The IBM System/370 vector architecture

by W. Buchholz

Discussed is the instruction-set architecture of the IBM System/370 vector facility, a compatible extension of the System/370 architecture. Both the base system, which is a general-purpose System/370 processor, and the optional vector facility employ a register type of organization. Data formats are the same, arithmetic operations produce exactly the same results, arithmetic exceptions are handled in the same way, and instructions are precisely interruptible for page faults and other causes in the same manner as those of the base system. This approach permits substantially increased performance on vectorizable programs with only a modest increase in hardware and software, while retaining the ability to run existing nonvector programs unchanged.

The architecture of the IBM System/370 vector facility[1] is a compatible extension of the System/370 architecture.[2,3] Use of the facility can substantially increase performance for applications in which a great deal of the time of the central processing unit (CPU) is spent doing arithmetic on vectors.[4]

Much of the numerical data for such computation-intensive applications has the form of an array. Vector processing can take advantage of the order inherent in array data by treating multidimensional arrays as sets of vectors (one-dimensional arrays). Vector-arithmetic operations may increase performance over loops of scalar arithmetic instructions in the following three ways:

- The fixed and predetermined structure of vector data permits housekeeping instructions inside the loop to be replaced by faster internal (hardware or microcoded) machine operations.
- Data-access and arithmetic operations on several successive vector elements can proceed concurrently by overlapping such operations in a pipelined design or by performing multiple-element operations in parallel.
- The use of vector registers for intermediate results avoids additional storage references.

Vector processors have been available for a number of years in two forms. One type of vector processor is a separately programmed special-purpose unit that attaches to standard commercial computers as a peripheral device; examples are the IBM 3838[5] and the Floating-Point Systems FPS 164[6,7] array processors. The other type of vector processor is integrated into a supercomputer design, as in the Cray series[8-10] or the CDC Cyber-200 series and its predecessors.[8,11] The IBM vector facility combines aspects of both types. It is designed as an optional feature which can be added to a general-purpose System/370 processor, the IBM 3090, operating in either the System/370 mode[12] or the System/370-XA (extended architecture) mode.[13]

When the vector facility is provided, it is viewed as an integral part of the CPU. Its instructions,

registers, and other functions are completely integrated into the System/370 architecture in the following ways:

- Regular System/370 instructions can be used for all scalar operations.
- Arithmetic operations on individual vector elements produce exactly the same result as do the corresponding System/370 scalar instructions.
- Vector instructions are interruptible, and their execution can be resumed from the point of interruption after appropriate action has been taken, in a manner compatible with the System/370 program-interruption scheme.
- Arithmetic exceptions are the same as, or extensions of, exceptions for the scalar arithmetic instructions of System/370, and similar fix-up routines can be used.
- Vector data may reside in virtual storage, with page faults being handled in a standard manner.

The thorough integration of the IBM vector architecture has a number of important consequences. Support of the vector facility is provided by appropriate additions to current operating systems. Existing application programs, language compilers, and other software can be run unchanged. Eligible programs can be modified at any time to take advantage of the vector instructions, with the system remaining productive doing scalar jobs whenever there are no vector applications ready to run.

Integrating the vector facility into a general-purpose System/370 processor also requires the adoption of certain restrictions of the System/370 scalar architecture. The vector architecture gives the same appearance of sequential instruction execution as the scalar architecture; this is extended to vector-element operations, which each vector instruction appears to perform sequentially. Only one exception at a time is allowed to cause an interruption. At the point of interruption, all preceding element operations have been completed successfully, and any subsequent operations that have already been performed are discarded as though they had not yet occurred.

