The Rice Research Computer—A tagged architecture*

by E. A. FEUSTEL

Rice University
Houston, Texas

INTRODUCTION

In this paper we report on a new computer with several novel features. These features are applications of the concept of tagged architecture, and although some of them are not unique to the Rice Research Computer (R-2), they focus our attention on this radical design form, its advantages and disadvantages. Since the work is still in progress, we limit this report to a discussion of the architecture and a few of its ramifications.

History of tagged architecture

The R-2 computer is an adaptation of a design of the Basic Language Machine (BLM)[1] of Iliffe. In his book and paper[2] he presents an argument for the utilization of a fraction of each memory word as tag bits. These tag bits are to be interpreted by the hardware as information about the data found in the referenced location, or its status with respect to the program or operating system.

The basic concept of tag bits is not new. Almost all computers employ a parity bit which the hardware uses to detect memory failure. In addition, many computers utilize a lock byte which limits access to an area of storage to the operating system or to those who have a key byte that opens the locked area.

Early machines also employed bits which were of special significance to the hardware. The Burroughs B5500 employed a flag bit to inform the hardware that the word at the location addressed possessed a non-numeric value which must be interpreted by the operating system.[3] The Rice Computer (R-1),[4] circa 1959, employed two bits for every word which could be set by the operating system or the programmer. These bits were used in an extensive debugging system wherein tracing, monitoring, or other procedures were carried out when a tagged data word or instruction was encountered.

Today the EAI8400 employs two tag bits for similar purposes. The Telefunken TR4 and TR440[5] employ two tag bits to denote the numeric type of data at an addressed location. The Burroughs B6700[6,7,8] and B7700 which were developed concurrently and independently of the R-2 employ three tag bits to identify types of numeric operands and special information used by the operating system.

What is new about Iliffe's concept is that it represents a rejection of the classical von Neumann machine in favor of something which may be better. In the von Neumann machine program and data are equivalent in the sense that the data which the program operates on may be the program itself. The loop which modifies its own addresses or changes its own instructions is an example of this. While this practice may be permissible in a minicomputer with a single user, it constitutes gross negligence in the case of a multi-user machine where sharing of code and/or data is to be encouraged.

Instead, Iliffe presents a different conjecture. All information which the algorithm needs to know about the data ought to be contained indivisibly in the data itself. For example, an algorithm to perform a *for*-loop on arrays ought to be the same whether the array is of length ten or length 100. Rather than record this information in a variable and use a loop with an index, Iliffe proposes to record the length of the array with the pointer to the array itself, as in Figure 1.

Rather than have several different algorithms for *add integer*, *add floating*, *add double precision*, and *add complex*, he proposes to make the data self-representing. An integer can only be used as an integer, a floating quantity as a floating quantity, etc. This idea is the fundamental difference between the class of machines represented by the BLM, the R-2, and the Burroughs B6700 on the one hand, and by those of more conventional architecture on the other.

Once the fundamental decision has been made to

* This work is supported in part by the Atomic Energy Commission under grants AT-(40-1)-2572 and AT-(40-1)-4061.

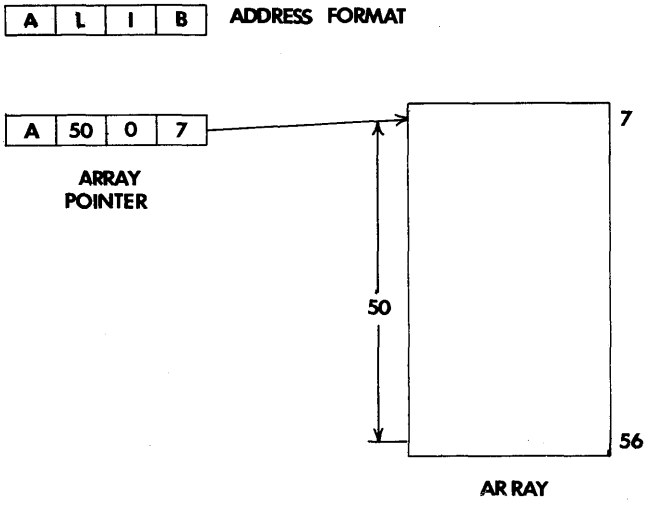

Figure 1—A typical array pointer and its array

adopt this goal in hardware, and the commitment to the use of memory bits for tags has been made, a great many benefits result. One of the most important is that the hardware, once informed of what each piece of data is, can perform run-time checks for consistency of data and algorithms, e g., bounds checking and type conversion.

