Design of the Burroughs B1700

by W. T. WILNER

Burroughs Corporation
Goleta, California

INTRODUCTION

Procrustes was the ancient Attican malefactor who forced wayfarers to lie on an iron bed. He either stretched or cut short each person's legs to fit the bed's length. Finally, Procrustes was forced onto his own bed by Theseus.

Today the story is being reenacted. Von Neumann-derived machines are automatous malefactors who force programmers to lie on many procrustean beds. Memory cells and processor registers are rigid containers which contort data and instructions into unnatural fields. As we have painfully learned, contemporary representations of numbers introduce serious difficulties for numerical processing. Manipulation of variable-length information is excruciating. Another procrustean bed is machine instructions, which provide only a small number of elementary operations, compared to the gamut of algorithmic procedures. Although each set is universal, in that it can compute any function, the scope of applications for which each is efficient is far smaller than the scope of applications for which each is used. Configuration limits, too, restrict information processing tasks to sizes which are often inadequate. Worst of all, even when a program and its data agreeably fit a particular machine, they are confined to that machine; few, if any, other computers can process them.

In von Neumann's design for primordial EDVAC,[1] ridigity of structure was more beneficial than detrimental. It simplified expensive hardware and bought precious speed. Since then, declining hardware costs and advanced software techniques have shifted the optimum blend of rigid versus variable structures toward variability. As long ago as 1961, hardware of Burroughs B5000[2] implemented limitless main memory using variable-length segments. Operands have proceeded from single words, to bytes, to strings of four-bit digits, as on the B3500. The demand for instruction variability has increased as well. The semantics of the growing number of programming languages are not converging to a small set of primitive operations. Each new language adds to our supply of fundamental data structures and basic operations.

This shifting milieu has altered the premises from which new system designs are derived. To increase throughput on an expanding range of applications, general-purpose computers need to be adaptable more specifically to the tasks they try to perform. For example, if COBOL programs make up the daily workload, one's computer had better acquire a "Move" instruction whose function is similar to the semantics of the COBOL verb MOVE. To accommodate future applications, the variability of computer structures must increase, in yet unknown directions. Such flexibility reminds one of Proteus, the mythological god who could change his shape to that of any creature.

DESIGN OBJECTIVE

Burroughs B1700 is a protean attempt to completely vanquish procrustean structures, to give 100 percent variability, or the appearance of no inherent structure. Without inherent structure, any definable language can be efficiently used for computing. There are no word sizes or data formats—operands may be any shape or size, without loss of efficiency; there are no *a priori* instructions—machine operations may be any function, in any form, without loss of efficiency; configuration limits, while not totally removable, can be made to exist only as points of "graceful degradation" of performance; modularity may be increased, to allow miniconfigurations and supercomputers using the same components.

Design rationale

The B1700's premise is that *the effort needed to accommodate definability from instruction to instruction* *is less than the effort wasted from instruction to instruction when one system design is used* for all applications. With definable structure, information is able to be represented according to its own inherent structure. Manipulations are able to be defined according to algorithms' own inherent processes. Given such freedom, it is easy to construct novel machine designs which are 10 to 50 times more powerful than contemporary designs, and which can be interpreted by the B1700's variable-micrologic processor using less than 10 to 50 times the effort, resulting in faster running times, smaller resource demands, and lower computation costs.

GENERAL DESIGN

To accomplish definable structure, one may observe that during the next decade, something less than infinite variability is required. As long as control information and data are communicated to machines through programming languages, the variability with which machines must cope is limited to that which the languages exhibit. Therefore, it is sufficient to anticipate a unique environment for each programming language. In this context, absolute binary decks, console switches, assembly languages, etc., are included as programming language forms of communication. Let us call all such languages "S-languages" ("S" for "soft," or also for "system" or "source" or "specialized" or "simulated"). Machines which execute S-language directly are called "S-machines." The B1700's objective, consequently, is to emulate existing and future S-machines, whether these are 360's, FORTRAN machines, or whatever. Rather than pretend to be good at all applications, the B1700 strives only to interpret arbitrary S-language superbly. The burden of performing well in particular applications is shifted to specific S-machines.

