2

Early Descriptor Architectures

2.1 Introduction

During the late 1950s and early 1960s a host of architectural experiments attacked significant problems in computer system utilization. Most computers of that era were batch systems that ran one program at a time. A program was loaded into a contiguous section of primary memory and run until completion; then another program was loaded and run. This static execution and memory environment made inefficient use of the costly processor, memory, and peripherals. In addition, programs had little flexibility for meeting dynamic programming demands.

Multiprogramming systems showed increased processor utilization as long as several runnable programs could be kept in primary memory. However, multiprogramming required more sophisticated memory management techniques and forced operating systems to deal with dynamic storage allocation and compaction. These tasks were greatly eased by the introduction of paged systems in which all storage units were the same size.

Although paging helped the operating system to manage storage, it did little to help the programmer with the task of programming. A program still had to manage a conventional linear address space. It was difficult to protect instructions or data separately, to catch array bounds violations, to increase the size of arrays and other data structures dynamically, or to create new data structures dynamically.

The concept of segmentation, however, aided both the programmer and the operating system. A segment is a contiguous section of memory that represents some logical entity, such as a procedure or array. The programmer views memory as a collection of segments, each separately addressable. A program addresses each memory element by a segment number and the offset of that element within the specified segment. Because each segment has a size, array bounds violations can be caught by placing the array within a single segment.

An operating system can load each segment into memory separately or relocate segments if needed (for example, to enlarge the size of the segment). However, for an operating system to manipulate segments easily, it must ensure that physical memory addresses are not embedded in the program. The simplest way to isolate the program from its physical memory location is to provide a level of indirection between program-generated addresses and the primary memory addresses of data elements. Just as page tables provide this indirection in the paged virtual memory system, segment descriptors—or segment base/limit registers in some hardware implementations—provide the indirection in a segmented system. A segment descriptor is a data element that contains the primary memory address and size of a segment. An operating system need only modify the relevant descriptors when relocating segments.

This chapter examines several early descriptor-based computer designs: the Burroughs B5000, the Rice University Computer, and the Basic Language Machine. Although these systems preceded the formal definition of capability, each system implemented capability-like structures in its addressing mechanisms. These machines were distinguished from their contemporaries by the generalized way in which they applied the concept of descriptor.

2.2 The Burroughs B5000

Much of the innovation in commercial computer architectures in the early 1960s emanated from the Burroughs Corporation. Introduced in 1961, the Burroughs B5000 system had several features unique for its time [Burroughs 61]. Most important was the use of segmentation for structuring memory and the use of descriptors for addressing segments. Also, the B5000 was geared to execute high-level language programs, particularly ALGOL and COBOL. In fact, assembly language was not available to the user. The system was designed to compile and execute high-level languages efficiently, and relied on a stack-oriented instruction set to aid in expression evaluation and procedure activation. The B5000 supported multiprocessing as well as multiprogramming by allowing connection of two processing units.

On the B5000 a program consists of many data segments and code segments. Each executing program has a local addressing environment consisting of its memory segments, its private stack, and a private *Program Reference Table* (PRT). The Program Reference Table, up to 1024 48-bit words in length, contains *descriptors* that locate the user's code and data segments in memory, and *values* of scalar elements, as shown in Figure 2-1. A *tag* field in each word in the table indicates whether the entry is a descriptor or a scalar data element. All memory references, including procedure calls, are made through Program Reference Table descriptors; thus, the Program Reference Table completely defines the domain of execution for each user program. When a program is running, a hardware register holds the address of its Program Reference Table.

The B5000 supports three different descriptor types: data descriptors, program descriptors, and input/output descriptors. The formats of these descriptors are shown in Figure 2-2. *Data descriptors* contain the size, primary memory address, and drum unit number and address of a data segment. *Program descriptors* are allocated for each procedure and every segment of the main program. Reference to a program descriptor automatically causes a procedure call. *Input/output descriptors* are

