The design and implementation of a small scale stack processor system*

by MICHAEL J. LUTZ

State University of New York at Buffalo
Amherst, New York

INTRODUCTION

A striking phenomenon in the current state of the art in computer technology is the rapidly growing power of mini-computers. One reason for this power is the ability of small computer systems to adapt to specific uses, making them an attractive and economical alternative to large- or medium-scale general purpose systems for many applications. The provision of micro-programming on many of these systems has much to do with this adaptability, since it permits the efficient design and implementation of a virtual machine suited to the needs of the particular application or intended use of the system. In this way the bare hardware can be molded to support the necessary (and often sophisticated) data and control structures desired.

In this paper one such application is presented—the implementation of a machine designed to support block-structured languages effectively in its "hardware". The resulting architecture is quite complex, and would most probably not be economical for such a small scale system (approximately 8K words of memory) were it to be built directly in hardware. Microprogramming made the implementation practical, however, and it is presented in this paper to provide an example of the potential which mini-computers possess in the current era of computer design.

The remainder of this paper is divided into 4 major sections. The following section gives the rationale for designing a machine around block-structured languages, and those data and control constructs which are considered important enough to be explicitly included in the design. It is followed by a section on the way these abstract constructs are actually supported in the architecture, as well as some examples of how this leads to conceptually clean implementations of various features in the system. A quick presentation of the micro-machine and the emulator which creates the virtual machine on top of it is then presented, with some emphasis on the resource mapping used and the microcoding techniques employed.

Finally, a brief discussion of the advantages of microprogramming is given, both from the economic standpoint and in relation to technical benefits.

DESIGN CRITERIA AND CONCEPTS

The design of this processor was motivated by the desire to create a computer whose organization was explicitly language-oriented in nature. To us, (and to others, see References 6, 7 and 11) the convolutions compilers must go through to produce decent object code for the majority of computers indicate a great disparity between the desires of the users of a language and what is provided for in the machine. The added overhead of run-time software to create the illusion of the required environment further attests to the fact that computers in general do not behave as we would like them to. Thus we desired a processor whose operation included features normally supported by these run-time routines, and whose organization was tailored to the compilation of efficient object code in a straightforward manner.

A critical parameter in the design was the class of languages around which to model our machine. It is not at all obvious that any one architecture could be found to provide effective support for every higher level language, since there are a multitude of different and conflicting conventions in use. For example, the definition of accessing environment in ALGOL-60 and APL/360 are quite different, and it may well be that any construct general enough to support both definitions may not enhance the implementation of either. The class settled upon was the block-structured languages, especially those semantically close to ALGOL-60.

This choice was not made capriciously; rather it was based on observations about the current state of programing practice. One strong motivation for the choice is that there is quite a bit of experience in the design of compilers for such languages, and the data structures most convenient for their support have become fairly well known. Also, many algorithms have been coded in these languages, giving us two additional benefits: (1) insight into the most used features of such languages and (2) a large base of programs with which to compare our design with other systems.

*This work was done as part of Project Mu in the Department of Computer Science and is supported in part by NSF Grant GI-993.
These features, attractive as they might be, are insufficient to justify the choice. More convincing are arguments which arise based on the desire to implement an entire system with a multitude of tasks, some of which are cooperating and others of which are independent. It was a major design goal to create a machine which was “complete” in the sense that there were few (if any) operations which necessitated an escape from the architectural framework. In short, a design was desired which allowed the operating system and all of its related tasks to be coded in a higher-level language of the type the computer was designed to support.

In this light, block-structured languages appear to be by far the most attractive alternative. The multi-tasking features necessary in any reasonable contemporary operating system can be incorporated quite elegantly when one considers nested tasks to be generalizations of nested procedures; we shall see that user jobs can all be considered sub-tasks of the operating system in this fashion. The fact that languages modelled on ALGOL-60 [NAU63] permit recursion provides impetus for an object code structure which operates in a re-entrant fashion, and the well-defined boundaries of procedures and blocks provide a natural division of a program into self-contained logical units. This in turn provides the opportunity for one copy of a procedure to be shared by many different tasks, decreasing the average amount of memory required by each. Finally, we will see that the rules of scope in block-structured languages provide a consistent and well defined manner in which to share code and data among several processes. Thus, in the context of an entire system, the choice of block-structured languages appears to be more than justified.

