Third Edition (August 1987)

This edition obsoletes and replaces the previous edition, SA22-7125-1. It contains a number of detailed changes, which are indicated by a vertical line to the left of the change.

Changes are made occasionally to the information herein; before using this publication in connection with the operation of IBM equipment, refer to the latest IBM System/370, 30xx, and 4300 Processors Bibliography, GC20-0001, for the editions that are applicable and current.

IBM may have patents or pending patent applications covering subject matter described herein. Furnishing this publication does not constitute or imply a grant of any license under any patents, patent applications, trademarks, copyrights, or other rights of IBM or of any third party, or any right to refer to IBM in any advertising or other promotional or marketing activities. IBM assumes no responsibility for any infringement of patents or other rights that may result from the use of this publication or from the manufacture, use, lease, or sale of apparatus described herein.

Licenses under IBM’s utility patents are available on reasonable and nondiscriminatory terms and conditions. Inquiries relative to licensing should be directed, in writing, to: IBM Corporation, Director of Contracts and Licensing, Armonk, NY, USA 10504.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates.

Publications are not stocked at the address given below. Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to: IBM Corporation, Central Systems Architecture, Department E57, PO Box 390, Poughkeepsie, NY, USA 12602. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1986, 1987
This publication contains, for reference purposes, a detailed definition of the machine functions provided by the IBM System/370 vector facility. The vector facility operates as a compatible extension of the functions of System/370 as described in one of the Principles of Operation publications, either the facilities of the System/370 extended architecture (370-XA) in *IBM 370-XA Principles of Operation*, SA22-7085, or those of the System/370 architecture in *IBM System/370 Principles of Operation*, GA22-7000.

The publication should not be considered an introduction or a textbook. It is written as a reference for use principally by assembler-language programmers, although anyone concerned with the functional details of vector operations may find it useful. It describes each function at the level of detail needed to prepare an assembler-language program which relies on that function.

This publication does not describe all the instructions or other functions needed to write a complete program using vectors. It includes a description only of functions which are added to System/370 as part of the vector facility. The reader is assumed to be familiar with either the *IBM 370-XA Principles of Operation* or *IBM System/370 Principles of Operation*, as appropriate. Terms and concepts referred to in this publication but explained in those Principles of Operation publications are not explained again in this publication.

Writing a program in assembler language requires a familiarity with the notations and conventions of that language, as well as with the facilities of the operating system under which the program is to be run. The reader should refer to the appropriate programming publications for such information.

**Terminology**

As used in this publication, a *scalar* is a single data item, which may be a floating-point number, a binary integer, or a set of logical data. A *vector* is a linearly ordered collection of such scalars, where each scalar is an *element* of the vector. All elements of a single vector are of the same type: floating-point numbers (floating-point vector), binary integers (binary vector), or logical data (logical vector).

*Scalar instructions* are instructions which perform load, store, arithmetic, or logical operations on scalars that may reside in storage, floating-point registers, or general registers. *Vector instructions* perform similar operations on vectors that may reside in storage or in registers of the vector facility. Only vector instructions and related operations are described in this publication. Scalar instructions are described in the *IBM 370-XA Principles of Operation* or *IBM System/370 Principles of Operation*. 

This page is intentionally left blank.
## Contents

Chapter 1. Introduction ........................................... 1-1
Compatibility Considerations ................................ 1-1
Vector and Scalar Operations .................................. 1-1
Model-Dependent Vector Functions .............................. 1-2

Chapter 2. Vector Facility ......................................... 2-1
Vector-Facility Structure ........................................ 2-2
Vector Registers .................................................. 2-2
Vector-Mask Register ............................................ 2-2
Vector Parameters ............................................... 2-2
Vector-Status Register ......................................... 2-3
Vector-Activity Count .......................................... 2-5
Modes of Operation ............................................. 2-6
Vector-Instruction Operands and Results ...................... 2-7
Arithmetic Vectors in Storage ................................ 2-8
Arithmetic Vectors in Registers ............................... 2-10
Bit Vectors ...................................................... 2-10
Vector Sectioning ............................................... 2-11
Conditional Arithmetic ........................................ 2-11
Common Instruction Descriptions .............................. 2-12
Instruction Classes .............................................. 2-12
Instruction Formats ............................................. 2-13
Summary of Instructions by Class and Format ............ 2-15
Class-IM and Class-IC Instructions ............................ 2-15
Class-NC Instructions ......................................... 2-18
Instructions In Other Classes ................................. 2-19
Vector Interruptions ............................................. 2-19
Interruptible Vector Instructions .............................. 2-19
Effect of Interruptions during Execution .................... 2-23
Program- Interruption Conditions ............................. 2-24
Priority of Vector Interruptions ............................... 2-26
Program Switching .............................................. 2-26
Program Use of the Restore and Save Instructions ........ 2-27
Clear Operations ............................................... 2-28
Save-Area Requirements ....................................... 2-28
Relationship to Other Facilities .............................. 2-28
Program-Event Recording (PER) ............................... 2-28
Vector-Store Operations ....................................... 2-29
Resets .......................................................... 2-29
Machine-Check Handling ....................................... 2-29

Chapter 3. Vector-Facility Instructions ....................... 3-1
ACCUMULATE ..................................................... 3-2
ADD ............................................................ 3-3
AND ............................................................. 3-4
AND TO VMR .................................................... 3-5
CLEAR VR ....................................................... 3-5
COMPARE ......................................................... 3-5
COMPLEMENT VMR .............................................. 3-7
COUNT LEFT ZEROS IN VMR ................................. 3-7
COUNT ONES IN VMR .......................................... 3-7
DIVIDE .......................................................... 3-8
EXCLUSIVE OR .................................................. 3-9
EXCLUSIVE OR TO VMR ....................................... 3-9
EXTRACT ELEMENT ............................................ 3-9
EXTRACT VCT ................................................... 3-10
EXTRACT VECTOR MASK MODE ............................... 3-10
LOAD ............................................................ 3-10
LOAD BIT INDEX ............................................... 3-11
LOAD COMPLEMENT ............................................ 3-14
LOAD ELEMENT ............................................... 3-14
LOAD EXPANDED ............................................... 3-14
LOAD HALFWORD .............................................. 3-15
LOAD INDIRECT .................................................. 3-16
LOAD INTEGER VECTOR ....................................... 3-16
LOAD MATCHED ............................................... 3-17
LOAD NEGATIVE ............................................... 3-18
LOAD POSITIVE ................................................ 3-18
LOAD VCT AND UPDATE ....................................... 3-19
LOAD VCT FROM ADDRESS .................................... 3-19
LOAD VMR ........................................................ 3-20
LOAD VMR COMPLEMENT ....................................... 3-20
LOAD ZERO ....................................................... 3-20
MAXIMUM ABSOLUTE ......................................... 3-21
MAXIMUM SIGNED .............................................. 3-21
MINIMUM SIGNED .............................................. 3-21
MULTIPLY ......................................................... 3-22
MULTIPLY AND ACCUMULATE ................................ 3-23
MULTIPLY AND ADD ............................................ 3-24
MULTIPLY AND SUBTRACT .................................... 3-24
OR ............................................................... 3-26
OR TO VMR ....................................................... 3-26
RESTORE VAC ................................................... 3-26
RESTORE VMR ................................................... 3-27
RESTORE VR ..................................................... 3-27
RESTORE VSR .................................................... 3-28
SAVE CHANGED VR ............................................ 3-29
SAVE VAC ........................................................ 3-30
SAVE VMR ....................................................... 3-30
SAVE VR ........................................................ 3-31
SAVE VSR ........................................................ 3-31
SET VECTOR MASK MODE ..................................... 3-32
SHIFT LEFT SINGLE LOGICAL ............................... 3-32
SHIFT RIGHT SINGLE LOGICAL ............................. 3-32
STORE .......................................................... 3-32
STORE COMPRESSED .......................................... 3-33
STORE HALFWORD ............................................. 3-33
STORE INDIRECT ............................................... 3-34
STORE MATCHED .............................................. 3-34
STORE VECTOR PARAMETERS ................................. 3-35
<table>
<thead>
<tr>
<th>Instruction/Operation</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORE VMR</td>
<td>3-35</td>
</tr>
<tr>
<td>SUBTRACT</td>
<td>3-35</td>
</tr>
<tr>
<td>SUM PARTIAL SUMS</td>
<td>3-36</td>
</tr>
<tr>
<td>TEST VMR</td>
<td>3-37</td>
</tr>
<tr>
<td>ZERO PARTIAL SUMS</td>
<td>3-37</td>
</tr>
<tr>
<td>Exception Avoidance</td>
<td>A-4</td>
</tr>
<tr>
<td>Add to Magnitude</td>
<td>A-4</td>
</tr>
<tr>
<td>Operations on Sparse Vectors</td>
<td>A-4</td>
</tr>
<tr>
<td>Full Added to Sparse to Give Full</td>
<td>A-5</td>
</tr>
<tr>
<td>Sparse Added to Sparse to Give Sparse</td>
<td>A-5</td>
</tr>
<tr>
<td>Floating-Point-Vector Conversions</td>
<td>A-6</td>
</tr>
<tr>
<td>Fixed Point to Floating Point</td>
<td>A-6</td>
</tr>
<tr>
<td>Floating Point to Fixed Point</td>
<td>A-6</td>
</tr>
</tbody>
</table>

**Appendix A. Instruction-Use Examples**  
Operations on Full Vectors  
Contiguous Vectors  
Vectors with Stride  
Vector and Scalar Operands  
Sum of Products  
Compare and Swap Vector Elements  
Conditional Arithmetic  

**Appendix B. Lists Of Instructions**  

**Appendix C. Condition-Code Settings**  

**Index**
Chapter 1. Introduction

The vector facility is a compatible addition to the IBM System/370 architecture. Use of the facility may benefit applications in which a great deal of the time of the central processing unit (CPU) is spent executing arithmetic or logical instructions on data which can be treated as vectors. By replacing loops of scalar instructions with the vector instructions provided by the vector facility, such applications may take advantage of the order inherent in vector data to improve performance.

When the vector facility is provided on a CPU, it functions as an integral part of that CPU:

- Standard System/370 instructions can be used for all scalar operations.
- Data formats which are provided for vectors are the same as the corresponding System/370 scalar formats.
- Long-running vector instructions are interruptible in the same manner as long-running scalar instructions; their execution can be resumed from the point of interruption after appropriate action has been taken.
- Program interruptions due to arithmetic exceptions are handled in the same way as for scalar-arithmetic instructions, and the same fixup routines can be used with at most some minor extensions.
- Vector data may reside in virtual storage, with access exceptions being handled in the customary manner.

Compatibility Considerations

Compatibility with System/370 scalar operations has been one of the major objectives of the vector architecture, so as to provide the same result data when equivalent functions are programmed on machines without the vector facility. Some departures from strict compatibility are introduced, however, for the sake of performance and to provide implementers of the vector facility more flexibility in making design choices.