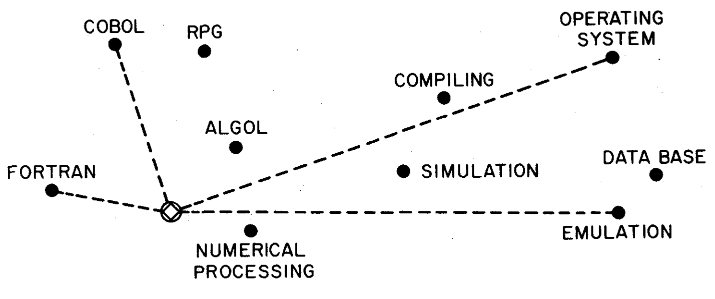

Figure 1—Typical machine design (○) positioned by goodness-of-fit to application areas (•)

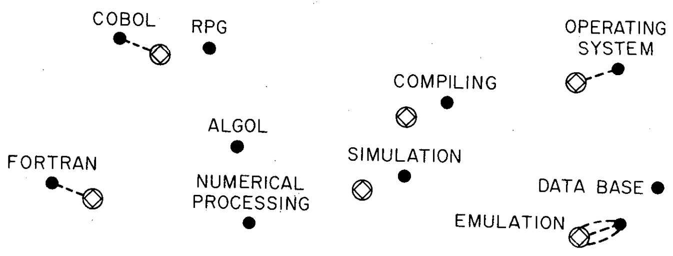

Figure 2—Typical B1700 S-machines (○) positioned by goodness-of-fit to application areas (•)

Throughput measurements, reported below, show that the tandem system of:

APPLICATION PROGRAM,
interpreted by an
S-MACHINE (which is optimized for the application area),
interpreted by the
B1700 HARDWARE (which is optimized for interpretation)

is more efficient than a single system when more than one application area is considered. It is even more efficient than conventional design for many individual application areas, such as sorting.

To visualize the architectural advantage of implementing the S-machine concept, imagine a two-dimensional continuum of machine designs, as in Figures 1 and 2. Designs which are optimally suited to specific applications are represented by bullets (•) beside the application's name. The goodness-of-fit of a particular machine design, which is represented as a point (○) in the continuum, to various applications is given by its distance from the optimum for each application; the shorter the distance, the better the fit, and the more efficient the machine is. Figure 1 dramatizes the disadvantage of using one design for COBOL, FORTRAN, Emulation, and Operating System applications. Figure 2 pictures the advantage of emulating/interpreting many S-machines, each designed for a specific application. Note that emulation inefficiencies must be counted once for each S-machine, since they are all interpreted.

HARDWARE CAPABILITIES

To allow the user's problem statement to dictate the structure of the machine and the semantics of machine operations, new degrees of flexibility and speed are required from hardware, firmware, and software.

Defined-field capability

All information in a B1700 system is represented by *fields,* which are recursively defined to be either *bit strings* or strings of fields. Specifically, bytes and words do not exist.

- All memory is addressable to the bit.
- All field lengths are expressable to the bit.
- Memory access hardware must fetch and store one or more bits from any location with equal facility. That is, there must be no penalty and no premium attached to location or length.
- All structured logic elements in the processor can be used iteratively and fractionally under microprogram control, thus effectively concealing their structure from the user. Iterative use is required for operands which contain more bits than a functional unit can hold; fractional use is required for smaller operands.

Defined-field design gives flexibility because information is represented by recursively defined structures of bits. It also gives speed because all bits in a field (and only those bits in a field) are processed in parallel. Additional speed is obtained from the advanced technology of the B1700 components. Main memory is constructed out of LSI MOS circuits with 1024-bit chips having 180-nsec access time. The B1700 is the first small-scale, general-purpose, commercial computer to use MOS/LSI circuitry in its main memory.

Generalized language interpretation

No machine language is built into the hardware. There is no processor structure or set of machine instructions for which compilers may generate code. Each language to be executed must first configure the B1700 processor into whatever structure is efficient for algorithms in that language. Defined operations on the defined structure are then executed by changeable microprogram. B1700 processors are specifically designed to avoid causing significant differences in efficiency due to differences in such "soft" machine structures and operations.