General structure of the R-2

We now turn to the general structure of the R-2 as shown in Figure 2, which represents only a minor modification from the original design which began in January 1969. The system consists of three major asynchronous subsystems: a memory system of 24K 64-bit words of core memory, a Digital Equipment Corporation PDP-11 I-O controller, and the CPU complex. While the details of the first and second subsystem are of interest, we will not be concerned with them in this paper.

The CPU complex consists of a set of 64 16-bit scratch-pad memory circuits organized as 64-bit registers (designated XØ through X15), whose cycle time is approximately 40 nanoseconds, an arithmetic unit and a CPU. The latter two units are built from RCA ECL integrated circuits with typical delays of from 3 to 5 nanoseconds. This results in a typical add time for two 54-bit floating quantities on the order of 50-200 nanoseconds, and a multiply consisting of additions and shifts typically requiring 3 microseconds.

The address calculator in the CPU is one of the most important features of the system. It functions as an automatic base-bounds calculator and is responsible for the high security of the system programs. For every fetch from memory the address calculator is given four quantities, a base address B of 20 bits, an initial index I of 14 bits, a length L of 14 bits, and the element selector D of 14 bits. In 150 to 200 nanoseconds the calculator performs the following algorithm:

```
Temp:=D-I;
if Temp<0 then Low_error_1;
if Temp≥L then High_error_1;
Actual_address:=B+Temp;
```

I is a number between $-(2^{13}-1)$ and $(2^{13}-1)$ in one's complement form. D has a maximum value of $2^{14}-1$ and this is the largest segment which one can practically use. This should be sufficient for all but the largest one-dimensional arrays of data. The base address B is of sufficient size for any program (or memory) that we can currently foresee, on the order of 1 million words.

Data formats

Before we can discuss the operation of the CPU we must understand the various data formats which the R-2 can deal with. These formats are given in Figure 3. Each word currently consists of 62 bits of information. A two bit field in every word contains a parity bit Z and a write lock bit L. The remaining 60 bits may be divided into four classes of words: numeric words, control words, address words (or partition words), and instruction words. The first three classes contain six bits used for tags as Iliffe suggested. Four bits are used to distinguish types and two may be set by the programmer to generate interrupts. The type codes are listed in Table I.

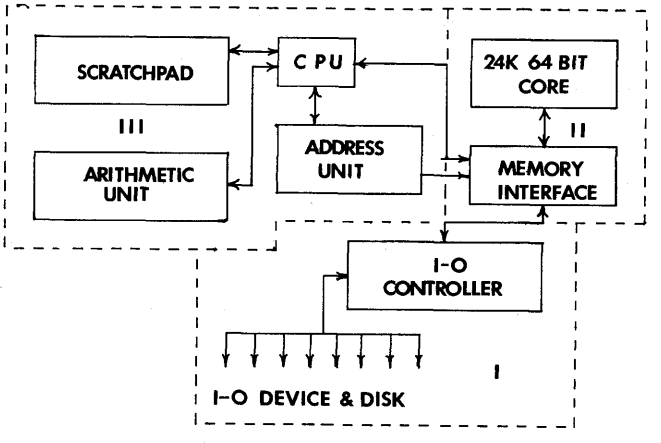

Figure 2—Subsystems of the R-2 Computer

The format of the various numeric types is of little interest. They are the same as those found in the original Rice Computer.[4] See Table I.

We should take greater note of the formats for address words. In addition to containing a base field (B), an initial index field (I) and a length field (L), each address word also contains an indirect reference field which indicates the type of the object in the block described by the address word. Two more bits are used. One indicates whether the block which is described is in core memory or in secondary storage (P). The other indicates whether the block may be relocated (Q).