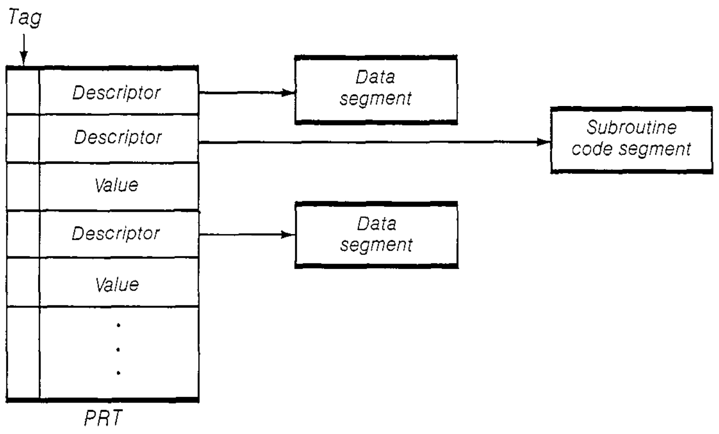


Figure 2-1: B5000 Program Reference Table

Tag	P	Drum number	Segment size	Drum address	Memory address

Data and Program Descriptor

Tag	P	Unit number	Operation size	Operation type	Format/ control	Memory address

I/O Descriptor

Figure 2-2: B5000 Descriptor Formats

command words for the operating system, specifying the size and type of transfer and any special device control or formatting information. The operating system selects a physical unit and allocates primary memory for the operation if needed.

The presence bit (P) in data and program descriptors indicates whether or not the segment is currently in primary memory. If reference is made to a segment not in primary memory, a trap occurs and the operating system automatically loads the segment from drum.

The B5000 is a stack machine and all instructions operate on the stack. The stack is stored in memory; however, the top two stack elements are held in hardware registers called the A and B registers. As items are pushed onto the stack, they move first into the A register, then to the B register, and finally into memory as more items are pushed. As items are popped from the stack, data moves from memory into the B register. All arithmetic operations are performed on operands held in the A and B registers, leaving a single result in the B register.

Each 48-bit B5000 instruction word is divided into four 12-bit instruction *syllables*. There are four types of instruction syllables: operators, literals, operand calls, and descriptor calls. An *operator* syllable operates on the top one or two elements of the stack, leaving a single-word result. A *literal* syllable simply causes a 10-bit literal field in the syllable to be pushed on the stack.

A program executes an *operand call* syllable to load a data item onto the stack. The operand call references an entry in the Program Reference Table, with three possible results depending on the type of entry encountered. First, if the PRT entry is a scalar, the scalar is pushed onto the stack. Second, if the PRT entry contains a program descriptor, a subroutine call takes place. Third, if the entry is a descriptor for a segment with length greater than zero, then array indexing takes place as follows. The contents of the B register, which contains the array index, is validated against the length stored in the descriptor. The index is then added to the segment base address to locate the selected word in memory. The word is read from memory and loaded into the B register, replacing the index.

Descriptors can also be loaded from the PRT onto the stack. This is required, for example, to execute the STORE operator, which saves the contents of the B register in the location addressed by the A register. A *descriptor call* syllable, used to push an address onto the stack, operates in a mode similar to the operand call. If the referenced PRT entry is a scalar, a descriptor is constructed pointing to its location in the PRT. If a PRT entry contains a descriptor, the descriptor is copied to the stack, with possible address modification by an index value in the B register. Reference to a program descriptor causes a subroutine call.

B5000 subroutines execute in *subroutine mode* which provides some special syllable formats. When a subroutine is called, input parameters (as well as linkage information) are saved on the stack by the caller. A hardware register is loaded with the address of the next available stack location past the saved parameters; this is the first location used by the subroutine for its local variables. One of the subroutine mode syllables allows stack addressing relative to the register in the positive direction (to access locals) or the negative direction (to access inputs). A subroutine can also address constants stored in the subroutine code segment using a type of program counter relative addressing. References to the caller's PRT are still permitted within the subroutine.

The B5000's use of the stack, segmentation, descriptor addressing, and high-level languages made it one of the most advanced systems of its time. These features have been expanded and generalized in later Burroughs systems and have had an effect on other manufacturers' products as well. The 16-bit Hewlett-Packard 3000 [HP 72], in particular, is an outgrowth of early Burroughs B5000 ideas. More important, the B5000 Program Reference Tables and their use in addressing and separation of process address spaces directly influenced early capability thinking.

2.3 The Rice University Computer

In 1959, development of a new machine began at Rice University. Called the Rice University Computer [Iliffe 62, Jodeit

The Rice University computer. Jane Jodeit is seated at the control console with Martin Graham looking on (Courtesy Dr. Martin Graham.)

