B1700 DESIGN AND IMPLEMENTATION

by Wayne T. Wilner, Ph.D

Burroughs Corporation
Santa Barbara Plant
Goleta, California

May 1972

ABSTRACT

Burroughs B1700 embodies a unique design tenet: the work done to accommodate definable machine structure from instruction to instruction is less than the work wasted from instruction to instruction when one machine structure is used for all applications. In other words, execution of machine language using procrustean hardware causes more inefficiencies than soft interpretation of arbitrary machine language on protean hardware.

The programs on the B1700 are not represented in B1700 machine language but in "S-language", that is, some other computer's machine language or a machine language invented (by Burroughs or by the user) expressly for the program's application area. Interpretation of S-language is aided by: (a) ~~bit-addressable memory,~~ (b) ~~parallel access of arbitrarily sized memory fields~~, (c) equal memory speed on all bit locations and bit string lengths, (d) clock-by-clock control over the effective width of processor data paths, registers, and structured logic elements, (e) soft microprograms, (f) English language microprogramming, (g) execution of microcode indifferently from main memory or control memory, (h) enough control memory to hold four interpreters and no limit on the number of interpreters that may be active at any given instant, (i) memory protection, (j) hard and soft interrupts, (k) stack organization, (l) Master Control Program taking full responsibility for efficient system utilization by means of interpreter multiprogramming and multiprocessing, user multiprogramming and multiprocessing, virtual control memory of 2^{24} (over 16 million) bits, virtual main memory of 2^{44} (over 17 trillion) bits, soft interpretation of I/O commands, and (m) automatic program profile statistics. Programs on

the B1700 are independent of: location, storage organization, I/O organization, processor organization, peripheral idiosyncrasies, mix size, and system configuration (except for unique devices). Priced in the lowest system range, the B1700 offers time- and money-saving features not found on systems 100 times more expensive.

The net result of this advanced system design is ease of use, simple programming, lack of conversion costs, improved utilization of system components, and better price/performance.

Keywords: computer architecture, microprogramming, definable structure, B1700, S-language, interpretation

1. 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 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], rigidity 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,

1. See EDVAC, Burks, Goldstine, and von Neumann.

2. See B5000, Lonergan and King.

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.

2. DESIGN OBJECTIVE

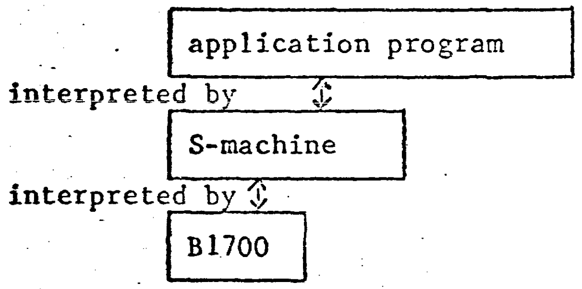
Burroughs B1700 is a protean attempt to completely vanquish procrustean structures, to give 100% variability, or the appearance of no inherent structure. Without inherent structure, there are no word sizes or data formats - operands may be any shape or size, without loss of efficiency; there are no a priori instructions--machine operations may be any function, in any form, without loss of efficiency; configuration limits, while not totally removable, can be beyond state-of-the-art extremes; modularity may be increased, to allow miniconfigurations and supercomputers using the same components.

2.1 Design Rationale

The B1700's permise 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. As long as one can define a machine environment which is more efficient for solving one's problems than a contemporary machine design, one can attain more throughput per dollar. As we shall see, there are novel machine designs which are 10 to 50 times more powerful than contemporary designs, and which can be interpreted by the B1700's variable-image processor using less than 10 to 50 times the effort, resulting in faster running times, smaller resource demands, and lower computation costs.

3. 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 machine 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 "systems" or also for "soft" 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. Throughput measurements, reported below, show that the tandem system of:



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.

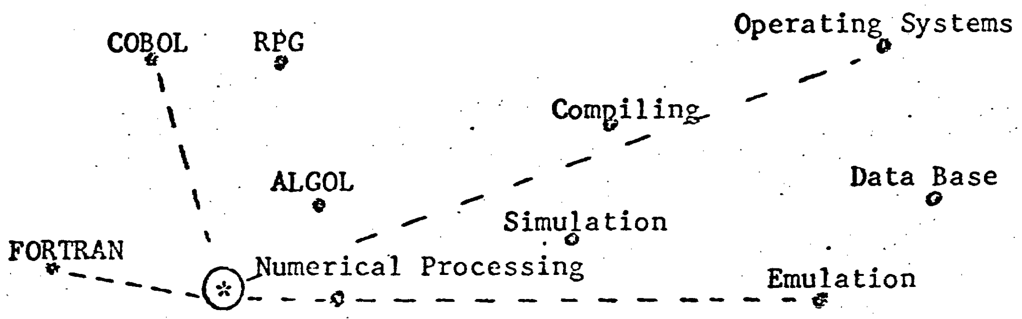


Figure 1. Typical machine design positioned by goodness-of-fit to application areas.

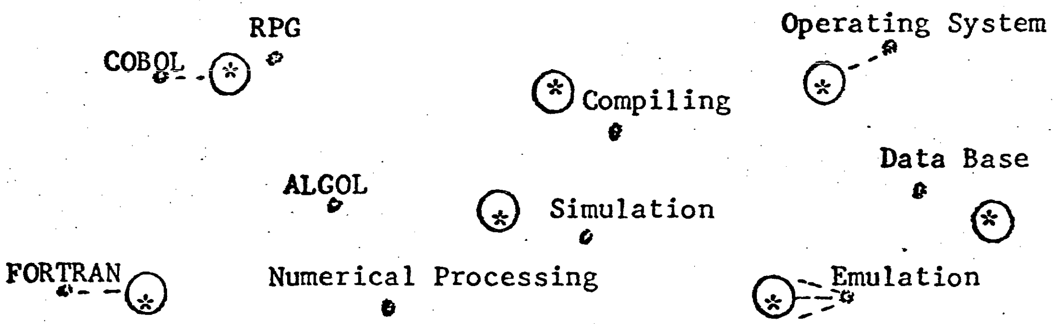


Figure 2. Typical B1700 S-machines positioned by goodness-of-fit to application areas.

4. HARDWARE REQUIREMENTS

Varying the processor's image for each application area implies very specific hardware requirements.

4.1 Defined-Field Capability

All information is represented by fields, which are (recursively) defined to be either bit strings or strings of fields; i.e., bytes and words do not exist.

(a) All memory is addressable to the bit.