Control words are an innovation. They are the only method of intersegment communication. They contain a 21-bit halfword pointer to an instruction (R-2 instructions are packed two per word). A mode field of 11 bits indicates the operating state of the computer at the time a jump to a subroutine is made. A two-bit condition code at the time the jump is made is stored in field C. Since the routine may be disk resident, a P field is provided to signal the routine's presence or absence in core memory. A chain field is provided in each control word. This field may be used to link together control words which are on the stack or are in a common linkage segment. Rather than scan storage linearly to find the previous control word, one merely follows the chain of links. Finally, a four-bit mark field may be used to indicate the level of the subroutine or the level of the subtask in the operating system. These marks are especially useful when employed with the control stack for exiting blocks and reestablishing an appropriate environment.

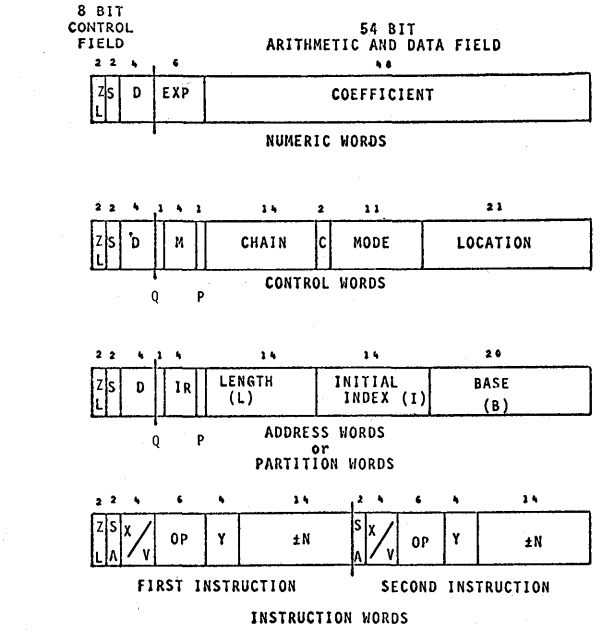

Figure 3—Diagram of R-2 word formats

TABLE I—Data Tag Assignment

TAG Number	*Meaning*
0000	MIXED OR UNTAGGED
0001	(unassigned)
0010	(unassigned)
0011	(unassigned)
0100	REAL, SINGLE PRECISION
0101	54 BIT BINARY STRING OR INTEGER
0110	DOUBLE PRECISION (real, fl. pt.)
0111	COMPLEX (two single precision fl. pt. words)
1000	UNDEFINED FOR NORMAL OPERATIONS
1001	PARTITION WORD
1010	RELATIVE CONTROL WORD
1011	ABSOLUTE CONTROL WORD
1100	RELATIVE ADDRESS, UNCHAINED
1101	ABSOLUTE ADDRESS, UNCHAINED
1110	RELATIVE ADDRESS, CHAINED
1111	ABSOLUTE ADDRESS, CHAINED

Processor facilities

Two major resources are available for use by the instruction unit. The first is the hardware stack and stacking mechanism. The second is the register set (previously described). The stack is maintained in memory and utilizes an address word held in register XØ as a stack pointer and bound. In order to utilize the address calculator in a consistent manner, the top of the stack is the location addressed by the base field of the address word and the bottom of the (accessible) stack is determined using the length field. Special words called partition words are stored by the operating system to denote the absolute beginning and end of the stack region or to point to a continuation of the stack in another segment (see Figure 4). Any word of the accessible stack within $2^{14}-1$ of the top may be accessed by an instruction. Partition words cannot be overwritten by normal stacking and unstacking operations.

Two hardware registers U and R constitute X1, the double length accumulator for arithmetic and logical operations. If X1 is loaded with an item of double precision or complex data, the second word is held in the R register. Special instructions are available to address the R register independently of U.