68], this system was designed for the single-program environment and was never intended to support multiprogramming. In fact, the original physical memory of the Rice machine was only 8K 56-bit words. However, this computer—operational until 1971—provided important experimentation with program addressing of memory.

The Rice architecture focused on several deficiencies in conventional linear address space machines. First, conventional hardware did not support entities corresponding to high-level programming objects. Second, for scientific problems, conventional architectures did not support the addressing of single or multidimensional arrays. Third, dynamic growth of data structures was difficult on conventional machines. Programmers had to code the maximum possible size of each array into their programs, so that contiguous storage could be preallocated. Support of ALGOL-like languages, with array size determination at block entry time, was difficult.

To solve these problems, the Rice designers chose a segmented architecture based on the use of *codewords*. Codewords are descriptors for logical program entities; they can be stored in the computer's memory or registers. Each program (as

15	12	1	1	1	8	15
L	*I*	*X*	*p*	*	*K*	*F*

F	*Physical address of the segment.*
K	*Specifies one of eight index registers whose contents can be used to select an array element at location F – I + (K).*
p	*Valid bit, indicates whether physical storage is allocated or not.*
*	*Indirect bit.*
X	*Specifies that the named segment contains codewords.*
I	*Index of the first array element (origin of the array).*
L	*Length of the segment in words.*

Figure 2-3: Rice University Computer Codeword Format

viewed by both the programmer and the machine) consists of a collection of segments, called blocks or arrays in the Rice design. A segment contains instructions, data, or codewords and is addressed indirectly by means of a codeword. Each segment is homogeneous, and data types cannot be mixed within a single segment. A single-bit tag within each codeword is set if the addressed segment contains codewords.

In one sense, a codeword is simply a single-word descriptor used to address a segment, similar to a segment base register or Burroughs B5000 descriptor. In another sense, a codeword *names* the block of storage it addresses. The logical machine address space seen by the program on the Rice system is totally defined by a list of *principal codewords* that it can access. The actual maintenance of codewords is provided by the operating system. The basic structure of Rice codewords (omitting unused bits) is shown in Figure 2-3.

The physically addressable memory of the Rice machine is divided into several fixed regions, as defined below:

- A 64-word table for accumulators, trap addresses, boot code, etc.
- Two 64-word directories of codewords defining array blocks for the operating system and programmer, respectively. These are the principal codewords through which all other storage is reached, including the following structures.
- A 128-word stack.
- A *symbol table* defining each named global object in the system.
- A corresponding *value table* containing values for scalars and codewords for arrays named in the symbol table.

The remainder of memory is allocated dynamically to user programs and data, including those addressed through the value table.

Figure 2-4 shows the structure of a Rice University Computer sample procedure. Procedure instructions can address variables within the procedure segment without reference to codewords (that is, relative to the program counter). However, external arrays, procedures, and variables are addressed through linkage words stored at the end of the procedure segment. When a procedure is compiled, the linkage words are initialized with the names of the global variables to be addressed. At procedure load time, the operating system locates the names in the symbol table and modifies the linkage words to point to the corresponding entries in the value table.

A value table entry can be a value if the object is scalar, or a codeword if it is an array, requiring one or more additional levels of indirection. Indirection is possible through a tree of codewords, and each successive level can specify one of eight index registers. For example, in addressing the two-dimensional array (2DArray) shown in Figure 2-4, each codeword in the secondary codeword segment addresses one row of the array. Indirection terminates when a scalar object is found. Measurements performed on the Rice University Computer showed that 10-15% of total data references were made through codewords.

Arrays can be extended in length by allocating additional storage and modifying the codeword. Multidimensional array addressing is aided by the fact that each codeword can specify an index register. For example, a two-dimensional array can be described by a primary codeword pointing to a table of codewords, one for each row. No address computation is required because the index registers are used to hold the column and row indices. In addition, the rows can be of different lengths.

Although the designers stressed the importance of array addressing and extensibility, perhaps more important is the use of codewords as object names. Using the Rice scheme, a procedure need only specify a codeword parameter to pass an object to another procedure. The codeword completely defines access to the object, including its address and length.

