Memory Reduction Through Higher Level Language Hardware

H. KERNER*
NASA Marshall Space Flight Center, Huntsville, Ala.

AND

L. GELLMAN†
Computer Sciences Corporation, Huntsville, Ala.

Traditional assignment of computer functions to logic and memory is evaluated based on trends in their cost and mission characteristics. The higher information content of a higher level language is exploited for the savings of memory. An instruction set was selected from the directly executable statements of FORTRAN and the remaining statements were identified for compilation. Programs written in this language occupied 75% less memory than those obtained by compilation. A computer for the execution of the selected FORTRAN statements was functionally designed in order to estimate the corresponding logic hardware effort. The tradeoff between costs saved through memory reduction and expenditures for the additional logic hardware considered development, manufacturing, power and weight costs. The proposed concept shows a cost advantage for missions with a high-weight penalty and large memory requirements. Another advantage is the higher execution speed inherent in this organization.

I. Introduction

INNOVATIONS in hardware technology and extraordinary performance demands will prove a major factor in the organization of future spaceborne computers. Advancing semiconductor technology such as large-scale integration (LSI) will make large-scale computers on board spacecraft a reality. Indeed, the sophistication of future spacecraft will make such an onboard computer a necessity.

One of these future spacecraft may be an Earth orbital space station that supports ten or more astronauts, carries equipment and supplies for some 300 scientific experiments, and has an orbiting lifetime of several years. The computer requirements of such a mission for experiment control and data processing, attitude control, guidance and navigation, performance monitoring, maintenance support, communication, etc., can be expected to be of a magnitude equivalent to a computer with real-time characteristics supporting a large ground-based laboratory. Another spacecraft, unmanned, on a journey to Mars would require extraordinary reliability and demand computer services for data compression, guidance and navigation, and communication. Without a man on board, a large-scale computer could increase the efficiency of the mission by data sampling and choice making through an adaptive mode of operation.[1]

Such extraordinary reliability and durability requirements combined with constraints on its power supply and launch costs of volume and weight affect directly the computer organization and may result in a substantial departure from present-day organization. In the design of spaceborne computers, these reliability and cost considerations prove especially relevant in the tradeoff between functions assigned to logic hardware and to memory. Presently, any particular balance of function allocation between logic hardware and memory is based on their respective manufacturing costs. Current trends in LSI technology promising manufacturing cost savings and weight reduction will shift that balance. These considerations demand a re-evaluation of logic-memory function assignments.

Expected LSI progress in the first half of the 1970's should reduce the weight of logic hardware by a factor of 30, whereas weight reduction in magnetic memories is expected to be much less drastic. This means that computers with present-day organization implemented in the technology of the mid-1970's will have the bulk of their weight concentrated in memory devices. For computers used in weight (or volume) sensitive missions, a design approach is indicated that will minimize memory at the expense of adding logic hardware. The validity of this approach is reinforced by the cost reduction expectations in logic hardware through LSI technology when compared with the limited cost reduction possibilities in memory technology.

For these reasons, this paper reinvestigates the traditional logic-memory balance that has survived three computer generations and offers a memory-saving approach that exploits the higher information content of higher level languages.

II. Problem Definition and Approach

Consideration of several types of computer languages provides insight into how memory savings can be realized. Computer languages can be categorized by levels, each characterized by a degree of generality (see Table 1).

A program written in a particular level language processed by the appropriate compiler produces a program in the language of the next lower level. The higher level language together with its compiler defines the detailed machine actions as completely as the compiled program does in the language of a lower level. The aforementioned points and the fact that higher level language programs occupy an order of magnitude of less memory than the compiled versions demonstrate the memory saving qualities of these languages.

To exploit these language properties, a technique was required that interprets higher language statements and immediately executes an equivalent group of lower level machine actions. Since it avoids the "assembly" of the total program into the next lower level this interpretive principle can be

Presented as Paper 69-963 at the AIAA Aerospace Computer Systems Conference, Los Angeles, Calif., September 8–10, 1969; submitted September 8, 1969; revision received April 29, 1970. The critical review and advice of S. Stein and J. Kennedy is greatly appreciated.

* Division Chief, NASA Manned Flight Space Center, Huntsville Ala.

† Senior Engineer, Computer Science Corp., Huntsville Operation.

used to reduce memory requirements. Whether in the microprogramed or hard-wired version, this approach is valid between any two language levels.

The goal of this study is to define a computer organization that introduces a level of control above the conventional level of arithmetic and logic processors (Fig. 1). This system features several processors on this control level capable of receiving higher level language statements, interpreting them each in their respective language, and controlling their immediate execution in lower level processors. The statement repertoire of each language and its associated control processor shall be sufficient for the description and control of complete programs or program segments.

The resulting computer organization will minimize computer memory by adding logic hardware. A comparison with conventional design will be made with respect to 1) generalized cost—a) manufacturing cost, b) power requirement, c) weight and size; 2) speed; and 3) reliability.

