ON THE ADVANTAGES OF TAGGED ARCHITECTURE

BY

E. A. FEUSTEL

Reprinted from IEEE TRANSACTIONS
ON *COMPUTERS*
Volume C-22, Number 7, July, 1973
pp. 644-656

On The Advantages of Tagged Architecture

EDWARD A. FEUSTEL

Abstract—**This paper proposes that all data elements in a computer memory be made to be self-identifying by means of a tag. The paper shows that the advantages of the change from the traditional von Neumann machine to tagged architecture are seen in all software areas including programming systems, operating systems, debugging systems, and systems of software instrumentation. It discusses the advantages that accrue to the hardware designer in the implementation and gives examples for large- and small-scale systems. The economic costs of such an implementation for a minicomputer system are examined. The paper concludes that such a machine architecture may well be a suitable replacement for the traditional von Neumann architecture.**

Index Terms—**Computer architecture, data representation, firmware, hardware–software tradeoffs.**

I. Introduction

A CENTRAL issue in the design of computer architecture is whether one should design large machines in the classic von Neumann form.[1] The von Neumann form states that data and program are indistinguishable. This form assumes fixed-size binary words or characters that allow programs to be treated as data. These computational units are manipulatable by a large, general-purpose set of operations. Meaning is not inherently represented in the contents of these units; rather it is assigned to the contents of these units by the program manipulating them.

In this paper we assert that for large, general-purpose computers, this form is not the optimal design form and that a better one requires self-identifying representations at all levels of storage. This assumption has great consequences in the architecture of computer systems and impacts the software, firmware, and hardware, changing the entire nature of the architect's job.

In this paper we will hypothesize a machine based on self-identifying data. Prior to 1969 and except for the Burroughs B5500 and BLM, machine design has included tagging as a relatively unimportant peripheral concept, one not at the heart of the design. In this work we attempt to make its use in the description of data, key to the design of hardware. In his paper, Reynolds [5] adopts the principles of completeness, consistency, and minimality. He attempts to use as few mechanisms as possible to implement his language and then to use them at every turn. He can represent all quantities in terms of basic atomic quantities and functions. He uses a uniform scheme of syntax and semantics to describe all actions of the language. In the same way we want to apply tagging to all aspects of software and hardware.

TABLE I
STANDARD HARDWARE-RECOGNIZED TYPE CODES

int:	integer
real:	real number
long int:	double-precision integer
long real:	double-precision real
complex:	single-precision complex
long complex:	double-precision complex
undf:	undefined
mixed:	mixed types (indirect only)
char:	character (indirect only)
Bool:	Boolean (indirect only)
vec:	vector of
ref:	reference to
label:	label in ith environment
matrix:	matrix of
svec:	sparse vector of
sll:	single-linked list of
dll:	double-linked list of
stack:	stack of
q:	queue of
ms:	machine state of
msg:	message from–to
ipt:	interrupt of
ev:	event
ps:	parameter set for
proc:	procedure-environment designator
name:	name of variable
i.d.:	i.d. of process or user
instr.:	instructions
file:	file
formal:	formal parameter
sema:	semaphore
garbage:	garbage

We do not know how many different types of data we need now or will need in the future. We can suggest a small number (32) sufficient for most needs (see Table I). Undoubtedly there are others. We can, however, provide a mechanism by which the programmer can define his own data types that will be "recognized" and "utilized" by the hardware. This mechanism is both simple and practical. Essentially it is a data driven interrupt. It can be made completely compatible with the hardware recognized types shown in Section VI.

A basic motivation for our work is to make computer systems more cost effective. In the past the cost of computer memory and logic was very high and software was relatively unsophisticated. It was cheaper to complicate the software than to increase the cost of the system through increasing the storage requirement or the processor complexity. In the past ten years the situation has reversed. Hardware costs have dropped radically while software complexity and cost has grown [63], [66]. We must now reconsider the balance of hardware and software

Manuscript received February 17, 1972; revised July 31, 1972. This work was supported by the Atomic Energy Commission under Contract AT-(40-1)-4061.

The author is with the Department of Electrical Engineering, Rice University, Houston, Tex. 77001.

[1] The SYMBOL machine takes a different view—that of specialized processors instead of specialized representations.

and to provide more specialized function in hardware than we have previously, in order to drastically simplify the programming process [1]-[4].

We will consider the hardware-software tradeoffs made previously. We will then review the software requirements for today's programming languages, operating systems, instrumentation systems, and debugging systems and attempt to show the simple, general solution that the introduction of tagged architectures produces. We will continue with a discussion of the requirements of large computing systems, indicating how tagged architecture can be used to simplify their design. Finally, we will briefly discuss a simple implementation and discuss the expected additional costs incurred in this implementation.

II. Previous Work

The most obvious kind of self-identification is a validity check. The parity bit has long been employed as a form of error-detecting code that helps the hardware in maintaining reproducible and correct results. Similar validity checks are made today in some MOS memories with complex error detection and correction codes. The CDC-6600 and its relatives feature data validity checking and have representations for numbers such as undefined, infinite, and infinitely small. If one attempts to operate on invalid data, a hardware trap to a particular location occurs. A control routine can be employed to check the state of the machine, correct the error, and return control to the interrupted routine.

The designers of the Rice Computer (R-1),[2] circa 1959 [6], felt that such bits which could be used by the programmer to cause the computer to trap—tag bits—might be useful to the software designer. They incorporated two tag bits on each memory word and also instructions to manipulate and test the bits. These instructions and bits were used to identify and monitor quantities of special interest when debugging software [7]. The designers of the EAI 8400 used its two tag bits for executive functions of the operating system in much the same way. Ingenious programmers found that the parity bit could be set on the IBM 7040 and used it to identify uninitialized at run time [49].

The designers of the Burroughs B5000, circa 1960 [8], [9] computer utilized a single flag bit to identify a memory quantity requiring special hardware interpretation. This quantity was viewed as describing some object in storage whether it be an address, a vector of words, or a procedure. This identification was essential to ensure a zero-address function set consisting of 12-bit operators and helped to assure protection as well as permitting a virtual storage mechanism [10].

Of course designers of character-oriented machines had long before utilized marks to delimit variable length character strings stored in their memories. One of the most interesting machines of this category was the IBM ADAM machine [11].

In the 1960's several machines were built that used bits to identify the arithmetic types of the data with which they were associated. Among them were Telefunken TR-4 and TR-440 [12], [13], the Burroughs B6500/B7500 [14]-[16], Iliffe's BLM [17], [18], and Rice Research Computer R-2 [19]. The first three machines were developed independently of one another. The R-2 was based on the BLM. All the machines featured hardware bits denoting the types of arithmetic data. In addition the latter three machines identified address-bounds pairs (denoted vec) and various other quantities of interest to the operating systems. The R-2 added additional quantities to those that are identified by the other machines.

Another kind of self-identification required by operating systems is access information. The IBM System 360, the RCA Spectra 70, and the Xerox Sigma families of computers made use of half-byte hardware keys that identify who may access physical blocks of storage. Other machines have READ-WRITE-EXECUTE lockout exercised over blocks of logical or physical storage. The R-2 features either word or segment write lockout. The GE645 features a segment lockout of writes by hardware interpretation of bits in the segment table. In all these machines if one attempts to access a memory location in an unauthorized way, control is transferred to system routines.

In the previous examples, we have seen how the designer delegated to the hardware various tasks with which the software could have dealt. A software solution to any of the problems mentioned above would have been very inefficient. The hardware was made more complex in order to handle the software functions, but the cost effectiveness of the system was enhanced. In effect certain oft-performed routines were factored from the general software and put in hardware.

III. Simple Computational Units in Programming Languages

In this section we will comment on the appropriateness of undifferentiated data cells for use in higher level, algorithmic programming languages. We will indicate how tagged architecture can simplify the chores of programming. Finally, we will suggest extensions of the set of simple computational units to assist the implementor in efficient implementation and to help the programmer determine or assure the validity of his program.

The majority of programmers who use higher level languages to program, do not require or use a von Neumann machine. There is no way to cause a Fortran or Algol, PL/1 or Cobol program to modify itself in a constructive manner. It is generally considered bad form to modify a program even in assembly language. The programmer is isolated from the machine by a level of abstraction, the programming language. Dijkstra contends that this is as it should be [20]. This isolation is achieved at great expense however, for the language of the programmer which is very rich in data types is mapped onto a typeless machine programmatically. In fact the typeless machine has a type: word or byte. A high percentage of the compile-time or run-time system is specifically for the purpose of providing access algorithms to make this storage type appear to be floating-point numbers, arrays of numbers, strings of characters, or Boolean elements. Generally this mapping reduces run-time efficiency substantially. For example on the CDC-7600 it is possible to obtain codes executing within the

[2] The first tagged data machine was the MERLIN machine. It was produced around 1957 at the Brookhaven National Laboratories. The R-1 was a successor and generalization of the MERLIN and MANIAC II. It was decommissioned in 1971.

stack at rates approaching 34 million instructions per second. A similar code in "optimized" Fortran runs at 30 to 40 percent of this rate.