Vector and Scalar Operations

Although operations on vector operands are generally compatible, element by element, with the corresponding scalar operations, there are certain differences between the vector and scalar architectures:

- Operands of vector-facility instructions must be aligned on integral boundaries; scalar-instruction operands need not be so aligned. (See the section "Vector-Instruction Operands and Results" on page 2-7.)
- Vector divide and multiply operations do not permit unnormalized floating-point operands; the corresponding scalar instructions do. Vector programs may encounter the unnormalized operand exception. (See the instruction descriptions and the section "Unnormalized-Operand Exception" on page 2-26.)
- Because the result of a series of floating-point additions may depend on their sequence, the results produced by the vector instructions ACCUMULATE or MULTIPLY AND ACCUMULATE, followed by SUM PARTIAL SU M S, are not necessarily identical with those produced by scalar summation loops, unless the scalar loops are written to perform the additions in exactly the same sequence as defined for the vector instructions. (See the instruction descriptions and the section "Partial-Sum Number" on page 2-2.)
- If, during execution of MULTIPLY AND ACCUMULATE, MULTIPLY AND ADD, or MULTIPLY AND SUBTRACT, the multiplication of an element pair results in an exponent underflow, a true zero is used in place of the product even when the exponent-underflow mask in the PSW is one. The vector and scalar results are the same, however, when the mask bit is zero or when an exponent underflow occurs during the addition or subtraction. (See the instruction descriptions and the section "Exponent-Underflow Exception" on page 2-25.)
- Vector-facility instructions cannot safely be used to store into the current instruction stream, whereas all other instructions are interlocked to prevent this. (See the section "Vector-Store Operations" on page 2-29.)
Model-Dependent Vector Functions

Programmers should keep the following restrictions in mind to ensure that programs will run successfully regardless of which implementation techniques have been chosen on a particular model.

The program should not depend on specific values of the model-dependent vector parameters (section size and partial-sum number). Likewise, the program should not depend on the contents of fields that are described as "reserved" or "undefined." Specifically:

• The section size should not be treated as a numeric constant. Thus, save-area sizes should be computed from the section-size value obtained at execution time. (See the section “Save-Area Requirements” on page 2-28.) The section size may be obtained by executing the instruction STORE VECTOR PARAMETERS.

• The exact result produced by the vector instructions ACCUMULATE or MULTIPLY AND ACCUMULATE, followed by SUM PARTIAL SUMS, may depend on the partial-sum number of the model because that number affects the sequence of performing the floating-point additions.

• The program should not rely on reserved bits 0-14 of the vector-status register being zeros when placed in a general register by the instruction EXTRACT VECTOR MASK MODE, or on the bits being stored as zeros by SAVE VSR. (See the instruction descriptions.)

• The program should not depend on any particular values being stored by the instruction SAVE VMR in the undefined part of the save area for the vector-mask register; nor should the program depend on the presence or absence of access exceptions for that portion of the VMR save area when executing the instruction RESTORE VMR or SAVE VMR. (See the instruction descriptions.)

• When a program using vector-facility instructions is interrupted, it cannot be safely resumed on another machine with a different section size or partial-sum number, unless the interruption occurred at a point that is known to be independent of the section size or partial-sum number, respectively.

• The program should not depend on the left-most bits of the address of a vector operand in storage being either set to zero or left unchanged by a vector instruction when no elements are due to be processed by the instruction. (See the sections “Storage Operands for QST and VST Formats” on page 2-17 and “VS-Format Instructions” on page 2-19.)

The program should not rely on receiving a specific program interruption, either operation exception or vector-operation exception, to indicate whether the vector facility is installed in any CPU of the configuration, since it depends on the model which of the two exceptions occurs. (See the section “Vector-Operation Control” on page 2-6.)

Problem-state programs should not depend on the setting of the vector change bits, which may be altered by actions of the control program that are unrelated to the actions of a problem-state program. Supervisor-state programs can depend on the accuracy of vector change bits that are zeros; vector change bits may sometimes be set to one, however, even when the corresponding vector-register pair has not been changed. Note also that the effect on the vector change bits of executing the instructions RESTORE VR and RESTORE VSR depends on whether the CPU is in the problem or supervisor state. (See the section “Vector Change Bits” on page 2-5.)

PER events for general-register alteration may or may not be recognized for vector-facility instructions.
Chapter 2. Vector Facility

Vector-Facility Structure .................................. 2-2
Vector Registers .......................................... 2-2
Vector-Mask Register ........................................ 2-2
Vector Parameters .......................................... 2-2
Section Size .................................................. 2-2
Partial-Sum Number ......................................... 2-2
Vector-Status Register ........................................ 2-3
Vector-Mask-Mode Bit ......................................... 2-3
Vector Count .................................................. 2-3
Vector Interruption Index .................................... 2-4
Vector In-Use Bits ............................................. 2-4
Vector Change Bits ............................................ 2-5
Vector-Activity Count ....................................... 2-5
Modes of Operation .......................................... 2-6
Vector-Operation Control ..................................... 2-6
Vector-Instruction Operands and Results .................. 2-7
Arithmetic Vectors in Storage ............................... 2-8
Access by Sequential Addressing ............................ 2-8
Access by Indirect Element Selection ........................ 2-9
Arithmetic Vectors in Registers .............................. 2-10
Operands in Vector Registers ................................ 2-10
Operands in Scalar Registers ................................ 2-10
Bit Vectors .................................................... 2-10
Vector Sectioning ............................................ 2-11
Conditional Arithmetic ....................................... 2-11
Vector-Mask Mode ............................................. 2-11
Instructions Controlling the Vector-Mask Mode ......... 2-12
Common Instruction Descriptions ............................ 2-12
Instruction Classes .......................................... 2-12
Instruction Formats .......................................... 2-13
Field Designations ........................................... 2-13
Three-Operand Instruction Formats .......................... 2-15
Summary of Instructions by Class and Format .......... 2-15
Class-IM and Class-IC Instructions ........................ 2-15
Class-IM Instructions ......................................... 2-16
Class-IC Instructions ......................................... 2-17
Storage Operands for QST and VST Formats .............. 2-17
Class-NC Instructions ......................................... 2-18

VS-Format Instructions .................................... 2-19
Instructions In Other Classes ................................ 2-19
Interruptible Vector Instructions ............................ 2-19
Units of Operation ............................................ 2-20
Operand Parameters .......................................... 2-20
Arithmetic Exceptions ........................................ 2-21
Exception-Extension Code .................................... 2-21
Types of Ending for Units of Operation .................... 2-21
Effect of Interruptions During Execution ................... 2-23
Setting of Instruction Address ................................ 2-23
Setting of Instruction-Length Code .......................... 2-23
Setting of Storage Address ................................... 2-23
Setting of Vector Interruption Index ........................ 2-24
Program-Interrupt Conditions ............................... 2-24
Access Exceptions for Vector Operands ...................... 2-24
Exponent-Overflow Exception ................................ 2-25
Exponent-Underflow Exception ................................ 2-25
Floating-Point-Divide Exception .............................. 2-25
Specification Exception ....................................... 2-25
Unnormalized-Operand Exception ............................. 2-26
Vector-Operation Exception ................................... 2-26
Priority of Vector Interruptions ............................. 2-26
Program Switching ............................................ 2-26
Program Use of the Restore and Save Instructions ....... 2-27
Restore Operations ............................................ 2-27
Save Operations .............................................. 2-27
Clear Operations ............................................. 2-28
Save-Area Requirements ...................................... 2-28
Relationship to Other Facilities ............................. 2-28
Program-Event Recording (PER) .............................. 2-28
Vector-Store Operations ...................................... 2-29
Storage-Operand Consistency ................................ 2-29
Storing into Instruction Stream .............................. 2-29
Sets .......................................................... 2-29
Machine-Check Handling ..................................... 2-29
Vector-Facility Failure ....................................... 2-30
Vector-Facility Source ........................................ 2-30
Validation of Vector-Facility Registers ...................... 2-30
Vector-Facility Structure

The vector facility provides:

- The vector-facility registers:
  - 16 vector registers
  - A vector-mask register
  - A vector-status register
  - A vector-activity count
- 171 instructions
- The following exceptions and exception indications:
  - An unnormalized-operand exception
  - A vector-operation exception
  - An exception-extension code for arithmetic exceptions
- A vector-control bit, bit 14 of control register 0

Figure 2-1 on page 2-3 shows the registers provided by the vector facility.

Vector Registers

There are 16 vector registers, numbered 0-15. They are used to hold one or more of the vector operands in most arithmetic, comparison, logical, load, and store operations. Unlike the general and floating-point registers, the vector registers are multipurpose in that vectors of floating-point, binary-integer, and logical data can all be accommodated.

Each vector register contains a number of element locations of 32 bits each. Depending on the operation, a vector operand may occupy a single vector register or an even-odd pair of registers. The element locations of a vector register are identified by consecutive element numbers, starting with 0.

Vector-Mask Register

There is one vector-mask register (VMR), which is used as:

- The target of the result of vector-compare operations,
- The source and target of logical operations on bit vectors, and
- The source of the mask for mask-controlled operations.

Vector Parameters

The section size and the partial-sum number are model-dependent parameters which control certain operations of the vector facility.

Section Size

The number of element locations in a vector register, which is also the number of bit positions in the vector-mask register, is called the section size. The section size is a power of 2; depending on the model, the section size may be 8, 16, 32, 64, 128, 256, or 512.

The element locations of a vector register, as well as the bit positions in the vector-mask register, are numbered from 0 to one less than the section size.

In a multiprocessing configuration, the section size is the same for each CPU which has the vector facility installed.

The section size of a model may be obtained by executing the instruction STORE VECTOR PARAMETERS, which places the value as a 16-bit binary integer in the left half of a word in storage.

Partial-Sum Number

The partial-sum number is the number of partial sums produced when executing the instruction ACCUMULATE or MULTIPLY AND ACCUMULATE. It is also the number of vector-register elements set to zero by the instruction ZERO PARTIAL SUMS, as well as the number of vector-register elements summed by the instruction SUM PARTIAL SUMS.

Depending on the model, the partial-sum number may range from 1 up to and including the section size.

In a multiprocessing configuration, the partial-sum number is the same for each CPU which has the vector facility installed.

The partial-sum number of a model may be obtained by executing the instruction STORE VECTOR PARAMETERS, which places the value as a 16-bit binary integer in the right half of a word in storage.
Vector-Status Register

The vector-status register (VSR) is 64 bits long and contains five fields of information, which describe the current status of the vector and vector-mask registers and of a mode of operation. The fields are arranged as follows:

<table>
<thead>
<tr>
<th>Zeros</th>
<th>M</th>
<th>VCT</th>
<th>VIX</th>
<th>VIU</th>
<th>VCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>32</td>
<td>48</td>
<td>56</td>
<td>63</td>
</tr>
</tbody>
</table>

The contents of the vector-status register as a whole may be examined by the instruction SAVE VSR and altered by the instruction RESTORE VSR. Bits 0-14 of the vector-status register are reserved for possible future use and are stored as zeros by SAVE VSR; if the instruction RESTORE VSR specifies other than all zeros for these bit positions, a specification exception is recognized.

