This paper discusses the design of an experimental character-processing computer for the interpretive execution of higher-level language programs.*

The design specifies a 100-nsec instruction cycle for the microprogram instructions stored in a read-only memory, a fast memory for intermediate "scratch pad" computation, and input/output through a conventional computer coupled to a 2-μsec main memory. The object program to be interpreted is stored in the main memory.

As a part of the research, the design was simulated on standard equipment.

A character computer for high-level language interpretation

by J. E. Meggitt

Interest in fast character-processing computers that are controlled by instructions in read-only memories is twofold:

- By using a design that permits the economical "writing" of microprograms into read-only memory—as, for example, by means of "pluggable" units—the apparent computer function can be modified to suit particular applications.
- The computer organization appears very attractive for higher-level language interpretation.

higher-level language

The design discussed here and shown schematically in Figure 1 was motivated by experimental interest in the problems of building a machine that interprets a high-level language.

A high-level language with sufficiently general properties was chosen to ensure that the resultant design would be suitable for other high-level languages that might evolve. This language[1] is FORTRAN-like, but with many extensions relative to (1) the variety of operators included, and (2) the generality with which arrays may be referenced through a concept of *structured* data.

* The research reported in this paper was sponsored in part by the Air Force Cambridge Research Laboratories, Office of Aerospace Research, under Contract AF19(628)-3257.

The notion of structured data can be explained by analogy with the traditional organization of text in a book. Information in a book is contained in a hierarchical structure, starting with the book itself, broken down into chapters, sections, paragraphs, sentences, words, and individual letters. Convenient reference to an item in the book can be made by referring to the item's position in the hierarchy. It is also possible and convenient to interpret literal reference to a particular part as reference to all of its sub-parts. Data is stored according to structure and is referred to by the high-level language in this manner. Consequently, an operation described as "A+B" may denote the addition of many pairs of data quantities if A and B are data items high in the hierarchy.

design considerations

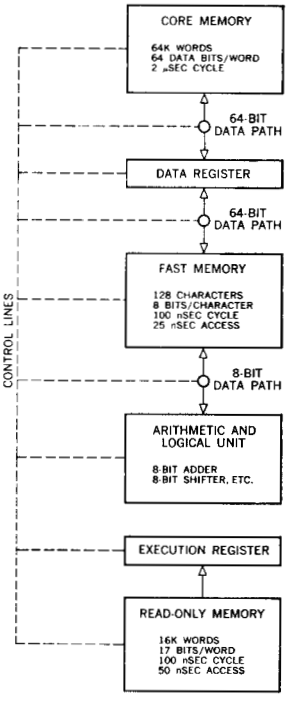
The original concept of microprogramming arose from the observation that certain gates in a computer are open at each instant of time, and that the control of the machine is specified by the sequence of signals sent to open the gates. The original microprogrammed machines held, in a suitable read-only memory, lists of gates to be opened at specified times, and they contained a mechanism for accessing the lists in sequence. However, this rather narrow concept broadened so that it now describes any machine having a somewhat simpler internal computer which executes interpretively the order code of the external machine. Thus, most machines involve a certain degree of microprogramming—as, for example, in executing multiplication by an internal "program" that controls a hardware adder.

A computer that "directly implements" a high-level language must perform many complicated operations that are clearly composite, i.e., composed of more elementary micro-operations. Therefore, it seems to be advantageous to directly implement a set of more restricted operations that represent the more complicated operations. In this way, a microprogrammed machine (according to the current definition) can be used for interpretive execution of high-level language programs.

The complex high-level language chosen is expressed by an 8-bit character set and conceived as machine independent. The machine word length is variable, and storage allocation is dynamic. Thus it is not possible to specify useful basic word operations, and it is necessary to process smaller basic units of data. Since the largest high-level language units with hardware identity are 8-bit characters, the value of an efficient character-processing machine is evident.