Knuth [22] suggests that we choose an internal representation for data in any algorithmic process that is appropriate for the operation that we wish to perform and the information that we must represent. These representations then become primitives in which we do all other operations for which our initial structures make sense. We suggest that this be done as a matter of course by the machine designer for frequently used types and that he provide a uniform firmware mechanism for other types. In fact, it has been done on the R-2 for a class of well defined and often used data types as suggested above. These types include numbers in floating-point notation (real), numbers in fixed-point notation (int), a pair of numbers in floating-point notation (complex), double-precision real (long real), undefined (undf), base-pounds pairs (vec), and a reference-like quantity (ref). The R-2 system has shown that even this small set of types is extremely useful in implementing programming languages. We will show a few examples before generalizing the concept of type.

The generation of code for a compiler is simplified because the operands carry their own semantics to the processor. Only one instruction from each class of arithmetic or conditional operations need be generated. The complexity that formerly went into class differentiation can be better invested in generality such as reverse operations, automatic restoring of a result, and testing operations. The machine can automatically perform run-time conversions from type to type when it is appropriate to do so and produce exceptions otherwise. Type conversion need be requested only in expressions. While run-time checking is not advocated as a solution to the detection of semantic errors in a program, the typing structure allows the hardware to do it automatically and the programmer is provided a foolproof check of the actual semantics at run time.

This concept of a simplified operations code set is also carried over in processing vectors, particularly in the BLM and R-2. A set of instructions called modify (MOD), limit (LIM), jump-last (JL), and jump-not-last (JNL) are provided.

A vector is completely described by a base-bounds pair denoted vec. In terms of processing algorithms, a vector may be considered an ordered set of elements that may be recursively specified in terms of its first or last element. On the R-2, instructions are provided that can access the first element of a vec (others access numerically indexed elements). Another set of instructions construct a new vec from an old one by deleting the first (N) or last (N) elements. Another set of instructions test a vec to see whether it is empty. If it is not empty, the first element of the old vec is discarded and a new vec results. A branch may be specified on an empty vector or on a non-empty one.

The use of these instructions facilitates instructions of the form

$$\underline{\text{forall}}\ X\ \underline{\text{do}}\ \{\ \}$$

where X is a vec or a list of vecs. This construct is often used in programming. For example see MADCAP II or MADCAP VI [23], [24]. This syntax means to perform the bracketed operation on every element of X or in parallel on multiple vectors, i.e.,

$$\underline{\text{forall}}\ X, Y, Z\ \underline{\text{do}}\ \{X = Y + Z\}.$$

In the R-2 machine error will occur if X, Y, and Z do not have the same number of elements.

In this section we have suggested that the word is not the appropriate choice for a simple computational unit. We have illustrated the power of this suggestion using a few of the types found on two machines, the R-2 and the Burroughs B6500. While these additional entities have not proven sufficiently general to relieve the programmer of many stereotyped tasks, they have convinced us that the principle is worth exploring further.

IV. Extensions for Languages

A large number of reasonable extensions suggest themselves. This discussion will touch on several. Barton [25] suggested that one could implement vector operators on a tagged architecture (and the B6700 permits this when working with characters). The CDC STAR-100 [26] has a set of operators that work on vectors of bits, bytes, or floating-point numbers at very high speed because of a pipeline organization especially suited to vectors. This set is almost replicated for work on scalars. Clearly it should be possible to declare

$$\underline{\text{vec}}\ a, b, c;\ \{\text{initialization of } b, c\};\ a = b + c$$

and have the operation performed as a vector operation because the machine knows the objects with which it is dealing. In this manner all the mechanics of the APL programming language could be transferred naturally to the hardware's task rather than the software's. Not only would the computation be speeded, but the hardware because of its awareness of the data it was handling, could access data more efficiently, thus improving the utilization of the memory and arithmetic unit.

In addition to this extension one would like to complete the set of quantities deemed either necessary or useful for the implementation of a complex language such as Algol 68 [27] or PL/1. To the preceding complement of types we would add character (char), Boolean (Bool), reference or pointer (ref), and label (label).

It is worthwhile considering these additional types in greater detail. In general, we are interested in individual characters only when we are operating on them in a stream or in isolation. Normally we consider strings, a basic element, made up of characters or bits; an isolated character or bit is treated as just another int. A method of treating these quantities without undue waste of memory is to allow the indirect reference of a vec to indicate char or Bool but not to allow a group of single characters or bits to each occupy a word with a tag on it. The upper and lower bounds are then in units of measurement, i.e., bits or characters, rather than in words. By the use of this indirect tag the hardware can easily make the adjustment of base address and do a bounds check on the string.

In one sense the ref and the label are instances of vec. They are pointers to either data or code. There are several practical

reasons why they should be differentiated from one another. Although ref points to data, it can only point to zero or one element and to an element that occupies memory or does not. It thus needs only an indirect type and does not require upper or lower bounds. The label points to code however and so we may delete the field for indirect type. Johnson [18] has suggested that a label must consist not only of a pointer into code but also an environment pointer that establishes what the environment is to be after transfer of control as is required in Algol 60 or PL/1. Obviously, jumps within an environment need only utilize jumps relative to the program counter (which in our case would also contain the environment pointer). Hence the operation codes need not be bulky. Thus a label would consist of an environment pointer and an instruction address.

In order to program the system in higher level languages we must add to the primitive data types events (ev), machine state (ms), and interrupt (ipt). In addition to these primitive elements we also include types to simplify higher level languages that represent matrix (matrix), a three-dimensional array (vol), sparse vectors (svec), single-linked list (sll), double-linked list (dll), stack (stack), queue (q), message (msg), and parameter sets (ps).

Let us deal first with matrix, vol, and svec and with the changes necessary to use vecs conveniently in Algol 68. This language requires that each array have upper and lower bounds and a base (as implemented on the MU-4 [29]) and bits that indicate whether the bounds are flexible. In addition, so that slices of arrays or subarrays may be selected, we are required to have a row offset (stride) for each component (or dimension) as well as an initial offset. Thus we are presented with the question of what to do about special representation for arrays.

Knuth recently studied a large volume of source language in Fortran [30]. He studied the way in which identifiers were used. About 59 percent of the time the identifiers were indexes or scalars.[3] One-dimensional array identifiers (i.d.'s) appeared 30 percent of the time; two-dimensional arrays 9 percent of the time and 4 or more 0.2 percent of the time. Thus it would appear uneconomical to attempt to achieve a special representation for them. One-dimensional arrays or vectors are used very frequently however and it is clear that we should provide a special descriptor for them. Since MOD and LIM will serve to slice the one-dimensional arrays we will provide only the following components: a base, an upper bound specified in elements, a lower bound in elements, flex bits for both upper and lower bound, and an indirect type field. The machine would be able to determine if the elements of the array were undefined before accessing them by consulting the indirect-type field.

Since the majority of the arithmetic used in many numerical problems involves matrix arithmetic, it seems worthwhile to include a special structure known as a matrix. While it would be possible to represent this structure as a tree, this representation is notoriously inefficient when transposing or accessing by column instead of by row. The special components for a matrix should include those necessary for Algol 68 as mentioned above. In addition it should include an indirect-type field.

Although the volume is used less frequently than the matrix, it is a desirable representation for many problems in three-dimensional space. The comments with respect to a slice are especially appropriate with respect to a volume. We will assign a vol one more quintuplet of bounds, stride, and flex bits than the matrix. Thus our representation for a descriptor block that describes a matrix and is recognized by the hardware assumes a standard storage layout as described in Knuth [22] except that flex bits are added (to indicate which bounds may be changed by a new assignment) and stride fields. The purpose of a stride field is to allow the manufacture of a descriptor to a subpart of an array such as a complete or partial row, a submatrix or a subvolume. Finally an indirect field permits the machine to deal efficiently with the quantities referenced such as characters, complex numbers, or other matrix or vol components.