Vector-Mask-Mode Bit

When the vector-mask-mode bit (M), bit 15 of the vector-status register, is one, the vector-mask mode is on, and arithmetic and logical instructions are executed under the control of bits in the vector-mask register. When the bit is zero, the mode is off. For details, see the section “Conditional Arithmetic” on page 2-11.

Vector Count

The vector count (VCT), bits 16-31 of the vector-status register, is a 16-bit unsigned binary integer. Together with the vector interruption index, it determines for most vector operations the number of element locations to be processed in vector registers or the number of bit positions to be processed in the vector-mask register.

Elements in register positions with element numbers less than the vector count are called the active elements of the vector register. Likewise, bits in bit positions of the vector-mask register with bit numbers less than the vector count are called the
active bits of the vector-mask register. Only the active elements or bits take part in operations where the number of elements or bits processed is determined by the vector count.

The vector count may range in value from zero up to and including the section size. A specification exception is recognized if the instruction RESTORE VSR attempts to place a value in the vector-count field which exceeds the section size. The instruction EXTRACT VCT may be used to examine the vector count.

The following instructions may be used to set the vector count. If they specify a number greater than the section size, they set the vector count equal to the section size.

LOAD BIT INDEX
LOAD VCT AND UPDATE
LOAD VCT FROM ADDRESS

For information on using the vector count with vectors of any length, see the section “Vector Sectioning” on page 2-11.

Vector Interruption Index

The vector interruption index (vix), bits 32-47 of the vector-status register, is a 16-bit unsigned binary integer. It specifies the number of the first element location in any vector register, or of the first bit position in the vector-mask register, to be processed by an interruptible vector instruction which depends on the vector interruption index. The vector interruption index is used to control resumption of the operation after such an instruction has been interrupted. It is normally zero at the start of execution, and it is set to zero at completion.

For details concerning the operation of the vector interruption index and the effect of an interruption, see the section “Vector Interruptions” on page 2-19.

The vector interruption index may range from zero to the section size. It may be examined by using the instruction SAVE VSR, and it may be set explicitly by RESTORE VSR. The instruction CLEAR VR sets the vector interruption index to zero. A specification exception is recognized if the instruction RESTORE VSR attempts to place a value in the vector-interuption-index field which exceeds the section size.

Programming Notes:

1. Since the vector interruption index is always set to zero upon completion of any instruction which depends on it, the program normally need not be concerned with setting its value.

2. The vector interruption index may be set to zero explicitly by use of the instruction CLEAR VR with a zero operand.

3. If it is desired to operate on a vector in a vector register starting at other than element location 0, this may be done by first setting the vector interruption index (vix) to the initial element number. The vix may be set by using the instruction SAVE VSR to place the current contents of the vector-status register (VSR) in storage, placing the initial element number in the field which corresponds to the VIX, and then returning the result to the VSR by means of RESTORE VSR. Such modification of the VSR can be performed safely when the CPU is in the problem state. If a program modifying the VSR is to be executed in the supervisor state, however, additional precautions may have to be taken; see the programming notes in the section “Vector Change Bits” on page 2-5.

Vector In-Use Bits

The eight vector in-use bits (viiu), bits 48-55 of the vector-status register, correspond to the eight vector-register pairs 0, 2, 4, 6, 8, 10, 12, and 14.

The vector in-use bits indicate which vector-register pairs are to be saved and restored by SAVE VSR and RESTORE VR. These instructions ignore vector-register pairs for which the vector in-use bit is zero.

During execution of instructions which use the vector registers, the vector in-use bit associated with a vector-register pair is set to one whenever any element in either or both of the registers is loaded or modified. When a register is used as the source of an operand, its vector in-use bit remains unchanged.

The vector in-use bits are set by the instruction RESTORE VSR. If that instruction changes a vector in-use bit from one to zero, it causes the corresponding vector-register pair to be cleared to zeros. A vector in-use bit is set to zero when the instruction CLEAR VR clears the corresponding vector-register pair to zeros.
See the section “Program Switching” on page 2-26 for a discussion of the vector in-use bits.

**Vector Change Bits**

The eight vector change bits (VCH), bits 56-63 of the vector-status register, correspond to the eight vector-register pairs 0, 2, 4, 6, 8, 10, 12, and 14.

The vector change bits indicate which vector-register pairs are to be saved by the privileged instruction SAVE CHANGED VR. That instruction saves a vector-register pair if the corresponding vector change bit is one; it then sets the vector change bit to zero.

If the vector in-use bit associated with a vector-register pair is set to zero by the instruction CLEAR VR or RESTORE VSR, the corresponding vector change bit is also set to zero.

During execution of an instruction which uses the vector registers, the vector change bit associated with a vector-register pair is set to one whenever any element in either or both of the registers is loaded or modified. An exception is the instruction RESTORE VR; when the CPU is in the supervisor state, execution of RESTORE VR leaves the vector change bits unchanged.

When a vector register is used as the source of an operand, its vector change bit remains unchanged.

See the section “Program Switching” on page 2-26 for further discussion of the vector change bits.

**Programming Notes:**

1. The vector change bit is always zero when the vector in-use bit is zero. When the vector change bit is set to one, the vector in-use bit is also set to one.

2. As pointed out in the section “Program Switching” on page 2-26, vector change bits are intended for use by control programs operating in the supervisor state. When the CPU is in the problem state, the value of the vector change bits stored by SAVE VSR is undefined; problem-state programs should, therefore, not depend on the value of these bits.

A program operating in the problem state cannot set a vector change bit to zero, except by also setting the corresponding in-use bit to zero (clearing the vector-register pair). In the problem state, the instruction RESTORE VSR sets the vector change bit to one for every pair of vector registers whose in-use bit is set to one.

3. If a program uses the instruction RESTORE VSR to modify the contents of the vector-status register while the CPU is in the supervisor state, and the program is subject to interruptions for which the interruption handler may cause a SAVE CHANGED VR instruction to be executed, care must be taken to ensure that the vector change bits reflect all modifications of the active vector registers. A safe procedure is to supply ones in all bit positions of the operand of RESTORE VSR which correspond to the vector change bits. This precaution is unnecessary in the problem state, because RESTORE VSR then sets the vector change bits to ones regardless of the operand.

4. A program operating in the supervisor state can depend on the accuracy of vector change bits that are zeros. When the program is a guest in a virtual-machine environment, however, vector change bits may be overindicated, so that a bit may be set to one even when the corresponding vector-register pair has not been changed.

**Vector-Activity Count**

The vector-activity count (VAC) provides a means for measuring and scheduling the machine resources used in executing instructions of the vector facility.

The vector-activity count has this format:

```
00000000
0  8  63
```

Bits 8-63 are a 56-bit unsigned binary integer. In the basic form, this integer is incremented by adding a one in bit position 51 every microsecond while a vector-facility instruction is being executed. In models having a higher or lower resolution, a different bit position is incremented at such a frequency that the rate of incrementing the vector-activity count is the same as if a one were added in bit position 51 every microsecond during those periods. Bits 0-7 are zeros.

The contents of the vector-activity count may be obtained by executing the privileged instruction.
SAVE VAC, and they may be set by means of the privileged instruction RESTORE VAC. Bits 0-7, and any rightmost bit positions which are not incremented, are stored as zeros by SAVE VAC and are ignored by RESTORE VAC.

When incrementing the vector-activity count causes a carry to be propagated out of bit position 8, the carry is ignored, and counting continues from zero. The program is not alerted, and no interruption occurs as a result of the overflow. Except for such wraparound, or an explicit restore or reset operation, the value of the count never decreases.

The vector-activity count is not incremented during execution of the instructions RESTORE VAC and SAVE VAC. In addition, depending on the model, the count may not be incremented during execution of some other short, uninterruptible instructions of the vector facility.

The vector-activity count is incremented only when the CPU is in the operating state.

Programming Notes:

1. The vector-activity count is not intended to be a precise measure of vector execution time. The count may or may not advance during the execution of a particular vector-facility instruction. In the aggregate, however, the count reflects the execution time of the vector portion of normal application programs.

2. The format of the vector-activity count has been chosen to permit the use of unnormalized scalar floating-point instructions to perform fast addition and subtraction of VAC values.

Modes of Operation

The operation of the vector facility is independent of the architectural mode, except for the range of storage addresses which can be specified. The 370-XA architectural mode provides the choice of operating in either a 31-bit or 24-bit addressing mode; the System/370 architectural mode does not.

On a CPU which provides both the 370-XA and System/370 modes, vector operations in the System/370 mode are the same as in the 370-XA mode when in the 24-bit addressing mode. Thus, an address size of 24 bits is available in either the 370-XA or System/370 mode, but vector operations with an address size of 31 bits can be performed only in the 370-XA mode. In the System/370 mode, instructions of the vector facility may be executed in both the EC and BC modes.

In both the 370-XA and System/370 modes, vector operations are governed by the vector-control bit.

Vector-Operation Control

When the vector facility is installed and available on a CPU, execution of vector-facility instructions can be completed only if bit 14 of control register 0, the vector-control bit, is one. Executing a vector-facility instruction when the vector-control bit is zero causes a vector-operation exception to be recognized and a program interruption to occur. The initial value of the vector-control bit is zero.

When the vector facility is not installed or not available on this CPU but is installed on any other CPU which is or can be placed in the configuration, executing a vector-facility instruction causes a vector-operation exception to be recognized regardless of the state of the vector-control bit.

If the vector facility is not installed on any CPU which is or can be placed in the configuration, it depends on the model whether executing a vector-facility instruction causes a vector-operation exception or an operation exception to be recognized.

A vector facility, though installed, is considered not available when it is not in the configuration, when it is in certain maintenance modes, or when its power is off.

Figure 2-2 on page 2-7 summarizes the effect of the vector-control bit according to whether the vector facility is installed and whether vector instructions can be executed by the program.
Vector Facility on This CPU

<table>
<thead>
<tr>
<th>Vector Facility Installed on Another CPU</th>
<th>In-stalled</th>
<th>Available</th>
<th>Effect of Vector-Facility Instruction</th>
<th>VC = 0</th>
<th>VC = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes or No</td>
<td>Yes</td>
<td>Yes</td>
<td>VOP</td>
<td>Execute</td>
<td>VOP</td>
</tr>
<tr>
<td>Yes or No</td>
<td>Yes</td>
<td>No</td>
<td>VOP</td>
<td>VOP</td>
<td>VOP</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>(NA)</td>
<td>VOP</td>
<td>VOP or</td>
<td>VOP or</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>(NA)</td>
<td>OP</td>
<td>OP</td>
<td>OP</td>
</tr>
</tbody>
</table>

Explanation:
- NA: Not applicable
- OP: Operation exception
- VC: Vector-control bit (control register 0, bit 14)
- VOP: Vector-operation exception

Figure 2-2. Vector Control

Programming Notes:

1. The control program may use the vector-control bit to defer enabling of the CPU for vector operations and to delay allocation of a vector-save area until a program attempts to use the facility by executing its first vector instruction. Because the resulting vector-operation exception nullifies the operation, the instruction address does not need to be adjusted in order to resume the program.