- Microinstructions are executed at 2, 4, and 6MHz rates using MSI CTL II logic with typical delay of 3 nsec per gate.
- Microcode executes out of main memory. It may be buffered through 60-nsec access bipolar circuits. Such buffering is invisible to the microprogrammer.
- Microprocedures are reentrant and recursively usable; each processor includes a 32-deep stack for fast entry and exit; stack operations are automatic, not microprogrammed.
- Microprograms are not limited in size, nor would large microprograms be inefficient because of size.
- Microcode on the B1700 is compact, economizing storage. COBOL, FORTRAN, BASIC, and RPG language processors as well as second-generation and third-generation emulators have been microprogrammed each in less than 4000 16-bit microinstructions.
- Hardware assists with the concurrent execution of many microprogrammed interpreters. It takes from 14 μsec to 53 usec (at 6MHz) from the completion of an S-instruction for one interpreter until the beginning of an S-instruction for another interpreter, depending on how much of the processor must be reconfigured.

Memory protection, fast interrupt response, and uniform status of microprograms allow each microprogrammer to be unconcerned that other interpreters may be running simultaneously.

Control over binding

While the hardware for defined-field and generalized language interpretation allows a varying processor image for microinstruction to microinstruction, it does not preclude taking advantage of a static processor image. For example, the number of bits to be read, written, or swapped between processor and memory can be different in consecutive microinstructions, but if an interpreted S-machine's memory accesses are of uniform length, this length can be factored out of the interpreter, simplifying its code. In other words, S-memory may be addressed by any convenient scheme; bit addresses are available, but not obligatory for the S-machine.

With these hardware advances, language-dependent features such as operand length are unbound inside the processor and memory buss, except during portions of selected microinstructions. Some of these features have, until now, been bound before manufacture, by machine designers. Language designers and users have been able to influence their binding only indirectly, and only on the next system to be built. On the B1700, the delayed binding of these features, delayed down to the

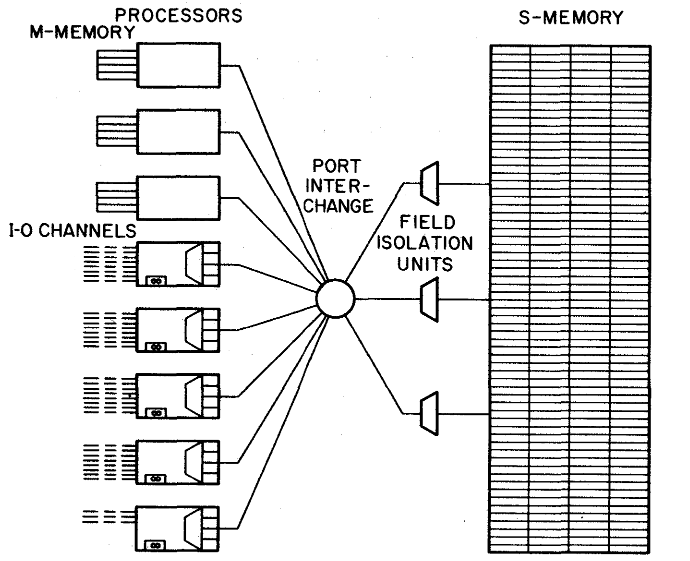

Figure 3—B1700 Organization—Peripherals include standard large-scale devices, data communications networks, and mass storage units as well as minicomputer devices such as paper tape and 96-column card equipment. Special purpose devices include graphics, document sorters, teller machines, etc.

clock pulse level of the machine, gives language designers and users a new degree of flexibility to exploit. Hopefully, this flexibility will lead to the design of languages which are levels closer to user problems. Because of the B1700's interpretation speed, there should be little execution penalty incurred by such advanced forms of man-machine communication.

SYSTEM ORGANIZATION

Extreme modularity improves the B1700's ability to adapt to an installation's requirements. There may be one to eight processors connected to one another and to two to 256 65,536-bit systems memory (*S-memory*) modules, interfaced by a field-isolation unit. ("Field-isolation" refers to converting defined-field memory requests [i.e., least- or most-significant bit address, field length, and direction] into whatever form actually drives the memory and to converting bit strings into whatever form is actually read and written by the memory.) Each processor also connects to one to eight I/O channels or to one to four microprogram memory (*M-memory*) modules. (See Figure 3.) Later systems may have several field-isolation units. With only one processor, the port interchange may be eliminated, as in Figure 4.

