Hardware Support for Program Debuggers in a Paged Virtual Memory

David Abramson
Department of Computer Science
Monash University
Clayton 3168

John Rosenberg
Department of Computer Science
Monash University
Clayton 3168

1. INTRODUCTION

This paper proposes a hardware unit which assists interactive debuggers in detecting breakpoint and watchpoint conditions. In the past debugger packages have used inefficient and expensive techniques for determining when a breakpoint has been reached. Most of these solutions have relied on large, inefficient and restrictive software routines, with very little support from the main processor. This paper examines the techniques used by conventional debuggers and highlights their disadvantages. We describe the support which could be provided by extra hardware attached to a virtual memory unit, and then develop an implementation of this hardware. Finally, we compare this implementation with the alternatives. The paper does not discuss the user interface provided by the debugger software. However, the primitives provided should allow most user functions to be supported.

2. INTERACTIVE DEBUGGERS

An interactive debugger assists users in debugging software by observing the data and control flow of an executing program. Among other functions, it allows a program to be stepped a selected number of machine instructions (or lines of source code) at a time and for the contents of variables to be examined and modified. These attributes are usually provided by code and data breakpoints, a single stepping facility and data watchpoints. Alternative terms for these functions may be found in [13]. Code breakpoint interrupts are caused if an instruction is fetched from a specific location. Data breakpoint interrupts are caused if a selected variable is addressed. A watchpoint allows a user to detect when a location has been modified.

3. CLASSICAL SOLUTIONS

A number of techniques have been used to detect code and data breakpoints, and watchpoints.

The simplest method of implementing code breakpoints is to insert a trap (or software interrupt) instruction into the program at the breakpoint [1]. When control reaches this point an interrupt is generated and the

debugger is activated. This has a number of disadvantages. First, because the trap instruction is actually inserted into the instruction stream, the debugger must be able to modify the code in main memory. Since most modern computers protect code from being altered, this operation may be difficult if not impossible. Second, because the code itself is modified, it cannot be shared between processes. Third, this method cannot be used to detect data breakpoints and watchpoints.

In another software technique the compiler inserts a subroutine call (or a special instruction) after each high level language statement. This subroutine checks if a breakpoint has been reached, and causes a software interrupt. The scheme has a number of disadvantages. First, it is extremely expensive in time. The program must execute a call instruction for every line of source code. Second, it cannot be used to detect data breakpoints or watchpoints. Third, it is only applicable to high level language programs. If a similar technique was used for an assembler program, then each machine instruction would require a subroutine call, an intolerable overhead in space and time. Fourth, the program must be recompiled with a special debug option.

Some processors have provided a small amount of hardware to assist the debugger software. In one such system the hardware generates an interrupt after each instruction has been executed [1]. This allows the program control flow to be monitored until a breakpoint is reached. The technique is extremely expensive, as each instruction which is executed causes the debugger to be activated. In addition it is unable to detect data breakpoints and watchpoints.

Another scheme uses an address trap register (or base and limit register [11]), which is loaded with a breakpoint address. When the selected location is referenced the debugger is alerted. This scheme allows both code and data breakpoints to be detected with very little effort. Unfortunately, only a limited number of the registers is provided (usually one), which complicates the debugger software significantly.

A similar level of support can be provided by using the paging hardware of processors with a virtual memory [2]. If the access rights associated with a page frame are set to no access, then each reference to the page will cause an interrupt. When coupled with some single stepping hardware (such as that described above) this scheme allows code breakpoints to be detected. Unlike the other schemes it can also be used to detect data breakpoints. If the page in which a variable resides is set to no access, then any reference to that page will cause an interrupt. The debugger must then determine whether the reference was for the correct variable. Watchpoints may be detected by setting the access rights of the page to read only. The only major disadvantage of this scheme is that it is still extremely expensive, as the debugger is activated each time a protected page is addressed. This method has the advantage that it can detect references to more than a single memory location, and may be used to monitor references to arrays and strings.

Watchpoints and data breakpoints may be detected in machines which require a variable to be addressed via an indirect address (or descriptor), provided that an interrupt is generated when a selected descriptor is referenced [3]. Such descriptors may be marked with a special tag field. This scheme has the disadvantage that it is usually necessary to recompile the program with a special debugger option. The method is not particularly effective for monitoring assembler programs as the descriptors may not be completely invisible.

Another method of detecting data breakpoints and watchpoints is to inspect the contents of the variable after each instruction (or memory

operation) [4]. This technique is not common because it is extremely expensive in time, and is typically only used to detect references to simple variables.