(b) All field lengths are expressable to the bit.

(c) 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.

(d) 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.

4.2 Generalized Language Interpretation

(a) The system should be capable of efficient interpretation of a variety of instruction formats.

(b) Format of interpreted instructions should not be predetermined or limited. The structure of the system should not cause any significant difference in efficiency due to selection of format.

(c) Interpretation should be by soft microprogram.

(d) Microprograms should be changeable, stored and executed in main memory. ~~Buffering through faster memory should be invisible to the microprogrammer.~~

(e) Hardware must assist with the concurrent execution of many interpreters, to make switching as rapid as possible.

(f) ~~Microprograms should be reentrant and recursively usable.~~ Microprogram execution is a critical part of the B1700. Some of the objectives involved in the design include: fast entry and exit of microprocedures, compactness of code and economy of storage, and ease of writing and maintaining microcode.

(g) Microprograms must not be limited in size. Execution of microprogram should be invisible to the user and not reflect any variation in microprogram efficiency due to size.

(h) Microfunctions must implement all present and foreseeable higher-level language functions efficiently but without prejudicing implementation of languages. Any function included solely for a single language should be in addition to basic microfunctions which could more generally implement the function.

While the hardware requirements for defined-field design and generalized language interpretation have been stated so as to allow a varying processor image from microinstruction to microinstruction, they do 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 clock pulse level of the machine, gives language designers and users a new degree of flexibility to exploit.

4.3 Advanced Design

On each newly designed system, professional responsibility dictates that previously proven advances in system organization be incorporated.

4.3.1 Virtual Memory

(a) S-programs should not be limited in size; all address fields should be variable.

(b) S-programs should not reflect the storage organization of the system.

(c) Programmers should be given feedback on the size and make-up of their working-sets.

4.3.2 Stack Organization

(a) Programs should be recursive and reentrant.

(b) Subprogram entries and exits should be very fast, to encourage decomposition of programming tasks into small, comprehensible units.

(c) Compilation and execution efficiency should not be dependent on a manufacturer's ability to solve register allocation problems.

4.3.3 Dynamic System Configuration

(a) Code should be independent of system configuration, to allow addition and deletion of processors, memory modules, I/O channels, and peripherals while programs are running.

(b) The system itself (not the user) should be responsible for full resource utilization; hence code should not have to change when the system is reconfigured.

4.3.4 Multiprogramming

(a) The system should run as many jobs at once as are necessary, subject to working-set limits, to keep each resource fully utilized.

(b) Code should be independent of the number of jobs in the mix, to provide equal efficiency when running alone as when running with 100 others.

(c) Memory must be protected from all invalid references (read or write).

(d) Hard interrupts are needed to manage asynchronous events with minimum overhead.

4.3.5 Multiprocessing

(a) The system should allow extra processors to be added to memory-rich or peripheral-rich installations, to balance the system to user workloads.

(b) Program state should be maintained outside of processors, to allow execution by any processor without excessive switching time.

4.3.6 Descriptor-organized I/O

(a) System I/O is itself a unique application, deserving its own S-language and interpreter.

(b) The I/O S-language interpreter may be soft microprogram or a separate, wired processor, incapable of interpreting other S-languages.

(c) With self-describing I/O requests, processor participation in I/O is reduced, improving the system's ability to keep many peripherals in operation simultaneously.

4.3.7 Soft Interrupts

(a) Asynchronous and infrequent events should not require explicit code for their individual detection.

4.3.8 Profile statistics

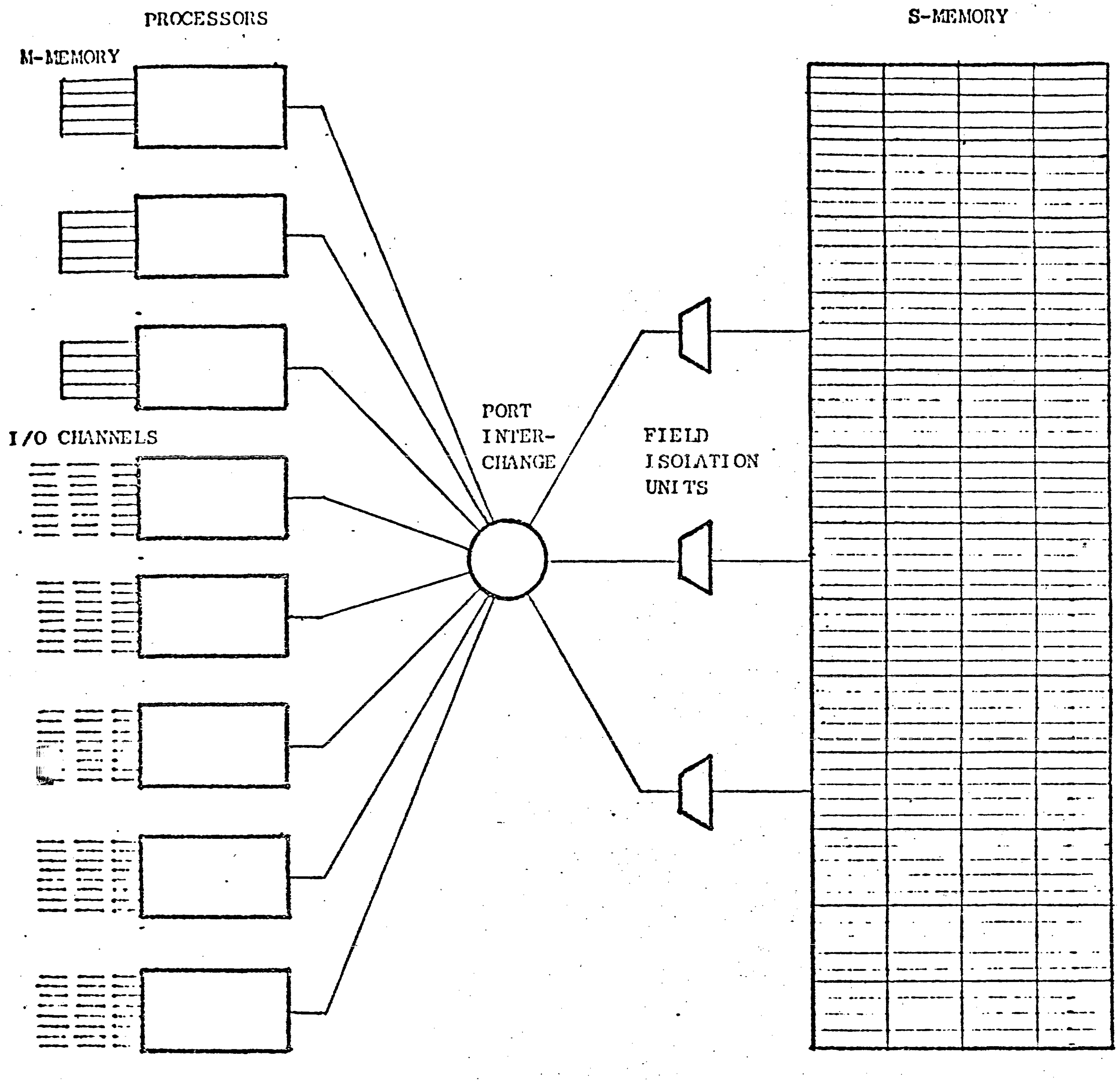
(a) Program behavior should be analyzed and reported back to the programmer, or filed for overall system analysis.