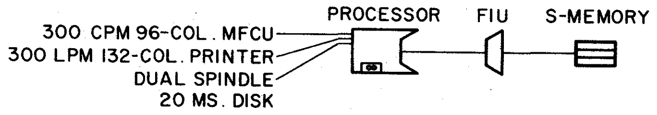

Figure 4—One of the smallest B1700's

EMULATION VEHICLE

Any computer which can handle the B1700's port-to-port message discipline may employ a B1700 for on-line emulation. (See Figure 5.) Programs and data

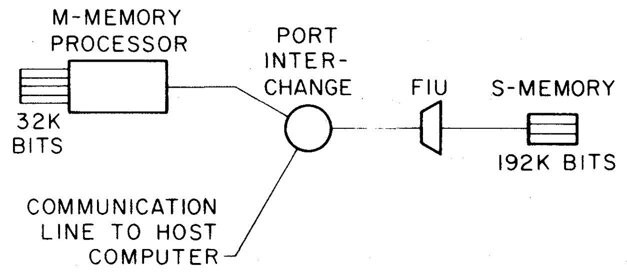

Figure 5—B1700 as an emulation vehicle

are sent to the B1700 for execution; I/O requests are sent back to the host which uses its own peripherals for them. Interpreters are loaded via the B1700's console cassette drive. Each Burroughs emulator can run standing-alone, or in an emulation vehicle, or in a multiprogrammed mix.

STATE OF THE ART DESIGN

The B1700's innovative features have been realized without diminishing the system's ability to provide many proven throughput enhancements. All Burroughs interpreters rely on the B1700's Master Control Program (MCP) for:

- Virtual memory—user programs are not limited in size by the amount of physical storage nor does the programmer ever need to know how much storage is available; compilers automatically segment programs, and the MCP automatically manages these segments without introducing any code into the user program.
- Multiprogramming—because common system

functions such as input/output, storage management, and peripheral assignment are removed from user programs and handled by the MCP, every pause in a running program becomes an evident opportunity to run other programs.
- Multiprocessing—with S-machine state kept in main memory and with every interpreter in main memory, any processor in the system can resume execution of an interrupted program.

The B1700 is the first small-scale computer to offer so comprehensive an operating system.

In addition to the MCP capabilities, there are notable system flexibilities, viz:

- Dynamic system configuration—processors, memory addresses, I/O channels, and peripherals are not uniquely coded into programs, so such entities can be brought on-line and used immediately without any reprogramming.
- Descriptor-organized I/O—in effect, I/O has its own S-language, interpretation of which causes data transfer; it is possible to build this interpretation in hardware, for maximum speed, or it may be soft for maximum flexibility, for example, to allow easy interfacing with new devices.
- System performance monitoring—interpreters automatically gather dynamic execution frequencies of program components to establish which parts of a program take the most time;[3,4] also, specific microinstructions can interface directly with external monitors, allowing soft event flagging.

Interpreter switching

Note that without a native machine language, the MCP itself must be written in higher-level language and interpreted just like any other program. It, and all other active jobs, are represented in memory according to Figure 6. There are read-only code segments which may be anywhere in memory and a write-protected area which contains the program's S-machine state, data segments, file buffers, and other work areas.

One of the MCP's data segments contains an interpreter dictionary that points to each interpreter which is active (i.e., interpreting one of the jobs in the mix). To reinstate a user's interpreter, the MCP extracts from the user's S-machine state the name of the interpreter being used, brings it into S-memory, and calls the interpreter interface routine which switches run structures. Associating S-machines and interpreters symbolically allows such things as several COBOL

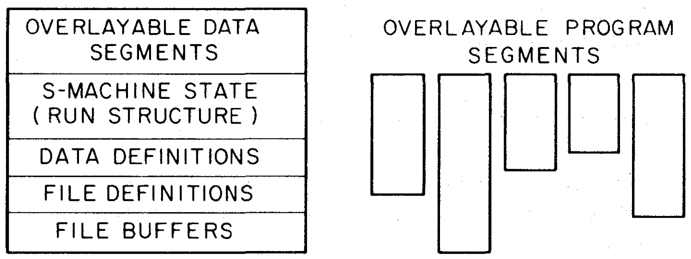
OVERLAYABLE DATA SEGMENTS
S-MACHINE STATE (RUN STRUCTURE)
DATA DEFINITIONS
FILE DEFINITIONS
FILE BUFFERS
OVERLAYABLE PROGRAM SEGMENTS

Figure 6—B1700 program S-memory components

interpreters active in one mix—one designed for speed, another for code compaction, etc.—all employing the same S-language expressly designed for COBOL, that is, a COBOL-machine definition. The interpreter name is looked up in the interpreter dictionary to yield a pointer to the interpreter code in S-memory.