If the processor has a tagged main memory, it is possible to reserve a tag field which causes an interrupt when the location is addressed [5]. This scheme can then detect code and data breakpoints, and also watchpoints, to either a specific location or a range of locations. Unfortunately, the tag fields are usually protected, and cannot be directly modified by the debugger software. Thus, special instructions must be provided for setting and clearing breakpoint tags (which is awkward in a code section). Also, the code cannot be shared amongst many processes, as they will all use the same breakpoint tags. Few processors have been built which use a tagged memory, and thus the technique has a very limited applicability.

From this examination we have shown that very few processors provide direct support for the detection of breakpoints. Most use techniques which are expensive and inefficient, and some require the compiler to insert special debugger code into the instruction stream. In this case the program must be specially compiled with a debug option. Very few debuggers allow breakpoints and watchpoints to be set over a range of addresses, thus is hard to detect references to part of a large structure, such as an array.

4. BREAKPOINT HARDWARE

Hardware support for the detection of breakpoint interrupts can be provided by a high speed associative memory device which holds a list of the breakpoint addresses (and possibly process numbers). Each time a processor address is generated, the associative memory is searched. If an address is found, then an interrupt is sent to the processor and the memory reference is aborted. If a breakpoint is not detected, then the memory reference may proceed. This hardware can be used to detect accesses to both data and code, and also may be used to detect watchpoints.

A number of different types of associative memory have been devised. Ideally, the memory should be large enough to hold all of the breakpoints which are current for all active processes. Thus, no extra work in required when a process change is executed. Unfortunately large true parallel memories (in which all of the cells are associatively matched at the same time) are not available. A small memory has the disadvantage that it can only hold a limited number of addresses, and thus must be reloaded on a context change. Larger memories can be built, but slower searching algorithms are used, such as a linear scan. If a slow searching algorithm is used the execution speed of the processor may be significantly reduced.

The use of an associative memory to hold all current breakpoints has a number of general disadvantages. First, all of the breakpoints must be checked before a reference can proceed to memory. Thus, the entire table must be searched regardless of the number of breakpoints that a particular process has set. Second, watchpoint conditions may be difficult to detect if the processor uses data registers to hold temporary results. Because of these registers the main memory location may never be addressed. Thus, some additional hardware must be included to detect accesses to the internal processor registers. This may in the form of a tag field attached to a register as suggested by Johnson [9]. An alternative is to instruct the compiler not to implement any register optimization if the program is being debugged, and always update a variable in memory. This is not a serious restriction as register optimization is only used to improve program performance, which is not particularly relevant while a program is undergoing debugging. Finally, it is not possible to build an associative memory which is large enough to hold all of the current breakpoints,

especially if a breakpoint is set on a range of addresses. Moreover, associative memories which can match a key within a range of addresses are even harder to build than those which retrieve a particular address pattern.

The hardware which is described in this paper is based around an associative technique, but alleviates many of these disadvantages. It has been proposed for the MONADS II computer, which is briefly described in the next section.

5. The MONADS II System

The MONADS II computer [6] uses a large paged virtual memory to hold all of its computational and permanent data. Virtual addresses are unique across the system, and a process may address any part of the virtual memory providing it has the authority. Unfortunately, conventional page tables are an inappropriate method of address translation because of the size of the virtual addresses (which is 47 bits in the MONADS II/2 processor). Consequently, special hardware is used to translate virtual page numbers into main memory page numbers.

5.1. The MONADS II Address Translator

Because the MONADS II virtual addresses are larger than used in many processors, a special hardware address translator was developed [7], [12]. This device accepts a virtual page number and either translates it into a main memory page number, or generates a page fault interrupt. If a page fault is detected the operating system searches page tables for the secondary memory address of the page [8] and copies the page into main memory.

The unit consists of a sparsely occupied hash table held in very fast addressable memory, a hashing unit and a fast comparator, as shown in Figure 1. When an address is generated, the hasher is used to find the start of a chain of addresses which hash to the same cell. As each cell is fetched the comparator determines whether the correct address has been located. Cells of the hash table may be chained together by a chain field. The hardware searches an entire list until either the page number is found, or the end of chain is detected.

This address translator can be modified so that certain addresses within a page will cause a breakpoint interrupt if they are referenced.

6. NEW BREAKPOINT HARDWARE

The new breakpoint hardware consists of an additional fast memory, which is used to hold a list of breakpoint addresses (or range of addresses), as shown in Figure 2, and three comparators. Also, an additional field is inserted in each cell of the hash table. This new link field is used to hold an index value into the breakpoint memory, as shown in Figure 3.