Having decided to design a computer targeted for this class of languages, what should be provided in “hardware” (quoted, since it is irrelevant at this point whether it is hardwired or microcoded) in the way of support? What internal data structures, and what instruction set is best suited to the needs of these languages? And given that the ultimate aim is an entire, cohesive system, can these be generalized or extended to provide such desirable features as protection and relocatability of code and data? The rest of this section is devoted to the answers chosen to these questions, with particular emphasis on the addressing scheme employed, the inclusion of explicit stack structures and the effects this has, the usefulness of information tagging in the design, and the provision of semaphore constructs for task co-ordination and access control in a fashion consistent with the rest of the architecture.

One common feature of block-structured languages is the static definition of directly accessible variables and procedures. The items which can be directly referenced at a given point within a procedure or block can be recursively defined as: (1) those variables, parameters and procedures declared local to the code body in question and (2) the variables, parameters and procedures accessible to the statically enclosing code body (i.e., the surrounding block or procedure in a listing), except those possessing names (identifiers) which are the same as some local item. This is combined with a definition of dynamic environment at any point in the execution of the program: since recursion results in many different allocations of variables, it is the most recent occurrence of the identifier which applies at any time during execution.

Based on the above observations, we note that any item which is accessible at a given point in a program can be identified by a pair of integers called an address couple. The first component is the static nesting depth of the variable or procedure declaration, the second is its ordinal in the declarations for the particular code body in which it occurs; for instance, the first declaration in the outermost block would be designated by the couple (0,1). While it is not true that a unique address couple is assigned to each identifier occurring in a program, it is true that at any point in the listing of a program, the scope definitions above insure that an address couple identifies a unique item.

Assuming the environment is set correctly, so that the first component of the couple can be used to locate the base of the procedure data area active at the indicated static level, this provides a convenient way of locating items in the machine. Since the object code is not directly concerned with setting the environment, this method automatically provides re-entrant code. These observations led us to adopt this as the normal addressing mode in our design.

As mentioned above critical to the success of the addressing scheme is the proper setting of the environment, and this motivates the introduction of the mechanism which is the central unifying concept of the design—the “hardware” recognition, maintenance, and use of stacks (or LIFO lists). In our opinion, though a stack-oriented system is not the most general support for the information structures that arise in the execution of block-structured language programs, it is the best trade-off between generality and practicality, especially for the majority of applications.

The reason the stack is so useful relates to the method in which procedure calls and returns are defined in block-structured languages. A dynamic sequence of called procedures are exited in the reverse order of the invocations, which is modelled exactly by the operation of a stack. Also, the facts that local variables need not be allocated until the procedure is called, and that they become inaccessible upon procedure exit means that the stack can be used to provide storage for the variables at each procedure invocation. Finally, if the machine provides a method of establishing the base of a procedure's data in the stack at each call, then the stack also provides support for the addressing scheme we desire.

The uses of the stack are extended to include the manipulation of data. For example, this provides the base for efficient evaluation of expressions in Polish postfix form; not only is this a convenient type of code for compilers to generate, but considerable code compaction can be accomplished by the elimination of explicitly addressed operands for such operations as ADD—they are implic-
From the collection of the Computer History Museum (www.computerhistory.org)

these systems, most other similarities were discovered by the operating system to queue further requests for the tasks active in the system. In particular, there must be one of suspended tasks can be re-activated. In this way, one of the most important to this paper: that to our knowledge this is the first time such a complex design has been provided on a small scale system.

A more complete description of the uses tagging has in the design will be given in the following section, but as an indication of the value of this construct, we note that besides “normal” data items, unique tags are used to specify: (1) words that mark the beginning of the accessing environment for each procedure activation in the stack, (2) indirect references to other in-stack items for passing parameters by name, (3) descriptors for code and data residing in memory separate from the stack, with automatic bounds checking and (4) link words to aid the operating system in memory allocation.