(b) Reports should be in terms of source language.

(c) Reporting which parts consume the most execution time is of primary importance.

(d) Compilers writers ought to be told how their languages are being used.

(e) By instrumenting soft microprogram interpreters for profile statistics gathering, overhead can be kept under 1%.



Each processor may have either 1-8 I/O channels or 1-4 micro-program memory modules of 16,384 bits each.

Each system may have 2-256 systems memory modules of 65,536 bits each.

Figure 3. B1700 Organization

Available peripherals include - Card Readers: 300-1400 cpm models, 80 col.; 300-1000 cpm models, 96 col.; Card Punch, 300 cpm, 80 col.; Card Read-Punch-Print-Sort, 1000/1000/120/120 cpm, 96 col.; Card Record-Read-Punch-Print-Sort, 300/60/60 cpm, 96 col.; Line Printers [illegible]-900 lpm models, 132 col.; Paper Tape Peripherals; Disk Storage 5ms-40ms models, Head-[illegible]-Track 2200-4400 bpi models, movable arm, cartridge; Magnetic Tape 7/9 Track, NRZI/PE models; Sorter-Reader 600-1625 dpm models; Data Communications 150Hz-48KHz+models; Graphics; Terminal Computers, Teller Machines, etc.

5. SYSTEM ORGANIZATION

Extreme modularity improves the B1700's ability to adapt to an installation's requirements. There may be 1-8 processors connected to one another and to 2-256 65,536-bit main memory modules, interfaced by one or more field-isolation units, described later. Each processor also connects to 1-8 I/O channels or to 1-4 microprogram memory modules. (See Figure 3.)

With only one processor, the port interchange may be eliminated, as in Figure 4.

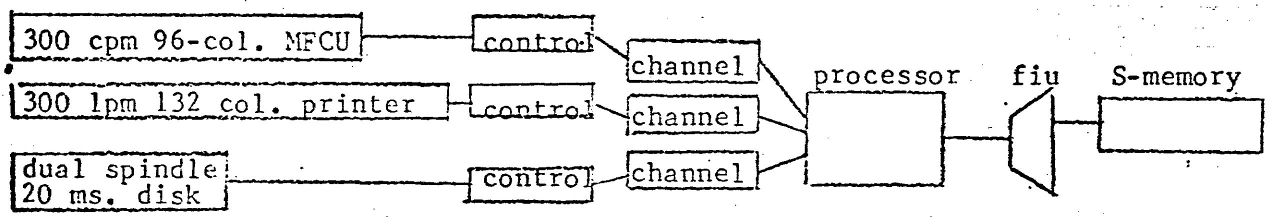


Figure 4. One of the smallest B1700's.

Rental of the system in Figure 4 is expected to be under $1500/month.

6. 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.)

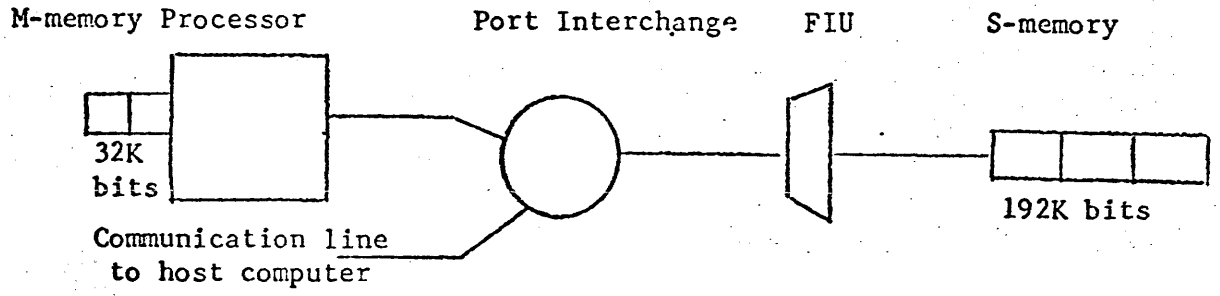


Figure 5. B1700 as an Emulation Vehicle.

Programs and data 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. Present interface specifications expect one interpreter to be in M-memory at a time. Rental of the system in Figure 5 is expected to be under $2200/month.

7. DEFINED-FIELD DESIGN

7.1 [redacted]s

The mechanisms by which the processor and microinstructions automatically handle variable operand lengths and formats have come to be called "bias" facilities. In addition to a normal complement of registers, functional units, and data paths, each processor has [redacted]

> [redacted] as many as the processor can handle, and
>
> [redacted]ich specifies the unit of information, viz., bit, BCD, USASCII byte, EBCDIC byte, along with some open codes for future use.

~~Length-dependent operations, such as add and subtract, are controlled by CPL, e.g.,~~ to determine when carries occur out of the apparently highest bit position. Memory-accessing microinstructions may choose to access only as many bits as CPL specifies. ~~Format-dependent operations always reference CPU to bias combinatorial circuits, such as add and subtract for selecting binary or decimal results~~. Thus, microprogram sequences which load CPL and CPU from data descriptors (or even from the computer operator's keyboard) during an instruction interpretation behave correctly whether binary or decimal operands are supplied and whatever size the operands are. The microprograms are invariant to actual operand details, so the B1700 hardware appears not to have inherent structure.