Many of the same considerations of cost versus use apply to the problem of the sparse vector. The designers of the CDC STAR-100 have provided a representation for a sparse vector. If our machine is able to process vectors automatically, we feel that such a representation is not only natural but adds power and elegance to the machine operation code. Basically an svec is a structure consisting of an ordered pair, a vector of 0's and 1's, and a vector of nonzero elements. This sparse vector must be processed serially for maximum effect, for example in a pipeline. The 0's stand as place holders for absent elements and the 1's stand for elements that are present. Operators such as compress and expand allow the conversion of vecs into svecs and vice versa. Sparse vectors allow substantial saving of space. Potentially they save processing time because in relatively sparse vectors, no access or computation is required for zero elements. It should be noted that we have not introduced a sparse matrix or sparse volume reasoning that the software implementation of these entities could be done more economically. In these cases the order of processing of the elements may be crucial for an efficient algorithm.

Next let us consider sll,[4] dll, stack, or q. These specialized structures are included because of their general usefulness in system programming and in nonnumeric programming such as graphics or simulation. A single-linked list element consists of a structure with those elements of a single information element (possibly a reference) and a reference without an indirect tag field. Burorughs found this structure so useful on the B5000

[3] It is of interest to note that APL and PL/1 have a smaller occurrence of scalars since the accessing algorithms are specified directly in the syntax of the language. Knuth found that one of the most frequent statements was of the form

$$A = A + 1$$

and this was only exceeded by the number of assignments

$$A = B.$$

Both of these uses bias the number of scalars to higher levels than would be expected in languages with clear flow of control and natural accessing of variables.

[4] All but dll are implemented in microcode by the Burroughs S machine [31].

that they implemented a linked list operator on the B5500 and upward machines. A double-linked list has two reference-like elements bound with a single information element. Since the information elements is self-identifying we eliminate the necessity of header bits as used in SLIP [32].

Stacks have proven so useful that hardware stacks have been provided in the B5000 family of computers, the KDF-9 [33]. the Sigma 7 family [34], the Digital Equipment Corporation PDP-6 family [35], the PDP-11 family [36], and the R-2 to name only a few. The language IPL-V [37] has shown the potential of a multiple-stack language. It features a program counter stack, ten workspace stacks, and a communications stack. It also has the feature that any data item is potentially a stack. One of the most novel features of the language is that when a subprocedure ends, it simply pops the program counter stack and resumes processing.[5] The R-2 combines all these stacks into one, as do all other current machines. One may simulate the above by simply swapping stack pointers. Clearly with a tagged machine one need not be limited to one stack and can have a stack item whose fields denote top of the stack, accessible lower bound, absolute lower bound, absolute upper bound, and element stacked type.

Queues also have a considerable number of uses in programming languages, especially those that will be used to build operating systems [38], [39]. The most commonly useful queue in an operating system is the priority queue where priority is a function of several variables and varies with time. The SYMBOL operating system and the recent Berkeley Computer featured hardware scheduling with parameterized priority queue management. Clearly a priority queue with continuously varying priority is difficult to implement in hardware although it has been done for disk queueing on the Burroughs B6700 [40]. Instead of this kind of queue we will build a modular queue of references to other things and associate them with their priority. This special representation q would possess a base, a length, flex bit, front offset, and rear offset. We will use it in conjunction with a queue-scanning operator to achieve priority queueing.

One of the things that we want to queue is a group of machine states m.s. or process states. This packet of bits is different for every machine or process organization. Basically it consists of the information that is necessary and sufficient to resume a process. Thus in order to schedule a processor we supply it with a queue of machine states and it will execute these processes in queue sequence.

In order to perform parallel processing and asynchronous processing we require events, interrupts, and messages. Messages consist of a string of characters that we wish to pass from one user to another via the operating system. They also have a header indicating the identity of the sender and who is to receive them or read them. In the most primitive systems a message may be used as a semaphore as in Dijkstra [41]. Alternatively a more general arrangement permits the message to coordinate a recursively defined operating system as in Brinch-Hansen [42]. Messages may be queued for the attention of a process using the mechanism described above.

In many cases a process may wish to wait on the completion of a set of external events. This could be handled by waiting for a message to be returned, but this would involve continual scanning of the input message queue. Rather than do this the process could set up an event and arrange to be put on the waiting queue pending its completion. Alternatively the process might wish to interrupt itself at a specific occurrence such as an incoming message. The event and the interrupt type give a mechanism suited to the above. Each event gives either a queue of processes or a reference to a process which are waiting for the event to happen and also a queue of processes that wish to be interrupted if the event occurs [43], [44]. We may separate these queues, or, using an indirect tag of interrupt or machine state achieve a single queue.

The interrupt is process specific. That is, each process features an interrupt item for each event to which it is attached. This item indicates whether the interrupt is enabled or disabled and it specifies an interrupt process to be executed if it is enabled. Of additional importance is a priority associated with the interrupt procedure that can be used if the interrupts are to be queued.

There are two definitions of hardware types remaining that are required by language designers. The first is a parameter set ps and the second a procedure, proc. The parameter set facilitates programming generality [45]. It can be used to pass actual parameters to a procedure or to access an array. It contains information about what is passed and about how it is to be used. One of the simplest parameter sets consists of the indices for accessing a volume, an ordered set of three subscripts. A parameter set attached to a procedure may indicate bound variables or formal parameters to be bound later. A formal parameter set associated with each procedure may be matched with the actual parameter set used at run time.

The parameter set is necessary for the implementation of general procedures. These procedures will be of two kinds: type procedures, used largely for structural access, and functional procedures. The mechanisms that we will describe are provided to allow implementation of functional data [46] and languages like Gedanken. In these languages all data are treated as if they were generated by a function instead of being accessed. Second, since a function is a data item, a function may be returned as a value. Further, functions may return multiple values. Although complex logical problems are created in cased where a function of n arguments can yield a function of m arguments as its value, recent work by Scott [47], [48] appears to indicate that such computations are logically consistent provided certain rules are followed.

If the realization of such functional data is to be efficient, the hardware must be able to distinguish procedures. If the hardware cannot, the problems of whether to execute the data or not, become overwhelming. In the following, we shall discuss two different realizations of the preceding. The first

[5] PL/1 provides controlled storage in which each variable serves as the top of a stack.

centers on type codes and the second immediate execution of a procedure.

The interpretation of operations has been done for some time. When the IBM 360/44 executes a decimal or character instruction, the computer traps and emulates the execution of it. The Digital Equipment Corporation PDP-11 features a large set of emulator traps for just such a purpose.[6] As another example, on the R-1 certain data words, designated code words, were passed to the operating system and interpreted as a sort of microcoded instruction to READ, WRITE, MARK, or MOVE data or to allocate or deallocate space.

One of the most important kinds of functional data is that whose representation or access is specially prescribed. For example on the R-2 double precision and complex are interpreted representations, as was floating point on the IBM 701. When two complex quantities are added together, the hardware does not do the job automatically, instead it traps to a special location with information about the operation and its operands. A software routine performs the operation and returns control to the main instruction flow.

The use of type codes to indicate to the hardware which item is being referenced and thus what method of access and representation is being used, is easily extended to permit programmers to specify special accessing algorithms or representations in a kind of microcoded data. As one example, a programmer wishing to deal with triple-precision numbers may select one of the unused type codes which is interpreted and provide software routines that deal with his special data for all operations, logical operations, and conversions from type to type.

Alternatively a programmer may define data access methods which the hardware does not recognize such as a sparse matrix, four-dimensional array, sequence of variable length quantities, dequeue, etc. The occurrence of the type code causes an interrupt that saves the location of the referenced items. The program at the indicated interrupt point handles the access and returns control to the hardware on completion.

It is quite important that the hardware handle these software interrupts and functional data in a uniform and efficient manner. An example of the way this is done is found on the Burroughs B5500. Suppose one attempts to load an item that is a function descriptor when referencing data. The old environment is automatically saved, parameters are removed from the communications stack, the function is calculated, the result is left on the communication stack, and the old environment is restored. The result has appeared as it would if we had simply requested data. Even if the function requires that other functions be called, this can be done without disturbing the initial environment. So it should be both in calculating a functional data item or in performing an access function. Functional data items may call for further calculations and one access may call for another recursively. One of the types in a typed architecture should be procedure. When an item marked proc is encountered it is treated in much the same manner as on the B5500. The current environment is saved. The parameter set of the procedure is examined and each of the actual parameters is evaluated. The procedure is then executed. If the procedure references a formal parameter, whose actual value has not been supplied or whose actual value does not match the formal parameters, an error trap occurs.

The parameter set permits us the additional flexibility which we require in order to implement Gedanken. In Gedanken it is possible for the invocation of a function to return either a new function or to return a version of the function with some of the parameters bound, that is, with formal parameters replaced with actual parameters. An example is given below in the Lisp [21] notation.