Any design which claims to support an entire system must provide some means for co-ordination among the tasks active in the system. In particular, there must be some mechanism for controlling access to critical portions of code and data. We have already mentioned that constructs exist to describe code and data outside the stack, and these were extended to provide “semaphore” constructs, based on the P and V operations of Dijkstra. In essence, the first task referencing the descriptor gains exclusive access to the resource described; an interrupt is generated when other tasks attempt to claim it, allowing the operating system to queue further requests for the resource. When the controlling task releases the resource, another interrupt is generated, informing the system that one of suspended tasks can be re-activated. In this way, primitive task co-ordination constructs are introduced in a consistent manner with the rest of the architecture.

We note in passing that many of the constructs and concepts we have employed parallel the features found in the Burroughs B6700/7700 Information Processing Systems. Though the choice of the stack model for block-structured processes was chosen with explicit knowledge that this was also the mode of operation of these systems, most other similarities were discovered by us only after inclusion in our design. For this reason, we feel that these constructs are a function of the general architectural framework employed. We believe our design is unique in many respects, however, not the least of which is the one most important to this paper: that to our

REALIZATION OF THE CONCEPTS IN THE ARCHITECTURE

We now turn to the subject of how these constructs that were considered desirable are physically realized in the “hardware” of the machine, which we call the Buffalo Stack Machine, or BSM. First we give a more detailed description of the tag bit concept and how it is used. We then turn to a brief description of how the stack for a process is maintained in memory and the manner in which the address couples are translated to absolute addresses. Next it is demonstrated how this concept is extended to link the code associated with the program and the operating system into the active stack. Finally, we briefly discuss the generalization to multi-tasking which is possible, and the manner in which interrupts are handled.

The BSM memory is composed of 36-bit words; however, only the low order 32 bits can be considered data in the traditional connotation of that term. The upper 4 bits serve to place the word into one of 16 “hardware” recognizable classes, which is the tagging described previously. Tag type 0, for example, is used for “normal” data words and is the type most frequently manipulated by user processes.

Other tag types are reserved for the code and data descriptors used to point to areas (segments) apart from the memory for the stack, but containing task related information (data or code). For example, the address couple of an array variable would locate a data descriptor for a separate segment allocated to the array. This is trivially extended to multi-dimensional arrays by allowing the segment itself to contain descriptors. This memory allocation scheme has two major benefits: (1) the average amount of contiguous area assigned to a segment (the stack itself is just a special segment) is lowered and (2) the arrays and code segments not currently being referenced can be removed from main memory, which aids the implementation of virtual memory. This second feature is made possible by a presence bit in each descriptor, so that attempts to access a segment through a descriptor whose presence bit is off causes an interrupt indicating which segment must be made present in memory. This shields the task from the fact that not all of its code and data is immediately accessible, and permits it to run in an effective memory larger than that physically available.

How the stack concept is implemented and the manner in which address translation is performed are interconnected to such a degree that it is easiest to describe both together. The following explanation uses the small program in Figure 1 and the corresponding stack structure at a particular point in its execution in Figure 2 as an example. The language used is PL-BSM, the implementation language used on BSM to write supervisor routines, including the entire operating system. Its syntax and semantics are very similar to XPL, and the functioning of the short program, whose main feature is a procedure to recursively compute the factorial of an integer, should be self-explanatory.
First, we note in Figure 2 the set of registers collectively called the Display, which are used to implement the address couple translation. Each task in the system is identified with a stack segment, and when the stack is assigned to a processor, each Display register is set to the base of the data area in the stack for the procedure activation whose environment is accessible at the corresponding level. The location addressed by the couple \((i, 0)\), for any level \(i\), contains an Environment Control Word (ECW), which is a distinct tag type used to insure that the proper accessing environment is set. An ECW has links in it to both its dynamic (calling) and static ancestors' ECWs, so that when a procedure is exited, the ECW for the caller can be found using the dynamic link, and the Display can be reset from the static link chain beginning there.