Structure of the vector facility

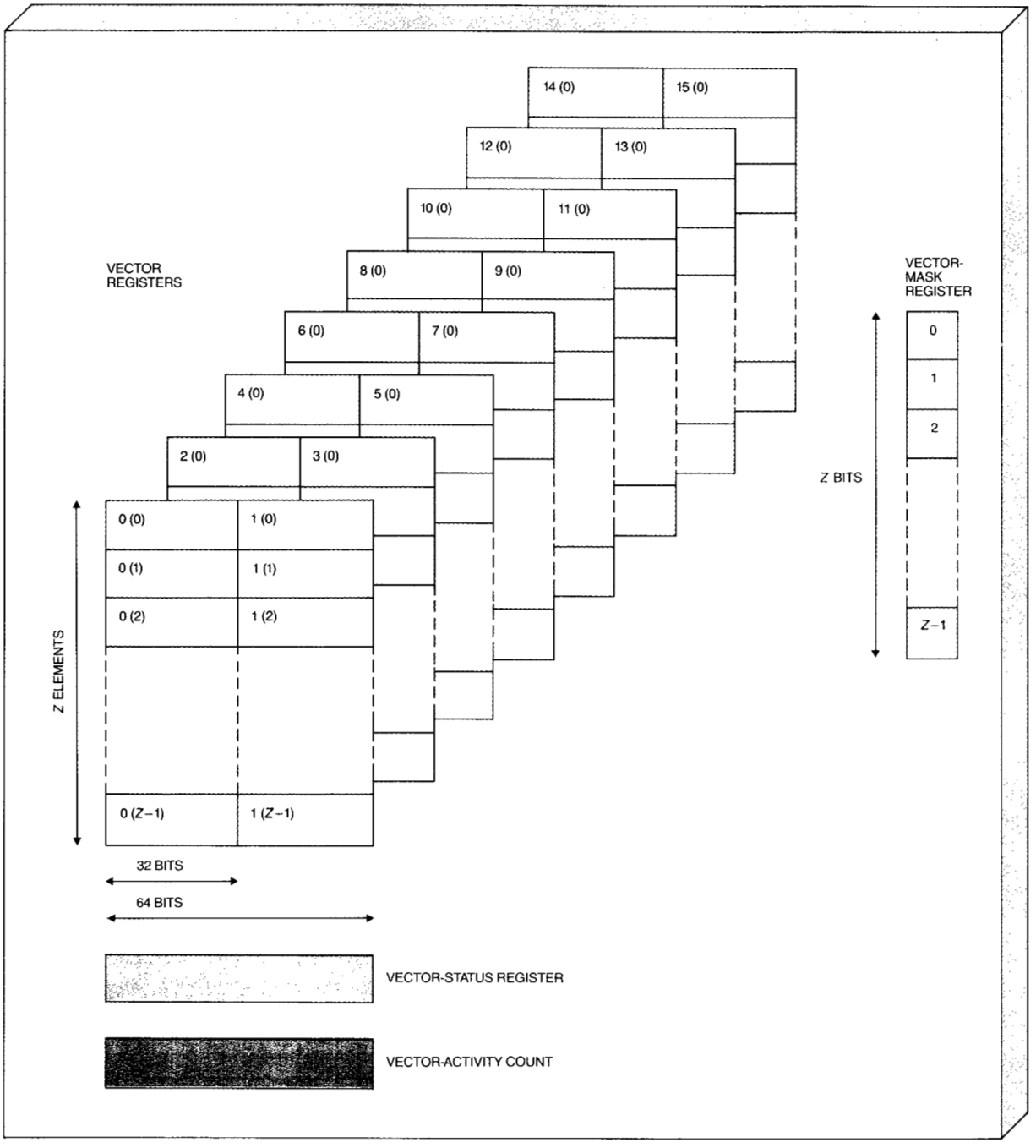
The vector facility may be viewed simply as an addition to the instruction-execution part of the base machine, which contains the additional registers and the specialized circuits to perform arithmetic and logic on a stream of data at high speed. There are also lesser additions to the instruction-handling part and other controls of the machine, but these merge readily into the existing control structure.

Registers. The vector facility has 16 vector registers, which are used as temporary containers for vector operands and results, as are the four floating-point registers and 16 general registers for scalar data. A vector register may contain a certain number of consecutive elements of a vector, the number depending on the model. Each element of a vector register is 32 bits wide. A pair of even-odd vector registers is used to hold 64-bit vector elements. Unlike the scalar floating-point and general registers, vector registers may be used interchangeably for floating-point or fixed-point data. Thus, a single vector register may hold a vector operand consisting of either floating-point numbers in the 32-bit short format or 32-bit binary integers. A vector-register pair may contain a floating-point vector in the 64-bit long format or the 64-bit product vector that results from multiplying two binary-integer vectors.

Vectors may be of any length that will fit in storage, from one element on up. Vectors longer than the number of elements that one vector register or register pair can hold are processed in sections. Thus, the length of a vector register is referred to as the *section size*. The vector architecture allows the section size to be any power of 2 between 8 and 512. The choice for a particular model is made by the designers as a performance-cost tradeoff. For the vector facility of the IBM 3090, the section size was chosen to be 128. The architecture provides special instructions to simplify the processing of long vectors, one section at a time, in a manner that is independent of the actual section size. This sectioning technique is discussed later in this paper.

Figure 1 shows the vector registers, as well as other registers of the vector facility that are available to the program. The *vector-mask register* contains mask bits that may be used to select which elements in the vector registers are to be processed. The *vector-status register* holds certain control fields, such as the vector count that determines

Figure 1 Registers of the Vector Facility, where *Z* is the model-dependent section size



how many elements in the vector registers are to be processed. The *vector-activity count* keeps track of the time spent executing vector instructions.

Instruction formats. Like their scalar counterparts, vector instructions can operate either on operands in registers or directly on operands in storage. The

resultant vector always goes to a vector register (not necessarily a register that contains one of the operands), except that the result of a vector comparison is placed in the vector-mask register. The arithmetic and logical vector operations are available in up to four formats, which differ as to the source of the operands, as follows:

- VST format—one operand in storage; the other operand, if any, in a vector register
- VV format—all operands in vector registers
- QST format—one operand in storage; the other operand in a scalar register

Processing a vector in storage directly can reduce the number of separate vector-load instructions.

- QV format—one operand in a scalar register; the other operand, if any, in a vector register

For the arithmetic operations, the four formats are repeated for long and short floating-point vectors and, with some omissions (such as division), for binary-integer vectors. Thus addition, subtraction, multiplication, and comparison operations are each available in twelve versions; division is available in eight.