For this comparison, the problem is restricted to direct code interpretation and execution between two adjacent language levels, specifically, the Procedure Oriented Language and the Machine Language levels. Further restrictions limit the system to one processor on each of these levels.

The selection of the higher level language will certainly influence the result of the comparison. Languages designed primarily for numerical calculations, well matched to existing general purpose computer organizations, offer scant opportunity for higher level execution. Therefore, languages rich in sophisticated control features are expected to yield more favorable results than languages strong in arithmetic and logic characteristics. For instance, a language designed especially for the writing of executive programs lends itself extraordinarily well to the implementation through high-level control processors. Such a language would also make less use of the lower-level processors.

FORTRAN IV was chosen as the "Higher Level Language" for this study for the following reasons: 1) its control features are balanced with arithmetic and logic features; 2) a great variety of programs including spaceborne applications were available for evaluation of the proposed design; 3) it has wide usage and a comparison is therefore of broad interest. The corresponding processor will be called FORTRAN Language Processor (FLP).

It is probable, however, that FORTRAN IV will not be the language best suited for spaceborne computer programing. If other, more specialized, languages prove better suited to space applications, the results of this study will represent a "worst case" for the proposed computer organization.

Aside from the language selected for this study, the sophistication of the higher-level language processor design constitutes a second influence on the results obtained. Since the computer is destined for spacecraft application, desirable design simplification can be attained through the use of a terrestrial computer preprocessor program. Such a ground-based preprocessor would "sort out" the program statements, "compile" those statements not required for execution, and format the others for interpretative execution by the spaceborne computer. Preprocessed programs can be loaded into the computer before and during a spacecraft mission. This approach simplified the FLP by avoiding the difficulties of a

Table I Types of Languages

Level	Types of languages
L4	Problem-oriented languages (COGO, GPSS, . . .)
L3	Procedure-oriented languages (FORTRAN, ALGOL, PL1, . . .)
L2	Machine peculiar languages
L1	Micro code

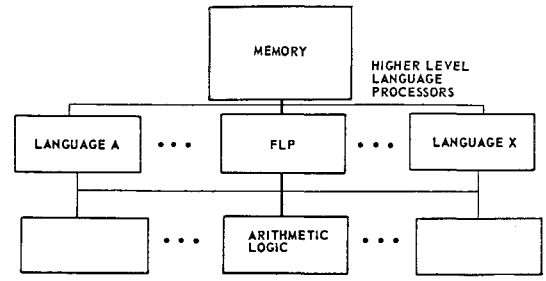

Fig. 1 Arithmetic and logic processors.

complex hardware compilation without compromising the memory-saving features of the design.[2-4]

Input/output (*I/O*) statements were eliminated from the study comparisons because of the difficulty of obtaining ground-based analogues to spaceborne applications. Terrestrial machines of the type available for the study, read cards, tapes, discs, drums, and output through these media, while the spaceborne computer will collect data from sensors through various data acquisition equipment and transmit results to actuators or to data links on Earth. Future spaceborne computer *I/O* repertoire is expected to include the combined *I/O* features of ground-based process control, checkout, navigation, communications, and general-purpose computers.

III. Instruction Set and Preprocessor

The allocation of functions to the Preprocessor and the Higher Level Language Processor was governed by the following considerations: 1) the requirement for maximum memory saving with minimum additional logic hardware cost; 2) that execution speed should not be less than that obtained with a conventional compiler approach; 3) that speed improvement possibilities uncovered be incorporated only if they can be obtained with minor hardware penalty. An analysis of the FORTRAN language in light of these considerations suggested the classification of FORTRAN statements into five categories: arithmetic, control, input/output, compilation, and comments.

Compilation aids and comments are handled by the preprocessor since they do not produce executable code. From the remaining three statement types, the control and the arithmetic statements proved candidates for implementation in hardware, because every instruction of this type offers opportunity for substantial memory savings through higher-level execution. For reasons previously mentioned, *I/O* statements were not exploited in this study, despite the fact that large memory savings and improved speeds are expected from hardware implementation of the specialized *I/O* requirements of spaceborne systems.

Arithmetic FORTRAN statements require special consideration. The analysis of the structure of an expression, as well as the optimization of computational sequences, does not directly contribute to memory savings. However, associated address modifications expedited by the controls of the arithmetic-logic processor and supported by the high efficiency of data transfer between the two levels of processors, are considered worthy of implementation in the Higher Level Language Processor.