It should be quite obvious now how the address couple scheme is implemented. The first component indexes into the Display and reads the address in the proper register; the ordinal is then added to this to get the address of the desired quantity, or of a descriptor or pointer to it. The CLEVEL register is used to hold the static nesting depth of the currently executing code, and determines the highest numbered Display register which contains a valid address; in this way, address couples can be checked for errors before any damage is done.

While we are discussing the addressing, the extension of the address couple method to locate code as well as data will be presented. At present, the first two Display levels are used to access the descriptors for the code segments. DR[0] points to an ECW in the “system stack”, and address couples of the form \((0, i)\) locate descriptors for system procedures. In a completely analogous manner, DR[1] is used to access the code descriptors for the particular user program being executed, which also reside in a separate segment. Technically, both of the segments are stacks, though they represent permanently suspended tasks and are never activated. The reason they occupy separate segments is to allow the sharing of the system stack by all tasks in the system, and in the same fashion, to allow several tasks executing the same program (e.g., a compiler) to share the same code stack. In this way the number of copies of procedures is kept down, with a consequent rise in memory utilization.

This also serves to introduce the fact that the conceptual stack for a task may include portions of other physical stacks. In this case, the other stacks are for tasks that are never active on a processor. It should be obvious that this need not be the only way to interlink stacks, and indeed, it is not. A procedure in PL-BSM may be called, in which case further processing by the calling routine halts until it is returned to, or it may be started as a separate task, in which case it is assigned a distinct stack, and both the created and invoking tasks can proceed in parallel. For either method of invoking the procedure's code, the accessing environment must be identical for consistency, and this may well mean that the Display settings for the new task refer to the creating task's stack, allowing the two to share information.

At this time we turn to a discussion of the constructs used to check the validity of code addresses generated. As shown in Figure 2, the code segment for the procedure currently being executed has associated with it 3 processor registers which keep track of the location of the next instruction (byte) (LOCCTR), and check to make sure the code addresses are in bounds (CPBASE, CPMAX). CPBASE is also useful on branch instructions, which are always relative to the start of the segment. Finally, we discuss the registers used to maintain and manipulate the stack structure. The BOS and EOS registers are used to check for stack underflow and overflow respectively; however, EOS, as can be seen in Figure 2, contains an address which leaves some space at the end of the stack when overflow is detected. This is used to provide room for handling the resulting interrupt; the method of interrupt processing in general will be explained later, at which time the necessity for this extra space should become evident. The MKSR register points to the start of the currently active environment and is used in setting the dynamic and static links at procedure call or return.

![Figure 2—Stack configuration at line 6 in program](image-url)
TOS contains the address of the next available word in the stack, and is used to locate the implicit operands for the arithmetic and logical operators, as noted previously. The TOS register also gets incremented and decremented by the code for procedure call and return, and insures the stack space being used by each active procedure is preserved.

We conclude this section with two miscellaneous remarks, the first in respect to the way stacks (and thus tasks) are identified, and the second on how a consistent implementation of interrupt processing is possible with this design. In this system, each stack segment has an associated descriptor pointing to it, and all of these are collected into one segment, known as the stack vector. This segment in turn has a descriptor pointing to it at a predefined location (address couple (0,2)). Thus a stack (and its associated task) can be identified by the index of its descriptor in the stack vector. Looking at Figure 2 once again, note that a portion of the user stack below the first ECW is marked "system space"; it is here that task-related control information, such as priority, is kept. The stack vector concept allows the system to access these items via a doubly subscripted reference to the stack vector descriptor.

The fact that the "hardware" has knowledge about the way stacks are used in procedure calls allows a very consistent implementation of the interrupt code. The primary interrupt procedure has its descriptor at the predefined address couple (0,1); when an interrupt is to be signalled to BSM, the stack of the current task is used by the "hardware" to force a procedure entry identical to a software call. In this way, the interrupt procedure is coded exactly as any other procedure, and uses the same exit mechanisms to return control to the interrupted process. This is all made possible by the fact that the "hardware" is knowledgeable of stack structures in memory, and that there is a well-defined sequence of operations to invoke any procedure. It should now be apparent why EOS is set to cause overflow while there is still some stack space remaining; it gives the interrupt procedure space to work in. Further stack overflow interrupts will not be generated after the original one, since this exception is detected only in the user mode, and invoking the interrupt procedure automatically sets the processor in system mode.