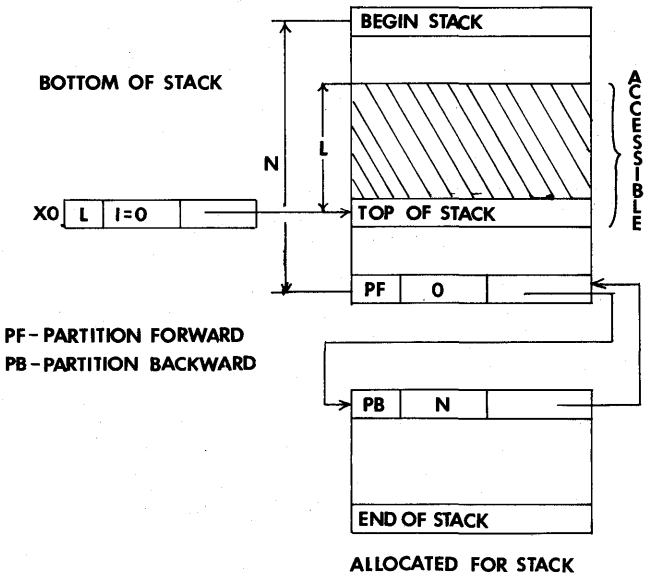

Figure 4—The stack pointer and stack layout

Registers X2 through X15 are implemented with the scratchpad memory integrated circuits mentioned previously, and a reserved, fixed core memory location is associated with each register to hold the second word of double word data when it appears. By recognizing the data tags "complex" and "double precision," the hardware automatically performs the appropriate double word transfers and storage so that no special programming is required. Thus, to the programmer X1 through X15 appear to be a set of general purpose double word registers. Since XØ is always used as the stack pointer, the hardware rejects (by interrupt) any attempt to store a word into XØ unless it is tagged as an address word.

All of the other registers except X15 may be used for temporary storage of numbers, addresses, or control words. X15 must contain an address word which describes where the interrupt vector is located. All interrupts transfer relative to this address. In the event of a catastrophe when there might not be an address word in X15, interrupts transfer to locations relative to absolute address 0.

Instruction formats

Every instruction contains a one bit software tag denoted by S and six bit function code field labeled OP in Figure 3. Each instruction also features an immediate bit labeled A, a numeric offset labeled N which may be plus or minus, and a register field Y. The remaining four bits labeled X/V in Figure 3 are interpreted differently in two classes of instructions. The first features a four bit result register field X. The second uses this field as a variant code for the operation, and for this class (which includes arithmetic and logic) the first operand is implicitly X1, which also contains the result.

Instructions are customarily written in the form S A X OP Y ± N or S A OP Y ± N where the function is to perform X OP $Y_{eff} \rightarrow X$. The first operand X is either stated explicitly or is implicit in the instruction. Y_{eff} depends on the contents of Y, the number N and the immediate bit A.

Rather than describe the operation of the addressing algorithm for Y_{eff} exhaustively by flow diagrams, we will describe instruction sequences for short algorithms. This will show the effect of Iliffe's conjecture, as well as illustrating the machine design.

EXAMPLE PROGRAMS

Example 1—Accessing vector elements

All elements must be accessed through an address word. Address words can be constructed by the operating system in a manner to be described in a following example. Suppose we have an address word in X2 which points to a vector in the manner of Figure 1, and suppose we wish to add the element with index ten of the array to a numeric quantity in X1. We could use the following instruction:

ADD X2.10 // Select the element of X2 indexed by 10 and add to X1. Suppose the initial index of X2, I(X2) is −4, the length 15, and the base address 1000. The sequence of computations would proceed as follows. The computer would check X2 and determine that it contained an address. It would issue B=1000, I=−4, L=15, and D=10 to the address calculator. The calculator would compute D-I (10−(−4)) or 14 which is greater than −1 and less than L. Since the element is within the vector, an address of 1014 would be generated and the element would be brought to the arithmetic unit. If the element is an integer it would be immediately added to the integer in X1 and the condition code would be set to reflect whether the result was less than, greater than, or equal to zero. If the element is real (fixed or floating point fraction), the computer would convert X1 to floating point form and perform the addition. If the element is anything but a numeric type, an exception occurs and an interrupt to a fixed location relative to the address in X15 takes place. If the instruction also invokes the auto-store option,

denoted ADD→X2.10, the result would be stored back into 1014.

Example 2—Accessing array elements

On the R-2 we usually represent arrays in a tree structure. The first index is used to determine an element in a vector composed of address words. The second index is used with this address element to select an element from a second vector which is an address word and so on until the last index which is used to select the desired element. This kind of an array is illustrated in Figure 5. Because of the fact that each address word carries with it the length of the vector it addresses, such arrays may be uniform or nonuniform as desired. They may also be so large that only one vector of data will fit in core memory at any time.