A 2-μsec conventional core memory is used to store high-level language programs. It was found that, on the average, about 50 basic character operations are required per core access, the specific ratio being determined by the particular language. Consequently, the character-processing machine must run as fast as possible to keep the logic processing in step with the core accessing. As a result, a simple character-processing machine was specified with a speed of one instruction execution per 100

Figure 1 Block diagram of system



nsec, the highest speed that currently seems possible.

control program store

It is clear that this high speed cannot be obtained if the control program for the character machine is stored in core, unless a complicated interleaved memory system is used. Therefore it is proposed to use a read-only memory for holding the instructions that constitute the control program. It is considered possible to build a read-only memory of sufficient size (16,000 words of 17 bits each) with a 50-nsec access time and a 100-nsec cycle time by using passive electrical networks. Such networks have propagation times measured in nanoseconds. A common arrangement involves a rectangular array of crossed drive and sense lines. At certain chosen intersections, the lines are coupled together by resistive, capacitive, or inductive coupling according to the particular design.

It should be observed that a read-only memory is entirely adequate for the present purpose, since the programs for interpreting the high-level language remain fixed. It is desirable that the read-only memory have its information set up initially by an automatic process which is controlled from punched cards or magnetic tape. This arrangement would make it possible to "write" into the memory an exact copy of a program that has been tested by simulation on another computer. To allow a change of application from time to time, it is also desirable that the read-only memory information be changeable with only modest effort.

working store

In addition to the control memory, the character machine needs a fast read-write memory to store temporary results and to act as a buffer between the relatively slow core memory and the character machine. This fast "scratch pad" may be small but should have a 100-nsec cycle time with an access time of 25 or 30 nsec. For a reasonably efficient interpretive high-level language program, a memory size of 128 8-bit characters would be sufficient. It is considered possible to build a thin magnetic film read-write memory to meet this specification. The actual memory specified by the design is word-organized and arranged to contain 16 words of 64 bits each. In the normal mode of operation, the addressing of a character causes the selection of one of the 16 words and of one of the eight characters within the selected word. However, it is also possible to address entire words in order to transfer them from the fast memory to the data register of the core memory and vice versa. This operation allows the fast memory to buffer the core memory. The core memory contains 72-bit words, but only 64 of these bits are used for data, the other eight being parity bits. Hence, the word lengths of the core memory and of the fast memory match.

memory addressing

The core memory, the read-only memory, and the fast scratch-pad memory are shown schematically in Figure 1. To complete the design, it is necessary to determine some data flow paths and to specify an instruction set. Since the machine discussed here is a character processor, and since eight bits are sufficient to express the character set, the data flow paths of the machine are generally

eight bits wide. The instruction format is of the simple single-address type, and the address field of each instruction is eight bits wide, as discussed below.

address indexing

The 128 characters of the fast memory can be addressed directly by the 8-bit field of an instruction, leaving one bit for future expansion. In many interpretive programs, however, it is necessary to scan successive characters, which is made possible by providing a set of index registers. In most cases, the scan is made over not more than eight characters (one word), since related characters usually originate in words that have been transferred, one word at a time, from the core to the fast memory. Accordingly, the length of the index register is three bits. For convenience, however, the indexing adder is provided with carries that carry into higher-order positions.

An examination of the interpretive programs proved that it would be desirable to scan several strings of characters simultaneously. For example, it is useful to scan the instruction string, two operand strings, and the resultant string at the same time. Thus, seven index registers were provided.

In addition to an 8-bit address field, each instruction has a 3-bit tag field. Except in one or two special cases, the effective address for an instruction consists of the specified address and the contents of the specified 3-bit index register. Of course, when some instructions use the address field to specify items other than addresses, the indexing logic is available.

read-only memory addressing

The read-only memory contains 16,000 words for implementing the high-level language. To reduce the amount of high-speed address decoding and to specify read-only memory addresses by 8-bit fields, the read-only memory is physically arranged in 64 *pages* (extendable to 256 pages) of 256 words each. The 8-bit word address is selected and decoded by instructions as they occur, but the page address is held in a 6-bit (or 8-bit) page register whose contents are relatively static.