To switch back to the MCP interpreter, a user interpreter performs the identical procedure. It calls the interpreter interface routine, which maintains a pointer to the MCP's interpreter, and switches run structures.

Interpreter switching is independent of any execution considerations. It may be performed between any two S-instructions, even without switching S-instruction streams. That is, an S-program may direct its interpreter to summon another interpreter for itself. This facility is useful for changing between tracing and non-tracing interpreters during debugging.

Interpreter switching is also independent of M-memory. Microcode always actually addresses S-memory. In case M is present, special hardware diverts fetches to it. Without M, no fetches are diverted.

Interpreter management

Entries in the interpreter dictionary are added whenever a job is initiated which requests a new interpreter. Interpreters usually reside on disk, but may be read in from tape, cards, cassettes, data comm, or other media. They have the same status in the system that object code files, source language files, data files, compiler files, and MCP files all share: symbolically-named, media-independent bit strings. While active, a copy is brought from disk, to be available in main memory for direct execution. The location may change during interpretation due to virtual S-memory management, so microinstructions are location-independent.

At each job initiation and termination, the MCP rearranges the interpreters in M-memory to try to avoid swapping. Interpreter profile statistics show that over 99 percent of all microinstructions are executed out of M-memory, even when the demand for M-memory space is double the supply. At higher demand rates, swapping occurs.

Ease of microprogramming

Writing microprograms for the B1700 is as simple, and in some ways simpler, than writing FORTRAN subroutines:

- Microprograms consist of short, imperative English-like sentences and narrative comments. For example, one microinstruction in the FORTRAN interpreter is coded as follows:

  Read 8 bits to T counting FA up and FL down.
- Knowledge of microinstruction forms is not beneficial. Although microprogrammers on other machines need to know which bits do what, on the B1700, there is no way to use that information. Once the function is given in English, its representation is immaterial. The B1700 microprogrammer has only one set of formats to worry about: those belonging to the S-language which he is interpreting.
- Multiprogramming of microprograms is purely an MCP function, carried out without the microprogrammer's knowledge or assistance. Actually, there is nothing one would do differently, depending on whether or not other interpreters are running simultaneously.
- Use of M-memory is purely an MCP function; users cannot move information in and out of M. Other than rearranging one's interpreter according to usage, there is nothing one should microprogram differently depending on whether microinstructions are executing out of M-memory or S-memory. Maximizing use of system resources is beyond the scope of any individual program; responsibility lies solely with the MCP and the machine designers.
- Since all references are coded symbolically, protection is easy to assure. Microprograms can reference only what they can name, and they can only name quantities belonging to themselves and their S-machines. Moreover, artificially generated names (e.g., negatively subscripted FORTRAN arrays) are checked for validity by concurrent hardware.
- Calling out interpreters is simplified by the continuation of Burroughs' "one-card-of-free-form-English" philosophy of job control language. Figure 7 shows the control information which creates a new interpreter (a) from cards, and (b) from a disk file named XCOBOL/SOURCE.
- Association of interpreters and S-language files occurs at run-time. Figure 8 shows the control information which executes a COBOL program named FILE/UPDATE with (a) the usual COBOL interpreter, and (b) another interpreter named XCOBOL/INTERPRETER.
- There is no limit to the number of interpreters that may be in the system (except that no more than 2^{44} bits are capable of being managed by the B1700's present virtual memory property, so a 28,000-bit average interpreter length means there is a practical limit of 628,292,362 interpreters . . . many more than the number of S-languages in the world).

```
(a) ? COMPILE XCOBOL/INTERP WITH
    MIL; DATA CARD
(b) ? COMPILE XCOBOL/INTERP WITH
    MIL; MIL FILE CARD=XCOBOL/
    SOURCE
```

Figure 7—Typical MCP control information for creating interpreters

```
(a) ? EXECUTE FILE/UPDATE
(b) ? EXECUTE FILE/UPDATE; INTERP
    =XCOBOL/INTERPRETER
```

Figure 8—Typical MCP control information for executing programs

Additional information about B1700 microprogramming may be found in Reference 5.