The QST and QV formats use the contents of a scalar (floating-point or general) register as one of the operand sources. The scalar operand is treated as though it were a vector of the same length as the vector operand and having all elements equal to the contents of the scalar register.

Processing a vector in storage directly and the result, not having to replace either operand, are capabilities that can reduce the number of separate vector-load instructions. This saving is nontrivial, because loading a vector may take about the same time as performing an arithmetic operation.

Vectors in storage. The primary focus of the vector instruction set is on vectors of floating-point numbers in the long format, where each element consists of 64 bits. For those applications that do not require the full precision, storage space can be saved by using the short floating-point format of 32-bit elements. Additional instructions perform operations on vectors of binary integers and logical data, but these are intended primarily to support the floating-point vector operations. Vectors of half-word (16-bit) binary integers may also be loaded and stored; these are expanded to the 32-bit integer format while in vector registers. The data formats are fully compatible with those of the scalar architecture.

The address of a vector in storage is the address of the first byte of the first element to be processed. The address is contained in one of the 16 general registers of the base machine. As execution of a vector instruction progresses from one element to the next, the address in the general register is incremented correspondingly. Upon completion, the general register points to the next element of the vector to be processed, so as to be ready if the program repeats the instruction for another vector section.

Successive elements of a vector in storage may be in adjacent storage locations (contiguous vector), or they may be separated by one or more element positions. The number of element positions in storage needed to advance from one vector element to the next is called the *stride* of the vector. Contiguous vectors have a stride of one. Each instruction which accesses a vector in storage can specify a stride; a contiguous vector is the default.

Consider, for example, an m-by-n matrix (i.e., a two-dimensional array) in storage. The address of a column or row is the address of its first element. If the matrix is stored in column order (i.e., the FORTRAN convention), each column is a contiguous vector of length m and stride one. Each row then is a noncontiguous vector of length n and stride m. For a row to be processed in reverse order, the instruction specifies the storage location of the last element and a stride of $-m$.

As a second example of a square n-by-n matrix stored in column order, a column is a contiguous vector, a row is a vector of stride n, the main diagonal is a vector of stride $n + 1$, and the sec-

ondary diagonal has a stride $n - 1$. All of these vectors have a length n.

Converting a stride to an address increment in storage amounts to multiplication by the element width in bytes. (Because the element widths are powers of 2, multiplication merely consists of a left shift of the binary stride.) Thus, a stride of 10 for a floating-point vector in the long format requires an address increment of 80 bytes. Much of this address arithmetic is done automatically by the machine.

Vectors in vector registers. When a vector is loaded into a vector register, its elements occupy consecutive register positions. The number of elements loaded is the *vector count*. A single vector count, which can be any integer from zero up to the section size, governs the processing of vectors in all vector registers.

Another quantity, the *vector interruption index*, indicates the element in the vector register or registers currently being processed. Element positions in a vector register are numbered from zero to $Z - 1$, where Z is the section size. During instruction execution, the vector interruption index normally starts at zero, advances until it reaches the vector count, and is reset to zero as the instruction is completed. If the vector instruction is interrupted for any reason, the vector interruption index marks the point that has been reached (hence its name), and execution resumes from that point if the instruction is reissued. This automatic operation of the vector interruption index usually requires no program intervention.

The vector count, vector interruption index, and certain other information which must be saved at the time of an interruption are located in the vector-status register shown in Figure 1.

Instruction execution

Vector sectioning. Sectioning refers to a technique for processing vectors of any length in sections, where the section size is the number of element positions provided in a vector register, except for the last section, which may be shorter than the section size of the machine. Sectioning is best explained by the use of two simple examples presented in assembler language.

The first example performs **C** = **A** + **B**, where **A**, **B**, and **C** are three contiguous vectors of length N. Figure 2 illustrates schematically the manner in which vectors are handled, one section of Z elements at a time, where Z is the section size of the model (i.e., 128 elements for the IBM 3090). The corresponding instructions for computing **C** = **A** + **B** are shown in Figure 3. The vectors in this example consist of floating-point numbers in the long format. Instructions that have mnemonics starting with the letter V belong to the vector facility; all others are regular System/370 instructions. (The numbers in parentheses are for use only in the description.)

Symbols G0, G1, G2, and G3 refer to the arbitrarily chosen general registers 0−3. V0 is vector register 0, although any even-numbered vector register would do. Identification letters are prefixed to the register numbers for clarity only.

Instructions (1) to (4) load the vector length N into general register 0 and the starting addresses of the three vectors into general registers 1 to 3. Instructions (5) to (9) form the sectioning loop. Each traversal of that loop processes one section of each of the three vectors.

Instruction (5) is a special instruction LOAD VECTOR COUNT AND UPDATE (VLVCU). It loads the vector count with the lesser of the vector length (specified by the instruction to be in general register 0) or the section size Z (supplied by the machine). Then the instruction reduces the contents of the general register by the number just loaded into the vector count. Finally, the VLVCU instruction sets the condition code of the machine to indicate whether the new general-register contents are zero (i.e., the last or only section has been processed) or greater than zero (i.e., more sections are to follow).