Thus, only the word bits of an instruction address must be decoded at high speed. The page bits are, in general, already decoded. This arrangement makes it possible for an instruction to describe branching within a page by specifying only eight bits. Thus, the branch instruction can use the same address format as other instructions.

This simple arrangement speeds up the instruction fetching mechanism and saves instruction bits. Programs are written in such a manner that branches to different pages of the read-only memory occur infrequently. In general, each high-level operation or suboperation is implemented on a separate page. Thus a relatively cumbersome page changing mechanism does not make the programs unduly inefficient. For page changing, a special 6-bit register, the next-page register, is loaded before the branch instruction is encountered. The loading of this register sets a latch which is tested on every branch instruction. If the latch is not set, the branch instruction merely selects a new line on the cur-

rent page. If the latch is set, the contents of the next-page register are transferred to the page register, a new line is selected, and the latch is turned off. In this way, both a new page and a new line are selected. Due to the additional register-to-register transfer and the additional decoding, a branch instruction that changes pages takes 300 nsec for its execution instead of the usual 100 nsec.

The index registers play an important role in indexing the read-only memory as well as the fast memory. Thus these registers may be used advantageously to modify the line part of branch addresses, providing a means of branching in as many as eight ways.

core memory addressing

The core memory, a conventional 2-μsec core, contains up to 65,000 words of eight information characters each. Since this memory looks like a slow I/O device to the fast character machine, conventional ways of controlling I/O devices may be used to control the core memory. Thus the address of the word to be accessed is first set up in the 16-bit memory address register. A read or write instruction is then given, which starts the reading or writing cycle. This action causes an autonomous data transfer between the memory and its data register while the character machine proceeds to execute further instructions. During the transfer, the data register is locked against further instructions, so that any instruction that requests use of the data register immediately stops the character machine until completion of the transfer.

The presence of this simple interlock makes it possible to have core access instructions in the microprogram as close together as desirable, without causing errors. However, it is better to anticipate core memory accesses and then space them appropriately to avoid time loss due to waiting.

Instructions are provided that allow the contents of the core memory data register to be transferred, eight characters at a time, to one of the 16 word locations in the fast memory and vice versa. This flow of information between the core memory and the actual character machine is inhibited when the core memory is busy.

arithmetic and logical unit

All other logic is accomplished by an arithmetic and logical unit that processes pairs of 8-bit characters. In general, one operand is fed to the logical unit from the machine's 8-bit accumulator, and the other from either the address field of an instruction or the fast memory. The result is returned to the accumulator.

binary adder

A binary 8-bit adder contains a 1-bit carry store to hold the high-order output carry. Instructions are provided to allow the low-order input carry either to be forced or to come from the carry store. When a subtract instruction is decoded, one of the sets of inputs is 1's complemented as it is fed to the adder. To get a true subtraction, the low-order carry must be correctly programmed.

Use of the carry store allows correct adding or subtracting of strings of characters. The carry store, used also for comparing two numbers, may be tested in a conditional branch instruction. In

implementing the high-level language, the binary adder is used mainly for housekeeping and address calculations.

decimal adder

It is essential to work internally in decimal arithmetic because the high-level language includes logical operations on decimal digits, and the identity of the decimal digits must be preserved.

For ease of processing, a pair of decimal digits is packed into an 8-bit character inside the character machine. Each digit is represented in a binary-coded excess-three form, which allows the binary adder to work rather simply in a decimal mode. First, a binary addition is performed, and the carries out of the low-order four bits and the high-order four bits are inspected. The presence of a carry to the left of a 4-bit block causes addition of a 3 to the 4-bit binary result, and the absence causes subtraction of a 3. In this way, the decimal sum of two pairs of digits as well as the correct output carry are generated. A low-order input carry can be handled exactly as in the binary case.

The adding or subtracting of 3's does not require adders but merely logical functional changes, in a conditional way, on the right and left four bits of the binary sum. The logic for this process is trivial.