DEFINE ((LARGE-FUNCTION (LAMBDA(X, Y) PLUS (X, Y)))) which defines the function $X + Y$ where both X and Y are variables. After this function has been defined it has the parameter set that includes X and Y as formal parameters. We should be able to do the following.

DEFINE((SMALL-FUNCTION (LAMBDA(Z) ADD1(Z)))) which adds 1 to X where X is the sole member of the parameter set and it is found, and finally we have the following.

DEFINE ((STRANGE (LAMBDA(U, V) LARGE-FUNCTION (U, SMALL-FUNCTION (V))))).

In the preceding example two functions are defined. Then a third function STRANGE is defined by binding the formal argument Y of the LARGE-FUNCTION to the actual parameter which is in turn SMALL-FUNCTION (Z). If we evaluate STRANGE (2, 3) first we evaluate the parameter set for STRANGE. U is replaced by 2 and V by 3 in the actual parameter set. Next the ps of LARGE-FUNCTION is evaluated. X receives the value Z and and Y the value of SMALL-FUNCTION (Z). In order to obtain this value the parameter set for SMALL-FUNCTION is evaluated and Z is replaced by 3. SMALL-FUNCTION is then executed resulting in 4. Y is replaced by 4. LARGE-FUNCTION is then executed. The result is 6. The actual parameters may depend on the environment that exists at the time of binding. It is therefore very important that this information be kept regardless of the number of levels of recursion required to evaluate all parameter sets and functions.

Type functions also have a parameter set. Consider as an example a volume. In order to access an element, all three subscripts must be known at the time of access. On the R-1 these parameters were held in three index registers. Or consider a complex number; the parameters were the values of the two successive storage locations, the real and imaginary parts themselves.

We are now equipped with all the hardware mechanisms that we shall need to construct languages. All other apparatus is simply constructed from the tools at hand. Consider the problem of the Algol 68 structure (struct). When one declares a structure he is in fact declaring an access rule for a new entity. The compiler writer can designate a specific type tag and provide the appropriate access routines at the interrupt control point. Or consider the problem of declaring a new operator for use with a given structure; this amounts to adding

[6] Several older machines provided this capability including members of the DEC-PDP family and the SDS-900 series.

an operator procedure at the interrupt control point. Since different structures are given different type tags, even though the access rules are the same, each is identified and one may not be mistaken for the other unintentionally.

As far as languages are concerned, we now meet Dennis' three criteria for programming generality [45]. We can create information structures of arbitrary extent. We can call on procedures with unknown requirements for storage and information structures, and we can create information structures of arbitrary complexity in a called procedure.

So we see that we have satisfied the major demands for data formats and procedure generality with the addition of the extra programmed and unprogrammed types. We have been careful to keep the mechanism for performing these actions general and simple. Thus the expense and complexity of the architecture have not been drastically increased, but the implementation of programming systems has been greatly facilitated.

IV. Operating Systems

The elements that we have included as primitive (hardware implemented) units of computation aid us in structuring programs written in an advanced programming language without exacting any undue penalty, although they do not constitute a complete set. In the previous sections, we have carefully avoided reference to the help that would be provided the operating system by increasing hardware complexity. In part this is because we feel that the programmer in an applications language should look at the system as if the hardware and software he is employing are integral and constitute a high-level language machine such as SYMBOL or ISPL [50].

We have not selected the units of computation by accident. The programming language based on the hardware features described above, constitutes a good one for most high-level language programming and in its provision of queues, list structure elements, structures, and other elements greatly simplifies the design of an operating system. However, five areas of concern remain untouched, including naming, protection and sharing, resource management including time, memory, and input-output, debugging, and system instrumentation. In each of these areas tagged architecture contributes significantly.

A. Naming

Several attempts have recently been made to solve the naming problem. Among them are the works of Dennis and Van Horn [51], Clingen [52], Fraser [53], and the SYMBOL, ISPL, and MU-5 [65] processors. In the last-mentioned processors, instead of striving for additional generality, the name management problem has been structured and is handled by the hardware. The designers' argument is that a name ought to refer to one thing and only one throughout the whole program. Since names correspond to a unique mapping into primary or secondary memory space, the names can be relegated to an associative memory and the mapping can be accomplished associatively.

If we reject this approach on the grounds that it makes naming of shared modules too difficult since names may not refer to more than one quantity, we might utilize a tag to identify a special quality called a name. Together with an identifier, which we will discuss next, we can develop a compound which when used associatively gives us the same ease as in the other scheme but which allows names with multiple associations.

B. Protection and Sharing

Protection and sharing are highly related to the implementation of names. The interested reader is referred to Dennis and Van Horn [51], Fabry [54]–[56], Graham [57], Lampson [58]–[60], and Schroeder and Saltzer [61]. These writers reflect the modern belief that sharing presents severe problems to most protection schemes. Second, they appear unanimous in the belief that physical protection as implemented by lock bytes and other schemes must give way to logical protection. Lampson [60] shows that the most general kind of protection mechanism makes use of an access matrix in which process identities are matched with names of domains of quantities to be accessed. In some cases the processes may execute, read, write, copy, or use the named quantities. Lampson points out that two ways of implementing protection have been used. The first is the capability list. This is a list by process of the domains which the process can access. The second is called the access list and is a list by domain of those processes which may access the domain.

In order to implement sharing most conveniently, it is important that the name of a domain of protected or shared things, cannot be created, altered, or destroyed, accidentally or maliciously. This implies that the name must be recognized by the hardware. Of course, it follows that the process i.d. must also be of the same nature. We propose that both be made types, name and i.d.

Given that only the operating system nucleus can create or change name and i.d. we may now build the access list or the capability list as structures or even implement Lampson's protection matrix. Further, because of our specification of functional data, it is possible that the data at the intersection of i.d. row and name column can be a function which changes dependent on time, the i.d. of user or the scheduler. Domains may be as small as necessary to allow sharing between as many people as necessary or as large as possible to preserve efficiency. The protection attribute may be checked as often as necessary to assure secure operation. For example, it would be checked more often for storage than for disk files.

The notion of gate and ring as proposed by earlier writers can now be simply implemented. The concept of gate is that one may only enter other procedures at selected points. Presumably procedures are prepared to protect themselves by argument validation at these points but not at others. Since the only method of transferring from one procedure to another is via a label and since labels are protected from tampering, all of the procedures in a domain may be associated with a named vector of labels and hence protected.

The concept of a ring or sphere of protection is a simple linear ordering of access rights. That is, a process may easily use procedures or data at one level but may not at another. Its rights while at one level may be different with respect to a data item than at another. Since we do not differentiate between procedures (their references) and data, it should be clear

that we can treat each or either as belonging to a named item which is thus protected. Thus we may conveniently implement lattices of protection rather than merely linear orderings as suggested by Arden in July 1969, at the University of Michigan Summer School on Advanced Programming Systems Design.

C. Resource Management

Resource management is made considerably simpler by the hardware implementations of msg, event, ps, ipt, q, sll, dll, stack, name, and i.d. and by the implementation of queue and list scanning operators and the functional data processing mechanism. We shall not comment on the scheduler but will dwell instead on the problems of addressing and memory management.

Discussions about the merits of paging are still being conducted.[7] We will assume that any paging mechanism together with its management can be committed to hardware or to a level of detail below that which we will consider. We then find ourselves dealing with three types of addresses. The first is the absolute address. It refers to an absolute address in the physical absolute address space. The second is the local relative to some named thing. The first two types of address are standard on the R-2. Each are recognized by a special tag value.

The relative address is of special use. It allows the operator using the address to know the location of the address and to access relative to it. This greatly facilitates relocation of data since only one external reference into a block of data need be used. When the data is moved only the external address need be modified in order to establish addressability.

The absolute address is required if multiple noncontiguous segments are to be used within one process. These addresses must be changed if the blocks to which they point are moved. Unfortunately, it is often the case that copies of these addresses are made and deposited in stacks, data areas, and elsewhere. Then when the blocks pointed to by the copies are moved, the absolute addresses must be found and modified. One method of alleviating this problem is to regulate the generation of copies by the conclusion of a bit in the tag indicating that such an item may not be copied. This bit is naturally useful in the protection process (see for example Lampson [60]).

An alternative is to have name-relative addressing, also known as segment relative. Here an address is specified relative to a named (protected or shared) thing in the manner described as NAME.OFFSET. Whether this is done via a hardware display as on the Burroughs B6700, or a segment table as on the GE645, or via an associative name table as on the SYMBOL and MU-5 machines is unimportant. If it is provided, it greatly simplifies the problems of sharing, protection, relocation, and binding.