**FIRMWARE IMPLEMENTATION OF THE BSM**

The quoting of the term "hardware" in the previous section when referring to some feature of BSM was done intentionally; since the design of a conceptual machine was being explained, it did not matter whether or not its features were actually hardwired or not. Of course, in our implementation the structures outlined above are realized by a microprogram which forms an emulator for BSM. In this section we turn to the design of this emulator, which we call Kielbasa II, a name which derives from a variety of Polish sausage and is thus a reminder of the Polish instruction set on BSM. It is the direct descendent of the original Kielbasa emulator (described in Reference 9), and though the two emulators were designed for two quite different micro-computers, many of the techniques used in the original have carried over to the current version. Thus we feel a discussion of the emulator is in order to present our observations.

In order to understand the design decisions made in the emulator, a short presentation of the facilities available in the Burroughs B1700, which is the computer used by Project Mu, will be given. Though most of the features to be described are present in all models, the particular model described here is the B1726.

Though the internal data paths in the machine are generally 24 bits wide, the memory of the machine is bit addressable, and can be read and written in units of 1-24 bits at a time in either direction from the address specified. The combinatorial section to do the arithmetic and logical functions can also be scaled to work on operands from 1-24 bits in length, and includes residual control to allow for multiple length operations. Also provided is the capability to shift both single and double length quantities, and the ability to extract an arbitrary field of contiguous bits from one of the 24 bit registers.

Hardware condition bits of the B1700 are held in a set of 4-bit registers (external interrupt pending, clock interval, arithmetic conditions, etc.), and certain 24-bit registers are re-mapped into 4-bit registers for added flexibility. Various micro operations can test and manipulate the bits in these registers, and the non-dedicated ones are especially useful for recording status of the machine being emulated. Finally, there is a scratchpad of registers for general purpose use; these can be thought of as either 32 24-bit registers or 16 double length registers, and are most useful for holding the contents of virtual machine registers.

In addition to this, the B1726 has a separate control memory for fast execution of microinstructions, though execution can also proceed from main memory at a reduced rate. The control memory can be overlayed from main memory, but its operation is effectively read-only (there are no operations for reading the contents of control store words). This added memory permits the emulator to run without conflicting with main memory operations, and micro-instruction execution can overlap reads and writes to main memory.

We now turn to the manner in which Kielbasa II uses these physical constructs to emulate the features of the BSM. First we describe the mapping of the resources in BSM onto those of the B1700, including resources used by the emulator which are hidden from the explicit view of the conceptual BSM. Then we turn to some examples from the emulator of its operation and the microcoding techniques employed. In this way we hope to demonstrate the viability of the use of microcode to implement complex constructs and systems.

Most of the registers used in BSM are kept in the scratchpad, where we use it as 32 24-bit quantities. Sixteen of the registers are used as the Display, the other
half being used to implement the remaining processor registers of the BSM. All of the stack and code page control registers are kept here, as well as some registers to hold state information about BSM used only by the emulator. For example, one register (NEXTINSTR) is used to hold the address of the instruction following the one currently being executed, which is addressed by LOCCTR. It is necessitated by the complexity of location counter updating, as will be shown later on in this section. Another pair of registers is used to hold internal interrupt parameters when such a condition is detected. The design of the system insures that only one internal interrupt can be set in the interval between successive checks in the emulator for interrupts to be generated to BSM, and since these conditions cannot be masked out, there is no danger in "losing" an interrupt by this method.

Unfortunately, the same is not true for external interrupts. Several such interrupts may be pending at any one time, and in addition, the BSM operating system can disable them. A small portion of main memory is set aside to queue these interrupts, and when the external interrupt pending flag is set and the interrupts are enabled, the first entry in the queue is "passed up" as an interrupt to the BSM processor. The external interrupt pending flag is reset if and only if the queue is emptied.