Iteration over operands which are longer than the processor can handle is accomplished by two unique microinstructions, Bias and Count F. ~~Operands are normally referenced in a pair of field definition registers called F and SF~~. These have 24-bit subregisters, FA and SFA, for the data's bit address, 16-bit subregisters, FL and SFL, for the data's length (bit strings are thus limited to 65,535 bits), and 4-bit subregisters, FU and SFU, for format information. The Bias instruction computes the format and number of bits to bring to the processor for an iteration by setting CPU from FU or SFU and by setting CPL to FL, SFL, itself, a literal value, or the minimum of any set of these. The Count F micro uses CPL or a literal value by which to increment and decrement FA and FL; this indexes through an operand by defining contiguous subfields on which the processor may operate. To handle indefinitely long operands, then, one first writes a microprogram which assumes that the processor registers are indefinitely long. One suffixes the program with a Branch micro to repeat the code. One prefaces the program with a Count F and Bias pair, the Count F to define suboperands that are small enough to fit in the physical processor, and the Bias to compute the suboperand length (so that the operand need not be a multiple of the processor width), to load the bias registers, to test for completion, and to bypass the program when the long operand is completely processed. Such a microsequence can handle zero-bit to 65,535-bit operands indiscriminately.

7.2 Bit-oriented Memory

To implement bit-oriented memory at low cost, one uses conventional memories and a memory-requestor interface which is called a field isolation unit, or F.I.U. (see Figure 6). The FIU's tasks are to convert bit addresses, field lengths, and field directions (i.e., address refers to most- or least-significant-bit) into conventional addresses, to align requestor bit strings with actual memory containers, and to mask off nonparticipating bits. During memory operations, bytes are read out of memory into the Memory Information Register, MIR, a bit string is extracted or inserted as the information passes through the rotator into either buffer, and then the buffer is gated to its destination.

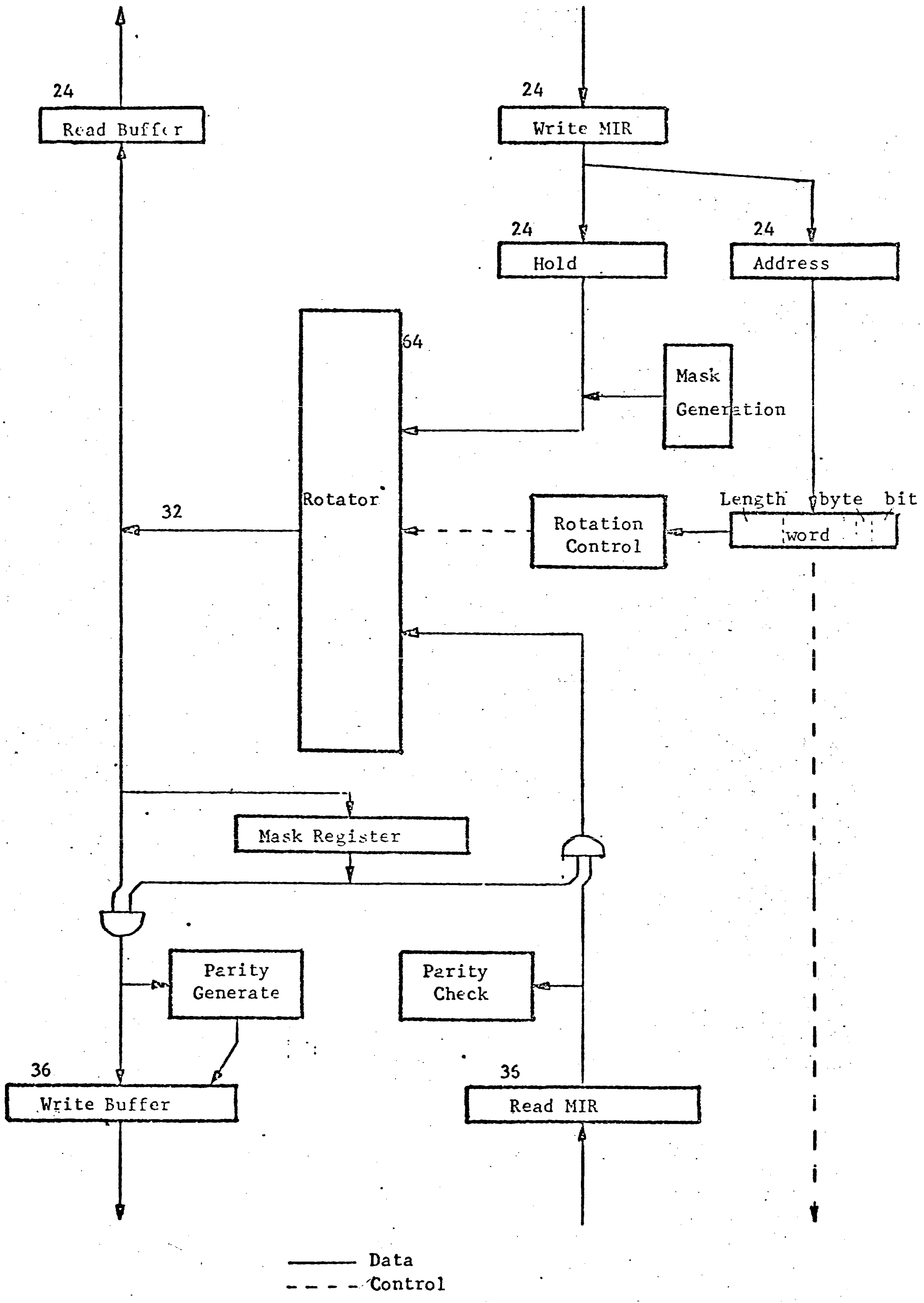


Figure 6. Field Isolation Unit (Memory/Requestor Interface)

Any port on the interchange (see Figure 3) may read bit strings from memory. The port must supply a bit location, a string length (in bits), and a field direction (one bit meaning forward or reverse) to the FIU, which places the information into its Memory Address Register. Because fields may be manipulated in either direction, locations actually refer to between-bit positions, as illustrated in Figure 7. The thirteen bits from location 13 forward are the same thirteen as accessed from location 26 backward. This simplifies microprogramming by naming bit strings in a manner similar to the way in which we think about them.

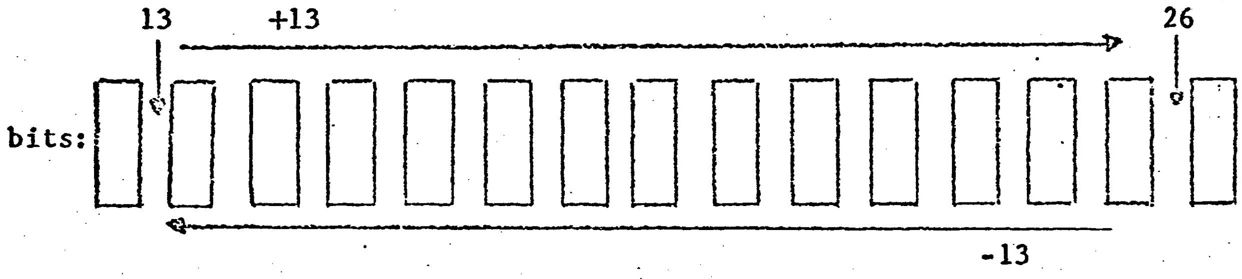


Figure 7. Defined-field addresses labelling between-bit positions.