We have satisfied the major demands for data formats and procedure generality with the addition of the extra programmed and unprogrammed types. We have been careful to keep the mechanism for performing these actions general and simple although the implementation may vary in complexity and cost. Thus the expense and complexity of the architecture need not be drastically increased, but the implementation of programming systems may be greatly facilitated.

D. Debugging and Instrumentation

Two features of a computer system not previously discussed and yet corequisite with programming languages and operating systems are the debugging and instrumentation systems. The debugging system is essential to obtain working programs on a working machine. The instrumentation system is to tune the system after the programs are working. Both should be useful in the certification of a program. Tags facilitate both.

On the R-1, they were used to denote one instruction or a sequence of machine instructions to be traced (interpretively) and they were used to monitor the values of data or addresses that were accessed by instructions. Although the tags could have been preassembled into object code, this was not their primary use in practice. Instead the operating system allowed the user to dynamically set or reset these tags during the execution of the program. The tags were set or reset by absolute reference, or by reference to symbolic names.

As with the R-1, tags can be used to mark instructions or data, directly or indirectly. When the tagged datum or instruction is encountered, a trap occurs to the instrumentation or debugging routine. At this point the appropriate routine determines why it has been activated. An instrumentation routine might enable or disable a clock, increment a counter, or validate the result of an arithmetic operation. A debugging routine might print out the contents of the machine's fast registers or offer the programmer a chance to interact with the program before resumption, just as if an ON-CONDITION had been encountered in PL/1.

The introduction of a more complete tagged architecture allows more programmatic assistance in determination of catastrophic failures. In the past it was necessary for the programmer to peruse octal (hexadecimal) dumps of memory. With tagged architecture, the system clean up routines can go through user memory recalling the structure of the program by the use of the tags and printing out each datum in its own predetermined format. Because the system can programmatically recover the structure and data, the programmer may more quickly identify undiagnosed errors. In the same manner the system may programmatically do its own error recovery.

Tagged architecture presents clear advantages in both instrumentation and error recovery.

V. Tags and Computer Architecture

In the previous sections we described advantages of tagged architecture apparent in the design, implementation, and use of software. In this section we investigate the ways in which tags may be employed to improve utilization of hardware in the medium- to high-performance categories. The sum of the advantages strongly suggests that the ramifications of a fully tagged architecture should be studied and exploited. These hardware advantages stem from the machine's ability to determine the context in which data is used, to achieve better register utilization, data scheduling, parallelism of functional units, or use of specialized functional units.

We will use conventional architectures as examples of how

[7] Paging is currently available on some machines produced by IBM, CDC, SDS, RCA, and Honeywell-GE. A form of "paged" segmentation is available on the B6700 and B7700 computers.

hardware is being employed in these areas. We will attempt to show how the use of tags would aid in current implementations, suggest generalization of current algorithms, or imply new areas of performance that should be investigated.

A. Register Allocation

To begin, we consider register allocation. One of the hardest problems in achieving efficient utilization of the central processor is in the allocation of the registers constituting the processor's register file. Often they have specialized functions such as indexing, floating-point operations, and basing, which make them nonhomogeneous. On almost any machine, their number is small for two reasons. They are accessible faster than the registers of central memory and much more expensive per bit. To be used effectively they require gating paths to arithmetic, logical, and control resources which may be very expensive. If we are to obtain high efficiency, we must keep these registers in constant use.

The register allocation problem is especially difficult because the same registers are usually used to perform the accessing algorithm and to do the arithmetic computations on the data. Usually both the accessing and the computation are independent, but because the machine typically can not recognize that one is the access algorithm and the other the computation, the two are not processed in parallel. In fact usually they are processed so that both are in contention for the register resources. If on the other hand the processor can recognize the data type by means of a tag and pick the access algorithm, it can perform the accessing algorithm and the computation in parallel and often at a much higher rate. In some cases a knowledge of the accessing algorithm permits concurrent execution of independent phases of the algorithm.

As an example let us consider the problem of accessing a matrix. A matrix has an access function that may be specified in many ways. Two different forms are specified below where A is declared: real $A[L_1 : H_1, L_0 : H_0]$ and row major order is assumed.

$$\text{addr}(A[I,J]) = \text{addr}(A[0,0]) + (H_0 - L_0 + 1)(I - L_1) + (J - L_0) \quad (1)$$

$$\text{addr}(A[I,J]) = \text{addr}(A[0,0]) + S_0 + S_1 I + J \quad (2)$$

where S_0 and S_1 are strides of A and, in order to be legitimate

$$L_1 \leqslant I \leqslant H_1 \text{ and } L_0 \leqslant J \leqslant H_0.$$

In the protected system, it is imperative that the last two conditions be verified on every access. If we know that the quantity in question is an array, by virtue of our possession of the accessing mechanism, a dope vector containing among other things: $H_0, H_1, L_0, L_1, S_0, S_1$, and addr $(A[0,0])$, we can conclude many quantities in parallel including: $I - L_1$, $H_1 - I, J - L_0, H_0 - J$, addr $(A[0,0]) + S_0, S_1 \times I$ in the first step. If any of the first four are negative, we can abort the access since the address is not legitimate. In the second step we add result four and J to select the proper column. On the third round we select element $S_1 \times J$ from the column. This kind of algorithm is simple to implement in hardware and is quite useful if the machine knows with what kind of data it is dealing, and where it may obtain the quantities I, J, L_1, L_0, H_1, H_0 and the base address of $A[I,J]$. With tagged data it does know all the preceding quantities.

Often the bounds calculations could be omitted without loss of security. If we were to take every element of the array in row major order we could simply form $T = \text{addr}(A[0,0]) + S_0$ and $L = S_1 \times (H_1 - L_1 + 1)$. To select the next element, add one to T and access; for correctness L must be decremented by 1 and checked for a zero or a negative value.

We observe that if the computer is to deal with an array without "knowing" that elements are to be accessed sequentially, seven arithmetic operations, four tests, and one indexed fetch must be performed for access assuming that registers contain the nine necessary values. On the other hand, if the computer recognizes the type and the context in which it is performing the access, after four initial operations overhead are performed, only two registers are required; only two arithmetic operations, one test, and an access are required per operand. As writers of compilers recognize, the method assumed in (1) is most expensive (especially in serial) and the last method is most preferred when applicable. If multiply's require 5 cycles, additions 2, tests 2, and fetches 8, the serial method requires 41 cycles including bounds checks and final fetch; the parallel method 13; and the serial selection 10, assuming all quantities are in registers. Thus if the machine can select an access algorithm to suit the context of the operation, it can greatly improve efficiency and register utilization. Tags greatly aid in this process.

In the above discussion we have assumed access of elements from one array. In practical circumstances we usually deal with several array accesses simultaneously in connection with some operation such as scalar product, inner product, or outer product. In these operations, register allocation is especially difficult because of the large number of quantities that must be shuffled in order to perform the accesses and the arithmetic functions. If the machine can keep track of the larger context of data and the instruction to be performed, there is some hope that register utilization can be optimized.

As specific examples of current mechanized solutions, we will examine the CDC STAR-100 and the TI-ASC computers. The STAR has 256 64-bit registers called a register file. Vectors are denoted by a single 64-bit word that contains a 16-bit length field and a 48-bit address (to the bit level). While the STAR does not have a tag to denote quantities, it does have vector instructions, which the programmer must initialize in the register file. A vector instruction can denote up to seven 64-bit registers each of which is assumed to contain a word denoting a vector. Given the operations $A = B + C$ where A, B, and C are vectors, the STAR will fill the result vector A, extending a short input vector B or C as necessary with the identity element. All the details of the access algorithm are automatically performed by the hardware and no additional registers are required. The ASC contains an arithmetic unit with a doubly nested iteration structure suitable for matrix or vector instruction. This unit is initialized by the programmer with "packets" of information describing the matrices or vectors to be manipulated and the operations to be performed.

The machine automatically performs the iteration from "hidden" parameter registers and performs the access. These hidden registers are special and may not be used for other functions. If we have: DIMENSION $A(10, 100)$, $B(10, 70)$, $C(70,100)$ the machine is simply programmed to perform

$$A_{ij} = B_{ik}C_{kj}$$

where the computer automatically performs incrementation in two layers. In this way both machines partially implement the PL/1 philosophy of implicit operations between compatible types.