A breakpoint address is composed of two fields, a virtual page number and a within page displacement. When a breakpoint is set, the within page displacement of the location and the process number are placed in a cell of the hardware breakpoint memory. If an access to a range of addresses is to be detected, then the start displacement and the end displacement of the range are placed in an entry of the table. If an access to a single location is to cause an interrupt, then the address is loaded into both the start and the end displacement fields. The cell of the hash table which holds the mapping information for the page is linked to the cell in the breakpoint memory by the new link field. If a page of virtual memory has more than one breakpoint current, they may be chained together by using the

link field in the breakpoint memory. If the range of addresses includes more than one page, then a separate range must be created for each of the pages.

Each time a virtual address is translated into a main memory address, the link field of the hash table is inspected. If it is zero, then the memory reference proceeds without delay. If it is not zero, it is used as an index into the breakpoint memory. At this stage the memory reference is suspended while a chain of cells in the breakpoint table is searched for the displacement (terminated by a zero link). If (a) the displacement is greater than or equal to a start displacement, (b) the displacement is less than or equal to an end displacement, and (c) the process number for the cell is the same as the current process number, then a breakpoint interrupt is caused. A 'breakpoint inhibited' state is set so that the next time this address is generated an interrupt will not be detected. This sequence of operations is summarized in Figure 4.

Each entry of the breakpoint table also holds three flags. The wild flag disables the process number check. This allows any process which generates a particular virtual address to cause a breakpoint interrupt, rather than a specific process. If the valid bit is clear then the displacement field of the cell is ignored, and the next cell in the chain is fetched. This allows the software to remove a breakpoint temporarily without the need to delete the cell from the breakpoint table. If the watch point flag is set, then an interrupt is only generated if a memory write operation was requested. If a watchpoint is detected, then the data present on the data bus is saved in a special register. This allows the debugger to show the contents of a location before and after the instruction.

This hardware has a number of advantages.

(i) Only memory references to pages which have current breakpoints are delayed. If a true associative memory is used then all memory references are suspended while the breakpoint list is checked.

(ii) Since it is unlikely that a page will contain more than a few breakpoints, the list which is searched is quite small. Thus, very little time is wasted if a breakpoint is not detected.

(iii) More than one process may set breakpoints in the same page of virtual memory, even at the same location. All breakpoints within a page are simply chained into a linked list.

(iv) Only part of the virtual address is actually compared with the contents of the breakpoint cells, thus a breakpoint may be detected very quickly. If the entire address were compared a much larger comparator would be required (three times the size), which is much slower.

(v) The breakpoint memory can be large enough to hold all of the current breakpoints in the system. Thus, the hardware does not affect the speed of a process change.

(vi) The size of the breakpoint table is independent of the virtual address size (apart from the displacement field), and only depends on the maximum number of breakpoints allowed.

(vii) The hardware can detect when an address is within a particular range, which is not possible with most conventional associative memories. This allows data breakpoints and watchpoints to be set on large structures, e.g. records, arrays and string variables.

(viii) No special compiler flags are necessary. Providing the debugger has access to the symbol table for the program, then a breakpoint may be set at any address, instruction or source line.

(ix) A section of code may be shared between processes, even though one or more processes may have breakpoints set. This allows a shared utility to be debugged without the need to recompile a special unshared version of the same code.

(x) The breakpoint hardware may be used to trap any reference to the virtual memory. Because MONADS II uses a uniform virtual memory, this allows references to all forms of data (e.g. stack data, permanent file data, etc) to be monitored.
(xi) Finally, the implementation is quite inexpensive. All that is required is fast memory for the breakpoint table, some extra memory for the hash table, and three comparators. Also, the debugger is only alerted when a breakpoint has been reached, unlike many other solutions.

7. TIMING CONSIDERATIONS

Because the breakpoint table is implemented by very fast addressable memory (similar to that of the hash table) it is possible to search for a displacement very quickly. To improve the search speed further, the comparison operation can be performed in parallel with the fetch of the next breakpoint table entry in the chain. Thus, if c is the time taken to perform a comparison operation, n is the number of breakpoints set per page, and r is the time taken to read an entry from the breakpoint table, the worst case time that a memory reference is delayed (e.g. if a breakpoint is not detected) is approximately:

```
time-delayed = if hash-table-link <> 0 then
                   r + c + (n-1) * max(r,c)
               else 0;
```

Using Schottky series logic $r \sim 45$ nsec; $c \sim 20$ nsec;

Thus: time-delayed = 45 + 20 + 45(n-1)
= 45n + 20 nsec.