The Rice University Computer had several limitations, but they were often due to implementation decisions. For example, codewords contained the length of the block they defined, but the length was not used by hardware to validate an array index. Instead, a trap facility was provided to allow software to check

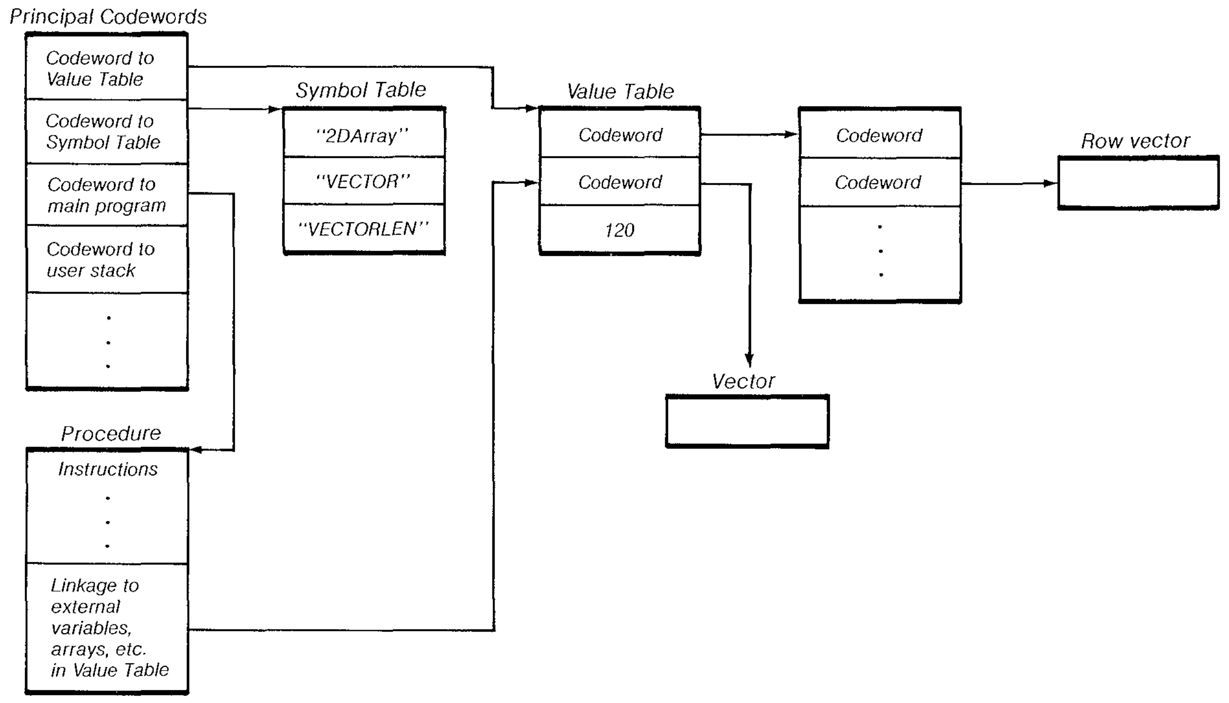


Figure 2-4: Rice University Computer Memory Organization

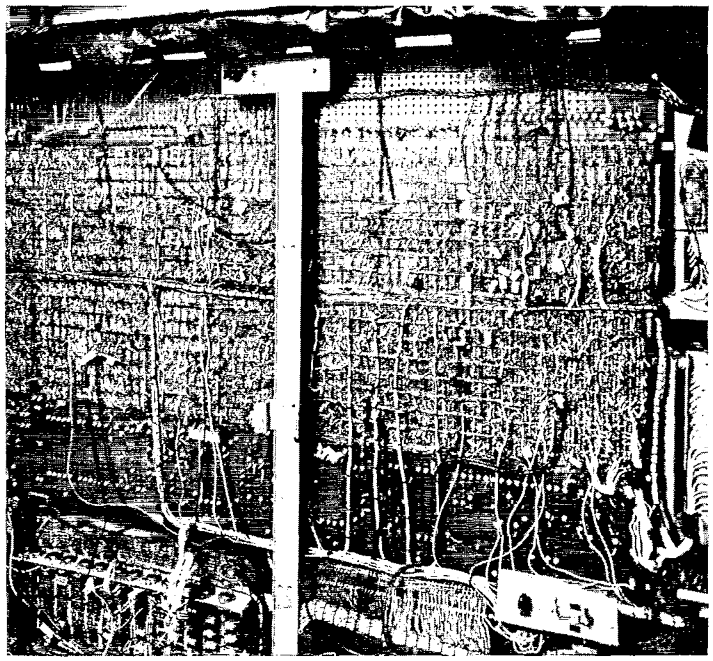
array bounds. There was also no hardware-enforced memory protection in the system; however, this was due to the simplified goals of the machine. One of the more troublesome shortcomings was that procedure return address links were stored as physical addresses, so procedures could not be relocated easily.