This analysis of FORTRAN statements resulted in the following instruction set, chosen for implementation in the Higher-Level Language Processor: 1) GO TO (includes COMPUTED GO TO); 2) IF (logical and arithmetic); 3) DO; 4) CALL (no argument); 5) library call (arguments included); 6) RETURN; 7) input/output (READ, WRITE); 8) parts of arithmetic statements—address processing, control of arithmetic and logic processor; 9) replacement (=). Although this list appears small, it can be verified that all

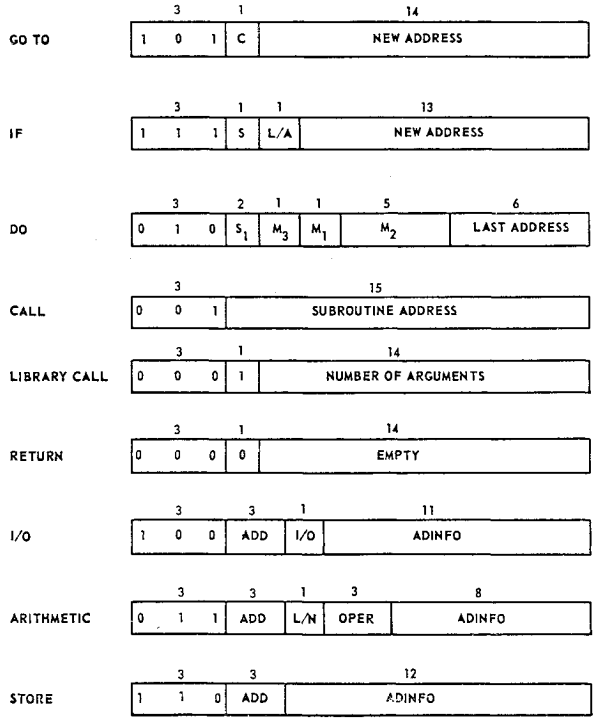

Fig. 2 Instruction set.

basic FORTRAN IV control operations can be performed using this instruction set.

Memory reduction goals also governed the instruction word format selected. A variable subfield format was used to conserve memory despite the additional decoding required. Whenever possible, two instructions were packed into one 36-bit word. More than one half-word is required for some arithmetic and *I/O* statements, the library call, COMPUTED GO TO, and arithmetic IF statements.

The only common feature of all instruction words is the 3-bit operation code (Fig. 2). The right-hand portion of the half-word contains a 13–15-bit address, or instruction modifiers supporting the operation code. The ADD field in the *I/O*, ARITHMETIC, and STORE instruction specifies the method of addressing data as follows: 000 = integer data accompanying the instruction, 001 = the actual data address in memory, 010 = an FLP register address, 011 = the previous address plus 1, 100–111 = dimensioned data. Another feature employed for code compacting was the construction of standard instruction modes for the commonly expected simple cases and the indication of extraordinary cases in short fields (labled S and S1) that control the decoding of words following in sequence.

The two approaches to memory compression discussed above, i.e., the use of a procedural language for the total program and compact coding within each specific instruction word, both require a substantial increase in logic hardware.

The search for a device to keep the hardware within bounds leads to the idea of the preprocessor executed on a ground computer, which would relieve the Higher Level Language Processor from all functions not absolutely necessary for the primary goal of code compression. This preprocessor was assigned functions of an assembler and of a compiler. The typical assembler tasks are 1) translation of the mnemonic code into binary code in a nearly one-to-one correspondence with each FORTRAN statement; 2) assignment of relative addresses for program variables, instructions, and statement labels; 3) formatting and packing of the code into the instruction word. Preprocessor compiler tasks are a) separation of FORTRAN statements directly executable by the FORTRAN Language Processor from statements requiring further analysis such as arithmetic statements; b) generation of additional instructions for initialization of registers before the beginning of computational sequences and for transitions from one sequence to another one; c) elimination of operations involving mixed modes of data types; d) data placement according to DATA statements; e) optimization of arithmetic operations and control paths; f) optimization of data addresses relative to the order of program execution.

IV. Organization and Operation

A quantitative evaluation of the proposed approach toward memory reduction required the design of a language processor capable of interpreting the selected higher level instruction