Given a field location, the FIU calculates which byte in the conventional memory contains the leading bit. On the B1700, there are four 9-bit memories (one bit of which is used for parity). Consequently, the low three bits of an address specify a bit position within a word, the next two bits specify one of the four memories, and the high 19 bits constitute a conventional address. The leading bit is in the word specified by the high 21 address bits. This word, and the corresponding words in the other memories, are brought to the FIU's Read MIR. For example, the accessed words which satisfy a request for the 13-bit field beginning at location 13, forward, are shaded in Figure 8. The field itself is doubly shaded. From the low three bits and direction of a request, the FIU is told how many positions and in what direction to rotate the received field in order to right-justify it. The request length is used to create a mask which zeros the unneeded high bits. The isolated field is then sent to the port interchange, as shown in Figure 9.

```
Request:
Location--
   0000000000000000000          word,
                      01        memory,
                        101;    bit
Length--
          0000000000001101;
Direction--
                         0;
```

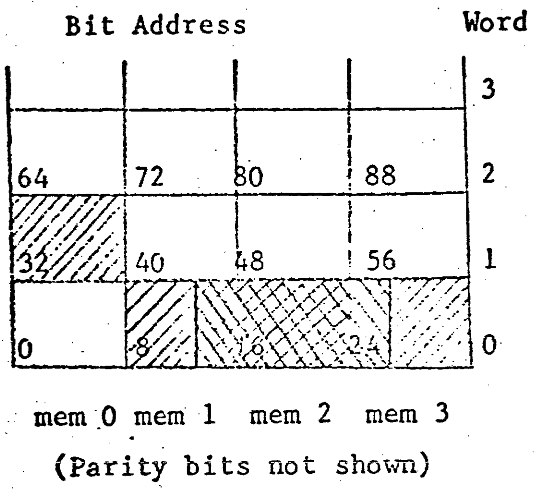


Figure 8. Words (shaded cells) delivered to FIU satisfying request for 13 bits beginning with location 13, forward.

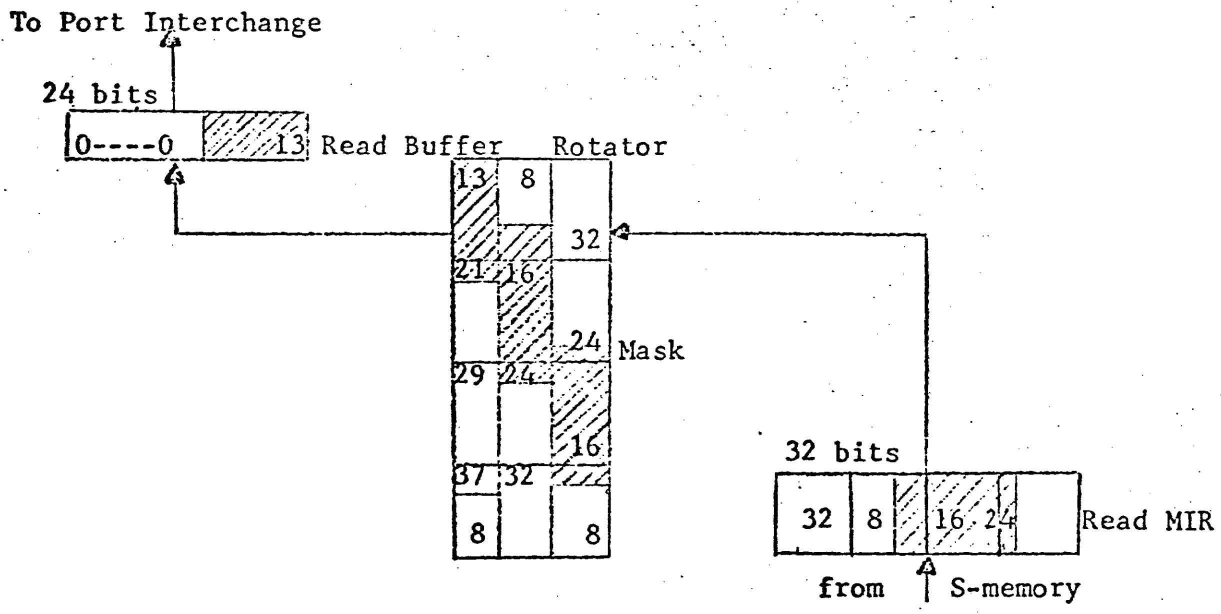


Figure 9. Read field isolate.

On the B1700, the field is available at a processor's port 668 nsec after the read request is initiated, assuming the port interchange and memory are both free. The port and FIU are involved during the last 500 nsec.

To write bit strings into memory, the FIU accepts a request (i.e., location, length, direction) and performs a read cycle to bring the receiving memory field to the Read MIR and right-justified through the rotator. It then accepts a bit string from the port interchange which goes through the Write MIR and is masked into the memory field at the rotator. Then the rotator returns the information to word-alignment, and passes it through to the write buffer, where it returns to memory. Write cycles involve ports for 500 nsec and the FIU for 1169 nsec.

Since writes are performed as Read-Modify-Write, the addressed field, as it was before modification, is available in the read buffer for the requesting port. Thus, fields may be swapped between memory and a processor by one microinstruction. It is also possible to force two back-to-back requests, which gives any processor the ability to test and set, and restore a bit string in one uninterruptable operation. This is vital for administering multiprocessor, multiprogramming environments because it can be used to prevent system deadlocks.

Because the amount of information which resides with the processor is so large (registers plus M-memory), one processor uses only 20% of the available S-memory cycles, which enables one FIU to support several processors. In addition, almost 80% of processor requests are reads.

Of the 41 bits required to specify a memory request, typically 5-10 come from the S-instruction. The rest come from other S-machine state. They represent the context within which the S-machine is working. The length field, for example, constantly is 16 when the S-machine is the IBM 1130. Consequently, little work is needed to maintain such large address fields.