Iliffe and Jodeit suggest that extensions for multiprogramming would be straightforward and require that each user have a separate primary codeword list. Virtual arrays would be possible also, but the only secondary storage on the Rice computer was a magnetic tape system. The Rice implementation of codewords closely resembles the capability concept in the sense that possession of the codeword (or knowledge of its address) is required to access an object. The designers also suggest that Rice codewords could be extended to include usage statistics and that device controllers could be developed to understand codeword formats. These additions were never made, but several architectural advances were made in a follow-on design, the Basic Language Machine.

2.4 The Basic Language Machine

The Basic Language Machine (BLM) [Iliffe 68, Iliffe 69] attempted to extend the capabilities of the Rice University Computer and correct some of its shortcomings. Like the Rice University machine, the BLM incorporated a codeword mechanism, but it added data type tagging and address manipulation as well. An additional goal of the BLM project was to build a machine defined in terms of higher level functions, hiding from the programmer the bit-level details of the machine. The Basic Language (not the familiar BASIC programming language used today) defined this high-level architectural interface in terms of an assembly-level command structure. Design of the BLM was started in 1964, and an experimental version was built by the research division of International Computers Limited (ICL) in the United Kingdom.

The Basic Language Machine supports 8-bit byte, 32-bit word, and 64-bit double-word information units. There are 16 general-purpose registers, each 64 bits long. One of the registers is the program counter (called the *control number*), one points to a data structure containing the context local to the current process (called the *Process Base*), and two are reserved for special escape actions. Memory on the BLM is segmented, the largest segment containing 64K elements of the largest information unit. The BLM supports a 24-bit physical address space.



The BLM computer. (Courtesy International Computers Ltd.)

BLM segments are addressed through codewords, as on the Rice computer. However, BLM codewords contain a *type* field indicating the type of information elements stored in the segment they address. The defined data types are:

- 32-bit binary word,
- 8-bit byte,
- 64-bit long numeric,
- 32-bit short numeric,
- mixed type,
- instruction,
- absolute codeword, and
- relative codeword.

The type field also indicates what access is permitted to the segment: data segments can be read-only or read/write; codeword and instruction segments are read-only.

Most of the type encodings specify segments that are homogeneous, that is, segments with only one data type. If the codeword type field specifies a mixed-type segment, the segment can contain elements of any type. However, in mixed-type segments, each element must contain its own tag. A tag is a field contained within the information unit indicating its interpretation. All elements in a mixed-type segment are 64 bits long and contain a 3-bit tag. The four tags defined are:

- 32-bit binary word,
- escape (an attempt to use such an element as an operand causes a trap to software),
- 45-bit address (stored in 64 bits), and
- 61-bit floating numeric element.

The BLM automatically performs conversion and tagging of data elements on fetch or store operations. In homogeneous sets, tags do not need to be stored with each data item, but are constructed from the type stored in the codeword used to load the item into a register. Therefore, homogeneous information can be tightly packed without tagging overhead. The format of 32-bit and 61-bit numeric elements when stored in registers, for example, is shown in Figure 2-5. The tag values of zero and three in the figure indicate 32- and 61-bit numerics, respectively. If an 8-bit byte is fetched, it is automatically sign-extended to 32 bits, and the tag is set to zero.

The BLM is a multiprogrammed computer, and a Process Base defines the execution environment for each process. It is possible for several processes to share the same base and, hence, share access to the same objects. The process address space is composed of a collection of segments, each of which is described by a codeword. The segments may be arranged in a tree structure, but all nodes are reachable only through codewords originating in the Process Base. That is, the terminal nodes of the tree structure contain data or instructions, while the intermediate or branching nodes are codeword sets. Codewords are thus used both to separate user address spaces and to separate logical entities within a program.