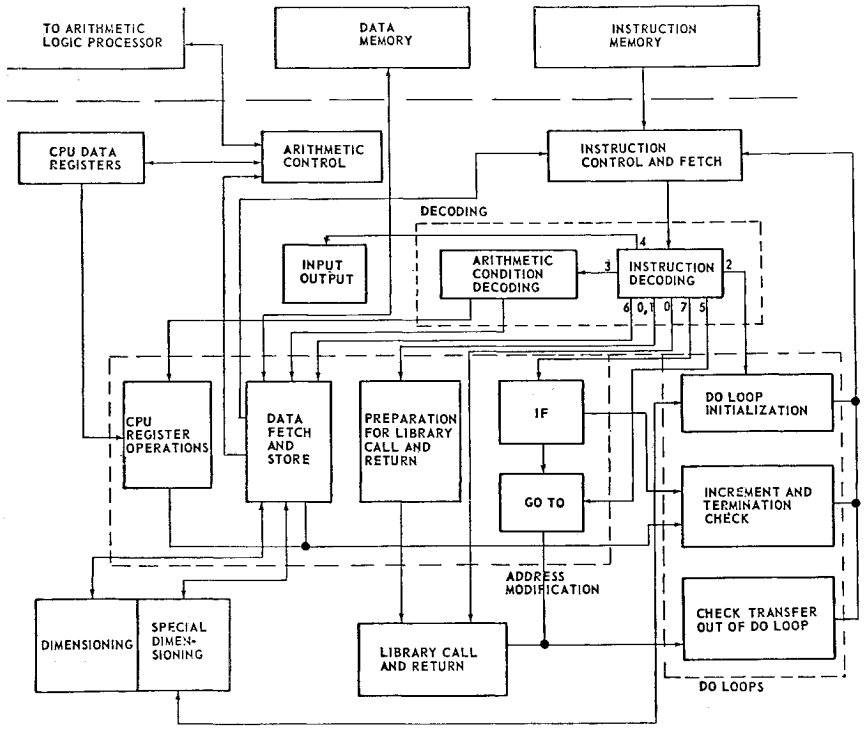

Fig. 3 FLP flow diagram.

set. A functional design of the Higher Level Language Processor proved sufficient to estimate its hardware complexity considering the heavy influence of other factors on the trade-off evaluation. The products of this functional design were a description of the decoding scheme for the variable format instructions, a chart depicting the flow of data between registers, and a definition of the pertinent program execution events and the corresponding control mechanism. A simplified flow diagram of the FORTRAN Language Processor illustrating the following discussion is shown in Fig. 3.

Instruction execution begins as soon as one memory word (containing two instruction words) has been obtained from memory and loaded in a 32-bit instruction register with a capacity of two memory words (four instruction words). A fetch overlap feature loads the next two instruction words into this register while the first two instruction words are decoded and executed. A memory address register specifies the next (double) instruction location in memory in the usual way, while a pointer locates the instruction to be executed within the instruction register. The memory address register and the pointer combined act as an instruction location counter. The multiple instruction register allows multiple word instructions to be executed without further access to memory. For example, DO loops of four instructions or less can be executed directly from the four instruction word register without accessing instruction memory until the loop is completed.

A special situation occurs during a fetch operation of an instruction word supported by more than four trailer instruction words. After decoding the first words, the resultant state is stored in a special register. The execution of this state is deferred until loading and decoding of the remaining instruction words has been completed.

The decoding of the variable field instructions requires a more complex decoding scheme than is usually employed. The main decoder analyzes the instruction code and routes the subfields to corresponding subsections for further decoding.

A special case of a multiple word instruction is an arithmetic statement that is represented (after preprocessing) by a string of arithmetic instructions. When encountering the first arithmetic instruction of the string, the decoder initializes processing of the string by setting control flip-flops that indicate that string processing is in progress and which discriminate between logical and numerical arithmetic and between real and integer operands.

All operations were coded in the most concise form by taking advantage of some high-frequency occurrences. These special cases offering maximum opportunity for shortening references to data are detected and specially treated by the processor in order to compress instruction memory.

The arithmetic instruction type will be used as one example to illustrate this method. Seven special cases of arithmetic operations, which are expected to occur frequently enough to be exploited for shortening memory references, are discussed below.

If the operand of an arithmetic operation is an integer, its value can be stored in the next consecutive instruction word or, if its value is less than 28, it can be stored in the ADINFO field of the same instruction. Some operands are intermediate results of previous arithmetic operations, which reside in one of the 32 internal CPU registers. These operands can be referenced in the CPU registers by the ADINFO field of the calling operation instruction. Frequently operands can be arranged in consecutive memory locations. Whenever this is possible, data of such a string can be obtained by retaining the address of the first data item and incrementing this address by one for each consecutive operation of this arithmetic string.

Arithmetic operations dealing with dimensioned variables are treated as three different cases. In the general case (seven dimensions are permissible in this language processor implementation), the address of the first word in the array is contained in the instruction word immediately following the calling operation instruction. Subscripts are referenced in the ADINFO field of this second instruction word. If the values of the subscripts do not reside in internal CPU registers, an additional instruction word refers to their memory locations. For variables in a dimension greater than 2, the data address will be computed by software algorithms. For one- or two-dimensioned variables, data addresses are computed by special hardware routines using subscript values stored in internal registers. Only one subsequent instruction word, locating the first word of the array, is required.

These special cases are differentiated by the preprocessor which relates this information to the FLP through the 3-bit ADD field of the arithmetic instruction.

A similar approach was taken in the case of the DO loop statement. The DO loop variables of this statement have the following standard meanings: $m1$—starting value of the loop index, $m2$—limiting value of the index, $m3$—increment by which the index is modified. Special values of these parameters occur frequently and this fact can be utilized as follows to save bits and time when referencing memory.

If the starting value of the index ($m1$) is either one or two, this value can be noted in the 1-bit "$m1$ field" of the DO instruction. If the maximal value of the index ($m2$) is less than 32, it will be stored in the 5-bit "$m2$ field" of the DO instruction word. Whenever $m1$ or $m2$ do not fall within the above range of "favorable" values, an additional instruction word is needed for references to memory locations containing the parameter. It can, however, be assumed that these general cases will not occur too frequently.

