Numerical Mathematics

Binary Summation

R. J. Walker
Cornell University, Ithaca, New York

Key Words and Phrases: summation, binary summation, floating-point addition, round-off errors
CR Categories: 5.11

In discussing his binary summation method [1] Linz mentions two defects: "It is more difficult to program than the standard method, and it is difficult to use unless all numbers are available at the start of the summation." A defect not mentioned is the extra storage space (approximately N words) required for the S_{ij}. The last two of these three defects can be removed by proper combination of partial sums. In the adjacent PL/I program, the only storage needed is for the M-component vector T, where $M \geqq [\log_2 N]$. N must be even, and B is a number not expressible as a sum of $A(I)$'s, e.g. $B = -1$ if $A(I) \geq 0$, or $B = 1E + 75$ for reasonably sized $A(I)$.

This program is written for pre-stored $A(I)$ but it is easily adapted to other cases by suitable change of statement Y. If N is a power of 2 (as in Romberg quadrature, for instance) the last three lines can be omitted.

```
   T = B;
   DO I = 1 TO N − 1 BY 2;
Y: S = A(I) + A(I + 1);
   DO J = 1 TO M;
      IF T(J) = B THEN GO TO X;
      S = S + T(J); T(J) = B; END;
X: T(J) = S; END;
   S = 0;
   DO J = 1 TO N;
      IF T(J)¬ = B THEN S = S + T(J); END;
```

References

1. Linz, Peter. Accurate floating-point summation. *Comm. ACM 13*, 6 (June 1970), 361–362.

Computer Systems

Interrupt Driven Programming

Marvin Zelkowitz*
Cornell University,† Ithaca, New York

Key Words and Phrases: interrupts, supervisors, monitors, debugging, parallel processing, associative memories, microprogramming
CR Categories: 3.51, 4.32, 4.42

This note is an extension of the ideas expressed by Morgan [1]. He suggests a new form of interrupt which he proposes to use to control the execution of a program, in his case a file management system called DPL [2]. Simply stated, he has attached to every subroutine a Boolean expression which, if true, would cause that subroutine to be executed. Since his implementation is via software, a relatively time-consuming check must be made whenever some condition in the program changes (e.g. some variable has a value changed). A rather simple hardware addition to existing machines can easily implement this interrupt. The following is such a proposal.

The specific details of implementation are not crucial to this discussion. What is important is the benefit of hardware implementation. The implementation is described in terms of an IBM 360 environment [3] but is certainly not limited to it. Essentially the implementation involves the addition of a small associate memory to the CPU which is $12N$ bytes long. Each entry is 12 bytes wide, and the number of entries is up to the designer of the computer (sixteen will probably be sufficient). An associative memory is a memory where all entries are searched in parallel for a given value. The associated functional value is retrieved if there was a match between the memory and the sought value. The format for each entry in this CHECK memory is pictured in Figure 1.

The CHECK memory operates as follows:

(a) For every store or fetch cycle of the main memory, the address of the location fetched and the ADDRESS fields in the CHECK memory are compared. This can be done in parallel with the actual fetching or storing of data, and thus should not slow down main memory. If there is a match then step (b) is executed.

(b) The contents of that fetched core location with the VALUE field in the CHECK memory are compared. If the relationship of memory contents to VALUE field is the same as CC, then an interrupt occurs and control transfers to LOCATION. As an initial implementation, since most 360's fetch information a word or a double-

* Present address: Department of Computer Science, University of Maryland, College Park, Maryland. † Department of Computer Science.

Fig. 1. Format of Check Memory

CC	ADDRESS	VALUE	LOCATION
1	3	4	4 BYTES

CC = Condition Code (for IBM 360)
= 02 (greater than)
= 04 (less than)
= 08 (equality)
ADDRESS = Address field
VALUE = Test Value
LOCATION = Location to begin execution if interrupt

Fig. 2. Interrupt if less than or equal to 0

0C	010000	00000000	00012340

word at a time, it would seem reasonable to have this comparison done for a fullword of data. While this would hinder the user slightly, it should be very easy to install in present-day hardware. Fullword checking also allows the system to be used as an address stop since addressing on the 360 is on a fullword basis. Multiple-precision variables can be trapped by a simple two part process. The first four bytes of the variable would be placed in the CHECK memory. At LOCATION, the remaining bytes could be checked. Since LOCATION would be activated only if the high order bytes agree, this should not occur with too much frequency.