The mechanism for binary subtraction is employed to perform decimal subtraction. Thus, when a decimal subtract instruction is decoded, the 1's complement of one of the operands is fed to the adder and is operated in the decimal mode as above. This operation produces the difference of two pairs of digits, since the 1's complement is exactly the 99's complement in the chosen representation.

logic and shifting

The adder is used also to function as a logical unit. It can generate the logical *and* as well as the logical *or* of corresponding bits in two characters.

The arithmetic unit contains a shifter in which the contents of the 8-bit accumulator may be shifted up to eight bit positions in either direction. Attached to this shifter is a normalizing mechanism. It is possible to normalize a number in the accumulator by left-shifting or by right-shifting until either the most significant or the least significant digit is a 1. The shift count for this operation is placed into index register 7 where it is accumulated. A left normalize causes the contents of index register 7 to be incremented by the shift amount, and a right normalize causes the contents to be decremented. In this way, index register 7 can be made to contain the bit address of the first non-zero bit in the accumulator and, by repeating the normalize operation k times, the bit address of the kth non-zero bit.

Since there is no built-in multiplication, this facility is very helpful for programming the multiplication of two strings. For this purpose, a binary-coded character of the multiplier is placed in the accumulator. Multiples of a multiplicand character are stored in successive character locations of the fast memory and are referenced by index register 7. Each successive normalize operation causes index register 7 to point to the appropriate character multiple that must be added to the partial sum.

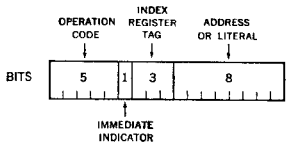
The normalize facility is also useful for a high-level language which uses a dynamic storage allocation scheme in which maps of the core store are kept in a special part of the store, one bit in the map representing one word. The normalize facility allows scanning of the maps in a very simple way.

instruction format

Instructions are held in the read-only memory, each instruction occupying 17 bits. It may seem strange that this number is not a multiple of 8. However, data and instructions, being held in separate memories, are fairly independent, so that a match between them is not essential. Some "immediate" instructions cause eight of the 17 bits to be used as data, and this amount of interconnection is sufficient.

The machine is organized as a single-address computer, and instructions have the format shown in Figure 2.

Figure 2 Instruction format



In general, an address is generated by adding to the address bits the contents of the index register that is specified by the tag.

If the immediate bit is 0, the effective address is used as the address of a character in the fast read-write memory. If the bit is set to 1, the indexed quantity is used as a literal. In an instruction such as an add or shift, this means that the literal quantity is added or that the shift is by a literal amount, whereas otherwise the add would be from the fast memory or the shift by an amount held in the fast memory.

For a branch instruction, an immediate bit 1 causes use of the indexed literal quantity as the "line" address of the read-only memory to which the branch is made. On the other hand, if the bit is set to 0, the line address is taken from an indexed location in the fast memory, thus obtaining an indirect-branch instruction. This instruction is essential since it is the means by which high-level language instruction strings, which appear as sequences of characters, are decoded.

internal instruction set

The instructions are listed in Table 1. AC indicates an 8-bit accumulator, X identifies an index register, CARRY means a 1-bit carry store, and C is a carry generated by the adder. Parentheses denote "contents of," and M denotes the contents of a fast memory character location or, when appropriate, a literal; the distinction is made in each instruction by means of the immediate bit. $\overline{M}$ denotes the 1's complement of M.

input/output

The character machine is not provided with any I/O instructions as such, but instead receives its information from a conventional computer which is coupled to the core memory. The core store of the character machine may be accessed by the conventional machine which has its own core memory as well as I/O channels and units. An interlock is provided for memory conflicts.

Mutual interrupt facilities are provided so that either machine can interrupt the other. Thus, the character machine has an additional instruction that causes the interruption of the other machine. This instruction is executed after the character machine has set up some control words in its core memory. The second machine inspects these words to determine its action.