8. SOFT INTERPRETATION

All Burroughs-supplied interpreters rely on the B1700's Master Control Program (MCP) for all input/output, virtual S-memory, virtual M-memory, multiprogramming and multiprocessing of user programs, multiprogramming and multiprocessing of interpreters, and standard functions. The MCP is written entirely in a higher-level language (a synergism of features from COBOL, PL/I, and XPL) which is interpreted itself. To provide for smooth change of control, a microprogrammed interpreter interfacing routine, named Gismo, exists in the beginning memory locations. By loading a pointer to another program into a processor register and clearing the microprogram address register, each interpreter transfers to Gismo. Gismo uses the program pointer to establish processor context within the new interpreter. Subsequent microinstructions are taken from the new interpreter.

8.1 Interpreter-Machine Interface

Given this interpreter interface, simple, hardware-oriented tasks such as interrupt decoding and priority resolution, M-memory overlay, and I/O transfer, can be included in the interface routine, simplifying the requirements of an interpreter.

An interpreter has control of a processor until interrupted by another element in the system or by programmatic interrupt. Between each S-instruction interpretation (approximately every 35 usec.), every interpreter must examine the processor's interrupt register to detect the need for change of control. If an interrupt is present, the interpreter calls (instead of transfers to) Gismo, leaving a constant in a register which directs Gismo to decode the interrupt. For simple functions such as timer interval or I/O transfer, Gismo actually performs the required actions. For other interrupts, Gismo returns an appropriate description to the calling interpreter.

If the interpreter can handle the interrupt, it does so and continues with the next S-instruction if all interrupts are quiet. If it cannot, it moves its S-machine's state outside of the processor, loads a constant which means "call MCP", and transfers to Gismo. Very little state needs to be saved because no S-instruction is in the middle of interpretation.

Real-time interrupts are distinguished by hardware, but are handled in exactly the same way. When Gismo is told to "call MCP", it may select a non-MCP program to process real-time interrupts (such as a COBOL routine which performs a pocket select for a document-sorter).

Gismo is also called by the MCP to move interpreter segments into M-memory, by all programs to initiate I/O, and by the computer operator to perform some start-up or post-mortem utility functions.

8.1.1 S-machine Switching

Note that change of control is between S-machines, which does not necessarily mean a different interpreter is needed. (When all user programs are written in the MCP's language, only one interpreter is active.) Each job is represented in S-memory according to Figure 10. All but one segment is read-only code. The one is called the "run structure" and consists of: I/O buffers; data; descriptions of devices and operands; and the run structure nucleus which contains the job's S-machine state and MCP-needed control information. All segments (except the run structure nucleus) move into memory under control of the MCP. The data section may or may not be administered internally by a virtual memory discipline. Note that code segments never are written out of memory because they never change. The space they occupy is always available for other uses.

The MCP's run structure includes an interpreter dictionary, each entry of which describes an interpreter (either active in S-memory or on disk).

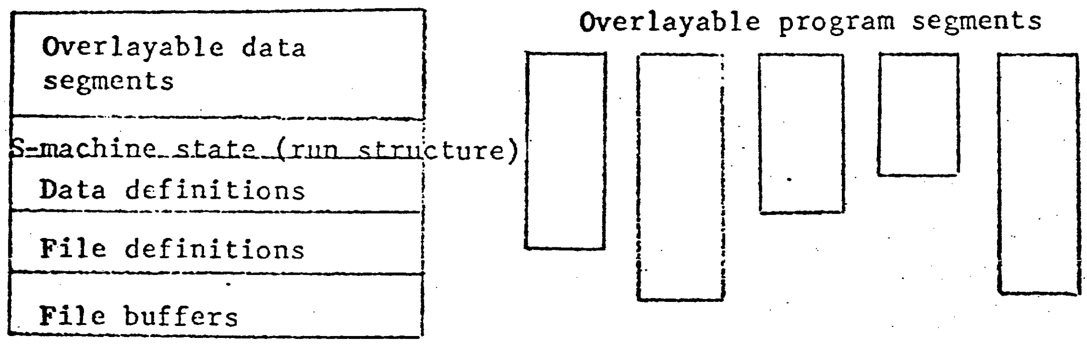

Overlayable data segments
S-machine state (run structure)
Data definitions
File definitions
File buffers
Overlayable program segments

Figure 10. Program components.

To reinstate a user's interpreter, the MCP extracts from the user's S-machine state the name of the interpreter being used. The interpreter name is looked up in the interpreter dictionary to yield a pointer to the interpreter code in S-memory. The MCP's interpreter then saves its S-machine state, loads the pointer into a hard register, and resets the Microprogram Base Register and Microprogram Address Register (to leave the MCP's interpreter's code space). The next micro is brought from Gismo, which uses the hard register to load the Microprogram Base Register, transferring microinstruction fetches to the new interpreter's code space. Associating S-machines and interpreters symbolically allows such things as several COBOL 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).

To switch back to the MCP interpreter, a user interpreter obtains a pointer to the MCP's run structure from the user program's run structure and performs the identical procedure.

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. The Microprogram Address Register always actually addresses S-memory. In case M is present, special hardware diverts fetches to it, whenever the Microprogram Limit Register indicates that M's contents mirror the portion of S-memory being addressed. Without M, no fetches are diverted, and Gismo sits in low S-memory.

8.2 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 M-memory for the processor being readied according to five strategies:

(a) Abundant M

Condition:	All active interpreters can fit in M.
Action:	Place all interpreters into M.

(b) Ample M

Condition:	All active interpreters can be granted at least their minimum M request (usually 1000 words = 1000 16-bit micros).
Action:	Divide M evenly and place part of each interpreter in M.

(c) Adequate M

Condition:	Several interpreters can be given their minimums, but not all.
Action:	Give the MCP's interpreter about 1000 words; divide the rest into 1000-word blocks and swap all user interpreters in and out during reinstate operations.

(d) Precious M

Condition:	Only two interpreters can be given their minimums.
Action:	Give the MCP its minimum; swap all users in and out of the rest.

(e) Bare M

Condition:	Only one interpreter can be given a minimum demand.
Action:	At each interpreter switch, place one interpreter into all of M.

Interpreter profile statistics show that 1000 micros (1000 words) account for over 99% of all instructions executed, even though most interpreters are 2000 words long. If a microprogrammer is prudent enough to rearrange his code according to usage, then an interpreter requesting 1000 words of M as a minimum may be as efficient as one requesting 2000 words.