It is important to note that in the above examples the programmer did not have to worry about the looping mechanism or attendant register conflicts, after he completed the initialization process and issued the correct instruction. Thus the problem is removed from the domain of compiler optimization in which the compiler writer strives to get the machine to do its best for the sake of efficiency, and is put in the domain of the machine designer who can optimize part of the machine hardware to solve the problem.

A second example of the reduction of the problem of register allocation and looping can be found on the CDC-1604 computer and its masked search operation. This computer features a mask register, a comparand register, a lower bound register, and an upper bound register. The machine instruction, masked search, causes the machine to automatically iterate through the array described by the upper and lower bounds register, masking the contents of each accessed word and testing it against the comparand. The accessing algorithm including iteration, and comparison are automatically performed by the computer without register side effects.

Burroughs incorporates a linked list lookup instruction in the B5000 and succeeding models which automatically performs the access algorithm by iterating through a linked list while testing a subfield for termination. This instruction greatly improves the performance of the operating system in the area of storage allocation and garbage collection since it eliminates instruction fetches from storage, prevents stack motion to memory, and does not require additional instruction execution cycles.

To summarize, the problems of register allocation are compounded because of contention between access algorithm and the calculation that is being performed for the limited register resources. The use of tagged architecture permits a single instruction to specify the calculation, i.e., add, multiply, linked list lookup. Examination of tagged descriptors permits the hardware to choose the most efficient access method. The hardware may then execute the access algorithm in parallel with the execution of the calculation.

B. Stream Processing

Since tagged data identifies the access algorithm and includes the parameters; parameter file setup is very simple. Once the operation is known, the machine can schedule delivery of the data in streams to pipelined functional units. If the access algorithm is mechanized in hardware no instructions need be fetched during access. The lower instruction-fetch-per-data-fetch ratio, allows a great economic benefit: the use of memory with high latency and high bandwidth which is relatively inexpensive, for example linear select memory.

The CDC STAR is an excellent example. It has three streamed arithmetic units, one for floating multiplication and addition, one for division and miscellaneous numeric operations, and one for string operations. The basic minor cycle time of the STAR-100 will be 40 ns. The pipe can accept operands at a rate of two per minor cycle. After an initial pipeline latency, which varies depending on the operation performed, results appear at the rate of one per minor cycle. Clearly it would be impossible to perform iteration and fetching at such a rate if the instructions for accessing were not known in advance. Since only the vectors are streamed, the access algorithm is to get the next element which must be in the next sequential (virtual) location. The processor can cause four coordinated input streams from memory and two to memory simultaneously, while using a very high percentage of the available memory bandwidth. Further, there is no need for an instruction fetch that might cause bank conflict or an instruction decode.

Thus, CDC is able to use memories of 1.2-μs cycle time with superwords of 512 bits plus a parity bit for each 32 bits. These memories are phased in such a way that 200 ns after the request is honored, the first $\frac{1}{4}$ of the superword is on the way to the CPU with the other quarter-words staged 40 ns behind. Since a minimum STAR has 512K–64-bit words in 8 independent modules, it is possible to keep the pipeline running at full speed for at least 64 000 pairs of operands at a data rate of one 32-bit word per 2.5 ns.

To summarize, the use of tagged descriptors to refer to long blocks of data in the form of vectors, as on STAR, or matrices, as on the TI-ASC, permits the hardware designer to use block transfer techniques and pipelines to great advantage. During these transfers no instructions are fetched from memory and the full memory bandwidth can be employed to deliver operands to the pipeline functional units.

C. Cache Memories

The ideas above may be extended to smaller blocks of data and instructions for use with cache memory. For example the CDC 6600 and 7600 have instruction buffers holding from 20 to 40 instructions in very high-speed memory. When the instruction loops occur within the buffer the machines issue instructions at a very high rate. If the instructions must be fetched from memory in a random manner, the execution slows down from 4 to 8 times. Unfortunately a large portion of the instructions used to fill the buffer memory are concerned with data access. If these access algorithms could be mechanized by the hardware and denoted by tagged data in a parameter cache, this would offer several advantages. First a larger number of calculations could be resident in instruction stack loops. This would mean no instruction fetches to disturb the well-regulated delivery of operands to the execution units. Second, the data access method might be selected by the hardware to make optimal use of available memory bandwidth.

One of the disadvantages of cache memory operating with conventional machines such as the IBM 370-155 is that data is

moved in small blocks. When a word in a data area is accessed, a 64-byte block containing the word is accessed whether one word or all words are desired. This method tends to work as long as data is in small working sets and is not accessed randomly. Severe deterioration results with a random access pattern as a high percentage of memory bandwidth and valuable cache storage is wasted. As long as the computational unit is the word and data access algorithms are embedded in the calculation, no better solution exists.

If, however, tagged descriptors are used to describe the data and they are kept in the cache, a better flow of data to the execution units can be maintained. This is true for three reasons. First, operations will tend to be on computational entities, e.g., add vector a to vector b. If this is so, the machine can recognize that the entire vector will be required and cause it to be transferred to local storage. Second, move operations such as copy vector a to vector b become block copy operations that do not utilize cache space. Third, if selection of an element of a quantity is required, only the required element need be moved from memory to cache, thus decreasing cache waste and the drain on available memory bandwidth.

To summarize, tagged architecture facilitates the use of cache memories. It helps to separate accessing from other computation, increasing the value of an instruction stack. It permits the use of tagged descriptors to represent logically related aggregates that may be manipulated as a whole without wasting memory bandwidth or cache space.

D. Parallel and Specialized Processors

Next we turn to the use of tags in systems of parallel processors or in systems of functionally specialized processors acting in concert. Provided that no more than one instruction stream ever accesses words of data simultaneously, we may assume that a well-designed compiler can assure protection of the data from mistaken access or other misuse leading to indeterminate computation. When several instruction streams acting independently have the capability of simultaneous access, such protection is extremely difficult to obtain and the probability of indeterminate computations rise. Similar problems have arisen in file systems giving rise to semaphores, locks, keys, and numerous other techniques of resource allocation.

This problem of protection is compounded if the entities on which we perform the computation are distributed in memory, unless we assign a lock to every addressable quantity. For example while user A is merely changing $A(I,J,K)$, user B can be changing $A(I, J, K + 1)$ without either being aware of the other. Or A may be changing a bound in a dope vector which will cause B's next store to change C's result or D may be interpreting E's pointer as a real number. If one is not to incur tremendous overhead in locking and unlocking everything, the hardware designer must provide more tools in systems of parallel processors.

One method of protection used in allowing parallel use of file structures is to control access logically at a directory level. This same concept is quite naturally implemented through the use of tags. The only method of access to anything should be via tagged descriptors. Because they are tagged the hardware can prevent their alteration. Since hardware access to the entity is funneled through one master descriptor, locking and unlocking is greatly facilitated and the hardware prevents misaccess or untimely access automatically. These data gates also assure that the things described are used in a uniform and supervised manner by all instruction streams accessing them, relieving the compiler writer from the responsibility of inserting instructions in the instruction stream to guarantee this function.

This last point, encompassing the idea of self-identifying data is especially important if specialized functional units are to be employed as on the SYMBOL computer, the Berkeley Computer Corporation's BCC-500, or the IBM 370/125. The idea in both these computers is to provide a specialized computer for a specialized task. Examples of the specialized functions include garbage collection, storage hierarchy management, communications processing, name management, automatic parsing, scheduling, arithmetic processing, and input-output processing. Each computer is designed to perform its task asynchronously, independent except for message communication and access to common main memory. Finally each computer has a specialized instruction set and register complement suited to its task.

Cooperation between these independent units is enhanced by the use of tags. For example the garbage collector could make a linear sweep over memory looking for garbage tags. These tags would occur in a reference to a block of storage and the processor could return the whole block to free storage. Or consider the scheduler. Because it recognizes a queue, an event, an interrupt, and a machine state, it can deal with them much more efficiently. A communications processor would automatically recognize a message and be able to speed it from sender to receiver. The hierarchy manager could recognize that large blocks of information were about to be required and transport them from serial or backup memory, communicating with the name manager to make the arithmetic processor aware of the in-core presence.

In summary we see that tags simplify the design of cooperating parallel processors and specialized functional units by standardization of communication and protection of computation entities from accidental misuse or untimely use.

VI. Implementation

There are too many forms that a tagged architecture might take to attempt a defense of a specific implementation in terms of cost performance [63], [64]. The costs of logic, main memory, mass serial and mass random memory change hourly and with the volume of units produced. We will examine how a low-cost implementation might be achieved and suggest a percentage of increased cost on a minicomputer system. A more detailed investigation into the cost of a system which is centered on the concept of tagging to attempt to extract every benefit from the concept in hardware and software is underway at Rice University.