Another portion of main memory is reserved to hold the routines for Kielbasa II which will not fit in control store. This is all that will be said on this subject at this point, but we will see later that a major decision in the emulator is what to keep in mainstore, and how to execute it.

Two of the non-dedicated 4-bit registers are used to hold processor status across all operations in the system. One is used as the CLEVEL register; the ability to increment and decrement these registers and test for overflow and underflow is quite helpful in detecting display overflow and underflow. The other 4-bit register is composed of 4 separate emulator state bits: internal interrupt pending, external interrupt pending, external interrupts enabled and a bit to decide whether or not LOCCTR is to be updated to the value in NEXTINSTR.

Now we turn to a discussion of the structure of the emulator which manipulates these resources. First, two techniques that have been used extensively (and quite successfully) in the design of Kielbasa are discussed: the structuring of all the code for BSM operations into subroutines and the decision to overlay certain routines from the main memory into control store; both of these techniques were used in the earlier version of Kielbasa and found to be quite advantageous. The fact that the B1726 provides hardware support for both of these techniques (an address stack for subroutines and an OVERLAY microinstruction) has made their use even more attractive.

Subroutining allows the use of code from several parts of the emulator, with many of the same benefits which accrue from this practice in programs for any machine. For example, the operations which compare the 2 top-of-stack items for various relations use the SUBTract routine, which is also a separate operation in the machine.

This method of organization is extremely valuable in the emulator, since many of the complex operations are implementable as calls on more primitive ones. The interrupt procedure, as has been noted, can have a call forced by the emulator; this is accomplished very easily in Kielbasa by having the routine which generates interrupts call the various opcode routines for procedure entry in the same sequence as a software invocation. This insures the criterion is met that a hardware or software call on a procedure be indistinguishable in its effects on the stack.

The provision of control store on the B1726 leads to considerable of the best allocation of routines to this fast memory. The indications are that the entire emulator will not fit into the control store, and thus some functions must be kept in main store. The questions that arise in this case are what routines should reside in control store, and of those consigned to main store, which (if any) should be overlaid into control memory when needed. At present, the routines for opcodes which are legal only in the system mode of BSM are kept in main memory, and they are overlaid only when the operation they perform involves considerable looping, it being felt that the overhead in overlaying other routines is greater than the loss in speed inherent in executing from main store. These are preliminary (and rather intuitive) decisions, and we are waiting until the system is in use to evaluate this allocation more thoroughly.

At this point, examples of the way in which the emulator operates are in order; it is hoped that providing these will give insight into convenient ways of structuring emulators, and also demonstrate the way in which complex virtual machines features can be effectively realized through microprogramming. The examples that have been chosen are instruction counter maintenance, address resolution, and the microprogrammed "intelligent" I/O channels.

Unlike conventional machine designs in which the location counter is advanced after each instruction, the operation of BSM causes the location counter maintenance to be more complex. For example, the location counter should not be advanced if a code or data segment is not present; after the segment is brought into memory, the operation must be re-started. On the other hand, detecting an interrupt for arithmetic overflow should cause the location counter to be advanced before the interrupt procedure is called, since if the system decides to resume the task in spite of overflow, it should continue with the next instruction. There are many other instances of this dilemma in the machine, and thus a mechanism was needed to update the location counter correctly. A bit in the emulator status register indicates whether or not the location counter should be advanced; it is set just prior to each instruction fetch for BSM, and if the routines detect conditions which imply the location counter should not advance, they reset the bit. At the end of the instruction cycle, this bit is tested to determine whether or not LOCCTR is to be set to NEXTINSTR.