2. The control program may also keep the vector-control bit set to zero to prevent a program from examining or changing the contents of the vector-facility registers. This may be useful when a program that does not use the vector facility is to be run after a program that does use the facility has been interrupted. If the next program to use the vector registers is the original program, then running the intervening program with the vector-control bit set to zero may eliminate the need for information held in the vector facility to be saved and later restored.

A possible exception is the vector-activity count (VAC). When the vector-control bit is zero, the VAC may or may not be incremented during the brief period of detecting that an instruction requires the vector-operation exception to be recognized. The number of times that the VAC might be stepped in this way is small, however, compared to the counts accumulated during execution of a vector-application program.

3. When a machine check indicating vector-facility failure occurs, the machine has made a previously available vector facility unavailable. Until the cause of the failure is removed and the facility is made available again, attempting to execute a vector instruction causes a vector-operation exception to be recognized even though the vector-control bit is one. (See the section "Vector-Facility Failure" on page 2-30.)

Vector-Instruction Operands and Results

The vector facility provides for operations on vectors of short (32-bit) and long (64-bit) floating-point numbers, 32-bit signed binary integers, and 32-bit logical data. A few operations deal with vectors of 16- and 64-bit signed binary integers. There are also operations on vectors of individual bits, which are generally used as mask bits.

All binary-arithmetic vector operations treat elements of 32-bit binary integers as signed; any fixed-point-overflow exceptions are recognized. Binary-comparison operations also deal with 32-bit signed binary integers. Logical vector operations, including shifts, treat elements as 32-bit logical data.

Most instructions which operate on floating-point, binary-integer, or logical vectors use a format that explicitly designates three operands: two source operands and one target operand. The operands may be:

- In storage,
- In a vector register, or a pair of vector registers, or
- In a scalar (general or floating-point) register.

Instructions which use mask bits generally designate an implicit operand in the vector-mask register, and they also may explicitly designate storage, vector-register, and scalar-register operands.
All vector operands in storage must be aligned on integral boundaries. When an instruction requires boundary alignment and the storage operand is not designated on the appropriate boundary, a specification exception is recognized.

An instruction which processes operands in vector or scalar registers must designate a valid register number for each such operand. If an invalid register number is designated, a specification exception is recognized.

Figure 2-3 on page 2-9 summarizes the vector-data formats, the associated operations, and the boundary-alignment and register-number requirements.

Vectors of 16-, 32-, and 64-bit elements containing arithmetic or logical data are collectively referred to as arithmetic vectors. Arithmetic vectors in storage must be on integral boundaries. The elements of arithmetic vectors have the same formats as scalar data of the same data type.

Vectors of individual bits are referred to as bit vectors (see the section "Bit Vectors" on page 2-10).

Programming Note: Logical-data elements may also be considered as 32-bit unsigned binary integers, but no arithmetic or comparison operations are provided to process such vectors.

**Arithmetic Vectors in Storage**

Arithmetic vectors in storage may be loaded and stored in one of two ways:

- By sequential addressing (contiguously or with stride)
- By indirect element selection

Most arithmetic, comparison, and logical instructions may also access one of the vector operands directly from storage by sequential addressing. Indirect element selection is available only for load and store operations.

**Access by Sequential Addressing**

Vector elements are most often accessed in storage in a regular sequence of addresses. The instruction specifies a general register containing the starting address and, optionally, another general register containing the stride. The stride, which is a 32-bit signed binary integer, is the number of element locations by which the operation advances when proceeding from one element to the next. The maximum number of elements to be accessed is specified by the vector count.

A stride of one specifies a contiguous vector, for which successive elements are in adjacent storage locations; this stride is the default when no general register is specified for the stride. A stride of zero causes the same element to be used repeatedly as the storage operand. A negative stride causes elements to be accessed in a descending sequence of addresses.

During the execution of instructions which access an arithmetic vector in storage sequentially, the starting address contained in the general register is updated as successive elements in storage are accessed. At the end of instruction execution, or at the time of any interruption, the contents of the general register have been updated to the storage address of the next vector element due to be processed if instruction execution had not ended or been interrupted. Likewise, when instructions process a bit vector in storage, the starting address in the general register is updated by the number of bytes accessed during execution.

Such automatic updating of vector addresses is used to process a vector in sections when the vector has more elements than will fit into a vector register. It also assists in resuming instruction execution after an interruption.

For more details on sequential addressing, see the section "Class-IM and Class-IC Instructions" on page 2-15. For more information on sectioning, see the section "Vector Sectioning" on page 2-11.

Programming Note: A contiguous vector is implied when zero is specified in the instruction field that designates the general register containing the stride. This differs from a zero stride, which is specified by placing a value of zero in the general register containing the stride, and which causes reuse of the same element in storage. A zero stride is generally not desired because the scalar form of an instruction is usually faster than repeated use of the same storage location. (See the section "Operands in Scalar Registers" on page 2-10.)
<table>
<thead>
<tr>
<th>Data Type</th>
<th>Width in Bits</th>
<th>Alignment Required In Storage</th>
<th>Valid Register Numbers</th>
<th>Scalar Register</th>
<th>Vector Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating point</td>
<td>1</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>Short</td>
<td></td>
<td>A</td>
<td>Word</td>
<td>Even FR</td>
<td>Any VR</td>
</tr>
<tr>
<td>Long</td>
<td></td>
<td>A</td>
<td>Doubleword</td>
<td>Even FR</td>
<td>Even VR</td>
</tr>
<tr>
<td>Binary integer</td>
<td></td>
<td>S</td>
<td>Halfword</td>
<td>--</td>
<td>Any VR</td>
</tr>
<tr>
<td>16-bit signed</td>
<td></td>
<td>B</td>
<td>Word</td>
<td>Any GR</td>
<td>Any VR</td>
</tr>
<tr>
<td>32-bit signed</td>
<td></td>
<td>P</td>
<td>--</td>
<td>--</td>
<td>Even VR</td>
</tr>
<tr>
<td>64-bit signed</td>
<td></td>
<td>L</td>
<td>Word</td>
<td>Any GR</td>
<td>Any VR</td>
</tr>
<tr>
<td>Logical</td>
<td></td>
<td>M</td>
<td>Byte</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

Explanation:

- -- Does not apply
- A All arithmetic, load, and store operations
- B Some arithmetic and all load and store operations
- FR Floating-point register
- GR General register
- L Logical and shift operations
- M Logical operations on bits in storage and in vector-mask register; comparison results
- P 64-bit binary integers, which occur only as the result of a binary multiply operation
- S Only load and store operations, which convert between 16 bits in storage and 32 bits in a vector register
- VR Vector register

Figure 2-3. Types of Vector Data

**Access by Indirect Element Selection**

Indirect element selection permits vector elements to be loaded or stored in an arbitrary sequence. With the instructions used for indirect element selection, LOAD INDIRECT and STORE INDIRECT, the locations of the individual operand elements to be loaded or stored are designated by a vector of element numbers in a vector register. Each such element number indicates the position of the corresponding operand element relative to the start of the operand vector. The number of operand elements accessed, which is also the number of element numbers used for indirect element selection, is equal to or less than the vector count.

The element numbers used for indirect element selection are 32-bit signed binary integers. They may be positive, negative, repeated, and in any order. Successive operand elements are located in storage at addresses $A + w \times E(0)$, $A + w \times E(1)$, $A + w \times E(2)$, ..., where $A$ is the origin of the operand vector in storage, $w$ is the width in bytes (4 or 8) of each element, and $E(0)$, $E(1)$, $E(2)$, ... are the successive element numbers in a vector register.

General-register address updating does not apply to the instructions LOAD INDIRECT and STORE INDIRECT.

**Programming Notes:**

1. For a discussion of address updating, see the programming notes under "Vector Sectioning" on page 2-11.

2. Vectors of element numbers may be stored as 16-bit signed binary integers when the element numbers remain within the range of such integers. The vector instructions LOAD HALFWORD and STORE HALFWORD perform the conversion between the 16-bit and 32-bit formats.
3. Accessing vectors in storage in the arbitrary sequence permitted by indirect element selection may be significantly slower than accessing contiguous vector elements.

**Arithmetic Vectors in Registers**

**Operands in Vector Registers**

Any vector register can be designated for a vector of short floating-point numbers, 32-bit signed binary integers, or 32-bit logical data. Even-odd vector-register pairs are coupled to hold long floating-point numbers or the 64-bit signed binary integers which result from binary multiplication.

When a vector register is modified, those elements in the vector register beyond the last element to be modified are left unchanged.

Most operations on floating-point, binary, or logical vectors which may be performed with one vector operand in storage and one operand in a vector register may also be performed with both operands in vector registers. When both operands are in vector registers, the corresponding pairs of elements from each vector-register operand generally have the same element number (but see the descriptions of ACCUMULATE and MULTIPLY AND ACCUMULATE for an exception to this rule).

**Operands in Scalar Registers**

Operations on floating-point, binary, or logical vectors may specify as one source operand the contents of a scalar register, that is, of a floating-point or general register, the other operand being a vector. This scalar operand is used repeatedly and treated as a vector of identical elements of the same length as the vector operand.

Some vector instructions which obtain one of the source operands from a scalar register also produce a scalar result, which replaces the contents of the same scalar register.

**Bit Vectors**

A group of bits in contiguous bit positions is called a bit vector. Bit vectors are the operands of logical operations where one of the operands is in the vector-mask register. They are used in operations on arithmetic vectors under mask control.

A bit vector in storage must begin on a byte boundary, but it may end at any bit position, the remaining bits of the rightmost byte being ignored. When the instruction STORE VMR stores a bit vector with the vector count specifying a number of bits that is not a multiple of 8, the final byte stored is padded on the right with zeros.

When used for the control of load and store operations or for arithmetic and logical operations in the vector-mask mode, the appropriate bit vector must first be placed in the vector-mask register. Each bit in the vector-mask register corresponds sequentially, one for one, to an element of one or both of the vector-register operands.

Bit vectors in the vector-mask register are generated or altered by the following vector instructions:

AND TO VMR
COMPARE
COMPLEMENT VMR
EXCLUSIVE OR TO VMR
LOAD VMR
LOAD VMR COMPLEMENT
OR TO VMR

**Programming Notes:**

1. Examples of the use of bit vectors for mask control are shown in Appendix A.

2. Since the section size is a multiple of 8 and bit vectors start on a byte boundary, every section of a bit vector also starts on a byte boundary. Thus, after an instruction has completed processing a full section of bits, the next bit is always the leftmost bit of the byte specified by the updated address.