Two different methods of addressing such arrays can be used. These methods are considerably more efficient than that used on the Burroughs 6700 because of the scratchpad registers X2-X14 which are available. One method involves element selection as in the first example. It is generally used when we wish to select only $X_{i,j,k^{th}}$ element of an array rather than to deal with every element. The following sequence of instructions indicates how this may be accomplished.

Suppose X2 contains the address word pointing to the vector and we desire to select $Z_{3,1,5}$ and assume

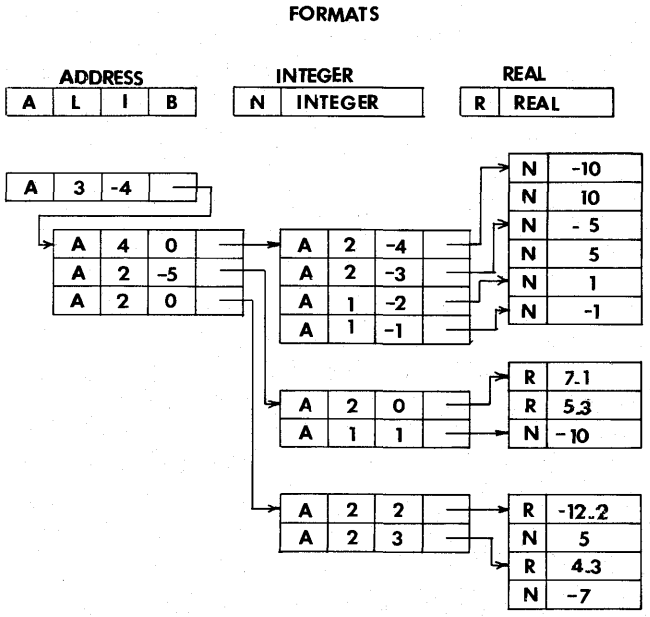

Figure 5—A nonuniform three dimensional array

that the initial indices are 0. Then we could use:

```
X2 DOT  = 3   // Replaces the contents of X2
                 with the third element of
                 first level tree.
X2 DOT  = 1   // Replaces of the contents of
                 X2 with the first element of
                 second level.
X2 MOD  = 5   // Generates the address of
                 Z3,1,5 in X2.
   ADD  → X2  // Adds (X1) to Z3,1,5 and auto-
                 stores back into the array.
```

This set uses the immediate address form of the instruction.

An alternative form might employ a vector of subscripts. Suppose that X3 contains the address word of a vector of subscripts to be applied to Z, i.e. (3, 1, 5). The following sequence indicates how this may be done.

```
X2 DOT  X3.1  // Obtains the third element of
                 first level.
X2 DOT  X3.2  // Obtains the first element of
                 second level.
X2 MOD  X3.3  // Generates a pointer to Z3,1,5.
   ADD  →X2   // Adds (X1) to Z3,1,5. and auto-
                 stores.
```

This sequence of computations is as follows. The first element of X3 (since I(X3) = 0) is selected and brought to the CPU. It has the value 3. If X3 had contained a number originally, 1 would have been added to that number and the resultant would have been used. This value is then used to obtain the third element of Z's first level subtree (denoted $Z_{3,*,*}$). This is left in X2. The next operation obtains an address word which is the first element of Z's second level tree of the third branch ($Z_{3,1,*}$). The next operation indexes this address to make its location field point to the desired element. This element is presumably a number (integer, real, complex, or double precision). It is added to the contents of the U register (X1), and the result placed in X1 and in $Z_{3,1,5}$, thus smoothly implementing the ALGOL 68 statement: Z[3, 1,5]+:=V; where V was the contents of X1. If the value of (X3.1), (X3.2), or (X3.2) had not been a number then an exception would have occurred.