Table 1 Instruction set

Type of operation	*Explanation of operation*
Clear and add	M → (AC); 0 → (CARRY)
Clear and add with carry	M + (CARRY) → (AC); C → (CARRY)
Clear and set	M → (AC); 1 → (CARRY)
Binary add	(AC) + M → (AC); C → (CARRY)
Binary subtract	(AC) + $\overline{\text{M}}$ → (AC); C → (CARRY)
Decimal add	(AC) + M → (AC); C → (CARRY)
Decimal subtract	(AC) + $\overline{\text{M}}$ → (AC); C → (CARRY)
And	(AC) *and* M → (AC)
Or	(AC) *or* M → (AC)
Store	(AC) → M
Right shift	(AC) shifted right M places → (AC)
Left shift	(AC) shifted left M places → (AC)
Load index register	M → (X)
Save index register	(X) → M
Increment and test index*	(X) + 1 → (X); if adder overflows, that is, if former (X) = 111, then branch to line M
Decrement and test index*	(X) − 1 → (X); if adder does not overflow, that is, if former (X) = 000, then branch to line M
Left normalize*	(AC) shifted left until most significant bit equals 1 → (AC); (X7) + shift amount → (X7); if (AC) = 00000000, branch to line M
Right normalize*	(AC) shifted right until least significant bit equals one → (AC); (X7) − shift amount → (X7); if (AC) = 00000000, branch to line M
Branch*	Branch to line M
Branch on zero*	If (AC) = 00000000, branch to line M
Branch on null*	If (AC) = 11111111, branch to line M
Branch on carry*	If (CARRY) = 1, branch to line M
Load data register	(Fast memory word) → (core memory data register)
Store data register	(Core memory data register) → (fast memory word)
Read	Start core memory read cycle
Write	Start core memory write cycle

* All branch instructions select a new page, in addition to a new line, if the latch associated with the next page register is set.

Similarly, the conventional machine writes control words into the character machine's core memory before interrupting it. When interrupting, the line and page registers for addressing the read-only memory are stored in a pair of special locations in the fast memory, and the line and page registers are reset to a special

value. A program that starts at this address is executed, storing contents of the other registers and part of the fast memory. The program then decodes the message that has been placed in the core memory and acts accordingly.

This I/O solution, adopted mainly to simplify the character machine's construction, has two attractive features. Since the I/O machine is intended for use in a multiprocessing environment, it is not taxed unduly by the I/O function and can perform other operations at the same time. Secondly, there is the possibility of regarding the character machine as the I/O machine's slave. In such an application, it would be possible to write some general-purpose character-handling operations for storage in the read-only memory. The I/O machine would then have an extended instruction set of fast character operations which could be very useful for compiling and editing.

registers

The fast memory has a 7-bit address register and a 64-bit data register. The memory address register is loaded with an effective address by every instruction whose immediate bit is set to zero. A read or write is then performed. Character operations use four of the address bits to select a memory word, and three bits to select a character from the data register.

The read-only memory has a 6-bit (or 8-bit) page address register and an 8-bit line address register. On certain branch instructions, the page register is loaded automatically from the next-page register, as explained earlier. On all branch instructions, the line register is set with the specified line address. The line register is connected to an instruction counter which increments the line register contents on all except branch instructions. The read-only memory also has a 17-bit data register, where instructions are delivered and decoded.

The next-page register is a hardware 6-bit (or 8-bit) register and, simultaneously, character location 2 of the fast memory. Thus, any instruction that writes into location 2 causes loading of the next-page register and setting of the associated latch. The next succeeding branch instruction causes transfer of the contents of the next-page register to the page register, as described earlier.

The core memory has a 16-bit memory address register and a 64-bit data register. Transfers may explicitly be made between the data register and the fast memory. The memory address register is loaded when a read or write instruction is decoded. Such instructions name a word in the fast memory (usually word zero) which is read, and the left-most two characters of the word are sent to the core memory address register. In this way, character locations 0 and 1 of the fast memory appear to the programmer as the core memory address register. Of course, these locations must be loaded before the read or write instruction is given.

The seven index registers are hardware registers, each three bits long. They are connected to a 3-bit indexing adder, having a carry extended to cover eight bits, so that indexing can take place across word boundaries.

The arithmetic and logical operations employ an 8-bit hardware accumulator and a 1-bit carry store as explained earlier.

program organization

The programs that interpret the high-level language are segmented in such manner that distinct programs are on distinct pages, as far as possible.