With a variety of constructs in the machine to cause indirect, (stack relative pointers, indexed descriptors,
etc.) the problem of address resolution becomes quite complex. In order to insure consistent interpretation of indirect chains, the addressing logic of the BSM is embodied in one subroutine, GEA (Get Effective Address). The purpose of this routine is to follow the chains of pointers and descriptors (doing indexing as necessary) until the desired item is found. Since the requirements of various calling routines are quite different (the store operator expects to find a data item address; the enter procedure operator expects a code descriptor), GEA is passed a set of parameters telling which types of items are expected. On return to the caller, the address of the last item in the chain is given, as well as an indication of any exceptional conditions which caused the exit. The calling routine may resolve any conflicts or cause an interrupt to be set for the BSM operating system to handle. In this manner, the complex address resolution is handled in a straightforward and consistent fashion.

The I/O channels BSM communicates with serve to interface the physical I/O of the B1700 with the logical I/O incorporated in BSM. For one thing, this permits the I/O channels to be "intelligent" in the sense that they obey the same tag-implied semantics as the BSM processor itself. An attempt to overwrite a memory link word, for instance, will cause an I/O error termination and interrupt the processor. This gives us an example where microprogramming not only extended the power of the machine, but allowed this complexity to carry over into its communication with the external world.

This concludes the section on the emulator which supports the BSM architecture on top of the B1700 hardware host. It is hoped that this presentation increases the understanding of the power given to systems through a microprogrammable processor. In addition, it should provide some insight into the techniques of microprogramming found useful in creating an integrated, consistent emulator for the desired virtual machine.

ADVANTAGES OF A MICROPROGRAMMED IMPLEMENTATION

This section is intended to detail some of the advantages microprogramming provides. Though the examples are drawn from the BSM implementation on the B1700, we believe these advantages apply in most cases where this technique is used to realize complex or special purpose designs.

All of these features can apply to any computer endowed with microprogramming capabilities, but they are especially important in small scale system and minicomputers, since they are a source of great power and adaptability. The two broad areas to be touched upon are those of the economics of these small systems, and the technical benefits.

A concise statement of the economic situation is that without a microprogrammable computer, BSM would be nothing more than a paper machine. Though it is economical to incorporate such complex constructs in the hardware of a system the size of the Burroughs B6700, it would not be competitive in the small machine market. However, as demonstrated in this paper, an efficient realization can be microcoded on relatively inexpensively, and sophisticated constructs become cost effective on small machines when they are microprogrammed.

Another indication of the economic attractiveness of such a system, especially one equipped with writable control memory, is its adaptability, which has been alluded to above. The same machine which has been so effectively in the support of BSM is also being used to emulate such diverse designs as an APL machine, a string processing machine (for SNOBOL4), as well as a wide variety of conventional designs, and there appears to be no reason why it could not also be tailored for such special purpose uses as a text editing machine or a data base management machine. We emphasize that this is not a feature of the particular machine used, but to a greater or lesser degree a characteristic of the entire genre of microprogrammable minis. This adaptability via microcode is one of the main reasons for the growing power of small systems.

There are also technical benefits which accrue to the use of microcode in systems implementation. One of the most striking benefits which was discovered during the design of BSM is the ability for primitives to migrate from one level of implementation to another. This means that a construct which is frequently used at the virtual machine level can be directly implemented in microcode, making it truly primitive to the emulated machine; conversely, a little-used feature in the virtual machine, or one whose complexity in microcode outweighs the advantages of its inclusion, can be broken up into less complicated operations, placing the burden of implementation on virtual machine software. Examples of both directions of migration will be given next to demonstrate the value of this feature inherent in the use of microcode.

It was mentioned in the section on the constructs included in the design of BSM that semaphore constructs were added to aid in task co-ordination and access control. In the original design of BSM there were no such constructs at all. Since the design was of a single processor system, it was felt that such controls could be implemented by explicitly called gate-keeper procedures (in BSM code) which would perform the same services as semaphores.

When the writing of the operating system was begun, it became evident that this mode of operation was unsatisfactory. The amount of code necessary in the procedures would cause high system overhead, but even worse, any coding flaws which allowed some task to ignore the conventions could have disastrous consequences for the entire system's integrity.

Thus, the decision was made to provide more immediate support in microcode, and it was discovered that the semaphore operations could be easily added in a consistent manner, as described previously. In this the first implementation of the BSM, these provide a sufficient means of control; we are awaiting the results of the analy-
sis of the system's performance (described later in this section) to see if the choice was optimal as well.