Most minicomputers feature an interrupt structure. When an interrupt line is activated the machine traps to a given location and stores its program counter in a second location. On more advanced minicomputers a different trap location is employed for each kind of interrupt. Given such a minicomputer with a 10-bit byte and a small amount of hardware, we have an elementary tagged computer. Let one bit of each byte be for

parity, one for a tag, and eight for data. When a tag bit is encountered and the machine is in user mode, defined by a single flip-flop, the hardware traps to the absolute location associated with the data byte. For example suppose the data byte is 250_{10}, the hardware traps to "location" 250 and executes the program to be found there. The only extra hardware required is a small number of gates and one extra bit per word. The cost of the extra bit per byte is the important cost, adding $\frac{1}{9}$ to the cost of all memory. Memory is an expensive component in minicomputer systems. On a 16-Kb PDP-11/20 at retail, it amounts to about 40 percent of the base cost of the machine. If we assume that increasing the width by 12.5 percent costs 12.5 percent more, the added cost is 5 percent of that of the basic machine. We have not considered the cost of software or peripherals. If we add a punch, a paper tape reader, and an inexpensive disk drive and controller to the basic machine, the added memory constitutes an increase of 3 percent. In a well-configured system it is not unlikely that the peripherals will cost more than the minicomputer and that the additional expense in memory will be from 2 to 3 percent of the system cost.

One also could question the space for programs to interpret the data. Since these programs constitute closed subprograms, the space that they require is less than or equal to the space that would be required to perform the same function in the user programs. This explains why Fortran IV interpreters are so widely used on small machines. Further, because the access routines are localized, standard data type programs can be relegated to READ-ONLY memory, cheaper at the same speed than random access memory, or to microprogram, which can be considerably faster. In either case the performance for a given cost is improved.

It is admitted that a larger data bus to secondary store and that more secondary store will be required. However, the ability to design an arbitrary segmented virtual memory scheme using the tag mechanism should more than justify this, since main memory is (conservatively) at least 30 times more expensive than disk memory and for the cost of 16 Kb one can get another disk unit.

In terms of software, much of what the purchaser of a minicomputer is paying for, is development. It has been conjectured but not conclusively demonstrated that software is easier to design if the appropriate data types are available. The development cost should be lower because high-level languages and most other software can be implemented in high-level languages.

As discussed initially we are not prepared to analyze the cost effectiveness in any more specific terms. We believe that as the cost of memory decreases with respect to the total system cost, that there is an excellent argument for tagged architecture form minicomputer to supercomputer and in the last two sections we have attempted to present our case for hardware.

VII. Summary

We have described some of the advantages of tagged architecture. Starting with its history we have shown the usefulness of tagged data types in the production of software for compilers and operating systems. We have shown some uses of tags in debugging of and instrumentation of programs. We have suggested ways in which tagged architecture alleviates problems in register allocation, parallel processing, and specialized processing. We have indicated reasons why tagged architecture facilitates streamed calculation and the use of cache memories. We have suggested that tagged architecture of lower performance can be easily implemented on minicomputers at low additional system cost.

VIII. Conclusions

We see that a complete tagged architecture provides us natural mechanisms for performing practical computations. It allows us to deal with stereotyped data as efficiently as the hardware is able and with new kinds of data in a natural and transparent manner. It simplifies the design, construction, and debugging of programming and operating systems. It can be implemented economically in minicomputers. It features numerous design tradeoffs that may be useful in medium- to large-scale systems in improving speed and memory utilization.

Taken together, the arguments we have advanced provide a powerful incentive for further investigation and exploitation of tagged architecture. Such a machine may soon well be a replacement for today's widely accepted von Newmann architecture.

Acknowledgment

The author wishes to thank the referees for their comments which helped in making this a better paper. He also wishes to acknowledge the works of J. K. Iliffe, M. Wells, J. Reynolds, and discussions with his many co-workers at the Institute for Defense Analysis, Rice University, and Lawrence Livermore Laboratories.

References

[1] G. D. Chesley and W. R. Smith, "The hardware-implemented high-level machine language for SYMBOL," in *1971 Spring Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 38. Montvale, N.J.: AFIPS Press, 1971, pp. 563–574.

[2] R. Rice and W. R. Smith, "SYMBOL—A major departure from classic software dominated von Neumann computer systems," in *1971 Spring Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 38. Montvale, N.J.: AFIPS Press, 1971, pp. 575–588.

[3] B. E. Cowart, R. Rice, and S. F. Lundstrom, "The physical attributes and testing aspects of the SYMBOL system," in *1971 Spring Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 38. Montvale, N.J.: AFIPS Press, 1971, pp. 589–600.

[4] W. R. Smith *et al.*, "SYMBOL—A large experimental system exploring major hardware replacement of software," in *1971 Spring Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 38. Montvale, N.J.: AFIPS Press, 1971, pp. 601–616.

[5] J. C. Reynolds, "Gedanken—A simple typeless language based on the principle of completeness and the reference concept," *Commun. Ass. Comput. Mach.*, vol. 13, pp. 308–318, May 1970.

[6] *Rice University Computer—Basic Machine Operation*, Rice Univ., Houston, Tex., revised 1962.

[7] *SPIREL Manual*, Rice Comput. Project, Rice Univ., Houston, Tex., revised 1968.

[8] *The Descriptor*, Burroughs Corp., Detroit, Mich., 1961.

[9] R. S. Barton, "A new approach to the functional design of a digital computer," in *Proc. Western Joint Comput. Conf., Ass. Comput. Mach.*, 1961, pp. 393–396.

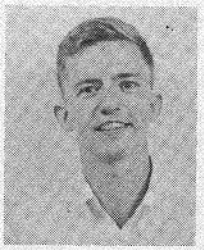
[10] B. Randell and C. J. Kuehner, "Dynamic storage allocation systems," *Commun. Ass. Comput. Mach.*, vol. 11, pp. 297–306, May 1968.

[11] A. D. Mullery, R. F. Schauer, and R. Rice, "ADAM—A problem-oriented symbol processor," in *1963 Spring Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 23. Washington, D.C.: Spartan, 1963, pp. 367–380.

[12] *"TR440 Eigenschaften des RD 441,"* AEG TELEFUNKEN, Konstanz, Germany, Rep. DBS 180 0470, Mar. 1970.