This time is significantly less than one memory cycle (which is between 300 nsec and 800 nsec), and is not a large overhead. Moreover, the delay only applies to pages which have breakpoints currently set. A further optimization may be to overlap the breakpoint search with part of the main memory reference. In the case of a read operation the reference can be aborted if a breakpoint is found. In the case of a write operation, the breakpoint list must be exhausted before the write is executed. These optimizations may mean that no memory reference is ever delayed.

8. BREAKPOINT SOFTWARE

The software support required by the proposed hardware is not particularly complex and could be provided by a set of routines in the kernel of the operating system. These routines could then be called by the high level debugger utility. Two basic functions must be provided. First, there must be procedures to insert and remove breakpoints and watchpoints. These routines must allocate space in the hardware breakpoint table and set up the chains. They must also ensure that a process only removes breakpoints which it has set up. The second function that the kernel must provide is a mechanism to allow the debug software to be entered after a breakpoint is encountered. This is a form of interrupt routine and is very much dependent on the structure of the operating system. It will not be discussed further here.

8.1. Kernel Primitives

The four proposed primitives are shown in Figure 5. The first function sets a breakpoint or watchpoint over a specified range of addresses. Privileged users may set the wild option so that the process number is ignored. The function returns a logical breakpoint number which will be used

for reporting an occurrence of the breakpoint as well as for permanently deleting it using the procedure clear_breakpoint. The last two procedures may be used to temporarily disable a breakpoint (using the valid bit) without deleting it.

8.2. Software Tables

Some proposed tables required by the software are shown in Figures 6 and 7. The logical breakpoint table has one entry for each breakpoint created by the set-breakpoint function. Each entry contains the number of the process that created the breakpoint (for protection reasons) and a pointer into the software breakpoint table. The latter points at the first entry in the software breakpoint table related to this logical breakpoint. It should be noted that if the breakpoint area spans a page boundary then more than one hardware breakpoint will have to be set. These related hardware breakpoints are linked via the software breakpoint table, which has the same number of entries as the hardware breakpoint table. The last related entry is indicated by the end-of-chain bit. The link field in this entry contains a pointer back to the logical breakpoint table. This is necessary so that the logical breakpoint number may be reported on the occurrence of a breakpoint.

Apart from the field linking all hardware breakpoints associated with one logical breakpoint, each entry in the software breakpoint table has two other fields. The first indicates the type of the second which can either be an index into the software breakpoint table, the hash table or the paged-out table.

The final table is the paged-out table. This has one entry for each virtual page containing at least one breakpoint, but not in main memory. It is addressed associatively and each entry contains the virtual page number and a pointer to the first breakpoint in this page in the hardware breakpoint table.

In addition to the above, the software would have to maintain data for the allocation of entries in the tables. This could easily be done using bit sets [10] and will not be described here.

8.3. Software Algorithms

There are two basic software algorithms involved. These are insertion/deletion of breakpoints and paging of pages containing breakpoints.

8.3.1. Insertion/Deletion of Breakpoints

Insertion and deletion of breakpoints is quite straightforward. On insertion the logical breakpoint must be divided into one hardware breakpoint for each virtual page involved. Each such breakpoint is then inserted into a free entry in the hardware breakpoint table. The corresponding entries in the software table are linked and pointed to by the logical breakpoint table. Finally the virtual page number is hashed to find the entry in the address translation unit (if present). Assuming the page is in main memory, this new breakpoint is added into the chain of existing breakpoints in this page (which may be null) and the back pointers in the software breakpoint table are initialised. This process is reversed for deletion, making use of the backwards pointers.

8.3.2. Paging of Breakpoints

The above discussion has assumed that the page containing a breakpoint is present in main memory. This is not necessarily the case and the hardware

and software described does allow breakpoints to be maintained in pages currently held in secondary memory. In this case the logical and hardware breakpoint tables are processed in exactly the manner described above. When a page is swapped out to secondary storage the link field in the address translation unit is examined. If there are any breakpoints (i.e. not null) then an entry is initialised in the paged-out table. This contains the virtual page number and the pointer to the start of the breakpoint chain. This process can also be executed if a breakpoint is created in a non resident page, without the need to swap the page into mainstore. Thus the entire breakpoint structure can be maintained even though the page is not present. This allows breakpoints to created and deleted at addresses currently paged-out without the need to load the pages into main memory.

This structure does incur an overhead. On every page-fault, after the requested page has been read into memory, the paged-out table must be searched to see if there are any breakpoints in this page. If there are then the pointer to the chain of breakpoints is copied into the link field of the address translation unit and removed from the paged-out table. However, it not expected that the paged-out table would be very long because breakpoints tend to be localised. Furthermore, it is advisable to keep pages which contain breakpoints in memory as they are likely to be addressed in the immediate future. Thus the cost of the scheme should not be very high.