An example of reverse migration is demonstrated by the memory management operations. In the original Kielbasa, memory management was incorporated into 3 system opcodes—GETSEG, RETSEG, and COMPACT. Using the first two, the operating system for BSM allocated and deallocated segments in memory, and the third was used for garbage collection in case no contiguous space was large enough for a segment which was to be brought into memory. However, the code for these three operations was considerably more complex than that for any of the other operations in BSM, and in addition, the BSM operating system did not have as much control over the allocation of memory resources as it did over the other resources of the system. These two observations together convinced us that memory management was not balanced with respect to the rest of the system.

To alleviate this situation, the current design has added more but less complicated operations. A better balance seems to have been struck by this approach; the coding in the emulator is proportionately less complicated, and supports memory management by the operating system without forcing any particular discipline on it (apart from that already dictated by the variable size segment organization implied by the architecture). This, then, is a case where an operation was considered to be unbalanced to the side of the emulator, and was broken up into simpler, lower-level primitives.

Another benefit in the use of microcode is the ability to design both the machine and the software that will run on it in parallel, allowing close co-operation. Instead of the usual process whereby the software has to be designed around an existing machine, the considerations of both software and the “hardware” can interact to produce a (hopefully) better design. Both the inclusion of semaphores in BSM and the dissolution of the original memory management opcodes described above were a direct result of this process.

Another example of the value of this co-operation arose when in the design of the compiler for PL-BSM it became apparent that there was a grave flaw in the environment setting mechanisms as originally defined. Since the correct environment for each call is essential to the proper operation of the entire BSM system, the error had been frozen in hardware, it should have been an embarrassing (and expensive) mistake. However, the changes that were necessary in the emulator were easily made with little cost.

These three examples, plus many other minor changes to improve the operation, were possible only because the BSM was being emulated. We believe this is one of the greatest benefits of microprogramming, and can be of even greater importance when trying to optimize a machine for a specialized set of tasks.

The final comment we have to make on the advantages offered by microprogrammed implementation of systems is in relation to the evaluation of the result and modifications based on this. Once a basic design outline has been chosen, the particular operations to include still remain to be specified. All too often the decisions are based upon limited previous experience and intuition as to what is appropriate, and little evaluation is done later of the result. Of course, if the machine is frozen in hardware, the evaluation would probably be of little use except as a guide for future designs.

When emulation is the implementation technique used, however, the evaluation can be very useful. Recent experiments have shown the power of microprogrammed measurement techniques and how these can indicate characteristics of a machine's operations which are not intuitively obvious. Thus, Kielbasa will have microcode embedded in it to gather statistics on the frequency of use of operations, length of indirect address chains and other relevant measures of BSM's performance.

Based on this, we plan to re-evaluate the decisions made in this, the first iteration of the design. It could well be that more constructs will be directly implemented in microcode, or that little used facilities will be even further subdivided. In the two instances cited above, where operations were changed in their level of realization, it is not inconceivable that they might be changed back to their former implementations. We are waiting for experience with the system to either confirm our original decisions or indicate alternate strategies.

CONCLUSIONS

Those of us in Project Mu who have participated in the BSM design and implementation have become increasingly impressed with the power small systems equipped with microprogramming capabilities possess. As technology continues to lower the cost of these systems we feel they will become even more attractive due to the sophistication they are capable of supporting and the ability to adapt to the needs of an application. It is these features of today's mini-computers which make them very attractive and contribute heavily to their growing power and potential.

ACKNOWLEDGMENTS

The author is pleased to acknowledge the help of all the faculty and students at SUNY/Buffalo who have participated in the BSM project and have helped make it a successful venture. Special thanks are due to Michael J. Manthey, who was the original driving force behind the effort and who contributed much of the original design, to Gideon Frieder and Robert Rosin, whose criticism, encouragement and guidance helped greatly in transforming rather abstract concepts into a coherent machine design, and to the National Science Foundation which helped sponsor this project through NSF Grant GJ-993.

REFERENCES