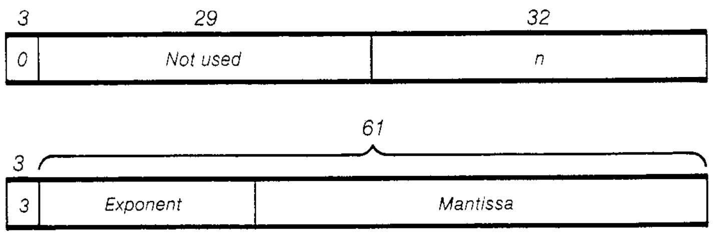


Figure 2-5: Example of BLM Numeric Formats

Relative codewords are provided so that, in situations where it is natural to do so, codewords can be stored in the same segment with the data they describe. To simplify packing, relative codewords are only 32 bits long and can only reference objects within 4096 bytes of their location. Relative codewords allow efficient storage of related data structures. A program can maintain several data structures in a single segment by placing relative codewords for the data structures in the first few segment locations.

Figure 2-6 shows a sample structure of a BLM process. In this case, the Process Base contains codewords for instruction segments, data segments, and codeword segments. The terminal nodes are all data segments. One of the terminal nodes is a mixed segment with relative codewords pointing to internal data structures.

Codewords define the address space and are read-only; they cannot be manipulated by users. BLM *addresses*, however, are quantities derived from codewords that can be user-manipulated. Both addresses and codewords contain the same information, as shown in Figure 2-7: the address and length of the defined set, its type, and a tag indicating an addressing element. Once an address is derived from a codeword, through an operation called *codeword evaluation*, it can be modified through special instructions. MOD and LIM instructions address a subset of the original segment by modifying the loca-

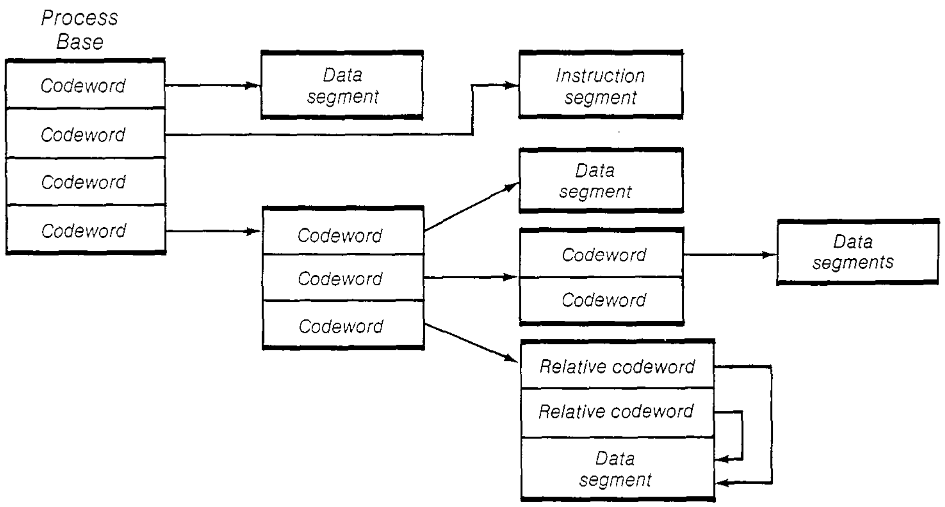


Figure 2-6: Basic Language Machine Addressing

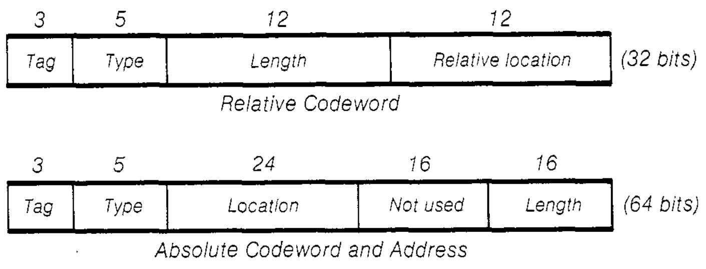


Figure 2-7: BLM Address and Codeword Formats

tion and length fields—to remove a specified number of elements from the beginning or end of the segment, respectively. Looping instructions are available to step addresses through consecutive elements of a segment (performing an implicit MOD by one each time) and to test when the last element has been examined. Iliffe notes that it would be possible also to use the 16 free bits in an absolute codeword to implement linked data structures.