As with the arithmetic operations, the preprocessor discriminates among the combinations of $m1$ and $m2$ for the general or the special cases and encodes this information in the 2-bit DOFLAG field of the DO instruction.

Since DO loop indices are most frequently incremented in steps of 1 ($m_3 = 1$) a special "$m3$ field" in the DO instruction format identifies this case. In the general case of $m3 \neq 1$, additional instruction space is required.

The value of the "last address" of the DO loop can be encoded in shorter form by measuring its numerical distance from the beginning address of the loop. If the distance is less than 63, it can be referenced in the 6-bit "last address field" of the DO instruction, and the absolute address can be calculated from this formation. The general case of a distance larger than 63 is indicated by zeros in this field and the value is carried in an additional instruction word.

DO loops can be nested within DO loops. In this implementation, the information concerning the first seven levels of nested DO loops are stored in a stack of seven special registers. An adjunct 3-bit register designates the level of the currently active loop.

Similar techniques exemplified by the arithmetic and DO loop instructions analyzed above were employed in handling the remaining FORTRAN Language Processor instructions. In summary, these techniques take advantage of high frequency occurrences to develop special case algorithms that allow more concise coding than a general approach covering all cases. High incidence of such opportunities appears in the location or arrangement of data, in the constraints imposed upon the length of data string addresses, and in the definition of statement parameters. Memory references are also reduced by storing intermediate results in a special set of internal CPU registers whenever possible.

In general, the techniques employed require that additional information regarding any special exploitable characteristics of the operation be part of the instruction word. The additional bits needed to signify these characteristics are expected to be highly repaid through the total memory savings achieved by this method. Also, the large number of vari-

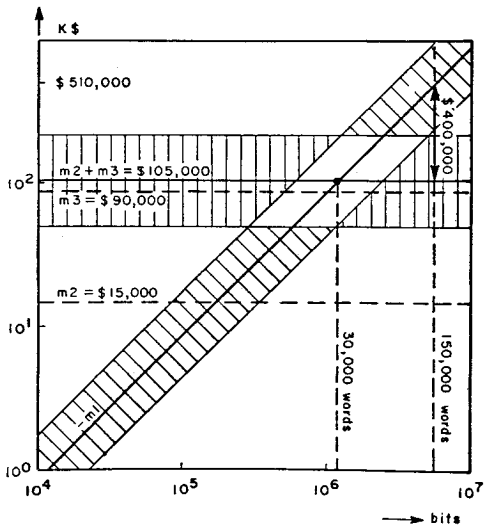

Fig. 4 Trade off equation (2), small quantity ($n = 5$), high reliability ($r = 3$).

ations in the instruction word format imply a complex execution logic. This situation demands a highly sophisticated preprocessor as an intermediate step in order to keep the size of the logic hardware within reasonable bounds.

In the following chapter the economical implications of these techniques will be evaluated in terms of memory savings. Classes of computer systems and applications for which these techniques appear to be advantageous will be explored.

V. Evaluation

Having functionally designed the FORTRAN Language Processor and defined the rules for the preprocessor program, we then evaluated the proposed approach to memory reduction by quantitative approximation.

Estimates were made of memory savings and of the added effort in hardware logic based on costs expected in the mid-1970's. These values were entered into a tradeoff equation that evaluates the relative influence of memory vs logic hardware. Four different missions are presented as examples of the memory-logic tradeoff evaluation.

For obtaining an estimate of memory savings, several programs were manually processed according to the rules established for the preprocessor. The programs were selected on the basis of their spaceborne application characteristics such as guidance and navigation programs. Manual preprocessing "assembled" and "compiled" the statements and packed the resulting instructions into the required FLP format. *I/O* instructions were omitted. A count of the FLP formatted words represents the FLP memory requirements of the programs.

The same programs were compiled on the IBM 7094. Code generated by *I/O* operations was deleted, and memory allocated for data storage was not taken into account. The number of words generated represents the comparative memory requirements for instruction storage on a computer executing machine language.

The ratio of the storage requirements for the machine language code over the FORTRAN Language Processor code, which we call compression ratio, resulted in an average of approximately 4:1 for the programs compared.

The estimate of the additional hardware cost was concerned only with the logic of the FLP that would have to be added to a conventional computer. Possible savings in the original computer control section, which might occur because of performance of similar functions in the FORTRAN Language Processor, were not considered. These concessions tend to produce conservative evaluation results. For purposes of this study the design of the FLP was limited to the definition of the various registers and the associated information flow. The logic requirements were estimated by multiplying the number of register flip-flops by the ratio of logic gates to flip-flops usually found in computers. This ratio was obtained by examining several computers and was found to vary between the values 20:1 and 40:1. The lower case "20" was used to calculate the number of logic gates required, because the FLP is a device using many more registers than is normal in computers. There are 2270 flip-flops employed in registers of the FORTRAN Language Processor, consequently the additional number of gates required was calculated to be 45,000 or less than 50,000. Since this amount of logic is common in medium- to large-scale computers, 50,000 gates is considered an upper limit to the size of the FLP.