9. MAPPING THE NEW HARDWARE ONTO DEBUGGER FUNCTIONS

The debugger functions we considered in section 2 were code and data breakpoints, data watchpoints and a single stepping provision. The hardware proposed in section 6 directly supports the setting and detection of breakpoints and watchpoints. Unlike many other solutions, only one mechanism is provided for detecting breakpoints in either code or data. A program may be stepped one instruction at a time by setting a breakpoint at the next instruction to be executed. When the breakpoint is encountered, it may be removed and replaced at the next instruction. This operation can easily be performed by the debugger software, and may be hidden from the user. Using the breakpoint hardware in this manner not only removes the need for extra single stepping hardware, but supports languages in which the high level instructions generate more than one machine code instruction.

10. COMPARISON WITH OTHER SYSTEMS

Because the hardware proposed in section 6 directly supports breakpoints and watchpoints, it is far more efficient than the conventional solutions. The amount of extra hardware required is quite small, and is relatively inexpensive. Unlike many other systems, the debugger is only alerted when a breakpoint has been encountered, saving a large amount of processor time.

11. GENERALIZATION

It may appear that the hardware proposed in section 6 will only operate with the MONADS II address translation hardware, which is different from most address translation schemes. However, the breakpoint table may be coupled to most conventional processors which support a paged virtual memory. In such systems, virtual addresses are usually translated into main memory addresses by page tables. This operation is made efficient by using small cache memories to hold the most frequently used address translation entries. If each cell of these cache memories is modified to include a link field, in the same manner as the hash table, then the breakpoint hardware may be used as in the MONADS II computer. Each time an address translation entry in the cache is updated, the link field must also be modified.

Some difficulty may be experienced in processors which use code and data caches. If the cache associates on a main memory address, then the scheme will operate correctly. Breakpoints will be validated when the virtual addresses are translated. However, if the cache associates on virtual address the page tables may never be consulted. Consequently, some breakpoints may not be detected. This problem is easily avoided by not placing copies of locations which are being monitored in the cache.

12. CONCLUSION

This paper has demonstrated that most of the conventional implementations of program debuggers are largely inefficient. The new hardware described is simple in design, inexpensive and far more efficient than other solutions. Further, it can be added to existing computers which have a paged virtual memory facility without difficulty.

ACKNOWLEDGEMENTS

The authors wish to thank Les Keedy and Chris Wallace for helping to improve the clarity of this paper.

REFERENCES

[1] Digital Equipment Corporation "The PDP 11/34 Processor Handbook"

[2] Digital Equipment Corp. (1979) "VAX 11 Architecture Handbook"

[3] Keedy J.L. (1977) "An Outline of the ICL2900 Series System Architecture", Australian Computer Journal, 9, 2, pp. 53-62.

[4] The UNIX Programmer's manual, SDB, 7th Edition, Virtual VAX 11 version, Computer Science Division, Dept. of Electrical Engineering and Computer Science, University of Berkeley, California.

[5] Myers, G.J. (1978a) "Advances in Computer Architecture", New York: Wiley-Interscience, 1978.

[6] Abramson, D.A. "The MONADS Series II Computer System", Proc. of 6th Australian Computer Science Conference, Australian Computer Science Communications, Vol 5, No. 1, Feb 1983, pp 1-10.

[7] Abramson, D. (1981) "Hardware Management of a Large Virtual Memory", Proc. 4th Australian Computer Science Conference, Brisbane (Australian Computer Science Communications 3, 1, pp. 1-13).

[8] Rosenberg, J. and Keedy, J.L. (1981) "Software Management of a Large Virtual Memory", Proc. 4th Australian Computer Science Conference, Brisbane, pp. 173-181.

[9] Johnson, M.S. (1982) "Some Requirements for Architectural Support of Software Debugging", Proceedings of Symposium on Architectural Support for Programming Languages and Operating Systems, ACM Publication 556811, Palo Alto, California, March 1982., pp 140-148.

[10] Keedy, J.L., Ramamohanarao, K. and Rosenberg, J. "On Implementing Semaphores with Sets", The Computer Journal, 22, 2, May, 1979, pp. 146-150.

[11] Case, R.P. and Padegs, A. (1978) "Architecture of the IBM System/370", Comm. ACM, 21:1, pp 73-96.

[12] Abramson, D.A. (1982) "Computer Hardware to Support Capability Based Addressing in a Large Virtual Memory", Ph.D. Thesis, Monash University.