Instructions (6) to (8) are long floating-point vector instructions. VLD loads a section of vector **A** from storage into vector register 0. VAD adds to that section a section of vector **B** from storage and returns the result to vector register 0. The first V0 of instruction (7) refers to the result register and the next V0 to the register location of one of the operands. (The two vector registers are the same here, but they could have been different.) VSTD stores the result into a section of **C**. All three vector instructions process one vector section at a

Figure 2 Sectioning example, C=A+B, where Z is the model-dependent section size

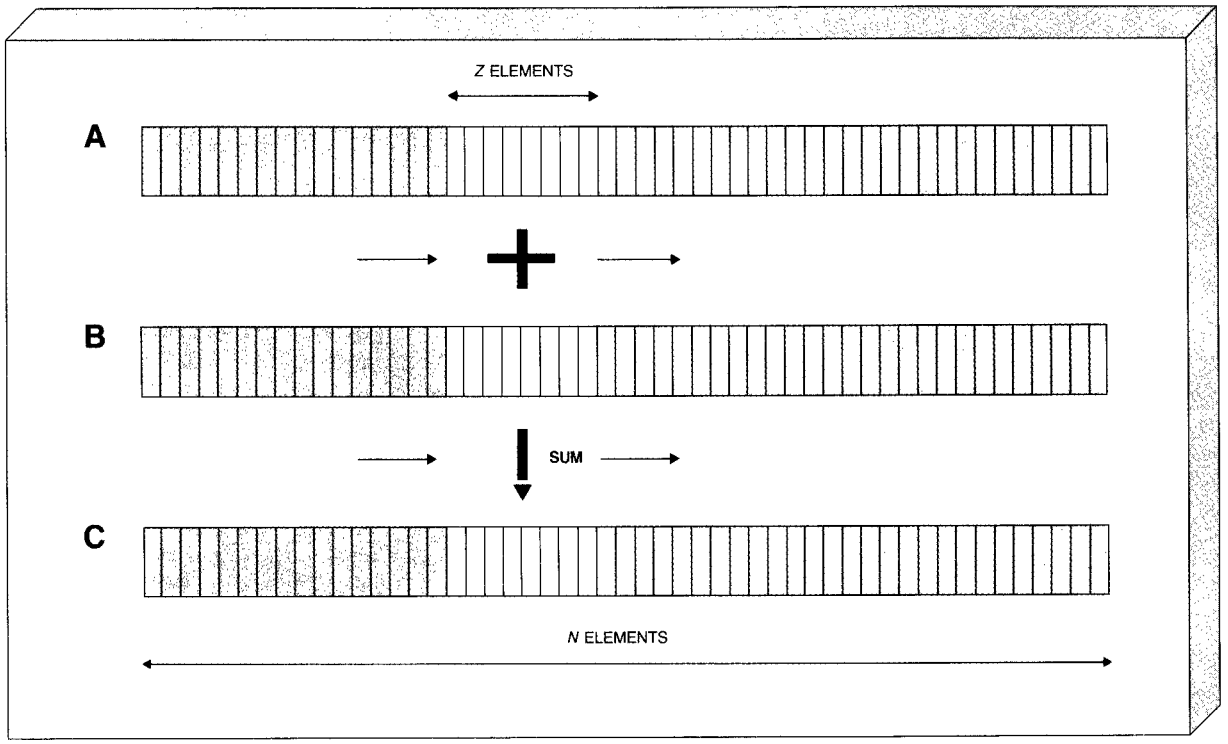


Figure 3 Instructions for the example C=A+B

```
         L     G0,N          (1)
         LA    G1,A          (2)
         LA    G2,B          (3)
         LA    G3,C          (4)
LOOP     VLVCU G0            (5)
         VLD   V0,G1         (6)
         VAD   V0,V0,G2      (7)
         VSTD  V0,G3         (8)
         BC    2,LOOP        (9)
```

time, the length of the current section being determined by the vector count as set by instruction (5). Normally, the vector count is set to Z, the section size of the machine, except for the last (or only) vector section, for which the vector count is set to the remaining number of elements.

Instruction (9) is a BRANCH ON CONDITION (BC) instruction, which tests the condition code set by VLVCU. If the condition code is 2, general register 0 is still greater than zero, and the instruction branches back to instruction (5) to set the vector count for the next section. Each of the vector instructions (6) to (8) advances the vector address in its general register to the first element of the next section, so that processing can continue directly from one section to the next. If the condition code is not 2, general register 0 is now zero, there are no more sections to be processed, and the program continues with the instruction following (9).

The example in Figure 4 evaluates the vector expression $\mathbf{B} = (S - \mathbf{A}) * \mathbf{B}$, where S is a floating-point scalar, **A** is a contiguous vector, and **B** is a vector of stride T. The vector length is N. All floating-point numbers are in the short format for this example.

The third and fourth instructions place the address of vector **B** in two general registers, 2 and 3, one register being used for fetching **B** as an operand and the other for storing the result back into **B**. Two copies of the address are used this time, because each of two vector instructions separately updates the address of **B**. Instruction (5) loads stride T into general register 4. Instruction (6) loads scalar S into floating-point register 0.