3. When a bit vector is used as a mask to identify selected elements of an arithmetic vector with one bits and the remaining elements with zero bits, the bit vector is logically equivalent to a vector containing a set of element numbers in ascending sequence, which may be used for indirect selection of the arithmetic-vector elements. The vector of element numbers consists merely of the bit indexes (bit numbers) of the one bits in the bit vector.

A bit vector may be converted to a vector of element numbers by the instruction LOAD BIT INDEX. This instruction operates directly on a bit vector in storage and produces a vector of element numbers in a vector register; the vector-mask register is not used.
Vector Sectioning

Vector sectioning is a programming technique for processing vectors the length of which may exceed the section size. Such vectors are processed by dividing them into smaller sections and using a loop of instructions, referred to as a sectioning loop, which repeats the appropriate sequence of instructions for all consecutive sections of the specified vectors. To assist with such sectioning, addresses of vector operands in storage and bit-vector parameters are automatically updated, and the instruction LOAD VCT AND UPDATE is provided.

The LOAD VCT AND UPDATE instruction specifies a general register that has initially been loaded with the total number of vector elements to be processed. The instruction sets the vector count to the lesser of the section size and the general-register contents. It also subtracts this value from the current contents of the general register, which then contains the number of elements remaining to be processed during subsequent passes through the sectioning loop.

LOAD VCT AND UPDATE sets the condition code to provide the program with an indication of whether a complete vector has been processed. The program may use the instruction BRANCH ON CONDITION for loop control to repeat the sequence of instructions for each section. A sectioning loop may also be closed by testing the residual count in the general register for zero and branching back to the start of the loop if not zero.

For most vector operations, the program can be written such that sectioning is independent of the section size. There are occasions, however, when knowledge of the actual section size is desirable; this value is available to the program by executing the instruction STORE VECTOR PARAMETERS.

Programming Notes:

1. Examples of sectioning are shown in Appendix A.

2. One method of controlling the vector count for sectioning is to place the instruction LOAD VCT AND UPDATE at the beginning of the loop and an appropriate BRANCH ON CONDITION instruction at the end of the loop. This is usually sufficient because most vector-facility instructions do not set the condition code. If the sectioning loop does contain an instruction that modifies the condition code, the final BRANCH ON CONDITION instruction could be preceded by a LOAD AND TEST instruction to test the general register containing the residual vector count.

Appendix A also illustrates other techniques.

3. If a sectioning loop contains more than one reference to the same vector in storage, such as a load followed later by a store, the program must ensure, by retaining a copy of the current address, that all addresses within the loop which specify the same vector refer to the same section.

4. The instructions which provide indirect element selection, LOAD INDIRECT and STORE INDIRECT, progress one section of element numbers at a time. But sectioning of the vector of element numbers used for addressing is performed by a preceding instruction which loaded or generated the element numbers by means of sequential addressing. The indirect-selection instructions themselves do not provide for address updating. Each element address is computed separately from an element number and from the specified starting address, which remains unchanged.

Conditional Arithmetic

Vector-Mask Mode

The vector-mask mode allows for conditional execution of arithmetic and logical instructions, depending on the mask bits in the vector-mask register.

When the vector-mask mode is in effect, operand elements are processed if they are in positions which correspond to mask bits that are ones. In positions which correspond to zero mask bits, the target locations remain unchanged, no arithmetic or operand-access exceptions are recognized for those positions, the corresponding change bits in storage remain unchanged, and no PER event for storage alteration is indicated. When the vector-mask mode is not in effect, the mask bits are ignored, and all active elements are processed.

The arithmetic and logical vector instructions which are under the control of the vector-mask mode are:
Except for LOAD COMPLEMENT, LOAD NEGATIVE, and LOAD POSITIVE, which are considered arithmetic instructions for this purpose, load and store instructions are not controlled by the vector-mask mode; neither are instructions which modify the vector-mask register, such as COMPARE. The instructions LOAD EXPANDED, LOAD MATCHED, STORE COMPRESSED, and STORE MATCHED do depend on the vector-mask register for their execution, but this is independent of the mode setting.

For more details, see the section “Class-IM and Class-IC Instructions” on page 2-15.

Instructions Controlling the Vector-Mask Mode

The instruction SET VECTOR MASK MODE turns the vector-mask mode on or off. EXTRACT VECTOR MASK MODE places the current value of the mode in a general register.

Programming Notes:

1. The vector-mask mode is useful when arithmetic vector operations depend on the result of a vector comparison. Only elements which are to be processed are subject to arithmetic and access exceptions.

2. Since loading, comparing, and storing are operations which are not subject to the vector-mask mode, it is frequently possible to leave the vector-mask mode in effect while performing the arithmetic for an entire sectioning loop.

Common Instruction Descriptions

Many vector-facility instructions have common characteristics and obey common rules for accessing the elements of their vector operands. This section describes the common aspects, which are not repeated in individual instruction descriptions.

Some instructions contain fields that vary slightly from the basic format, and in some instructions, the operation performed does not follow the general rules stated in this section. Any exceptions to these rules are noted in the individual instruction descriptions, as are the rules for instruction formats and types not covered in this section.

The rules are grouped according to instruction classes and formats.

Programming Note: Many load and all store operations on vectors are the same for binary and short floating-point operands, so that only a single set of operation codes is provided for them. However, for programming convenience, both binary and short floating-point mnemonics are assigned to these operation codes.

Separate operation codes are provided for short floating-point and binary operands when the operation must distinguish between floating-point and general registers, as in loading or extracting an element, or when the operation depends on the data type, such as LOAD COMPLEMENT.

Instruction Classes

Vector-facility instructions are classified into one of nine classes: 1M, 1C, 1G, 1P, 1Z, NC, NZ, N1, and NO. The properties of these nine instruction classes are summarized in Figure 2-4 on page 2-13.
Instruction Formats

The instruction formats used by vector-facility instructions are shown in Figure 2-5 on page 2-14. The first four are the base formats — QST, QV, VST, and VV, where Q indicates that the format provides for a scalar-register operand, ST indicates a storage operand (with stride), and V indicates a vector-register operand. Most of the arithmetic instructions are available in all four of these base formats. For the vector-comparison instructions, the VR1 field of the base formats is interpreted as a modifier (M1).

Bit positions which are shown in instruction formats as shaded (////) are unassigned.

Field Designations

The field designations in the instruction formats indicate the use of the field and the type of operation in which the field participates.

B2 and D2 Fields: B2 designates a base register, and D2 is a displacement. They are used for addressing in the same way as with scalar instructions.

FR3 Field: FR3 designates a (scalar) floating-point register. It is a more specific description of the QR3 field used in some instruction descriptions, and the same rules and restrictions apply as for QR3.

GR Field: GR designates a (scalar) general register or a pair of general registers. Unless otherwise indicated in the individual instruction descriptions, the contents of the general registers designated by the GR1 and GR2 fields are called the first operand and second operand, respectively. When designating the third operand (GR3), it is a more specific indication of the QR3 field used in some instruction descriptions, and the same rules and restrictions apply as for QR3.

QR3 Field: QR3 designates a scalar register, with the operation code determining whether it is a floating-point or general register. In the QST format, the QR3 field must not designate a general register which is the same as that designated by the RS2 field; otherwise, a specification exception is recognized. For instructions in the QV or VR formats with only two operands, one a vector and
<table>
<thead>
<tr>
<th>First Halfword</th>
<th>Second Halfword</th>
<th>Third Halfword</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Base Formats**

**QST Format**

<table>
<thead>
<tr>
<th>Op Code</th>
<th>QR3</th>
<th>RT2</th>
<th>VR1</th>
<th>RS2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

**QV Format**

<table>
<thead>
<tr>
<th>Op Code</th>
<th>QR3</th>
<th>VR1</th>
<th>VR2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>20</td>
<td>24</td>
</tr>
</tbody>
</table>

**VST Format**

<table>
<thead>
<tr>
<th>Op Code</th>
<th>VR3</th>
<th>RT2</th>
<th>VR1</th>
<th>RS2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

**VV Format**

<table>
<thead>
<tr>
<th>Op Code</th>
<th>VR3</th>
<th>VR1</th>
<th>VR2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>20</td>
<td>24</td>
</tr>
</tbody>
</table>

**Other Formats**

**RRE Format**

<table>
<thead>
<tr>
<th>Op Code</th>
<th>GR1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

**RSE Format**

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R3</th>
<th>VR1</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>32</td>
</tr>
</tbody>
</table>

**S Format**

<table>
<thead>
<tr>
<th>Op Code</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

**VR Format**

<table>
<thead>
<tr>
<th>Op Code</th>
<th>QR3</th>
<th>VR1</th>
<th>GR2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>20</td>
<td>24</td>
</tr>
</tbody>
</table>

**VS Format**

<table>
<thead>
<tr>
<th>Op Code</th>
<th>RS2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

Figure 2-5. Vector-Facility Instruction Formats

One a scalar, the scalar operand is called the second operand and is designated by a QR3 field.

**R3 Field:** R3 is shown in individual instruction descriptions as either VR3, to designate a vector register, or GR3, to designate a general register.

**RS2 Field:** RS2 designates a general register containing a storage-operand address. The address is updated during execution. The RS2 field must not designate the same general register as the RT2 field or, in the QST format, as the QR3 field.

**RT2 Field:** RT2 designates a general register containing a stride. The field cannot designate general register 0; if the RT2 field is zero, a stride of 1 is specified. It also must not designate the same general register as the RS2 field.
VR Field: VR designates a vector register or a pair of vector registers. The VR₁, VR₂, and VR₃ fields designate the first, second, and third operands, respectively, in vector registers or pairs of vector registers, as required for the data type specified by the operation code.

Three-Operand Instruction Formats

All nonstore vector instructions which explicitly specify three operands in the QST, QV, RSE, VST, and vv formats use the first-operand location as the target for the result and the second- and third-operand locations for the source operands. These three-operand operations may be shown symbolically as:

Operand 1 = Operand 3 • Operand 2

where • represents an arithmetic or logical operation. Operand 1 is always in vector registers. Operand 2 is in storage or in vector registers. Operand 3 is either in vector registers or in a scalar register. An instruction may specify the same or different vector registers for the target and source operands.

Vector-comparison instructions are similar to these three-operand instructions, except that they designate a modifier (M₁) instead of a first operand (VR₁), and they place the result in the vector-mask register.

Programming Note: The base address of a storage operand is placed in a general register designated by the following: the RS₂ field in the QST, VS, and VST formats; the GR₁ field in the RRE format; and the B₂ field in the RSE and s formats. The RS₂ and GR₁ fields may designate general register 0; but the B₂ field cannot, since a zero B₂ field specifies that the base address itself is zero. Use of general register 0 for storage addresses should, in general, be avoided to keep storage addressing consistent among all instruction formats for both vector and scalar instructions.

Summary of Instructions by Class and Format

Figure 2-6 on page 2-16 briefly lists all instructions of the vector facility according to class and format within the class.