For the tradeoff analysis, a memory compression ratio of 4:1 and a corresponding hardware estimate of 50,000 gates was used to arrive at a comparable cost figure. The total cost differential (C) between a computer using a FORTRAN Language Processor and one without this addition consists of differentials in manufacturing and development cost (M), in direct power cost (P), and in launch cost (L). The following equation (1) expresses the generalized cost (C) for a total of n computers manufactured and one launched:

$$C = nM + P + L \tag{1}$$

The manufacturing and development cost differential (M) contains three terms. The first term represents memory cost saving, the second term represents manufacturing cost of the additional logic hardware, and the third term represents the LSI chip development cost.

$$\begin{aligned} M &= (s \cdot b \cdot r) - (g \cdot cg \cdot r) - (g \cdot d \cdot cc \cdot co)/n \\ &= \quad m1 \qquad -m2 \qquad\quad -m3 \end{aligned} \tag{2}$$

where M = net cost savings realized (memory reduction vs added logic); s = size of memory saved (bits); b = bit cost ($); r = cost factor because of reliability requirements; g = number of gates added; d = chips per gate packing density; cg = manufacturing cost per gate ($); cc = development cost per chip ($); co = fraction of total chips requiring new development; n = number of computers to be manufactured. Cost savings as a result of reduced power consumption and launch weight follow from Eqs. (3) and (4):

$$P = (pm - pl) \cdot cw \tag{3}$$

$$L = (wm - wl)l + (pm - pl)ww \cdot l \tag{4}$$

where P = power consumption cost savings ($); L = launch weight cost savings ($); wm = weight of saved memory; wl = weight of FLP logic; pm = power saved because of memory reduction (w); pl = power penalty because of the added logic hardware (w); cw = cost per watt ($/w$); ww = weight per watt (lb/w); l = cost to launch 1 lb ($/lb).

As a first example for the tradeoff evaluation expressed by Eq. (1), consider a computer aboard an Earth-orbiting space station in the last half of the next decade. This com-

Table 2 Values needed to solve tradeoff equations

s	Memory saved	5.7E6, bits
b	Bit cost	$3E - 2$, $
r	Cost factor because of higher reliability	3
g	Number of gates	50,000
cg	Cost per gate	$1E - 1$, $
d	Chips per gate	$1.5E - 3$
cc	Development cost per chip	2E4, $
co	Fraction of new chips	$3E - 1$
n	Number of computers	n

puter controls a portion of the 300 experiments aboard and processes their output data, a task that may vary from simple reformatting of data to fairly complex pattern recognition. It monitors spacecraft subsystems, and predicts and diagnoses failures. Data management and transmission are important functions of the computer, which also serves as the center of command and control activities and is the heart of the guidance, navigation, and attitude control systems.

The memory requirements for this mission were estimated to be 260,000 38-bit words. Sixty thousand words were allocated for storing the executive program, data, and work area leaving 200,000 words available for instruction storage. The instruction storage compression ratio of 4:1 expected with the FLP approach indicates a memory savings of 150,000 words or 5.7 million bits.

The other values needed for solving the tradeoff equations (1–4) were obtained from the literature[5–7] and updated with current information obtained from component vendors. Table 2 lists these values (E means the exponent of the base 10, as used in FORTRAN notation).

Figure 4 shows the plot of the three terms of Eq. (2). Manufacturing cost is represented by $m2$, and $m3$ represents development costs. The term representing the savings through memory reduction, $m1$, is shown as a function of the reduced bit size. The manufacturing cost ($15,000 each) represented by $m2$ was calculated for a production run of five computers: the flight computer, another for testing, two for program development, and one backup computer. The development cost, $m3$, calculated as $90,000 per computer is nearly one order of magnitude larger than $m2$, the manufacturing costs ($15,000) of the FLP. The combined chip development and manufacturing cost ($m2 + m3$) equals $105,000, which differs from $m1$ by $410,000 for a memory savings of 150,000 words. Five computers could therefore be produced at a savings of more than 2 million dollars ($2,050,000).

Figure 4 also shows, that the costs for logic and memory break even at a memory size of a million bits. For larger memories, the FLP approach should be economically superior.

In order to compensate for the uncertainty of the values, the plots of the $m1$ term and of the combined $m2 + m3$ term were represented by bands whose limits were defined by twice and one-half the calculated values. Figure 4 shows that even in the worst case (the lower boundary of $m1$ and the upper boundary of $m3$) the tradeoff is in favor of the FLP approach for memories of over 4 million bits.