[13] Johnson, M.S. (1982) "A Software Debugging Glossary", ACM Sigplan Notices, Vol 17, No. 2., Feb 1982.

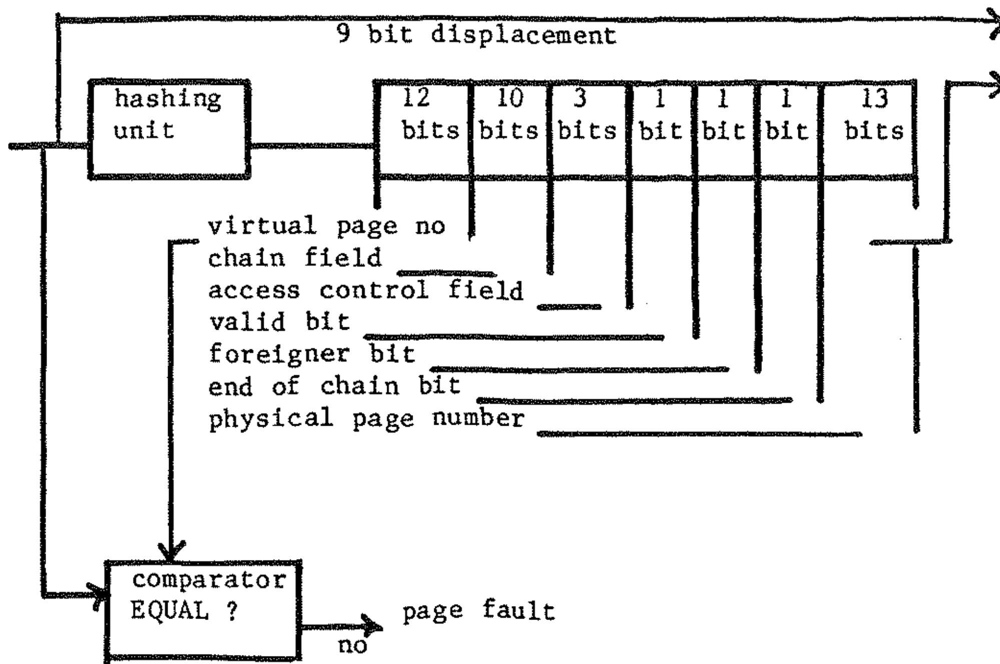

Figure 1 - the hash table format

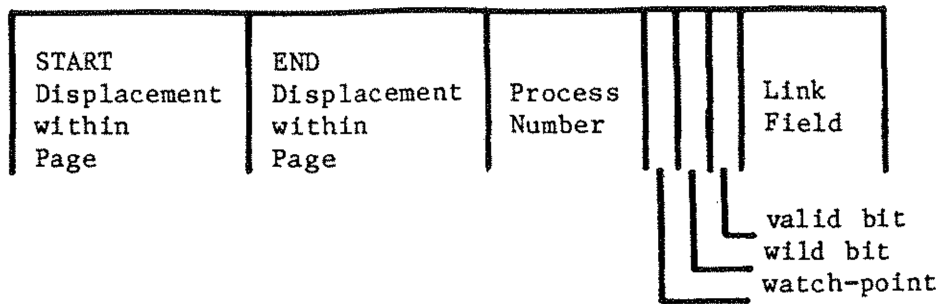

Figure 2 - the hardware breakpoint table format

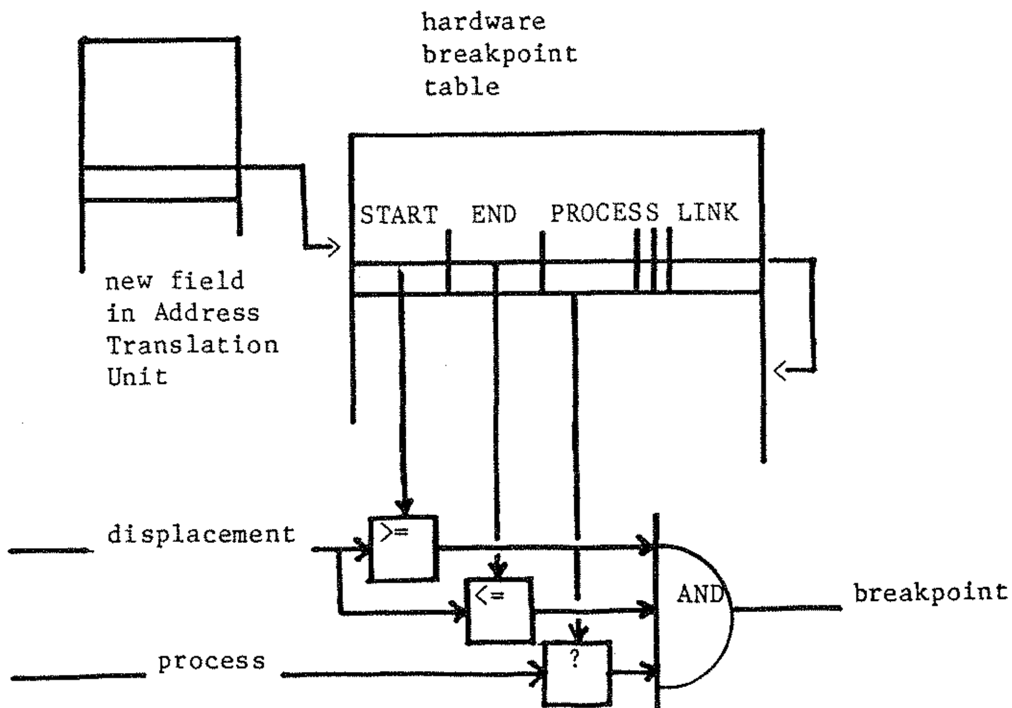

Figure 3 - the breakpoint hardware

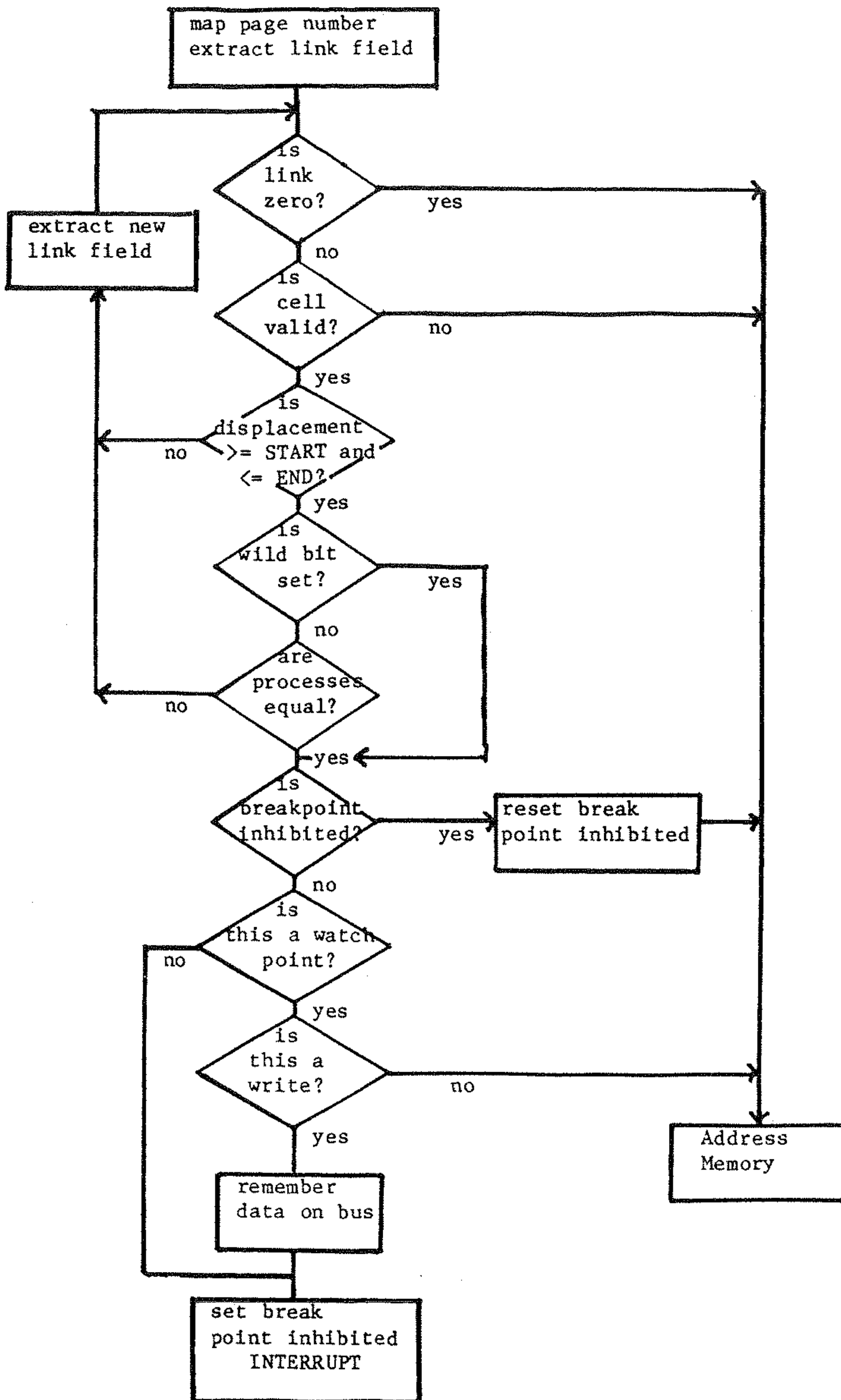

Figure 4 - the breakpoint cycle

```
function set_breakpoint      (start_address : address;
                              length        : integer;