Example 3—Array processing

In some cases vector processing is desired. For this purpose a different kind of access is desired. In this

mode one systematically examines all the elements of an array or vector in turn, and performs some operation on them. For example, to sum a vector pointed to by an address word in register X2, one might use the following sequence of instructions. Assume (I(X2)) equals 0.

```
        X3 LOAD   4     // To sum five elements of
                          vector X2
        X1 LOAD   0     // Initialize X1 to Zero
AGAIN   ADD X2.0        // Add first element of
                          vector
        X2 MOD   =1     // Adjust X2 to point to
                          the next element
        X3 JGE AGAIN    // Continue to CONT if
                          X3 is zero or negative
CONT                    // Decrement and jump to
                          AGAIN otherwise.
```

This sequence loads X3 with the number 4 and X1 with the number 0. The next instruction causes the number pointed to by the B field of X2 to be added to the number in X1. If the length field is set to −1, an exception will occur. The next instruction uses the literal 1 to decrement the L field of the address in X2 and simultaneously increase the B field by 1. If L is less than zero an exception will occur. The next instruction examines X3. If it is a number less than or equal to 0, the next sequential instruction is taken. Otherwise the number is decremented by one and control passes to AGAIN. Using this sequence the first five elements of the vector pointed to by X2 are summed. If there are fewer than six elements in this vector, an exception occurs. If there are exactly five elements in the vector when the program gets to CONT, L(X2) equals zero. An attempt to do X2 MOD=1 again will cause an error exception. Otherwise the new L(X2) is five less than before and the new B(X2) is five more than previously.

A shorter sequence may be used if it is desired to sum all the elements of the vector. Here the programmer need not even know how long the vector is.

```
        X1 LOAD  =0     // Set X1 to zero
AGAIN   ADD X2.0        // Add elements to X1
        X2 JNL AGAIN    // See the discussion below
```

The last instruction checks to see if L(X2) is greater than 0. If it is it performs X2 MOD=1 and transfers control to AGAIN. If it is not, control passess to the next instruction.

As a final example of the power of this approach, assume that we have three arrays A, B^T, and C and that we desire to compute

$$C_{i,k} = \sum_{j=0}^{n} A_{i,j} B_{k,j}{}^{T}.$$

This can be calculated simply in the following routine assuming X2 is an address word pointing to A, X4 is an address word pointng to B^T, and X6 is an address word pointing to C.

```
INIT    X9 COPY  X4    // Copies (X4) to X9
BEGIN   X7 LOAD  X6    // Get ith subtree of
                          C
FIRST   X3 LOAD  X2    // Get ith subtree of
                          A
        X5 LOAD  X4    // Get kth subtree of
                          B^T
           ZERO  X7.0  // Put zero in C_{i,k}
SECOND  X1 LOAD  X3.0  // Get A_{i,j}
           MUL   X5.0  // Multiply by B_{k,j}^T
           ADD→  X7.0  // Add and autostore
                          to C_{i,k}
        X5 MOD   =1    // Next consider
                          B_{k,j+1}^T
        X3 JNL SECOND  // Next consider
                          A_{i,j+1}
CONT                   // if no more j con-
                          tinue here
        X4 MOD   =1    // Next consider
                          B_{k+1,j}^T
        X7 JNL FIRST   // Consider C_{i,k+1} if
                          any left
        X6 MOD   =1    // Consider C_{i+1,k}
        X4 COPY  X9    // Start over with
                          B_{0,0}
        X2 JNL BEGIN   // Consider A_{i+1,j} if
                          any left.
```

This routine destroys pointers located in X2, X4, and X6. The steps

```
           MUL   X5.0
        X5 MOD=1
```

may be combined into MUL! X5 which uses a variant option for the arithmetic operation code to modify the address word in X5 after the element has been fetched.

Example 5—Use of the stack

The stack may be used for intermediate storage in the following manner. It is first necessary to get the operand in a scratchpad register. Suppose we wish to

stack X5. Then we write X5 STORE XØ. The contents of X5 is pushed onto the stack. On the other hand, X5 STORE XØ.7 stores the contents of X5 in the seventh location of the stack without altering other elements. If there are less than seven elements on the accessible stack an exception occurs. The top element of the stack may be changed without pushing by the use of X5 STORE XØ.1.

Elements are removed from the stack in an analogous manner. For example, ADD XØ adds the top element of the stack to the accumulator and pops the top element. ADD XØ.1 adds the top element of the stack but does not pop it; ADD XØ.7 adds the seventh element of the stack without affecting the stack.

This stack arrangement affords many conveniences in arithmetic operations. An example of this is the instruction save and fetch. Take as an example X2 SVF X3.12. This instruction gets the twelfth element of the array pointed to by X3, after saving the old value of X2 on the stack, and places the new value in X2. In compiling, the instruction X1 SVF Xi.N will occur frequently. Intermediate results can be saved on the stack for later use. Alternatively, when the value is to be used many times they can be stored in a register using COPY.