The sectioning loop in this example consists of instructions (7) to (11). The VSES instruction (8) performs a scalar-vector subtraction. This instruction subtracts each element of the current section of vector **A** in storage from scalar S and places the difference in vector register 5. VME in instruction (9) multiplies this difference by a section of vector **B** in storage. The stride register for the noncontiguous vector **B** is specified in parentheses following the address register. The stride cannot be in general register 0, because a zero in the stride-register field of the instruction is defined instead to indicate a contiguous vector. [The assembler permits this zero field to be omitted, as in instruction (8).] The product of the two short vectors, which is in the long format, is placed in the vector-register pair consisting of registers 2 and 3, but only the left half is stored by the VSTE instruction (10).

These sectioning loops are quite general and work with vectors of any length N and with any section size. *Short vectors*, which are less than or equal to the section size, are processed in a single pass through the loop. If N is a variable that happens to be zero (or even negative), VLVCU sets the vector count to zero, and the vector instructions inside the loop are executed once but without processing any vector elements. The section size does not appear explicitly in these examples because they are independent of the actual section size.

Interruptible vector instructions. All long-running vector instructions, that is, those which can operate on multiple vector elements, can be interrupted during execution due to a variety of causes. One cause of interruptions is the recognition of one of the standard arithmetic exceptions, such as overflow or division by zero. There is a new exception for unnormalized floating-point numbers, which, for performance reasons, are not permitted as operands of vector multiplication and division.[14] Page faults and other exceptions may result from

Figure 4 Instructions for the example $B=(S-A)\cdot B$

```
        L     G0,N            ( 1)
        LA    G1,A            ( 2)
        LA    G2,B            ( 3)
        LR    G3,G2           ( 4)
        L     G4,T            ( 5)
        LE    F0,S            ( 6)
LOOP    VLVCU G0              ( 7)
        VSES  V5,F0,G1        ( 8)
        VME   V2,V5,G2(G4)    ( 9)
        VSTE  V2,G3(G4)       (10)
        BC    2,LOOP          (11)
```

attempts during execution to access vector operands in virtual storage. Finally, there are asynchronous interruptions from input/output or other external sources that the machine may allow during execution of long instructions to provide greater responsiveness.

Some of these interruptions may require the job to be terminated, others provide an opportunity for the program to fix up the result and then resume normal operation, and still others are expected to be handled transparently by the operating system in such a way that the application continues to run as though nothing had happened other than a short delay.

The updated storage address in the general register designated by the instruction and the vector interruption index are key to the interruptibility of most of the vector instructions. The general register and vector interruption index serve as placeholders in storage and in the vector registers, respectively, so that resuming the program by reexecuting the vector instruction causes the instruction to continue from the point of interruption. Multiple interruptions during the execution of a single instruction are handled by maintaining the appearance of sequential execution, instruction by instruction and element by element. At the point of interruption, all preceding instructions and all preceding elements of the current instruction have been completed, and any operations that may have been started on elements or instructions beyond this point are nullified and appear as though they had never occurred. A few instructions are interruptible in a different manner, but the same principles apply.

Saving and restoring of vector status. If more than one program must share the use of the same vector facility, it becomes necessary to save the contents of vector facility registers for the just-interrupted program and to restore the previous contents for the program that is to be resumed. Unlike the scalar parts of the machine, where the saving or restoring of registers involves less than 200 bytes of data, saving and restoring the registers of the vector facility can involve thousands of bytes, which may have a noticeable performance impact.

The structure of the vector instruction set is simpler than the count of 171 new instructions suggests.

Several measures are available to help reduce the save-restore overhead. Saving and restoring are handled by special instructions which differ from the regular vector store and load instructions for optimum performance. Each pair of vector registers has an in-use bit. When the bit is off, the register pair is known to be cleared and does not participate in saving and restoring. The bit is turned on as soon as any part of the register pair is loaded. A clear instruction is provided to allow the program to turn an in-use bit off and thus indicate that those vector registers are no longer needed. A vector-change bit for each vector-register pair indicates whether the contents have been changed since the last time the control program saved the register contents; this allows redundant saves into the same storage area to be bypassed, but it cannot avoid the time needed to restore all registers that are in use, whether changed or not.

Vector instructions

Arithmetic and logical operations. The structure of the vector instruction set is simpler than the count of 171 new instructions would seem to suggest, because each arithmetic and logical operation is repeated several times for different operand types and instruction formats. The following are the basic arithmetic and logical operations on vectors:

ADD
SUBTRACT
MULTIPLY
DIVIDE
COMPARE
AND
OR
EXCLUSIVE OR

The following three compound operations have direct application in array arithmetic:

MULTIPLY AND ADD
MULTIPLY AND SUBTRACT
MULTIPLY AND ACCUMULATE

The following instructions operate on vector operands to produce scalar results, as does MULTIPLY AND ACCUMULATE:

ACCUMULATE
MAXIMUM ABSOLUTE
MAXIMUM SIGNED
MINIMUM SIGNED

The maximum and minimum operations, when used in a sectioning loop, produce as the result a single number which is the maximum or minimum element of an entire vector regardless of length, together, optionally, with its position in the vector. ACCUMULATE has as its result the sum of all elements of a vector, and MULTIPLY AND ACCUMULATE gives the sum of the product elements obtained by multiplying a pair of vectors (the inner or dot product). However, there is a complication.