These findings were applied to a production run of 50 of the same computer for possible use in military ground or air-

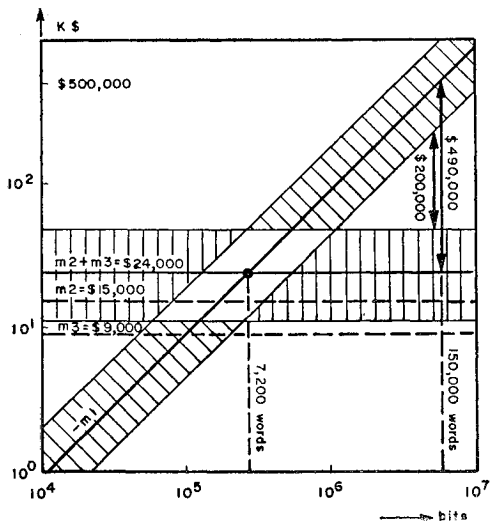

Fig. 5 Tradeoff equation (2), large quantity (n = 50), high reliability (r = 3).

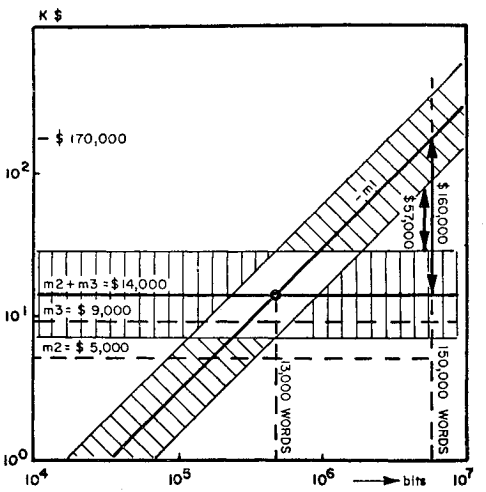

Fig. 6 Tradeoff equation (2), large quantity (n = 50), commercial reliability (r = 1).

borne applications. The chip development costs, $m3$, distributed over the 50 computers were reduced to $9000 per unit. The combined chip development and manufacturing cost, $m2 + m3$, per unit totalled $24,000. This indicates an advantage for the FLP approach when memory savings are 270,000 bits (7200 words) or more (Fig. 5).

For commercial computers (Fig. 6, r = 1), with chip development and manufacturing costs ($m2 + m3$) of $14,000, a break-even point appears at 500,000 bits or 13,000 words. Consequently, a memory reduction of 150,000 words in a commercial version of the FLP would save $146,000 per computer.

Assuming the memory compression ratio of 4:1, we come to the conclusion that military computers with more than 7,000 words and commercial computers with more than 13,000 words required for instruction storage can benefit from a high-level language processor.

For spaceborne computers, the cost of the power supply and the launch cost of the computer may constitute a substantial portion of the cost of the total computer system.

The cost savings resulting from the reduced power requirements (P) of the FLP design were calculated by Eq. (3), $P = (pm - pl) \cdot cw$, using the following values: pm—memory reduction of 150,000 words saves, 580 w; pl—the added hardware logic consumes, 50 w; cw—cost per watt (solar cells), 200 $/w. Then P = $106,000.

Launch cost savings (L) attributable to the FLP design were calculated by Eq. (4), $L = (wm - wl) + (pm - pl)ww \cdot l$, using the following values: wm—weight of saved memory, 230 lb; we—weight of added logic, $\ll$230 lb; $(pm - pl)$—reduced power requirements, 530 w; ww—weight per watt, 0.16 lb/w; l—cost of launching one lb., 1000 $/lb. Then, L = $230,000 + $85,000 and L = $315,000. (The weight of added logic hardware was considered negligible when compared with the weight of the memory saved.)

Equation (4) indicates that the weight of a computer with a 260,000 word memory could be reduced 315 lb by adopting the FLP approach (230 lb could be saved through memory reduction and 85 lb through reduced power requirements).

This launch cost savings of $315,000 is for an Earth-orbiting mission where current launch costs average about $1000 lb. These savings appear insignificant when compared to the 2 million dollar reduction in manufacturing costs or when compared to the cost of the total mission.

Launch costs of $20,000 lb are predicted for interplanetary missions of the mid-1970's. In this case, launch cost savings because of weight reduction of the FLP approach will exceed 6 million dollars in addition to the 2 million dollars saved in manufacturing costs. The ultimate payoff of this savings is

the opportunity it affords for payload sophistication, thereby increasing the scientific value of the mission.

However encouraging this result is for interplanetary missions, the final analysis of a spaceborne computer system must include its reliability. This task must be postponed until sufficient data on LSI reliability is available.

VI. Conclusions

This paper attempts to answer the question of whether the traditional allocation of functions between logic and memory will change under the impact of new technology or extraordinary performance requirements such as those imposed by space flight. The conclusions reached as a result of this investigation stand on projections of manufacturing costs in the mid-1970's, on a particular language, compressed instruction coding, and interpretive logic. Despite wide variances, tolerated in all numerical values used in this paper, the authors contend that some positive conclusions were reached regarding the benefits of the Higher Level Language Processor approach.