Class-1M and Class-IC Instructions

Most vector instructions are in either class 1M or 1C. Instructions in both classes are interruptible, the number of elements processed is determined by the vector count, and they depend on the vector interruption index. Class-1M instructions are also under the control of the vector-mask mode; class-1C instructions are independent of the vector-mask mode.

For both classes, the elements of each operand are processed in sequence from element X, where X is the initial value of the vector interruption index (normally zero), to C−1, where C is the vector count.

The number of elements that are processed for each operand is called the net count. If C is greater than X, then the net count is C−X; otherwise the net count is zero. For vector instructions which combine vector operands with a scalar operand, the scalar operand is considered to be replicated as many times as indicated by the net count.

If the net count is zero at the start of instruction execution, the vector interruption index is set to zero, and execution is completed immediately. No elements are processed, no operand-access exceptions occur, the change bits for any storage operand remain unchanged, and no PER event for storage alteration is indicated. Operands in vector, floating-point, and general registers that are due to be modified, the vector in-use bits, and the vector change bits remain unchanged.

If the instruction is interrupted during execution, Y−X pairs of elements have been processed, where X and Y are the values of the vector interruption index at the beginning of execution and at the time of interruption, respectively. Y is then the element number of the next element, if any, to be processed for each operand.

When a class-1M or class-1C instruction designates a scalar register as the location of the third operand (in the QST or QV format), and the scalar register is a floating-point register, the instruction must designate register 0, 2, 4, or 6 in the third-operand field; otherwise, a specification exception is recognized.
<table>
<thead>
<tr>
<th>Instructions</th>
<th>Class</th>
<th>Long</th>
<th>Short</th>
<th>Binary</th>
<th>Other</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD, SUBTRACT</td>
<td>IM</td>
<td>Four^1</td>
<td>Four^1</td>
<td>Four^1</td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>AND, EXCLUSIVE OR, OR</td>
<td>IM</td>
<td>Four^1</td>
<td>Four^1</td>
<td>Four^1</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>DIVIDE</td>
<td>IM</td>
<td>Four^1</td>
<td>Four^1</td>
<td>Four^1</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>MULTIPLY</td>
<td>IM</td>
<td>Four^1</td>
<td>Four^1</td>
<td>Four^1</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>MULTIPY AND ADD</td>
<td>IM</td>
<td>QST/QV/VST</td>
<td>QST/QV/VST</td>
<td>QST/QV/VST</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>MULTIPY AND SUBTRACT</td>
<td>IM</td>
<td>QST/QV/VST</td>
<td>QST/QV/VST</td>
<td>QST/QV/VST</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>MULTIPY AND ACCUMULATE</td>
<td>IM</td>
<td>VST/VV</td>
<td>VST/VV^2</td>
<td>VST/VV^2</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>ACCUMULATE</td>
<td>IM</td>
<td>VST/VV</td>
<td>VST/VV^2</td>
<td>VST/VV^2</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>LOAD COMPLEMENT</td>
<td>IM</td>
<td>VV</td>
<td>VV</td>
<td>VV</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>LOAD NEGATIVE, LOAD POSITIVE</td>
<td>IM</td>
<td>VV</td>
<td>VV</td>
<td>VV</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>SHIFT LEFT SINGLE LOGICAL</td>
<td>IM</td>
<td>RSE</td>
<td>RSE</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>SHIFT RIGHT SINGLE LOGICAL</td>
<td>IM</td>
<td>RSE</td>
<td>RSE</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>MAXIMUM ABSOLUTE</td>
<td>IM</td>
<td>VR</td>
<td>VR</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>MAXIMUM SIGNED, MINIMUM SIGNED</td>
<td>IM</td>
<td>VR</td>
<td>VR</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>COMPARE</td>
<td>IC</td>
<td>Four^1</td>
<td>Four^1</td>
<td>Four^1</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>LOAD, LOAD MATCHED</td>
<td>IC</td>
<td>QV/VST/VV</td>
<td>QV/VST^2/VV^3</td>
<td>QV</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>STORE, STORE MATCHED</td>
<td>IC</td>
<td>VST</td>
<td>VST^2</td>
<td>VST</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>LOAD EXPANDED, STORE COMPRESSED</td>
<td>IC</td>
<td>VST</td>
<td>VST^2</td>
<td>VST</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>LOAD INTEGER VECTOR</td>
<td>IC</td>
<td>VST</td>
<td>VST</td>
<td>VST</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>LOAD HALFWORD, STORE HALFWORD</td>
<td>IC</td>
<td>VV</td>
<td>VV^2</td>
<td>VST</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>LOAD ZERO</td>
<td>IC</td>
<td>RSE</td>
<td>RSE^2</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>LOAD INDIRECT, STORE INDIRECT</td>
<td>IC</td>
<td>RSE</td>
<td>RSE^2</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>LOAD BIT INDEX</td>
<td>IG</td>
<td>RSE</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>SUM PARTIAL SUMS</td>
<td>IP</td>
<td>VR</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>ZERO PARTIAL SUMS</td>
<td>IP</td>
<td>VR</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>RESTORE VR</td>
<td>IZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAVE VR, SAVE CHANGED VR</td>
<td>IZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESTORE VSR</td>
<td>IZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLEAR VR</td>
<td>IZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2-6 (Part 1 of 2). Summary of Vector-Facility Instructions by Class and Format

**Class-IM Instructions**

For instructions in class IM, all elements are processed as described above when the vector-mask mode is off. When the vector-mask mode is on, however, operand elements are fetched from storage or from operand registers, and result elements are placed in the target register, only for those elements which correspond to ones in the vector-mask register. Element positions in the target register corresponding to zeros remain unchanged; no arithmetic or operand-access exceptions are recognized for those positions, the corresponding change bits in storage remain unchanged, and no PER event for storage alteration is indicated.

The first mask bit used, when the vector-mask mode is on, is bit X of the vector-mask register, which corresponds to vector-register element X. The last mask bit and vector-register element processed are numbered C−1, if the instruction is completed, or Y−1, if the instruction is interrupted during execution.

Class-IM instructions in the QST and VST formats have the storage address in the RSt register updated during execution for every element posi-
Table 2-6. Summary of Vector-Facility Instructions by Class and Format

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Class</th>
<th>Long</th>
<th>Short</th>
<th>Binary</th>
<th>Other</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT LEFT ZEROS IN VMR</td>
<td>NC</td>
<td></td>
<td></td>
<td></td>
<td>RRE</td>
<td>1</td>
</tr>
<tr>
<td>COUNT ONES IN VMR</td>
<td>NC</td>
<td></td>
<td></td>
<td></td>
<td>RRE</td>
<td>1</td>
</tr>
<tr>
<td>COMPLEMENT VMR, TEST VMR</td>
<td>NC</td>
<td></td>
<td></td>
<td></td>
<td>RRE</td>
<td>2</td>
</tr>
<tr>
<td>AND TO VMR, EXCLUSIVE OR TO VMR</td>
<td>NC</td>
<td></td>
<td></td>
<td></td>
<td>VS</td>
<td>2</td>
</tr>
<tr>
<td>LOAD VMR, LOAD COMPLEMENT VMR</td>
<td>NC</td>
<td></td>
<td></td>
<td></td>
<td>VS</td>
<td>2</td>
</tr>
<tr>
<td>OR TO VMR, STORE VMR</td>
<td>NC</td>
<td></td>
<td></td>
<td></td>
<td>VS</td>
<td>2</td>
</tr>
<tr>
<td>RESTORE VMR, SAVE VMR</td>
<td>NZ</td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td>2</td>
</tr>
<tr>
<td>EXTRACT ELEMENT, LOAD ELEMENT</td>
<td>N1</td>
<td>VR</td>
<td>VR</td>
<td>VR</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>EXTRACT VCT</td>
<td>N0</td>
<td></td>
<td></td>
<td></td>
<td>RRE</td>
<td>1</td>
</tr>
<tr>
<td>EXTRACT VECTOR MASK MODE</td>
<td>N0</td>
<td></td>
<td></td>
<td></td>
<td>RRE</td>
<td>1</td>
</tr>
<tr>
<td>LOAD VCT AND UPDATE</td>
<td>N0</td>
<td></td>
<td></td>
<td></td>
<td>RRE</td>
<td>1</td>
</tr>
<tr>
<td>LOAD VCT FROM ADDRESS</td>
<td>N0</td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td>1</td>
</tr>
<tr>
<td>RESTORE VAC, SAVE VAC, SAVE VSR</td>
<td>N0</td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td>3</td>
</tr>
<tr>
<td>SET VECTOR MASK MODE</td>
<td>N0</td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td>1</td>
</tr>
<tr>
<td>STORE VECTOR PARAMETERS</td>
<td>N0</td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td>1</td>
</tr>
<tr>
<td><strong>Totals</strong></td>
<td></td>
<td>53</td>
<td>51</td>
<td>41</td>
<td>26</td>
<td>171</td>
</tr>
</tbody>
</table>

**Explanation:**

1. Four instruction formats are provided: QST, QV, VST, and VV.
2. Operand 1 is in the long format; operands 2 and 3 are in the short format.
3. Instruction in this format may be used for both short and binary operands.

Figure 2-6 (Part 2 of 2). Summary of Vector-Facility Instructions by Class and Format

tion, regardless of whether the corresponding mask bit is one or zero (see the section “Storage Operands for QST and VST Formats” on page 2-17).

**Class-IC Instructions**

Execution of instructions in class IC is independent of the vector-mask mode. The following instructions depend on mask bits in the vector-mask register, but their execution is the same whether the vector-mask mode is on or off: LOAD EXPANDED, LOAD MATCHED, STORE COMPRESSED, and STORE MATCHED. The first mask bit used for those instructions is bit X, corresponding to vector-register element X. The last mask bit and vector-register element processed are numbered C–1, if the instruction is completed, or Y–1, if the instruction is interrupted during execution.

**Storage Operands for QST and VST Formats**

In the QST and VST formats, the RS2 field designates a general register containing the starting address, that is, the address of the first element of the vector operand in storage which is to be processed. The RT2 field, if not zero, designates a general register containing the stride; if the RT2 field is zero, general register 0 is not used, and a stride of one is assumed.

The addresses of successive vector elements in storage are \( A, A+wT, A+2wT, \ldots \), where \( A \) is the starting address, \( T \) is the stride, and \( w \) is the size of each element in bytes. The value of \( w \) is 2, 4, or 8, depending on whether the operation code specifies the storage-operand elements to be halfwords, words, or doublewords.

Each address may be obtained by adding to the previous address the value \( wT \), which is the stride \( T \).
shifted to the left by one, two, or three bit positions. Any carries or ones shifted out of bit position 0 are ignored. Depending on whether the address size is 31 or 24 bits, the rightmost 31 or 24 bits of the sum are used as the storage address, which is also returned to the general register containing the initial address; the leftmost one or eight bit positions, respectively, of the register are set to zeros. The register is thus updated after each unit of operation to hold the address of the next element, whether an element of the storage operand has been accessed or not. All bits in the general register containing the stride take part in the operation, with the contents of the stride register remaining unchanged.