A subroutine linking mechanism is set up by assigning two words of the fast memory to hold a list of line and page-return addresses. This list is pointed to by one of the index registers which is assigned for this purpose. As subroutines are entered, this index register is incremented, and vice versa. The programmer is responsible for the storing of addresses before the jump to the subroutine is made. The return consists of explicitly setting the next-page register from the subroutine return-page address list and transferring it indirectly to the line address.

push-down organization

Since one of the main purposes of the interpreter program is the decoding of high-level language instruction strings, a programmed push-down organization is used. There are two main push-downs which work together: the instruction push-down in which operators and core addresses are stored, and the data push-down in which data are stored. Two push-downs are needed because the high-level language uses data of variable length. For convenience, operators and their associated addresses occupy single 8-character words in a fixed format, whereas data occupy as much room as necessary.

The top two levels of the instruction push-down store are programmed to be physically in the fast memory. The other levels are put into the core store. The top location in the core is pointed to by the contents of a pair of locations in the fast memory. The data store is entirely in the core store and is addressed through the fast memory.

scan and execution

The current word of the high-level language instruction stream occupies a word in the fast memory. As a scan is completed, a new word is obtained by the program.

Stated with some oversimplification, the high-level language instruction string consists of names of variables, separated by operators. The allocation of storage is dynamic, and high-level language programs refer to data through symbolic names. The occurrence of a name causes the addresses of the associated data to be looked up in lists that are provided in a programmed way.

One purpose of the instruction scan is to decide when names are encountered and to call the name look-up program. Another purpose is to see whether the current operation can be executed or not. If it can be executed, the appropriate program is called to implement the operation; if it cannot be executed, the instruction push-down is pushed down, and the top location is loaded with the operator and address of that data in core on which it is to operate.

An operation can be executed or not, depending on its relative precedence in the instruction stream. The allocation of precedences is part of the specification of the language. In the present system,

this precedence is expressed directly by the arithmetic values that are given to the character codes representing the operators.

When an operation has been performed, the instruction push-down is pushed up again, and a test is made to see whether the operation now at the top may be executed.

The data push-down is used to store temporary results. Every operation that generates temporary results places these results into the data push-down. Similarly, every operation using an operand that is in this push-down causes a pushing-up operation after the operand has been used. Because of the structure of the push-downs, the required operands are always at the top of the data push-down.

simulation

One of the attractive features of implementing a high-level language machine in the way described is that the logic can be checked by simulation. When the character machine was simulated on a 7094, each page of the microprogram was assembled as though it were a FAP subroutine, and microinstructions were expressed as though they were 7094 instructions. In brief, a dummy operation was written in the FAP operator field, a symbolic address referring to either the read-only memory or the fast memory was written in the FAP address field, the character machines tag was written in the tag field, and a symbolic name of a character machine operation was written in the FAP decrement field.

The 7094 interpreter was arranged not to execute these assembled subroutines, but to interpret them. In this way, it was possible to write read-only memory programs in a symbolic manner and have all the advantages of an assembly program, without actually writing one.

It has proved possible to write the programs that interpret the high-level language in a direct way, with an effort similar to that needed to write a compiler.

concluding remarks

It is perhaps worthwhile emphasizing again the two points of view from which the system may be seen. On the one hand, it is a direct implementation of a high-level language machine in which the various logical suboperations are rather formalized. On the other hand, it is a character-handling machine, one of whose jobs is the interpretation of a high-level language.

The discussion points to the value of regarding logical design and programming in an integrated way. If all operating and control programs being used on current computer systems were regarded in this way, the effect on systems design might well be significant.

FOOTNOTE

1. A. P. Mullery, R. F. Schauer, and R. Rice, "ADAM: A Problem Oriented Symbol Processor," *Proceedings of the Spring Joint Computer Conference*, 1963. The reference describes the high-level ADAM language as well as a tentative machine design. The character computer discussed here implements this language, but differs from the machine design of the reference.