From the economical point of view, the example worked suggests advantages for all four cases whenever the memory required for the application was sufficiently large. For a computer with a memory of 260,000 words it was considered feasible to reduce its size to 110,000 words through the approach offered. This reduction of 150,000 words implied a manufacturing cost savings of about $400,000 for computers demanding high reliability, when produced in a batch of 5. For computers manufactured in batches of 50, a break-even point between the additional logic hardware and the saved memory occurred around 13,000 words for commercial computers and around 7000 words for high-reliability hardware. Since these represent rather small memories, the addition of a Higher Level Language Processor appears to be advantageous even in commercial quality hardware. The reduction in manufacturing costs for such computers containing larger memories appears to be the significant factor (some $100,000). For interplanetary missions, however, the reduction in weight and volume proved more important. A key consideration for these long duration missions is the impact the additional processor would have on system reliability.

The economies of the Higher Level Language Processor approach demonstrated in this paper suggest that additional benefits can be derived through improvements in the scheme. Investigations of languages other than FORTRAN may reveal even greater opportunities for code compression. Some excellent candidates should be found among real-time control languages and algebraic types capable of processing arrays of data or executing complex algorithms by single instruction.[8–10] Problem-Oriented Languages, which we ranked one level above Procedure-Oriented Languages in code compacting qualities, seem to offer great opportunities for code compressions. An investigation of data structures should uncover additional memory-saving possibilities. A more detailed analysis than the one done for this paper should investigate individual features implemented in logic hardware and evaluate the tradeoff involved.

Further improvement of the tradeoff results seems to be possible through standardizing some of the logic components, especially heavily used stacks and counters. Employment of microprograming schemes could replace specialized logic by standardized microcontrol hardware. This approach may improve the reliability of the system through the interchangeability of the microprogramed control devices.

Besides cost, the computer organization implied by the higher level language approach can influence other factors of computer architecture. It can, for example, offer a solution to the program segmentation problems encountered in multilevel memory schemes (paging, cache technique[11–13]), because complete statements are transferred to lower level processors, where the equivalents of small segments are generated. Furthermore, since the higher level statement is not compiled into memory machine steps, extensive look-ahead features can be more easily implemented.

Future computers also face a bandwidth problem in the internal communication between computer subsystems. This is especially serious in the case of distributed logic computers. The instruction stream through such computers can be greatly reduced through the employment of Higher Level Language Processors.

In conclusion, this paper indicates that employment of more comprehensive instructions or complete higher level languages is desirable not only because of their inherent speed and programing advantages, but because they also provide economical advantages by virtue of their memory-saving properties.

References

[1] "Spaceborne Multiprocessing Seminar," Oct. 1966, NASA TMX 59362, Electronics Research Center, Cambridge, Mass.

[2] Bashkow, T. R., "A Sequential Circuit for Algebraic Statement," *IEEE Transactions on Digital Computers*, April, 1964.

[3] Lawson, H. W., "Programming Language Oriented Instruction Streams," *IEEE Transactions on Digital Computers*, Vol. C17, No. 5, May 1966.

[4] "Burroughs 6500/7500 Information Processing System Characteristics Manual," Sept. 1968, Burroughs Corp.

[5] Wainer, R. M., "Comparing MOS and Bipolar Integrated Circuits," *IEEE Spectrum*, June 1967.

[6] Einhorn, R. N., "LSI Improves Computer Memory Bit by Bit," *Electronic Design*, April 1, 1968.

[7] Brown, D. W. and Burkhordt, D. L., "The Computer Memory Market," *Computer and Automation*, Jan. 1969.

[8] Seitz, R. N., Wood, L. H., and Ely, C. A., "AMTRAN: Automatic Mathematical Translation," Interactive Systems for Experimental Applied Mathematics, *Proceedings of the Association for Computing Machinery Inc.*, Washington, D.C., Aug. 1967, p. 44.

[9] Falkoff, A. D. and Iverson, K. E., "The APL/360 Terminal System," *Proceedings of the Association for Computing Machinery Inc.*, 1967, p. 22.

[10] Symes, L. R. and Roman, R. V., "Structure of a Language for a Numerical Analysis Problem Solving System," *Proceedings of the Association for Computing Machinery Inc.*, 1967, p. 67.

[11] Wilkes, M. V., "Slave Memories and Dynamic Storage Allocation," *IEEE Transactions on Digital Computers*, April, 1965.

[12] Conti, C. J., Gibson, D. H., and Petowski, S. H., "Structural Aspects of System/360 Model 85," *IBM Systems Journal*, Vol. 7, No. 1, 1968.

[13] Liptay, J. S., "The Model 85 Buffer Storage," *IBM Systems Journal*, Vol. 7, No. 1, 1968.