The stack is also useful in control actions. The instruction JUMP AND SET MARK, e.g., 4 JSM LABEL, causes a control word to be made up pointing to the next sequential instruction. This control word contains the current mode, the current condition code, and the mark specified by the JSM instruction, in this case 4. The chain field is loaded with the current value of L(XØ). The resulting control word is pushed on the stack. XØ is then updated to reflect a new stack regime. L(XØ) is set to 0 and B is set to the location that is one less than that occupied by the control word. This stack regime is completely disconnected from the prior one; there is no way save through the registers or memory constants to reach any of the members of the previous stack regime (see Figure 6). One can return to the previous environment by the use of the return instruction: RET 4. The previous control word is found by examining the address B(XØ)+L(XØ)+1. This address contains the last control word or a partition word pointing to another section of the stack regime or a partition word marking the absolute beginning of the stack. If a control word occupies this position, B(XØ) is set to point to this address. The chain field of the control word is copied into L(XØ), the mode field to the mode register, the condition code to the condition code register, and the 21 bit address to the program counter. If the mark field is less than the literal used with the return instruction, the computer resumes processing. Otherwise the process

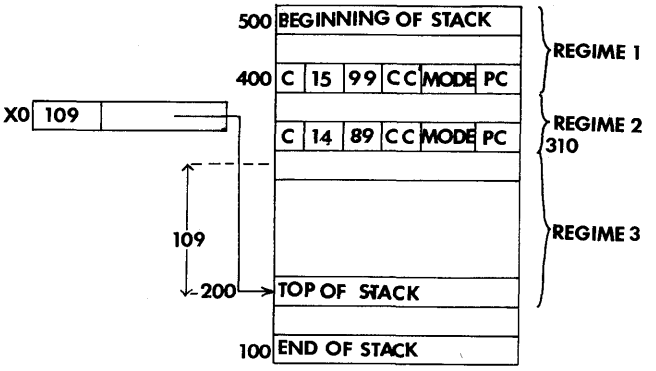

Figure 6—The use of the stack in programming systems

described above is repeated until a partition word marking the absolute beginning of stack is encountered. If the beginning of stack is encountered, an error exception will result. This form of return has the advantage that it is efficient and can be used to return several levels in a block structured language.

Relative addressing

In an earlier section we stated that all addressing was done via an address word. In fact this is not quite true. Addresses may be relative to the program counter or to the location in which the address word or control word is stored. One may then jump minus five halfword instructions. This feature was available on the R-1 in 1959 and made possible the development of very efficient relocatable code. It is frequently found on machines today. One can also access fullword data in the same manner so that constants can be stored with the code.

In the same way relative codewords and addresswords can be used to address other quantities. The address of the location in which the address or control word is stored must be determinable by the computer. An offset relative to this address is used to point to the jump location or to the data. This feature makes possible relocation of large blocks of data in a very efficient manner and minimizes the number of address words which contain absolute addresses. This is important because it drastically reduces overhead in the reorganization of storage. Only a few locations need to be modified to relocate all programs and their data.

Chained Addressing

A second kind of addressing is also provided. Chained addressing provides for efficient parameter passing

mechanisms, particularly of the call by reference variety. The addressing algorithm is modified to include indirection. When a chained address or control word is encountered the machine causes an indirect reference through the address or control word to the next quantity. A counter is employed to assure that chaining beyond 32 levels is not allowed. Special instructions are employed to defeat chaining for loading and storing. These instructions allow complete control of the addressing mechanism.

Miscellaneous instructions

The R-2 is designed to be used with compilers rather than assemblers. It has many useful miscellaneous instructions including reverse divide, variants of pre- and post-complemented addition and logical operations, and integer and floating multiplication and division. Various instructions allow extraction and replacement of the predefined data and instruction fields. A large number of shift, bit count, and test instructions complement the arithmetic and logical set of instructions, thus facilitating the development of operating systems and compilers.

Ramifications of tagged architecture in the R-2

The use of tagged architecture has many ramifications. In a future paper we will discuss them more fully. Here we will comment on a few of interest to those who write or use compilers and operating systems or who must debug programs.