EVALUATION

Evaluation of novel architecture is not merely an unsolved problem; most rational attempts produce worse results than subjective guesses. Consider benchmarks, which measure more system parameters than any other technique. Any benchmark program which runs on the B1700 develops not only an observed running time, but also a program profile which indicates how to reduce that time (possibly by 50 percent or more). What, then, is the true performance of the system? The observed time, even though known inefficiencies are pin-pointed? Half the observed time? Not until the benchmark has been changed.

The point of benchmarks is to have a standard reference which allows the customer to characterize his work and obtain a cost/performance measure. What customer would be satisfied with an inefficient characterization? If the B1700 can show that a program is not using the system well, what good is it as a benchmark? If we change the program to remove the inefficiencies, it is no longer standard. This is a pernicious dilemma.

Even the simplest measure, add time, still published as if it hasn't been a misleading and unreliable indicator for the past 15 years, is void. What is the relative performance of two machines, one of which can do an almost infinite variety of additions and the other of which can do only one or two? The B1700 can add two 0-24 bit binary or decimal numbers in 187 nsec; how fast must a 16-bit binary machine be in order to have an equivalent add time?

Assuming reasonable benchmark figures are obtainable, they would say nothing about the intrinsic value of a machine which can execute another machine's operators, for both existing and imaginary computers; which can interpret any current and presently conceivable programming language; which can always accept one more job into the mix; which can add on one more peripheral and one more memory module, to grow with the user; which can interpret one more application-tailored S-machine; which can tell a programmer where his program is least efficient; which can continue operation in spite of failures in processing, memory, and I/O modules. These characteristics of the B1700, shared by few other machines—no machine shares them all—save time and money, but are not yet part of any performance measurement.

Despite the nullification of measures with which we are familiar and the gargantuan challenge of measuring the B1700's advancements of the state-of-the-art, there are, nevertheless, some quantifiable signs that the system gives better performance than comparably-priced and higher-priced equipment.

Utilization of memory

Defined-field design's major benefit is that information can be represented in natural containers and formats. Applied to language interpretation, defined-field architecture allows S-language definitions which are more efficient in terms of memory utilization than machine architectures which have word- or byte-oriented architecture. For example, short addresses may be encoded in short fields, and long addresses in long fields (assuming the interpreter for the language is programmed to decode the different sizes). Alternatively, address field size may be a run-time parameter determined during compilation. That is, programs with fewer than 256 variables may be encoded into an S-language that uses eight-bit data address fields. Even the fastest microcode that can be written to interpret address fields is able to use a dynamic variable to determine the size of the field to be interpreted.

Just how efficient this makes S-languages is difficult to say because no standard exists. What criterion will tell us how well a given computer represents programs? What "standard" size does any particular program have? We would like a measure that takes a program's semantics into account, not just a statistical measure such as entropy.

If we simply ask how much memory is devoted to representing the object code for a set of programs, we find the statistics of Figure 9.

Language of Sample	Aggregate Size on B1700	Aggregate Size on Other	Other System	Percent Improved B1700 Utilization
FORTRAN	280KB	560KB	System/360	50
FORTRAN	280KB	450KB	B3500	40
COBOL	450KB	1200KB	B3500	60
COBOL	450KB	1490KB	System/360	70
RPG II	150KB	310KB	System/3	50

Figure 9—Amount of program compaction on B1700

In short, the B1700 appears to require less than half the memory needed by byte-oriented systems to represent programs. Comparisons with word-oriented systems are even more favorable.

As to memory utilization, the advantage of the B1700 is even more apparent. Consider two systems with 32KB (bytes) of main memory, one a System/3, the other a B1700. Suppose a 4KB RPG II program is running on each. If we ask how much main memory is in use, we find the comparison of Figure 10.

The utilization at any given moment may be 30 times better on the B1700 than on the System/3. At least, with all program segments in core, it is seven times better (4.5KB vs. 32KB). Even if we assume the RPG interpreter is in main memory and is not shared by other RPG jobs in the mix, the comparison varies

System	Bytes in Use	Percent	Comment
System/3	32K	100	28K is idle without multiprogramming and virtual memory.
B1700	1K	3	Assumes 500B run structure and 500B of program and data segments.

Figure 10—Hypothetical RPG memory requirements

from 6:1 to 4:1, 5KB to 8KB (vs. 32KB), 84 to 75 percent better utilization. As more and more RPG jobs become active in the mix, the effect of the interpreter diminishes, but then comparison becomes meaningless, because other low-cost systems cannot handle so large a mix. (Note that these figures change when a different main memory size is considered, so the comparison is more an illustration of the advantage of the B1700's variable-length segments and virtual memory than of its memory utilization.) More detailed information on memory utilization may be found in Reference 6.