BLM addresses allow users to save intermediate address computations through a tree of codewords. (In contrast, on the Rice University Computer, a full address computation is required on each access to an indirectly referenced object.) On the BLM, the programmer can compute the object address once and save it. The address for a single element in a set can also be computed and saved. Of course, relocation is difficult because addresses as well as codewords must be examined when an object is relocated; that is, BLM addresses are not virtual but contain the primary memory location of a data element.

The Basic Language Machine made several important advances over the Rice University Computer. First, it extended the design to encompass multiprogramming, using a separate Process Base for each process. Second, it provided a more general addressing structure to give users flexibility in performing address arithmetic and saving results. Third, it used a relatively efficient typing mechanism to reduce the number of operators in the instruction set. However, despite the advantages of its structure, the experimental BLM was dismantled in 1970 and no product evolved from the research effort.

2.5 Discussion

The machines described in this section share two major traits: segmentation and the use of descriptors (called codewords in the Rice and BLM machines) for segment addressing. Segmentation of programs was used:

- to separate programs into logical entities (procedures and arrays, for example),
- to separate user processes from each other,
- to represent and address complex data structures in hardware, and
- to allow relocation and dynamic growth of data structures.

In general, an address is specified by two parts: a segment descriptor and an offset. However, different approaches for the specifics of addressing and address manipulation were used for each machine. For example, array addressing on the Burroughs B5000 required the index to be pushed onto the stack before the array reference was made. Multidimensional array address calculation required a series of index pushes and evaluations. The Rice University Computer used index registers, and multilevel indexed addressing was performed automatically with an index register specified for each level in the addressing tree. With the BLM, this idea was abandoned and replaced by address modification instructions that allow a controlled form of user-modifiable codewords.

All three machines provide a single base segment that defines a program's execution environment: the B5000 Program Reference Table, the Rice University Computer primary codeword list, and the BLM Process Base. The address of the base segment is usually held in a hardware register. From the base segment, the addressing mechanism provides for the representation of programs and data structures as tree structures. The trees are slightly different in each case due to the differences in addressing. The root of the tree is the base segment hardware register, and the first level nodes are in the Process Base. Starting at the Process Base, the branchpoints of the tree are codewords or descriptors and the leaves are data elements (in the case of the Rice University Computer) or data segments (in the case of B5000 or BLM). The BLM allows a program to traverse several levels and save the intermediate address of a subtree, but the Rice machine requires a complete multilevel scan for each access. The tree structure allows the user to represent complex data structures directly in hardware and to share substructures. Different processes can share subtrees by sharing subtree descriptor segments.

One of the major reasons for segmentation in these systems was to simplify relocation of programs and data. Relocation is facilitated by forcing all references to flow through descriptors. To relocate a segment, the operating system needed only to modify its descriptors. The additional level of indirection provided by descriptors also made segments easily "virtualizable," that is, all segments did not have to occupy primary memory while a program was running. Of course, the complexity of relocation is greatly influenced by the generality with which descriptors can be used. For example, if descriptors are stored in a single descriptor table, relocation involves only a scan of that table. However, if descriptors are stored in segments and each descriptor contains a segment base address, then many segments may need to be searched. Such a memory search can be simplified if segments are typed, as on the BLM, because only mixed or codeword segments would need to be examined.

Care must be taken in any scheme in which multiple copies of the physical segment information can exist for a single segment. This problem could be reduced if the descriptors themselves referred indirectly to a second-level segment descriptor. However, on the machines examined in this chapter, a descriptor contains all of the physical information describing a segment. Thus, copying a descriptor duplicates the physical address.

Descriptors on the B5000 can be copied onto the stack, requiring a possible stack search in order to relocate a segment. However, because it is exclusively a high-level language machine, the use of descriptors can be restricted by the B5000's compilers. The Rice University Computer allows descriptors to exist in any segment of codeword or mixed type, so these segments would need to be scanned. The BLM, on the other hand, allows pure codeword segments and relative codewords within other word-oriented segments. Both the Rice and BLM machines require a tree search to find descriptors for segments to be relocated.