A stride of zero \( (T=0) \) means that the same element location is used repeatedly. When storing with a zero stride, only the last element stored is retained in the addressed location.

A nonzero \( R_{T2} \) field must not designate the same general register as the \( R_{S2} \) field; likewise, the third-operand field of a \( QST \) format instruction must not designate the same general register as the \( R_{S2} \) field. Otherwise, a specification exception is recognized, and the operation is suppressed.

When the net count is not zero and one or more elements have been accessed, the address is updated. The leftmost bits of the \( R_{S2} \) register, depending on the address size, are set to zeros; this is done even when the address remains unchanged because the stride is zero. No storage accesses are made for elements that are skipped when the stride is greater than one.

If no elements are processed because the net count is zero at the start of instruction execution, no access exceptions are recognized for the storage operand. The storage address in the \( R_{S2} \) register and the change bits for the operand remain unchanged, and no PER event for storage alteration is indicated.

When the net count is not zero for the instructions \( \text{LOAD EXPANDED} \) and \( \text{STORE COMPRESSED} \), but no elements are processed because the bits of the vector-mask register which correspond to vector-register element locations to be loaded or stored are all zeros, no operand storage accesses are made, and the storage address in the \( R_{S2} \) register remains unchanged.

In either of these two cases where no elements are processed, it is undefined whether the leftmost \( n \) bit positions of the \( R_{S2} \) register, where \( n \) is 1 or 8 depending on whether the address size is 31 or 24 bits, are set to zeros or remain unchanged.

When class-IM instructions are executed with the vector-mask mode on, no access exceptions are recognized for elements corresponding to zeros in the vector-mask register.

**Programming Notes:**

1. For instructions which produce a vector result, result elements corresponding to ones in the vector-mask register are the same whether the vector-mask mode is on or off. The vector-mask mode does affect the results produced by instructions which reduce vector operands to a partial sum (\( \text{ACCUMULATE} \) and \( \text{MULTIPLY AND ACCUMULATE} \)) or to a single scalar result, because those results may depend on the presence or absence of each operand element.

2. The address-updating operation consists of unsigned shifts and additions of binary integers without overflow. Nevertheless, it is useful to consider the stride as a signed quantity, because adding the two’s complement of an integer to an unsigned binary number is the same as subtracting that integer.

**Class-NC Instructions**

Class-NC instructions process a variable number of bits in the vector-mask register but do not process any arithmetic-vector elements. The number of bits processed is determined by the vector count. The instructions are not interruptible and do not depend on the vector interruption index.

Class-NC instructions use the \( \text{RRE} \) or \( \text{VS} \) format. Class-NC instructions in the \( \text{RRE} \) format operate on bits in the vector-mask register. Class-NC instructions in the \( \text{VS} \) format operate on bits in the vector-mask register and on a bit vector in storage.

When instruction execution is completed for an operation that modifies the contents of the vector-mask register, any remaining rightmost bits of the register are set to zeros.

When the vector count is zero, execution of the instruction is completed without any bits being processed. For an instruction of a type that modi-
fies bits in the vector-mask register when the vector count is not zero, a vector count of zero causes all bits of the vector-mask register to be set to zeros. Any general register due to be modified remains unchanged.

**VS-Format Instructions**

The vs format is used for instructions which operate on bit vectors in storage and in the vector-mask register. All vs-format instructions are in class NC.

The RS2 field designates a general register that contains the storage address of the first byte of the second operand, the leftmost bit of which is the first bit of the storage operand to be processed. The first bit in the vector-mask register is the leftmost bit, bit 0. The operation proceeds with successive bits in contiguous bit locations of the second operand and in the vector-mask register.

When instruction execution is completed, the address of what would have been the next byte of the second operand is placed in the general register designated by RS2; that address is the integral part of the expression \( A + (C+7)/8 \), where \( A \) is the starting address in the RS2 register and \( C \) is the vector count. The updated address occupies the rightmost 31 or 24 bit positions of the RS2 register, depending on the address size; the leftmost bit or eight bits, respectively, are set to zeros.

If the vector count is not a multiple of 8, the remaining bits in the last byte used in storage are ignored on fetching and set to zeros on storing.

If no bits are processed because the vector count is zero, no access exceptions are recognized for the storage operand, the storage address in the general register designated by the RS2 field and the change bits for the operand remain unchanged, and no PER event for storage alteration is indicated.

**Programming Note:** Only class-NC instructions which modify the vector-mask register set bits beyond the active bits to zeros. This contrasts with COMPARE (class IC), which leaves bits in the vector-mask register beyond the active bits unchanged, and RESTORE VMR (class NZ), which ignores the vector count and replaces all the bits.

**Instructions In Other Classes**

Details of instructions in classes IG, IP, IZ, NZ, N1, and N0 are contained in the individual instruction descriptions.

**Vector Interruptions**

**Interruptible Vector Instructions**

All instructions which can operate on multiple elements of arithmetic vectors in storage or in vector registers are interruptible. Their execution generally consists of multiple units of operation with interruptions being permitted between these units of operation.

Vector instructions which can operate on only one arithmetic-vector element, or on none at all, are not interruptible; that is, the entire execution consists of one unit of operation. They include instructions which operate on multiple bits in the vector-mask register but not on elements of arithmetic vectors.

Conceptually, vector instructions are executed sequentially, elements of the vector operands of a single vector instruction are processed sequentially, and any resulting exceptions are recognized sequentially. Any program interruption is due to the first exception which is recognized and for which interruptions are allowed.

At the time of an interruption, changes to register contents, which are due to be made by an interruptible vector instruction beyond the point of interruption, have not yet been made. Changes to storage locations, however, which are due to be made by an interruptible vector instruction beyond the point of interruption, may have occurred for one or more storage locations beyond the location containing the element identified by the interruption parameters, but not for any location beyond the last element specified by the instruction and not for any locations for which access exceptions exist. Changes to storage locations or register contents which are due to be made by instructions following the interrupted instruction have not yet been made at the time of interruption.
If an instruction is due to cause more than one program interruption other than for PER events, only the first one is indicated.

**Units of Operation**

The execution of an interruptible vector instruction is considered to be divided into units of operation, such that an interruption is permitted between these units of operation.

The unit of operation for program interruptions, other than for PER events alone, is one vector element. After the last vector element has been processed without a program interruption, the instruction is completed in a final unit of operation. This final unit of operation consists in advancing the instruction address to the next instruction, setting the vector interruption index to zero if the instruction depends on the vector interruption index, and, for some instructions, setting the condition code.

Performing the final unit of operation cannot create any program-interruption conditions. If a program interruption occurs while processing the last element of a vector, the instruction remains partially completed, because the final unit of operation has not yet been performed. Thus, all elements of a vector are processed alike, including the recognition of any program exceptions.

Only the final unit of operation of advancing the instruction address, setting the vector interruption index to zero, and possibly setting the condition code is performed without processing any elements, when an interruptible instruction which depends on the vector interruption index is executed in the following situations:

- For class-1M and class-1C instructions, the vector interruption index equals or exceeds the vector count.
- For the class-IP instructions SUM PARTIAL SUMS and ZERO PARTIAL SUMS, the vector interruption index equals or exceeds the partial-sum number.
- For the class-1G instruction LOAD BIT INDEX, the specified bit count is zero, or the vector interruption index equals the section size.

For interruptions due to an asynchronous condition (external, I/O, repressible machine-check, or restart), the unit of operation may be one or more elements, depending on the model, the particular instruction, and the condition causing the interruption. If a PER event is held pending at the time an instruction is due to be interrupted by such an asynchronous condition, a program interruption for the PER event occurs first, and the other interruptions occur subsequently (subject to the mask bits in the new PSW) in the normal priority order.

PER events alone do not normally cause execution of a vector instruction to be interrupted prematurely. For possible exceptions, see the subsection “Priority of Indication” of the section “Program-Event Recording” in Chapter 4, “Control,” of *IBM 370-XA Principles of Operation* and *IBM System/370 Principles of Operation*.

**Operand Parameters**

Execution of interruptible vector instructions involves the updating of information referred to as the operand parameters. The operand parameters include:

- The vector interruption index, for instructions which depend on that index,
- The storage address in a general register, for instructions in the QST and VST formats,
- The bit index and bit count in a general register, for LOAD BIT INDEX,
- The floating-point scalar operand, for MAXIMUM ABSOLUTE, MAXIMUM SIGNED, MINIMUM SIGNED, and SUM PARTIAL SUMS,
- The element numbers in a general-register pair, if specified, for MAXIMUM ABSOLUTE, MAXIMUM SIGNED, and MINIMUM SIGNED,
- The vector in-use bits, for CLEAR VR and RESTORE VSR, and
- The save-area address and element number in general registers, for RESTORE VR, SAVE CHANGED VR, and SAVE VR.

Upon interruption, the operand parameters are adjusted so as to indicate the extent to which instruction execution has been completed. If the instruction is reexecuted after the interruption, execution resumes from the point of interruption.
Arithmetic Exceptions

The arithmetic exceptions which may be caused by interruptible vector instructions are:

- Exponent overflow
- Exponent underflow
- Fixed-point overflow
- Floating-point divide
- Significance
- Unnormalized operand

In the following respects, the arithmetic exceptions are the same for vector instructions as for the corresponding scalar instructions: the program mask in the PSW controls the occurrence of a program interruption for fixed-point overflow, exponent underflow, or significance; the result for the current target element is the same as the result for the corresponding scalar operation; and bits 8-15 of the program-interruption code indicate the type of exception.

The binary ADD, LOAD COMPLEMENT, LOAD POSITIVE, and SUBTRACT instructions for vectors do not indicate fixed-point overflow when a program interruption is disallowed by the fixed-point-overflow mask in the PSW, unlike the corresponding scalar instructions which can indicate overflow by setting the condition code. Other differences, including the definition of the unnormalized-operand exception, which does not apply to scalar instructions, are described in the following sections.

Exception-Extension Code

When an arithmetic exception is recognized during execution of an interruptible vector instruction, a nonzero exception-extension code is stored in bits 0-7 of the program-interruption code. The exception-extension code indicates whether the interruption was due to a noninterruptible scalar instruction or an interruptible vector instruction, whether the result, if any, was placed in a scalar or vector register, the width of the result, and the number of the register.

The arithmetic-partial-completion bit, bit 0 of the program-interruption code, indicates that the exception-extension code has been stored. If the arithmetic exception is due to an interruptible vector instruction and causes an interruption which leaves instruction execution partially completed, bit 0 is set to one, and bits 1-7 contain further information. If a scalar instruction was executed, bits 0-7 are set to all zeros.

If not all zeros, the information in the exception-extension code is as follows:

<table>
<thead>
<tr>
<th>avwwrrrr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 7</td>
</tr>
</tbody>
</table>

**Bit 0 (a)** is the arithmetic-partial-completion bit; when one, it indicates that the interrupted instruction was partially completed and that bits 1-7 have the meaning shown below. If bit 0 is zero, bits 1-7 are also zeros.

