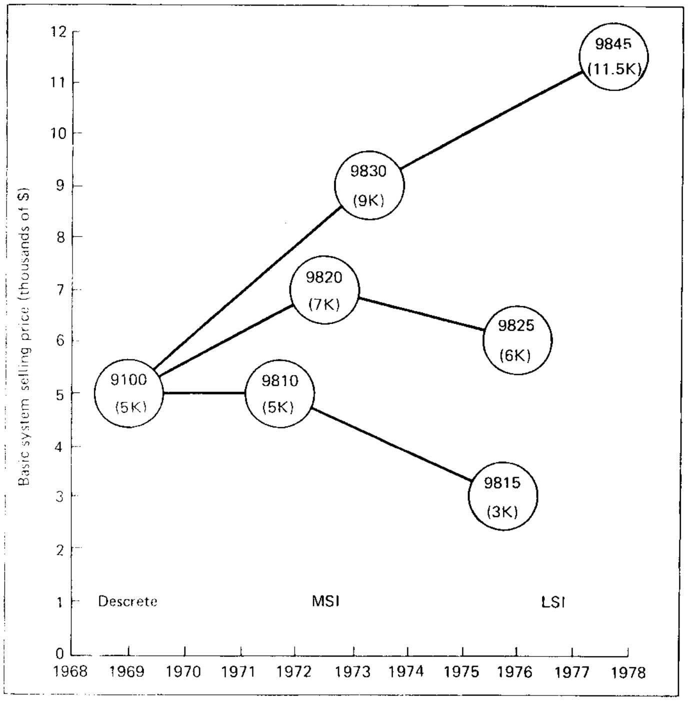


Fig. 1. Selling price (thousands of dollars) versus introduction date.

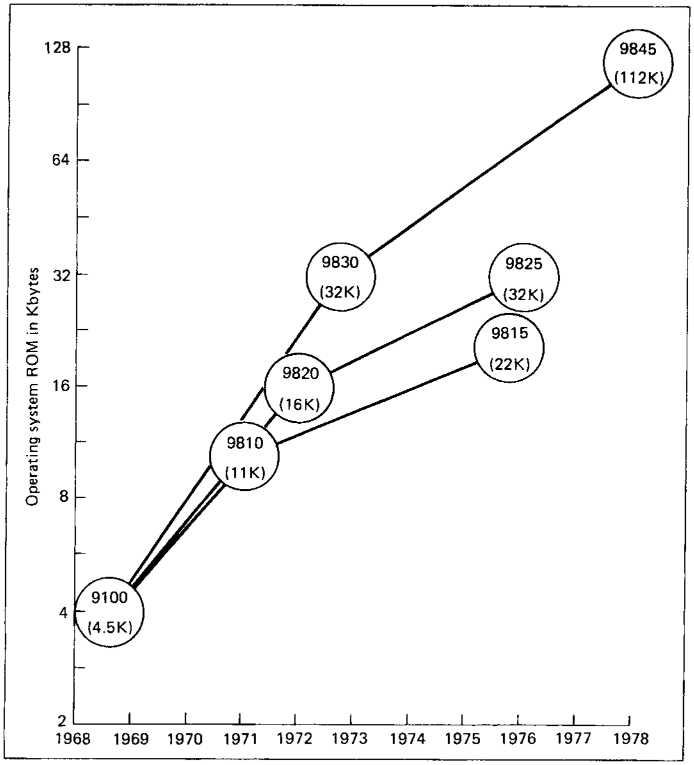


Fig. 2. ROM operating system (Kbytes) versus introduction date.

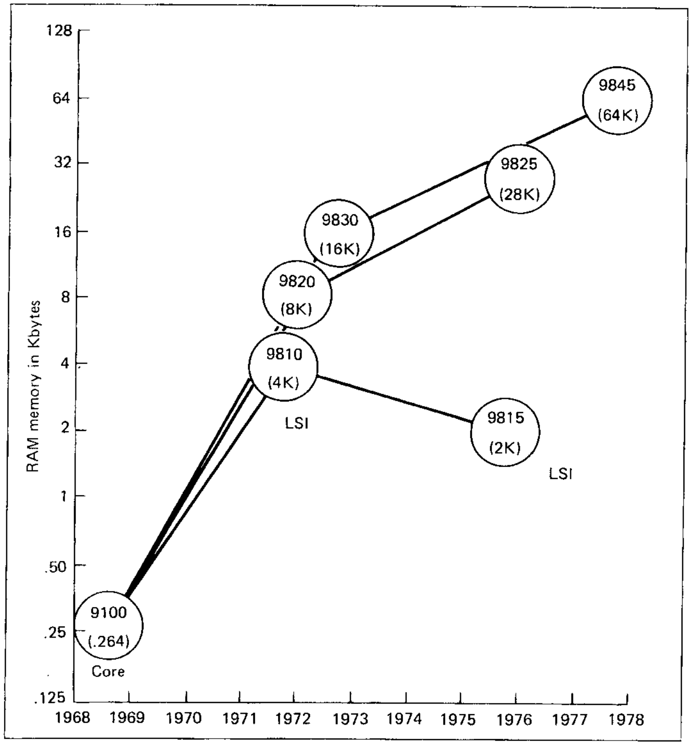


Fig. 3. RAM (minimum configuration) versus introduction date.

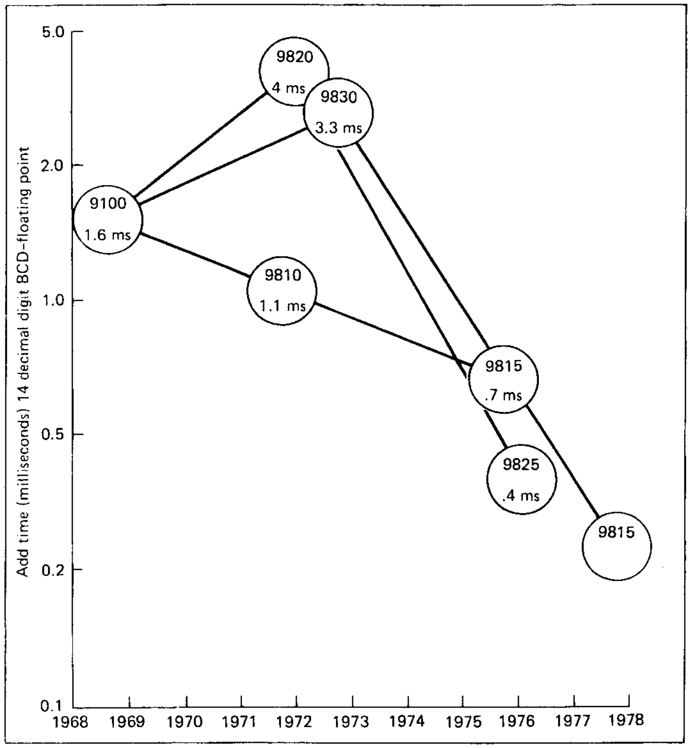


Fig. 4. Floating point add times versus introduction date.

constraint on user compatibility between generations. Chapters 48, 49, and 31 sketch the designs of the three major generations of Hewlett-Packard desk-top calculators. Functionality has increased with each generation and is best exemplified by the programming interface. HP 9100A programs consisted of arithmetic keystroke functions and program control operations (e.g., GO TO and IF). The HP 9800 series ranged from an algebraic language through BASIC, traditionally a computer-based language. The perception is that the HP 9810/20/30 series is matched to user functionality rather than software compatibility. The HP 9845 also supports BASIC. This section concludes, in Chap. 50, with some observations by Tom Osborne, one of the architects of the Hewlett-Packard desk-top series.