Compilers can be made simple and more efficient. Since tags indicate what each numeric quantity is and since the hardware will correctly perform the appropriate operation on all legal combinations of data, the compiler need not deal with semantic operations referring to basic types of identifiers. This is now handled conveniently at run-time. The problem of temporary storage is largely solved by the stack whose implementation is greatly facilitated by the address calculator and the tagged address type. The tagged registers allow simple manipulation and mechanization of vector and array operations and allow dynamic variables much more freedom than do previous machines. They also permit the optimum calculation of expressions of type *address* and type *numeric*. Finally, tagged addresses and numbers greatly simplify the problems of run-time systems for use with particular compilers; note that even undefined quantities have a distinct representation.

Operating system design is facilitated by tagging. The difference between a label and an address can be determined at run-time. This means that one cannot jump through an address word or a number but only to approved points in a subroutine via a control word. Attempting to do otherwise produces an interrupt.

Secondly, addresses can only be manipulated by means of a special set of instructions. The more powerful instructions may be denied to the user and he may be given the use of MOD, TAG, and LIM. TAG is an instruction with an immediate operand. The compiler monitors all TAG instructions assuring that no unauthorized user can construct illegal address or control words. Any user may employ MOD or LIM. They modify an address to point to a subset of the elements in an addressed space. Since an unprivileged user may never generate an address outside the initial space to which he has been given access, protection of user programs is enhanced if not insured. This protection mechanism appears to be exactly what is required for recursively defined operating systems.

The design and use of debugging systems is greatly simplified. A program can be written to dump core memory using the type of each datum. This means that complex, double precision, floating, integer, and undefined types can be used to interpret the data contained in the cells. This can be used in the analysis of dumps. Addresses and labels are also distinct in this scheme; the fact that they are not in other systems has been a recurring problem for those who must debug.

Dynamic debugging is also easily implemented. By the use of symbolic locations, relative locations, or absolute locations, data or instructions may be tagged with software tags. Whenever such tagged data is encountered, an interrupt occurs. The programmer may supply his own programs to analyze or monitor the data values which are encountered. The software tag in each of the instructions may also be set to cause interrupts related to tracing any or all control actions. Again the user may write a program to analyze the results. Finally since the tag bits may be set either at compile time by the compiler, or arbitrarily at run-time by the user through the operating system, code which is verified as being correct need not be recompiled, thus simplifying and expediting the process of debugging.

SUMMARY

This paper has reported the state of the Rice Research Computer in its development. It has emphasized the features of the R-2 which arise from accepting the principle of tagged architecture. For a particular implementation, we have shown by example the power of tagged architecture in application to compilers, operating systems, and debugging systems.

ACKNOWLEDGMENTS

I would like to acknowledge the work of the designers of R-2 including John Iliffe, I.C.L. (London, England), Walter Orvedahl (Colorado State University), Dr. Sigsby Rusk (Rice University), Douglass DeLong (Graduate Student, Rice University), Dr. Ernest Sibert (Syracuse University), and Dr. Robert C. Minnick (Rice University). Also, I owe thanks to Dr. J. Robert Jump and the graduate and undergraduate students at the Rice University Laboratory for Computer Science and Engineering who have made and are making the machine a reality.

REFERENCES

1 J K ILIFFE
Basic machine principles
American Elsevier New York 1968

2 J K ILIFFE
Elements of BLM
Computer Journal 12 August 1969 pp 251-258

3 *A narrative description of the Burroughs B5500 disk file master control program*
Burroughs Corporation Detroit Michigan Revised October 1966

4 *Basic machine operation*
Rice University Computer Project Houston Texas January 1962

5 *TR440 eigenschaften des RD441*
AEG Telefunken Manual DBS 180 0470 Konstanz Germany March 1970 (German)

6 *Burroughs B6500 information processing systems reference manual*
Burroughs Corporation Detroit Michigan 1969

7 J G CLEARY
Process handling on the Burroughs B6500
Proceedings of the Fourth Australian Computer Conference Griffin Press Adelaide South Australia 1969 pp 231-239

8 E I ORGANICK J G CLEARY
A data structure model of the B6700 computer system
SIGPLAN Symposium on Data Structures and Program Languages ACM New York New York 1971 pp 83-145