[13] F. von Sydow, "Der zentrale rechner des TR440," *Beihefte der*

Technischen Mitterlungen AEG TELEFUNKEN, vol. 3, pp. 104–109, 1970.
[14] *Burroughs B6500 Information Processing Systems Reference Manual*, Burroughs Corp., Detroit, Mich., Publ. 0143676, 1969.
[15] B. A. Creech, "Architecture of the B6500," in *Software Engineering Coins III*, vol. 1, J. Tou, Ed. New York: Academic, 1970, pp. 29–43.
[16] *Burroughs B6500 Information Processing System Master Control Program*, Burroughs Corp., Detroit, Mich., Publ. 1042447, 1969.
[17] J. K. Iliffe, *Basic Machine Principles.* New York: Elsevier, 1968.
[18] —, "Elements of BLM," *Comput. J.*, vol. 12, pp. 251–258, Aug. 1969.
[19] E. A. Feustel, "The Rice research computer—A tagged architecture," in *1972 Spring Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 40. Montvale, N.J.: AFIPS Press, 1972, pp. 369–377.
[20] E. W. Dijkstra, "Notes on structured programming," Univ. Maryland Comput. Sci. Cen., College Park, Aug. 1969.
[21] J. McCarthy *et al.*, *LISP 1.5 Programmer's Manual*, Res. Lab. Electron., Mass. Inst. Technol., Cambridge, Aug. 1962.
[22] D. E. Knuth, *The Art of Computer Programming*, vol. 1. Reading, Mass.: Addison-Wesley, 1968, ch. 2.
[23] D. H. Bradford and M. B. Wells, "MADCAP II," in *Annual Review of Automatic Programming*, vol. II, R. Goodman, Ed. New York: Pergamon, 1961.
[24] M. B. Wells *et al.*, *MADCAP VI Reference Manual*, Los Alamos Lab., Atomic Energy Commission, Los Alamos, N. Mex., 1971.
[25] R. S. Barton, "Ideas for computer systems organization: A personal survey," in *Software Engineering Coins III*, vol. 1, J. Tou, Ed. New York: Academic, 1970, pp. 7–15.
[26] *STAR Computer System Hardware Reference Manual*, Control Data Corp., St. Paul, Minn., Doc. 60256000, 1970.
[27] A. Van Wijngaarden (Ed.), B. J. Maillous, J. E. L. Peck, and C. H. A. Koster, *Report on the Algorithmic Language ALGOL 68, MR101.* Amsterdam: Mathematisch Centrum, Feb. 1969. (Available from Ass. Comput. Mach., New York, N.Y., in offprint.)
[28] J. B. Johnson, "The contour model of block structured process," in *Proc. Symp. Data Structures*, J. T. Tou and P. Wegner, Ed. New York: Ass. Comput. Mach., 1971, pp. 55–82.
[29] R. A. Brooker, "Influence of high-level languages on computer design," *Proc. Inst. Elec. Eng.*, vol. 117, pp. 1219–1224, July 1970.
[30] D. E. Knuth, "An empirical study of FORTRAN programs," *Software - Practice & Experience*, vol. 1, pp. 105–133, Apr.–June 1971.
[31] R. L. Davis and S. Zucker, "Structure of a multiprocessor using microprogrammable building blocks," *SIGMICRO Newsletter.* New York: Ass. Comput. Mach., 1971, pp. 28–42.
[32] J. Weizenbaum, "Symmetric list processor," *Commun. Ass. Comput. Mach.*, vol. 6, pp. 524–544, Sept. 1963.
[33] R. H. Allmark and J. R. Lucking, "Design of an arithmetic unit incorporating a nesting store," in *Proc. IFIP Congr. 62*, 1962, pp. 694–698; or see C. G. Bell and A. Newell, Ed., *Computer Structures: Reading and Examples.* New York: McGraw-Hill, 1971, ch. 21.
[34] *XDS Sigma 9 Computer*, Xerox Data Systems, El Segundo, Calif., Publ. 90 17 33A, Oct. 1970, p. 93.
[35] *PDP-10 Systems Reference Manual*, Digital Equipment Corp., Maynard, Mass., June 1968, pp. 2–13.
[36] *PDP-11 Handbook*, 2nd ed., Digital Equipment Corp., Maynard, Mass., 1969, p. 13.
[37] A. Newell *et al.*, *Information Processing Language - V Manual*, 2nd ed. Englewood Cliffs, N.J.: Prentice-Hall, 1964.
[38] D. M. Lyle, "A hierarchy of high order languages for systems programming," in *Proc. SIGPLAN Symp. Languages for Syst. Implementation.* New York: Ass. Comput. Mach., 1971.
[39] *Burroughs B6500 Information Processing System ESPOL Reference Manual*," Burroughs Corp., Detroit, Mich., Publ. 1042744, Jan. 1970.
[40] J. Abate and H. Dubner, "Optimizing performance of drum-like storage," *IEEE Trans. Comput.*, vol. C-18, pp. 992–996, Nov. 1969.
[41] E. W. Dijkstra, "The structure of T.H.E.–Multiprogramming system," *Commun. Ass. Comput. Mach.*, vol. 11, pp. 341–346, May 1968.
[42] Per Brinch Hansen, "The nucleus of a multiprogramming system," *Commun. Ass. Comput. Mach.*, vol. 13, pp. 238–242, Apr. 1970.
[43] J. G. Cleary, "Process handling on Burroughs B6500," in *Proc. 4th Australian Comput. Conf.* London: Griffin, 1969, pp. 231–239.
[44] E. I. Organic and J. G. Cleary, "A data structure model of the B6700 computer system," in *SIGPLAN Symp. Data Structures and Programming Languages.* New York: Ass. Comput. Mach., 1971, pp. 83–145.
[45] J. B. Dennis, "Programming generality, parallelism, and computer architecture," in *Proc. IFIP 1968.* Amsterdam: North Holland, 1968, C1-7ff.
[46] G. Mealy, "Another look at data," in *1967 Fall Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 31. Washington, D.C.: Thompson, 1967, pp. 525–534.
[47] D. Scott, *Outline of a Mathematical Theory of Computation*, Oxford Univ. Comput. Lab. Programming Res. Group, Tech. Mono. PRG-2, Nov. 1970.
[48] —, *The Lattice of Flow Diagrams*, Oxford Univ. Comput. Lab. Programming Res. Group, Tech. Mono. PRG-3, Nov. 1970.
[49] P. W. Shantz *et al.*, "WATFOR—The University of Waterloo FORTRAN IV Compiler," *Commun. Ass. Comput. Mach.*, vol. 10, pp. 41–44, Jan. 1967.
[50] R. Balzer, *ISPL Machine*, The Rand Corp., Santa Monica, Calif., Oct. 1971.
[51] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Commun. Ass. Comput. Mach.*, vol. 9, pp. 143–155, Mar. 1966.
[52] C. J. Clingen, "Program naming problems in a shared tree-structured hierarchy," presented at the Conf. Techniques in Software Engineering, Rome, Italy, Oct. 27–31, 1969.
[53] A. G. Fraser, "On the meaning of names in programming systems," *Commun. Ass. Comput. Mach.*, vol. 14, pp. 409–416, June 1970.
[54] R. S. Fabry, "A user's view of capabilities," *ICR Quart. Progr. Rep. 15*, Inst. Comput. Res., Univ. Chicago, Chicago, Rep. C00-2094-5, pp. 1–157, Nov. 1967
[55] —, "Preliminary description of a supervisor for a machine oriented around capabilities," *ICR Quart. Progr. Rep. 18*, Inst. Comput. Res., Univ. Chicago, Chicago, Sec. 1B, Aug. 1968.
[56] —, *List Structured Addressing*, Inst. Comput. Res., Univ. Chicago, Chicago, Rep. C00-2094-5, pp. 1–157, 1971.
[57] R. M. Graham, "Protection in an information processing utility," *Commun. Ass. Comput. Mach.*, vol. 11, pp. 365–369, May 1968.
[58] B. W. Lampson, "Dynamic protection structures," in *1969 Fall Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 35. Montvale, N.J.: AFIPS Press, 1969.
[59] —, "On reliable and extendable operating systems," presented at the 2nd NATO Conf. Software Engineering, Rome, Italy, Oct. 27–31, 1969.
[60] —, "Protection," in *Proc. 5th Annu. Princeton Conf. Inform. Sci. and Syst.*, Dep. Elec. Eng., Princeton Univ., Princeton, N.J., 1971, pp. 437–443.
[61] M. D. Schroeder and J. H. Saltzer, "A hardware architecture for implementing protection rings," *Commun. Ass. Comput. Mach.*, vol. 15, pp. 157–170, Mar. 1972.
[62] *Rice Computer - 2, General Specifications*, prelim. ed., Rice Univ., Houston, Tex., 1970.
[63] S. F. Dennis and M. G. Smith, "LSI-implications for future design and architecture," in *Spring Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 36. Montvale, N.J.: AFIPS Press, 1970, pp. 343–352.
[64] M. J. Flynn, "Towards more efficient computer organizations," in *Spring Joint Comput. Conf.*, AFIPS *Conf. Proc.*, vol. 36. Montvale, N.J.: AFIPS Press, 1970, pp. 1211–1217.
[65] R. N. Ibbett, "The MU-5 instruction pipeline," *Comput. J.*, vol. 15, pp. 42–50, Feb. 1972.
[66] G. Weinberg, *Psychology of Computer Programming.* Princeton, N.J.: Van Nostrand–Reinhold, 1971.



Edward A. Feustel (A'59–M'64) was born in Fort Wayne, Ind., on June 18, 1940. He received the B.S.E.E. and M.S.E.E. degrees simultaneously, in 1964 from the Massachusetts Institute of Technology, Cambridge, and the M.A. and Ph.D. degrees from Princeton University, Princeton, N.J., in 1966 and 1967, respectively.

From February 1967 through December 1967 he was a Research Fellow at the California Institute of Technology, Pasadena, Calif. From February 1968 to August 1968 he was on leave at the Institute for Defense Analysis, Princeton, N.J. While on leave at Princeton he was a Lecturer in the Department of Electrical Engineering, Princeton University. From May through December 1972 he was on academic leave from Rice University while at the Lawrence Livermore Laboratories, University of California. From February 1968 to the present he has been an Assistant Professor of Computer Science and Electrical Engineering, Rice University, Houston, Tex. His current research interests are in computer architecture, programming, languages, operating systems, artificial intelligence, and nonparametric statistics.

Dr. Feustel is a member of the Association for Computing Machinery, the Institute for Mathematical Statistics, the American Association of University Professors, the American Association for the Advancement of Science, Tau Beta Pi, Sigma Xi, Eta Kappa Nu, and is a Registered Professional Engineer in the State of Texas.