The basic arithmetic and logical instructions produce as their result a vector of independent elements, whose value is not affected by the order in which they are generated. Execution of the element operations can, therefore, be readily overlapped, as in a pipelined arithmetic unit. The same is true of the maximum and minimum operations, which produce a scalar result that remains independent of the order of comparing individual elements.

The accumulation instructions are different. The design of an arithmetic pipeline, particularly the number of stages in the pipeline, places limitations on the order in which the elements can be added at top speed. The order of addition may affect the rounding error and, on occasion, whether and when overflow or underflow takes place. To overcome this problem, the accumulation operation is carried out in two phases: First, the vector elements or products are reduced to a few partial sums, using the arithmetic pipeline at full speed; then the partial sums are added sequentially to form the desired single result.

The result of the first phase of an accumulation operation is a partial-sum vector that is placed in a vector register. The length of this result vector is p, the partial-sum number, which is a small number that depends on the model and is essentially the length of the pipeline. (The 3090 has a partial-sum number of 4.) Given the number p, however, the result is precisely defined. Accumulation of the elements of an operand vector **B** to produce the partial-sum vector **A** is performed just like the vector addition **A** = **A** + **B**, except that the result vector **A** is wrapped back on itself. Thus, elements 0 to $p - 1$ of vector **B** are added to elements 0 to $p - 1$ of vector **A**; but then element p of vector **B** is added to element 0 of vector **A**, element $p + 1$ to element 1; and, in general, operand element i is added to partial-sum element i modulo p in ascending order of i. The only difference between accumulation and addition is a simple change of the counter that advances from one vector element in the target register to the next, so that it progresses from element $p - 1$ to element 0. Vector accumulation is interruptible in the same way as vector addition. Sectioning loops for vectors that are longer than the section size continue to add accumulation results to the same partial-sum vector.

The second phase of accumulation is performed by the sequential, unoverlapped instruction SUM PARTIAL SUMS, which reduces the partial-sum vector to a scalar sum. Its performance is not critical because only a few vector elements are involved.

Although the partial-sum technique produces a result which may differ from one model to another and from the result of sequential addition, the result is precisely defined, is independent of any interruptions, and can be duplicated by means of a corresponding scalar loop.

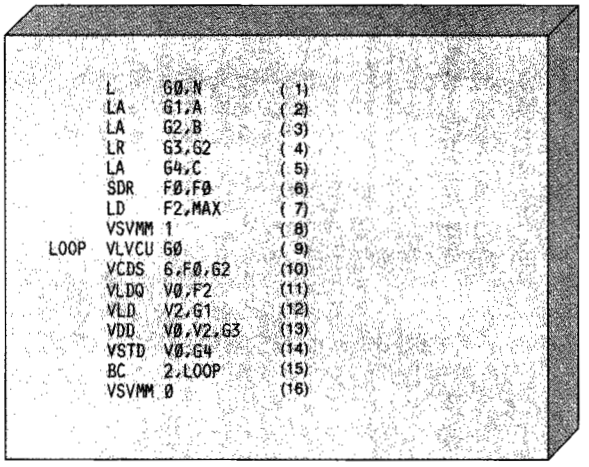
The partial-sum technique produces a result which is precisely defined.

Conditional vector processing. Vector elements may have to be processed conditionally, depending on the outcome of a comparison between two vectors or between a vector and a scalar. The desired one of the high-low-equal relationships is specified by the vector-comparison instruction. The true (one) or false (zero) comparison results are recorded as a set of mask bits in the vector-mask register, one bit for each vector element. The number of active bits in the vector-mask register is the same as the number of active elements currently in the vector registers, as determined by the vector count.

Arithmetic and logical operations may be performed conditionally by turning on a *vector-mask mode*. When the mode is on, only those elements are processed that correspond to a mask bit of one. Where the mask register contains zeros, no result element is produced, the target register remains unchanged, and any arithmetic exceptions are suppressed. The division of two vectors provides an example. If the divisor may contain zero elements, disruptive divide exceptions are avoided by first comparing the divisor vector with a scalar zero and then performing the division with the mask mode on.

LOAD and STORE instructions are not under the control of the vector-mask mode. Separate instructions, LOAD MATCHED and STORE MATCHED, are provided to load and store elements only where the vector-mask register contains ones, and they do so regardless of the mask mode. This separation of functions allows conditional arithmetic to be interspersed with unconditional loading or storing

Figure 5 Instructions for the example C=A/B



```
        L     G0,N        ( 1)
        LA    G1,A        ( 2)
        LA    G2,B        ( 3)
        LR    G3,G2       ( 4)
        LA    G4,C        ( 5)
        SDR   F0,F0       ( 6)
        LD    F2,MAX      ( 7)
        VSVMM 1           ( 8)
LOOP    VLVCU G0          ( 9)
        VCDS  6,F0,G2     (10)
        VLDQ  V0,F2       (11)
        VLD   V2,G1       (12)
        VDD   V0,V2,G3    (13)
        VSTD  V0,G4       (14)
        BC    2,LOOP      (15)
        VSVMM 0           (16)
```

inside the same sectioning loop, without repeatedly changing the mode setting.