                              type          : (break,watch);
                              wild          : boolean) : integer;

procedure clear_breakpoint   (breakpoint_number : integer);

procedure disable_breakpoint (breakpoint_number : integer);

procedure enable_breakpoint  (breakpoint_number : integer);
```

Figure 5 - The Kernel primitives.

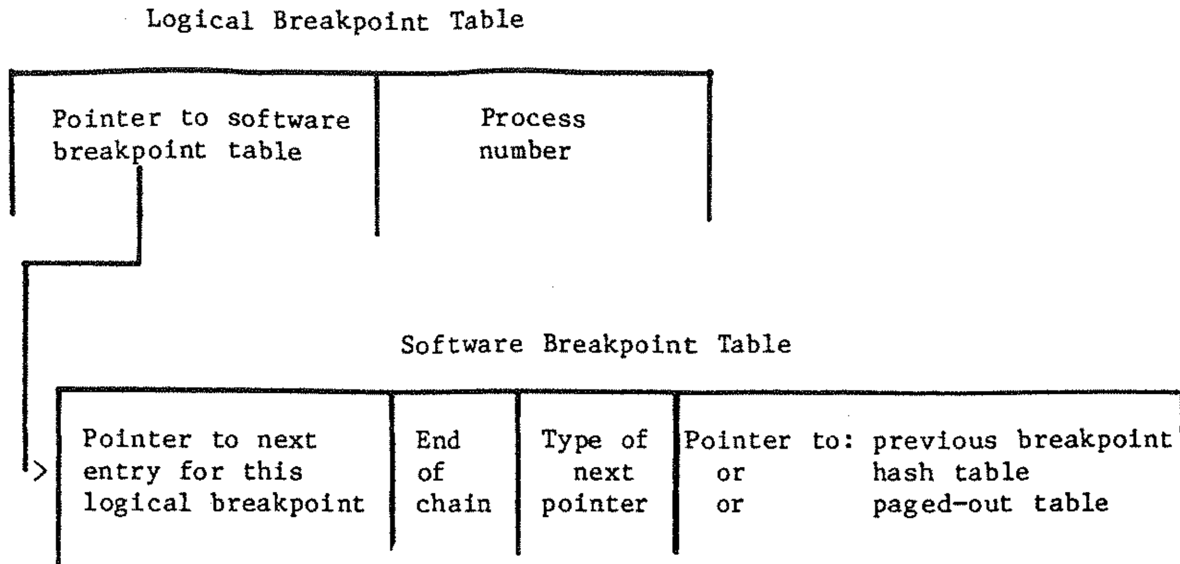

Figure 6 - The logical and software breakpoint tables

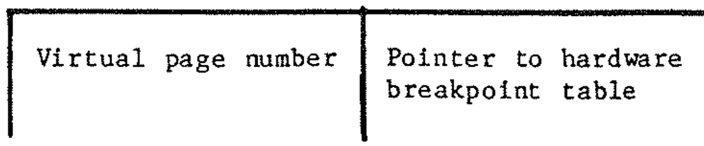

Figure 7 - The paged-out table

COMPUTER ARCHITECTURE NEWS

A Quarterly Publication of the Special Interest Group on Computer Architecture

VOL. 11, NO. 2, JUNE 1983

Contents

SCANNING THE ISSUE 1

LETTERS 2

CONTRIBUTIONS

Hardware Support for Program Debuggers in a Paged Virtual Memory, by David Abramson and John Rosenberg 8

Word Length of a Computer Architecture: Definitions and Applications, by Dennis J. Frailey 20

BOOK REVIEWS, edited by M. David Freedman

Computer Design, by Glen G. Langdon, Jr., reviewed by Lee A. Hollaar 27

RECENT TECHNICAL REPORTS, compiled by H. J. Siegel 29

SIGARCH BUSINESS 31

ANNOUNCEMENTS 34