8.3 Ease of Microprogramming

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

(a) Microprograms consist of short, imperative English sentences and narrative comments. For example, one subroutine in the FORTRAN interpreter reads as follows:

```
*     Decimal to binary conversion
*             Source:  addressed by F;  1-13 digits
*             Destination:  L Y, initially zero
Decimal-to-binary
     Read 8 bits to T counting FA up and FL down    obtain a char
                                                    and address the next one
     Move Tf to X                                   strip off zone bits
     Call Add-X-to-LY                               LY ← LY + X add to partial sum
     If FL=0 then exit                              quit after last char
     Move L to X                                    allow another digit:
     Call Multiply-XY-by-10                         TL ← 10 x XY
     Move L to Y                                    LY ←
     Move T to L                                    LY ←
     Go to Decimal-to-binary                        repeat
```

(b) 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.

(c) 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.

(d) Use of M-memory is purely an MCP function; the resident interpreter interface alone can 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.

(e) Since all references are made 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, names cannot be artifically generated, as they can in FORTRAN (e.g., by negative subscripts, or by call-by-value parameters used in call-by-reference constructs).

(f) Calling out interpreters is simplified by the continuation of Burroughs' "one-card-of-free-form-English" philosophy of job control language. Figure 11 shows the control information which creates a new interpreter (1) from cards, and (2) from a disk file named XCOBOL/SOURCE.

```
(1)   ?  COMPILE XCOBOL/INTERP WITH MIL; DATA CARD

(2)   ?  COMPILE XCOBOL/INTERP WITH MIL; MIL FILE CARD =
         XCOBOL/SOURCE
```

Figure 11. Typical MCP control information for creating interpreters.