Figure 5 illustrates the conditional division of vector **A** by vector **B** so as to avoid any interruptions due to zero elements in the divisor. The largest possible floating-point number is arbitrarily placed in element locations of the result vector **C** which correspond to zero divisor elements. All three vectors are contiguous and consist of long floating-point numbers.

The first five instructions set up four general registers with the vector length N and the addresses of the three vectors. The address of the divisor is loaded into both general registers 2 and 3 because two vector instructions in the following sectioning loop refer to that vector in storage. Instruction (6) sets floating-point register 0 to zero. Instruction (7) loads the largest positive floating-point number (a constant at storage location MAX, not shown) into floating-point register 2. The vector-mask mode is turned on by instruction (8) before starting the sectioning loop at (9).

Instruction (10) is a vector-comparison instruction that compares a section of vector **B** with the scalar zero set up in floating-point register 0; the 6 is a modifier field that specifies a not-equal comparison, so that the vector-mask bits are set to ones wherever the vector elements are nonzero. Vector registers 0 and 1 are loaded unconditionally with the largest positive value by instruction (11). Instruction (12) loads the dividend vector into vector registers 2 and 3. The division is performed by instruction (13), and the result placed temporarily in vector registers 0 and 1 is stored by instruction (14).

The only instruction that is affected by the vector-mask mode inside the loop is the division instruction (13). The comparison instruction (which sets up the vector-mask bits) and the load and store instructions all operate unconditionally. The division instruction places the proper quotient in every element position of vector registers 0 and 1 that corresponds to a mask bit of one. The constant loaded by instruction (11) remains only in element positions that are skipped because the mask bit is zero. Instruction (16) turns off the vector-mask mode upon exit from the sectioning loop.

Conditional operations on vectors are performed differently from such operations on scalars. The three or four possible outcomes of a single scalar comparison are recorded in a two-bit condition code which is then tested by a conditional branch instruction for the desired condition. When an arithmetic operation is not to be performed, the program branches around the corresponding instruction. Vector comparisons obviously cannot use the branching technique, and most vector instructions do not even set the condition code (which remains free for other uses, such as loop control). The vector-comparison instructions specify a more restrictive test that has only two possible outcomes, so as to limit the comparison result to a single bit per vector element.

Indirect element selection. The elements of a vector **V** are considered to be numbered in sequence from zero to $N - 1$, where N is the length of the vector. The accessing of elements in this sequence is fastest. The vector elements can also be accessed in an entirely different order as vector **V(A)**, where **A** is an auxiliary vector consisting of a rearrangement of these element numbers. Auxiliary vector **A** may have a length different from N, and each element number may appear zero, one, or more times. The resulting vector **V(A)** has the same length as **A**.

To perform such indirect element selection, a section of the auxiliary vector is placed into one vector register, and the corresponding section of vector elements is then loaded into another vector

register or register pair by means of the instruction LOAD INDIRECT. The selected elements can later be returned to their storage locations by using the companion instruction STORE INDIRECT.

Processing of sparse vectors. Sparse vectors have a large number of zero elements. Such vectors may often be processed and stored more efficiently by retaining only the nonzero elements in storage. The positions of the nonzero elements in such a sparse vector are recorded in an auxiliary vector. One type of auxiliary vector is a vector containing the element numbers of the nonzero elements. Another type is a bit vector which has ones in bit positions corresponding to nonzero elements of the full vector in storage and zeros as place-holders for the zero elements of the full vector that are not stored.

A bit vector is first created in storage by comparing the full vector with zero (or some tiny value) to detect nonzero elements. The bit vector may be converted to a vector of element numbers by means of the instruction LOAD BIT INDEX. These element numbers are the positions of all the one bits in the bit vector.

The instructions LOAD INDIRECT and STORE INDIRECT may be used to perform indirect element selection when the auxiliary vector is or has been converted to a vector of element numbers. Two other instructions, LOAD EXPANDED and STORE COMPRESSED, work directly with a section of a bit vector in the vector-mask register.

Discussion

The IBM System/370 vector facility can provide a substantial performance increase for vectorizable applications with relatively modest additions to hardware, software, and application programming. The thorough integration of the vector facility into the existing System/370 base architecture provides common data formats, produces compatible results, and allows for common exception handling. Among the features believed to be novel for a register type of organization are machine-assisted vector sectioning and precisely interruptible instructions.