Another problem in multiprogramming systems is controlling access to shared segments. A user (or I/O device) wishing to perform a multistep transaction on a shared segment must gain exclusive access to that segment. This can be achieved by disabling interrupts or context switching (usually via executive procedures), through the use of explicit software locks, or through the use of a "lockout" or software trap bit in the descriptor. If lockout bits are used, then the executive must find all copies of descriptors for the target segment.

Another issue in descriptor design is the cost of indirection.

All of the examined machines allow tree-structured data. Although the Rice machine has automatic multilevel addressing, the Burroughs and the BLM require several manual steps. However, the Burroughs and the BLM allow for partial address computations to be saved.

One of the perpetual debates in computer architecture is the tradeoff between the use of tag bits in data elements and the larger operation code set needed in non-tagged architectures. The BLM scheme seems to answer the concern for tagging overhead by only storing tags in the codeword or address for homogeneous segments. However, for mixed or heterogeneous structures, each element must still carry a tag. In addition, the elements in a mixed set must all be of the same size as the largest element in the set; that is, all elements must have the same alignment to protect against addressing the middle of some element and interpreting data bits as tags. This is not particularly efficient because any segment containing a codeword pointer must use 64 bits for each element. Still, there are benefits to tagging besides the possible savings of operation code bits, including automatic conversion and checking by the hardware. A certain amount of error detection may also be gained by self-tagging of information units.

A likely problem with these machines was that of garbage collection. If a program can write a descriptor to a descriptor segment, the descriptor previously occupying that memory word could be overwritten. If the overwritten descriptor were the only one referencing some segment, that segment would then be unreachable. In general, this problem was prevented by making descriptor segments read-only. The B5000 PRT was not read-only; however, this system relied heavily on the compilers for proper system operation. User programs did not have direct control of the PRT or descriptors. Garbage segments were considered a problem on the BLM, and a garbage collection process was written to search for unreachable segments.

One of the more important gains from the use of descriptors is the protection of procedures. If procedures can be invoked only by referencing a descriptor, then two benefits are realized. First, a procedure can only be invoked at its entry point contained in the descriptor; it cannot be entered at a random point. Second, procedure code is protected from accidental or deliberate modification.

Despite their differences, all of these machines have a common link to capability architectures: they all use descriptors to name programming objects. The objects are generally simple, for example, a segment containing an array, a procedure, or a list. I/O operations are also described by descriptors on the B5000.

It is important to note that all of these machines support large word lengths. A single word is large enough to contain all of the segment base and limit information as well as various other bits. In general, although bytes may be supported as data types, byte addressing is not provided; that is, memory is word-addressable. The descriptor is a single word that contains all of the physical information needed to locate the object in primary or secondary memory. In retrospect, this fact is important because duplicating the descriptor duplicates all of the segment mapping information. Descriptors are therefore different from virtual addresses or modern capabilities where a second level of addressing is employed.

Although the Rice family of machines was not directly continued, the B5000 led to many stack and descriptor machines in the Burroughs family, and other manufacturers were also influenced by its design. Whether or not they were long-lived, these machines demonstrated the feasibility of using descriptors and segmentation to greatly increase programming flexibility for the user, the compilers, and the operating system.

2.6 For Further Reading

The Burroughs B5000 is described in *The Descriptor* [Burroughs 61], a remarkably modern document for the time it was written. One section of the manual is devoted to the advantages of high-level language systems (ALGOL in this case), such as reduced programming time, simplified debugging, and program maintenance. Such goals are remarkably similar to the objectives of today's object-based systems.

Two papers that discuss storage allocation in the Rice University Computer are [Iliffe 62] and [Jodeit 68]. A book is available on the Basic Language Machine [Iliffe 68]; however, it is unfortunate that more was not published on the machine's design and use. Perhaps this indicates the fate of industry's research projects that never become products. However, an excellent discussion of the BLM within the context of modern capability systems appears in [Iliffe 82].

Following the BLM, design of a third member of the Rice computer family, called the Rice Research Computer, was started at Rice University [Feustel 72]. The Rice Research Computer was to be a high-performance tagged architecture, but technological problems caused the termination of the project in 1974. A discussion of the general advantages of tagged architectures can be found in [Feustel 73].