(g) Association of interpreters and S-language files occurs at run-time. Figure 12 shows the control information which executes a COBOL program named FILE/UPDATE with (1) the usual COBOL interpreter, and (2) another interpreter named XCOBOL/INTERPRETER.

```
(1)  ?  EXECUTE FILE/UPDATE

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

Figure 12. Typical MCP control information for executing programs.

(h) 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).

9. VIRTUAL MEMORY

On the B1700, S-language addresses may be 44 bits long (disregarding the possibility of using magnetic tape as a backup for main memory), even though each system will have 16 million bits of S-memory or less. M-addresses are 24 bits long, even though each system will have 65,536 bits of M-memory or less. Virtual memory is the mechanism by which the B1700 accommodates memory references to more bits than are physically attachable.

9.1 Virtual M-memory

As control is switched to an interpreter, two registers are set by the M-memory manager: [redacted], which contains the [redacted]. Micros are always addressed relative to the start of an interpreter. [redacted]

This is not a true virtual memory scheme since early interpreter locations are always in M-memory, and later locations never are. It does have the necessary property, however, that the actual amount of M-memory never impacts the program. Interpreters execute identical sequences of instructions for a given task as the amount of M-memory varies. The microprogrammer never takes cognizance of the actual amount of M-memory that is present. (He should, of course, arrange his interpreter with the most often used parts first.)

9.2 Virtual S-memory

The B1700 MCP maintains a large disk area for program pieces that are not in use and can be kept outside of main memory. These pieces may be any number of bits long (up to 2^{24}). Whether they are segments, pages, arrays, or otherwise, depends on the S-machine from whose environment they came. All Burroughs S-machines designed for specific programming languages (e.g., COBOL, FORTRAN, RPG, BASIC, ALGOL) make references through descriptors, or interpretable pointers. These descriptors define areas within an S-machine's data or code space; references are relative to the start of the described area. The descriptions themselves are relative to the start of the S-machine's space, or to an MCP back-up area. When a reference selects an area which is not in memory, the MCP initiates a disk request and temporarily runs another job. When the absent area is brought in, its descriptor is changed to indicate the new location. The reference is retried when the job is selected again.

S-machines are free to manage their own data and code spaces, with and without the MCP's assistance.

Within an S-machine, only its own data and code spaces are accessable. Each machine environment is represented as if it began in location zero and extended throughout all memory, possibly up to 2^{44} (over 17 trillion) bits' worth. The B1700 monitors all references to main memory to verify that they lie between locations contained in the hardware's Base and Limit registers which are set by the MCP during reinstating. Illegal references cause a hard interrupt which transfers control from a user interpreter back to the MCP's interpreter, preventing meddling in other S-machine's areas.

10. STACK ORGANIZATION

Ea~~ch B1700 processor has a hardware stack of 32 24-bit cells. This machinery is intended solely for microprogram recursion during the interpretation of a single S-instruction.~~ Present interpreter switching assumes that no information remains pushed into the stack.

Other hardware makes S-machine stacks easy to implement. The ability to read and write in both directions in S-memory, and to count field definition registers' subfields up and down independently give microprograms read-and-pop and write-and-push operations which can be carried out by one microinstruction.

11. DYNAMIC SYSTEM CONFIGURATION

Since no information (other than a few MCP tables) contains absolute memory locations, the presence or absence of particular memory modules is irrelevant. Should one fail, it may be taken off-line and repaired without disturbing the rest of the system. Conversely, when memory modules are added to the system, the B1700's location-independent segments can be placed in them as soon as they are brought on-line. No code in the MCP nor in any user program need ever be revised due to memory reconfiguration.

Likewise, the identity of any processor, I/O channel, or peripheral is not represented in any user program or MCP routine. When particular devices are needed, their identity is looked up using a symbolic reference each time a device is accessed. Consequently, devices may leave the system without inhibiting any program from running (unless the MCP is not able to create a pseudodevice to hold the I/O requests until the real device is available again). Further, devices may enter the system and be fully utilized without changing any user or MCP code.

12. MULTIPROGRAMMING

A misunderstood concept, multiprogramming refers to the interlaced processing of independent programs using as much of an entire system's resources as are required. It is usually confused with partitioning, which is the interlaced execution of independent programs using part of a system's resources. Under multiprogramming, two three-tape sorts which use 24K of core can be run together on a three-tape, 24K system (the MCP must utilize three pseudo-tapes); a partitioned system needs six tapes or 48K or both.

A simple way to implement multiprogramming is to represent each program in main memory exclusively; that is, no state information or temporary results are kept in a processor...everything is available in memory. On the B1700, this is true of each S-machine. To interlace processing of each program, imagine directing the processor to execute exactly one instruction from each program, in round-robin fashion. Since everything necessary for any instruction's execution is represented in memory, no difficulties are encountered in passing from one program to the next. Such an approach, however, denies the efficiencies which can be obtained by successive instruction executions in one processor. The B1700 takes an intermediate approach which is made feasible by its descriptor-organized virtual memory scheme. [redacted], either for I/O or virtual memory management, [redacted] When a program can go no further by itself, its state is recorded in memory and another task is taken up. It is a happy discovery that such breakpoints occur often enough to keep the MCP attentive to the needs of many jobs. This scheme is further refined by allowing programs to become dormant, which permits their state information to be taken out of main memory. Subsequent I/O operations on which dormant programs may be waiting carry some form of identification which ties them to the dormant program, and causes program state information to be brought back.

13. MULTIPROCESSING

Multiprocessing is the concurrent execution of more than one processor on independent programs. The processor-independent program state, which is used to keep multiprogramming simple, automatically allows any program to be resumed by any processor, as long as each processor can address all of memory, communicate with the entire I/O system, and has access to the interpreter named in the program's run structure (see Figure 10). All of these conditions are always true of B1700 processors.

14. DESCRIPTOR-ORGANIZED I/O

Because I/O processing is often not directly dependent on subsequent program steps, greater throughput can be achieved by overlapping I/O processing with other processor activity. So, to reduce B1700 processor involvement with I/O, requests take the form of descriptors (interpretable programs) whose effect, when interpreted, is the I/O function. To activate a request, a processor sends a port-to-port message across the port interchange. The message locates an I/O descriptor in main memory which an I/O processor will interpret. Non-I/O processors are thus relieved of the intricacies of device and channel communication.

On one-processor B1700 systems, the only I/O descriptors which are fabricated are self-evident to the device controls themselves. Each contains a literal device name and an opcode for the device, as well as the addresses to be used for information transfer in and out of memory. On multi-processor B1700 systems, the opportunity to create arbitrary descriptors is present, enabling file-oriented activities, such as record accessing, searching, and sorting, to be carried out in the I/O realm. In addition, new device disciplines can be accommodated by new microcode for the I/O processors.

15. SYSTEM PERFORMANCE MONITORING

15.1 Profile Statistics

Whereas hardware receives extensive and penetrating scrutiny while it is being designed, software is normally constructed with only the programmer's intuitions about its efficiency to guide its design. The performance measurement technique of profile statistics, the association of code usage with a program's source language, has been reported to help improve a program's running time by a factor of two to ten. (See Darden and Heller, or Knuth [Profile]). To help B1700 users obtain the greatest throughput per dollar, each Burroughs interpreter can gather profile statistics about a program which it is interpreting and present them at the end of a run.

At compile time, a user may indicate which portions of his program are to receive more or less scrutiny. These indications define a set of program segments whose usage is to be recorded by means of an inserted S-language monitoring command.

The compiled program consists of: the code segments expanded with monitors (by less than 1%), textual information which will be used to describe the participating program segments in terms of source language, and an array of cells for the frequency counts. Interpretation of the monitors appears to extend execution times by less than ½%. After execution, the weighted frequency counts show which program segments account for most of the running time. Reprogramming these critical segments for efficiency will reduce running times the most.

Microprogramming can easily allow dynamic measurements of other properties as well, with similarly small overhead.

15.2 "Monitor" Microinstruction

One microinstruction, Monitor, simply presents a user-specified bit pattern at designated pins in the processor backplane and frontplane. This allows unique software "events" to be identified by external hardware, which greatly simplifies the task of knowing what the system is doing.

Each higher-level language has been extended to include a construct which generates a Monitor S-instruction for each interpreter to carry into an identical microinstruction. Event flagging is thus available to all programmers.

16. EVALUATION

The B1700's ability to provide profile statistics at negligible cost voids all known system performance measures. 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 an indication of how to reduce that time (often by more than 50%). What, then, is the true performance on the system? Not the observed time, because inefficiencies are pin-pointed. Half the 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 more performance than comparably-priced and higher-priced equipment.

16.1 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 following statistics:

Language of Sample	Aggregate Size on B1700	Aggregate Size on Other	Other System	% 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%

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:

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

In other words, 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 that the RPG interpreter is in main memory and is not shared by other RPG jobs in the mix, the comparison varies from 6:1 to 4:1, 5KB to 8KB (vs. 32KB), 84% to 75% 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.)

16.2 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 particular configurations leased for $3500 (B1700) vs. $2000 (System/3). 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. When one considers that RPG is the only programming language on the System/3, it is incredible that System/3 seems so poorly equipped to run RPG programs. It is even more incredible because the B1700 really has no S-language expressly for RPG; it uses the COBOL S-language instead. The likelihood of an S-machine more than 50 times more efficient than System/3 is almost certain. This seems to support the B1700 philosophy, that interpretation of S-machines for each application is more efficient than using a general-purpose architecture.

Using another set of benchmark programs (for banking applications), and another B1700 which leases for $2000, throughput comparisons are again astounding. On the one hand, we are comparing a defined-field, soft-interpreting, soft-I/O-processing machine using pre-release compilers, interpreters, and MCP routines, under multiprogramming, multiprocessing, virtual memory systems design, against, on the other hand, a byte-oriented, hard-wired system with two years' field testing, five software releases, batch-processing, one cpu, and 32K maximum main memory. Despite all of the B1700 features, which supposedly trade speed for flexibility, the B1700 executes RPG programs in 50% to 75% of the System/3 time, and compiles them in 110% 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.)

17. 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. Standard system modules may be tailored 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. Both are advancements of the state-of-the-art. Now one machine can execute every machine language well, eliminating nearly all conversion costs. One machine can interpret every programming language well, reducing problem-solving time and expense. The B1700 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.

18. ACKNOWLEDGEMENT

Many of the design objectives were first articulated by R. S. Barton [BARTON]. 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.

19. BIBLIOGRAPHY

BARTON Barton, R. S., "Ideas for Computer Systems Organization: A Personal Survey", Software Engineering, vol. 1, Academic Press, New York, 1970, pp. 7-16.

B5000 Lonergan, W., and King, P., "Design of the B5000 System", Datamation, vol. 7, #5, (May 1961), pp. 28-32.

EDVAC Burks, A. W., Goldstine, H. H., and von Neumann, J., "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument", in Taub, A. H. (ed.), Collected Works of John von Neumann, vol. 5, The Macmillan Co., New York, 1963, p. 34-79.

Also in Bell, C. G., and Newell, A., Computer Structures: Readings and Examples, McGraw-Hill Book Co., 1971, pp. 92-119.

PROFILE Darden, S. C., and Heller, S. B., "Streamline Your Software Development", Computer Decisions, vol. 2, #10 (October 1970), pp. 29-33.

Knuth, D. E., "An Empirical Study of FORTRAN Programs", Software--Practice and Experience, vol. 1, #2 (April 1971), pp. 105-134.