To give the appearance of sequential instruction execution and thereby facilitate the interruptibility of vector instructions, there is no *chaining* of these instructions.[9] Chaining would allow the execution of two or more successive vector instructions to be overlapped, where vector elements produced as the result of one instruction are passed on-the-fly to a subsequent instruction which needs them as operand elements. Some of the advantages of instruction chaining are obtained, however, by providing several of the most important combinations of operations with single compound instructions that are cleanly interruptible. They include compound instructions that directly process vector operands in storage and, most particularly, the instruction MULTIPLY AND ADD.

Having instructions which are cleanly interruptible is especially helpful in a virtual-storage environment. Long vectors may span many storage pages, especially when the stride is large; in the extreme, each element may be on a different page. There is no need to prefetch all the pages and tie up corresponding space in real storage, just in case the pages are needed. Although occasional page faults while loading or storing vectors could be handled by re-executing a noninterruptible instruction from its beginning, such restarting might be difficult when instructions also perform arithmetic on vectors in storage.

Users should bear in mind that relying too heavily on the automatic vectorization of existing programs may cause performance to be less than that of which the vector facility is capable. Since optimally designed scalar programs do not necessarily run in optimal fashion in a vector version, it may be desirable to tune important programs to the characteristics of the particular vector hardware. Further improvements may be obtained by developing new algorithms that are better suited to vector processing. Such extra efforts can be applied selectively over a period of time, while obtaining the initial performance gain available from a simple conversion.

Acknowledgments

Some of the basic concepts of this vector architecture were derived from two earlier IBM development projects by G. Paul and by D. S. Wehrly. Credit for major innovations in the present version belongs to R. M. Smith. Among other contributors, particular mention should be made of R. J. Stanton, J. Thomas, and S. G. Tucker for their contributions from an engineering perspective, and

F. G. Gustavson, W. P. Heising, and T. C. Spillman for their contributions from a programming perspective. A. Padegs provided technical guidance and assistance from the beginning.

Cited references and notes

1. *IBM System/370 Vector Operations*, SA22-7125, IBM Corporation, available through IBM branch offices; contains a full description of the vector architecture and more programming examples than are included in this paper.
2. R. P. Case and A. Padegs, "Architecture of the IBM System/370," *Communications of the ACM* **21**, No. 1, 73–96 (January 1978).
3. A. Padegs, "System/370 Extended Architecture: Design considerations," *IBM Journal of Research and Development* **27**, No. 3, 198–205 (May 1983).
4. A *vector* is a linearly ordered collection of data items. A *scalar* is a single data item. The term "scalar" is also used here to mean "nonvector"; thus, scalar instructions are those that are not part of the vector facility.
5. P. M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill Book Co., Inc., New York (1981); contains an extensive bibliography.
6. A. E. Charlesworth, "An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family," *Computer* **14**, No. 9, 18–27 (September 1981); part of special issue on peripheral array processors.
7. T. Louie, "Array processors: A selected bibliography," *Computer* **14**, No. 9, 53–57 (September 1981); 116 references for special issue on peripheral array processors.
8. E. W. Kozdrowicki and D. J. Theis, "Second generation of vector computers," *Computer* **13**, No. 11, 71–83 (November 1980); contains extensive references.
9. R. M. Russell, "The CRAY-1 computer system," *Communications of the ACM* **21**, No. 1, 63–72 (January 1978).
10. R. L. Sites, "An analysis of the CRAY-1 computer," *The 5th Annual Symposium on Computer Architecture (IEEE and ACM)*, Publication 78CH1284-9C, pp. 101–106 (April 1978); available from the Institute of Electrical and Electronics Engineers, Inc., 345 East 47 Street, New York, NY 10017.
11. D. J. Theis, "Vector supercomputers," *Computer* **7**, No. 4, 52–61 (1974).
12. *IBM System/370 Principles of Operation*, GA22-7000, IBM Corporation; available through IBM branch offices.
13. *IBM System/370 Extended Architecture Principles of Operation*, SA22-7085, IBM Corporation; available through IBM branch offices.
14. Unnormalized numbers can occur only when introduced as constants or input data, because all floating-point vector instructions produce normalized results.

Werner Buchholz *Information Systems and Storage Group, Department E57, Building 901, P.O. Box 390, Poughkeepsie, New York 12602.* Dr. Buchholz is a senior engineer in the Central Systems Architecture Department. He received the B.A.Sc. and M.A.Sc. from the University of Toronto, Canada, in 1945 and 1946, and the Ph.D. from the California Institute of Technology in 1950, all in electrical engineering. In 1949, he joined IBM at the Poughkeepsie laboratory, where he has worked on various assignments in architecture and performance evaluation, starting with the IBM 701 and 702, later as manager of systems planning for Stretch, and most recently on extensions of the System/370 architecture. He received two IBM Invention Achievement Awards and an IBM Outstanding Invention Award. He has published a number of papers on computer organization and edited the book *Planning a Computer System (Project Stretch)*, McGraw-Hill Book Co., Inc. Dr. Buchholz is a Fellow and past Director of the Institute of Electrical and Electronics Engineers, and a past Chairman of what is now the IEEE Computer Society. He was a Director of the American Federation of Information Processing Societies. He is on the editorial board of the *AFIPS Annals of the History of Computing*.

Reprint Order No. G321-5261.