Since step (a) is done in parallel with main memory accessing, it should not slow the machine. Step (b) is only executed for a few special addresses; thus if several additional machine cycles must be added, it should not be a significant factor in total system performance.

Entries would be added to this memory by an instruction such as INSERT CHECK-OPERAND. CHECK-OPERAND would be a 12 byte field. The first four would be used to scan to table to see if a previous entry exists for that address. The 360 condition code could be set after this INSERT instruction to the following:

0 if the entry was added to the memory,
1 if it was added, but there was a previous entry for it,
3 if the CHECK memory was full.

As an example in the use of this memory, consider the problem of going to location 12340 if location 10000 becomes nonpositive. The number 8 is the condition code for "equality" and 4 is the condition code for "less than"; thus (in hexadecimal) $C = 8 + 4$ is the condition code for "less than or equal to." The CHECK memory entry for this condition is pictured in Figure 2.

Thus one now has the power to do much more checking than is now possible by hardware without slowing the program by adding additional software checks. Some of the possible uses of this memory are:

1. The concept of event sequenced programming mentioned at the beginning of this article is achieved almost without execution time penalty. While it may not be possible to automatically check complicated Boolean expressions (e.g. $A < B$ or $C > D$), this CHECK memory does enable the programmer to test against many useful conditions.
2. Statements of the form ON CONDITION ($X < 0$) CALL ERROR-ROUTINE could be added to PL/I (or other higher level languages). These would have a simple implementation at almost no cost in speed of execution. It would greatly enhance the diagnostic capabilities of present-day higher level languages.
3. One could easily envision an operating system where each parallel process is kept in synchronization with other processes by this check memory. For example a process I may be waiting for a process J to access a buffer X. If X is placed in this CHECK memory, with I as the LOCATION field, I will be activated when X is accesses by process J. While this description has been naively simplified, it could be expanded into the design of a real system.

Thus it is seen that many varied uses are possible with this new interrupt. While only a few have been outlined here, a few moments' thought will reveal other practical uses.

Even though there appears to be a need for this interrupt, no present-day computer has anything like it. The G-20 machine [4] of the early 1960's did have a flag attached to each word which, if set and that word was accessed (as instruction or data), would cause an interrupt. This form of address stop would also be an effective means of implementing the above proposals, but would not be as versatile (e.g. an address stop could be achieved with the CHECK memory by setting the condition code to 0F).

The most serious flaw with the G-20 scheme, in the light of present-day architecture, is that it took one additional bit of memory for each address in the machine. In the 360 that would mean approximately one-eighth more memory and a totally different byte structure, whereas the CHECK memory could probably be implemented within the present microprogramming framework of the 360.

In conclusion, this note attempts to show the utility of this small associative memory under a wide selection of applications. It is by no means definitive in its description. Its only goal is to show the desirability of implementing such a check at the hardware level. If DPL or other future systems should prove successful, then hardware solutions will be sought to the problems raised by these languages. This proposal is one simple way to carry this out.

Received October 1970, revised December 1970

References

1. Morgan, H. L. An interrupt based organization for management information systems. *Comm. ACM 13*, 12 (Dec., 1970), 734–739.
2. Morgan, H. L. DPL: A language for instruction in contemporary data processing concepts. Tech. Rept. No. 53, Dep. of Operations Res., Cornell U., Ithaca, N. Y., 1968
3. System 360 Principles of operation. IBM manual A22-6826-7.
4. Bendix G-20 Central processor machine language. BET-10601-3. Bendix Corp., 1961.