Running time

Although program running time is said to involve less annual cost at installations than the unquantifiable parameter which we may call "ease of use", let us mention some current observations. When the B1700 interprets an RPG II program, the average S-instruction time is about 35 microseconds, compared to System/3's 6-microsecond average instruction time. On a processor-limited application (specifically, calculating prime numbers), the identical RPG program runs in 25 seconds on a B1700 and 208 seconds on a System/3 model 10. Both systems had enough main memory to contain the complete program; only the memory and processor were used.

The B1700 lease rate was 75 percent greater than the System/3's. In terms of cost, the B1700 run consumed 30¢ while the System/3 run took $1.60. In terms of instruction executions, the B1700 was 50 times faster. That is, each individual interpreted RPG instruction, on the average, contributed as much to the final solution as 50 System/3 machine instructions. The fact that the B1700's S-machine for RPG is 50 times more efficient than System/3 seems to support the B1700 philosophy, that interpretation of S-machines which are optimized for each application yields better performance than using a general-purpose architecture.

Using another set of benchmark programs (for banking applications), and another B1700 which leases for the same as the System/3 with which it was compared, throughput comparisons are again noteworthy. Despite defined-field design, soft-interpretation, soft I/O, multiprogramming, multiprocessing, and virtual memory, all of which supposedly trade speed for flexibility, the B1700 executes RPG programs in 50 to 75 percent of the System/3 time, and compiles them in 110 percent of the System/3 time, for the same monthly rental. In applications of this type, compilation is expected annually (monthly at worst) while execution is expected daily. (Systems used for this comparison included a multi-function card unit to read, print, and punch 96-column cards, a 132-position 300 lpm printer, a dual spindle 4400 bpi disk cartridge drive, and operator keyboard. The System/3 could read cards at 500 cpm, while the B1700 could read at 300 cpm.)

CONCLUSION

Microprogramming, firmware, user-defined operators, and special-purpose minicomputers are being touted as effective ways to increase throughput on specific applications while decreasing hardware costs. One standard system tailors itself to an installation's needs. Effective as these approaches are, they are all held back by procrustean machine architecture. Burroughs B1700 appears to eliminate inherent structure by its defined-field and soft interpretation implementation, advancements of the state-of-the-art. Without a native machine language, the B1700 can execute every machine language well, eliminating nearly all conversion costs. Designed for language interpretation rather than general-purpose execution, the B1700 can run every programming language well, reducing problem-solving time and expense. It does not waste time or memory overcoming its own physical characteristics; it works directly on the problems. Furthermore, these innovations are available in low-cost systems that yield better price/performance ratios than conventional machinery.

ACKNOWLEDGMENT

Many of the design objectives were first articulated by R. S. Barton.[7] The author wishes to thank Brian Randell, R. R. Johnson, Rod Bunker, Dean Earnest and Harvey Bingham for their conscientious criticism of various drafts of this article.

BIBLIOGRAPHY

1 A W BURKS H H GOLDSTINE
J VON NEUMANN
Preliminary discussion of the logical design of an electronic computing instrument
A H TAUB (ed) *Collected Works of John von Neumann* Vol 5
The Macmillan Co New York 1963 pp 34-79
Also in
C G BELL A NEWELL
Computer structures: Readings and examples
McGraw-Hill Book Co 1971 pp 92-119

2 W LONERGAN P KING
Design of the B5000 system
Datamation 7 5 May 1961 pp 28-32

3 S C DARDEN S B HELLER
Streamline your software development
Computer Decisions 2 10 October 1970 pp 29-33

4 D E KNUTH
An empirical study of FORTRAN programs
Software—Practice and Experience 1 2 April 1971 pp 105-134

5 W T WILNER
Microprogramming environment on the Burroughs B1700
IEEE CompCon '72
For reprints write to the author at Burroughs Corporation 6300 Hollister Avenue Goleta California 93017

6 W T WILNER
Burroughs B1700 memory utilization
Proc FJCC '72 this volume

7 R S BARTON
Ideas for computer systems organization: A personal survey
Software Engineering 1 Academic Press New York 1970 pp 7-16