**Bit 1 (v)**, when one, indicates that the arithmetic result is in vector registers. When bit 1 is zero, the arithmetic result is in a scalar register.

**Bits 2-3 (ww)** contain the width of the arithmetic result:

| 01 | 4-byte result (short or binary) |
| 10 | 8-byte result (long)            |

**Bits 4-7 (rrrr)** contain the register number of the result register designated by the interrupted instruction.

Types of Ending for Units of Operation

When execution of an interruptible vector instruction is interrupted, the current unit of operation may end in one of five ways: completion, inhibition, nullification, suppression, or termination. Termination of a unit of operation of a vector instruction causes termination of the instruction; it can occur only as the result of an exigent machine check and will not be discussed further.

When an interruption occurs after completion, inhibition, nullification, or suppression of a unit of operation, all prior units of operation have been completed. The effect of the interruption on the instruction address in the old PSW stored during the interruption, on the operand parameters, and on the result location for the current unit of operation is as follows:

**Completion:** The instruction address in the old PSW designates the interrupted instruction or an EXECUTE instruction, as appropriate. The result location for the current unit of operation contains
the new result, as defined for the type of exception. The operand parameters are adjusted such that, if the instruction is reexecuted, execution of the interrupted instruction is resumed with the next unit of operation.

**Inhibition:** Same as completion, except that the result location for the current unit of operation remains unchanged. The exception-extension code is stored the same as if a result had been placed in that location.

**Nullification:** The instruction address in the old psw designates the interrupted instruction or an execute instruction, as appropriate. The result location for the current unit of operation remains unchanged. The operand parameters are adjusted such that, if the instruction is reexecuted, execution of the interrupted instruction is resumed with the current unit of operation. Interruption occurs before any arithmetic operation on the current element has started. Because access exceptions which nullify execution may be recognized for elements beyond the current unit of operation, access to the current element may or may not be the cause of the exception.

**Suppression:** Same as nullification, except that the instruction address in the old psw designates the next sequential instruction. Because access exceptions which suppress execution may be recognized for elements beyond the current unit of operation, access to the current element may or may not be the cause of the exception.

The following chart summarizes the differences between the four types of ending for a unit of operation:

<table>
<thead>
<tr>
<th>Unit of Operation Is</th>
<th>Instruction Address</th>
<th>Operand Parameters at</th>
<th>Current Result Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completed</td>
<td>Current Instruction</td>
<td>Next Element</td>
<td>Changed</td>
</tr>
<tr>
<td>Inhibited</td>
<td>Current Instruction</td>
<td>Next Element</td>
<td>Unchanged</td>
</tr>
<tr>
<td>Nullified</td>
<td>Current Instruction</td>
<td>Current Element</td>
<td>Unchanged</td>
</tr>
<tr>
<td>Suppressed</td>
<td>Next Instruction</td>
<td>Current Element</td>
<td>Unchanged</td>
</tr>
</tbody>
</table>

**Programming Notes:**

1. After a program interruption due to an arithmetic exception, an interruption handler may perform any desired fixup of the result before resuming execution of the program.

2. When an instruction which depends on the vector interruption index is interrupted because of an arithmetic exception for the last element to be processed by the instruction, and the instruction is later reexecuted, it is completed by advancing the instruction address, setting the vector interruption index to zero, and possibly setting the condition code, without further processing or program interruptions for this instruction. The same may happen after the vector interruption index has been set to too high a value by the instruction RESTORE VSR.

If the last element processed before an interruption due to an arithmetic exception is the last element of the vector register, then the vector interruption index contains the section size.

3. The floating-point-divide and unnormalized-operand exceptions are defined to inhibit execution of the current unit of operation. Inhibition differs from completion only in that no result is defined for these exceptions, and that the result location for the current element remains unchanged. Inhibition differs from nullification in that an arithmetic operation has been performed for the current element and the operand parameters have been adjusted to point to the next element.

4. When an arithmetic exception is recognized and bit 1 of the exception-extension code is one, the number of the associated result element in the vector registers is always one less than the current vector interruption index, since all arithmetic exceptions cause either completion or inhibition of the current unit of operation.
Effect of Interruptions during Execution

Interruptions occurring before instruction execution has begun, or after completion of the entire instruction, are the same as for nonvector instructions.

The effect of interruptions which occur during execution of vector-facility instructions depends on the type of ending. Figure 2-7 shows the effect for each interruption type that can occur during execution.

Setting of Instruction Address

The instruction address in the old psw designates the interrupted vector-facility instruction or an execute instruction, as appropriate, after completion, inhibition, or nullification of a unit of operation. The instruction address designates the next sequential instruction after suppression of a unit of operation.

Setting of Instruction-Length Code

When a program interruption occurs during the execution of an interruptible vector instruction, the instruction-length code (ILC) that is stored is 2 or 3, depending on whether the instruction length is two or three halfwords, respectively. When the vector instruction is executed under the control of an execute instruction, the ILC is always 2.

The ILC is stored as described regardless of whether the instruction address is advanced to the next instruction (the unit of operation is suppressed) or the instruction address designates the interrupted instruction (the unit of operation is completed, inhibited, or nullified).

For information on the ILC setting for a program interruption that occurs while fetching the instruction, see the section "Instruction-Length Code" in Chapter 6, "Interruptions," of IB M 370-XA Principles of Operation and IBM System/370 Principles of Operation.

Programming Note: Unless an interruption occurs during instruction fetching and prevents interpretation of the instruction, the instruction-length code is determined entirely by the leftmost two bits of the operation code. The ILC value does not depend on whether the operation code is assigned, or whether the instruction is installed or executed.

---

<table>
<thead>
<tr>
<th>Type of Interruption</th>
<th>Type of Ending</th>
<th>Exception-Extension Code Stored?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addressing</td>
<td>S</td>
<td>No</td>
</tr>
<tr>
<td>Exponent overflow</td>
<td>C</td>
<td>Yes</td>
</tr>
<tr>
<td>Exponent underflow</td>
<td>C</td>
<td>Yes</td>
</tr>
<tr>
<td>Fixed-point overflow</td>
<td>C</td>
<td>Yes</td>
</tr>
<tr>
<td>Floating-point divide</td>
<td>I</td>
<td>Yes</td>
</tr>
<tr>
<td>Page translation</td>
<td>N</td>
<td>No</td>
</tr>
<tr>
<td>Protection</td>
<td>S</td>
<td>No</td>
</tr>
<tr>
<td>Segment translation</td>
<td>N</td>
<td>No</td>
</tr>
<tr>
<td>Significance</td>
<td>C</td>
<td>Yes</td>
</tr>
<tr>
<td>Translation specification</td>
<td>S</td>
<td>No</td>
</tr>
<tr>
<td>Unnormalized operand</td>
<td>I</td>
<td>Yes</td>
</tr>
<tr>
<td>PER event alone</td>
<td>C</td>
<td>No</td>
</tr>
<tr>
<td>PER event with another exception</td>
<td>E</td>
<td>E</td>
</tr>
</tbody>
</table>

External, I/O, Repressible Machine Check, and Restart

All                  C       No

Explanation:

C Completed unit of operation
E Action determined by the exception reported with the PER event
I Inhibited unit of operation
N Nullified unit of operation
S Suppressed unit of operation

Figure 2-7. Interruptions during Execution of Interruptible Vector-Facility Instructions

Thus, the ILC is set to 2 or 3 for a vector instruction, depending on the instruction length, even when a vector-operation exception or an operation exception is recognized.

Setting of Storage Address

When a vector-facility instruction which updates a vector-operand address in a general register is interrupted, the address in the general register has been updated to the point of interruption.

After completion or inhibition of a unit of operation, the updated address designates the next
operand element in storage following the one causing the interruption.

After nullification or suppression of a unit of operation, the updated address designates the current operand element; this may or may not be the same as the element that caused the interruption, because of access exceptions which may be recognized for elements beyond the last one processed. If the exception occurs before the first element has been processed, the entire instruction is nullified or suppressed, and the general register containing the storage address remains unchanged.

When the entire instruction has been completed before an interruption takes place, the updated address designates the operand element following the last element processed.

**Setting of Vector Interruption Index**

At the start of execution of an interruptible vector instruction which depends on the vector interruption index, the vector interruption index contains the number of the next element to be processed in the designated vector registers or the vector-mask register. When such an instruction is interrupted, the vector interruption index is set to indicate the element within the registers at which execution may subsequently be resumed.

After completion or inhibition of a unit of operation, the vector interruption index identifies the next element, if any, to be processed after the one causing the interruption.

After nullification or suppression of a unit of operation, the vector interruption index identifies the current element; this may or may not be the element which caused the interruption, because of access exceptions which may be recognized for elements beyond the last one processed.

During the final step of completing the entire instruction, the vector interruption index is set to zero. This final step cannot cause any further interruptions.

When the entire instruction is nullified or suppressed, the vector interruption index remains unchanged. It also remains unaffected by the interruption of interruptible vector-facility instructions which do not depend on the vector interruption index and which do not set it explicitly. The vector interruption index is explicitly set to zero by CLEAR VR and to a specified value by RESTORE VSR.

**Programming Notes:**

1. Proper resumption of an interrupted instruction depends on the vector interruption index and the appropriate general registers being left unchanged.

2. If it is desired not to resume a program that was interrupted during execution of a vector-facility instruction but, instead, to store the current vector-register contents by means of vector-store instructions, or to load different data using vector-load instructions, care must be taken to set the vector interruption index to zero explicitly. This may be done with a CLEAR VR instruction; specifying a second operand of zeros leaves the vector-register contents unchanged.

**Program-Interruption Conditions**

When the vector facility is installed, two additional program exceptions can occur: unnormalized operand and vector operation. A vector-operation exception may also occur on CPUs without the vector facility. All arithmetic exceptions for vector instructions cause an exception-extension code to be stored as part of the program-interruption code. There are also modifications to access exceptions and to some of the arithmetic exceptions, and additional causes for the specification exception.

**Access Exceptions for Vector Operands**

When a vector-facility instruction specifies an arithmetic or bit vector in storage, access exceptions may be recognized for one or more storage locations beyond the location containing the element being processed, but not for any location beyond the last element specified by the instruction.

For contiguous operands, that is, for arithmetic vectors which are addressed sequentially with a stride of one and for bit vectors, access exceptions are not recognized more than 2K bytes beyond the current location. For noncontiguous operands, that is, for vectors which are addressed sequentially with a stride not equal to one and those which are loaded or stored by indirect element selection, access exceptions are not recognized more than seven element locations beyond the current one.
No access exceptions are recognized for the storage location of an operand when:

- No vector elements are to be processed because the net count is zero,
- The instruction operates under the control of the vector-mask register and the location of a vector element in storage corresponds to a zero mask bit,
- For the instruction LOAD BIT INDEX, the specified bit count is zero or the vector interruption index equals the section size,
- For the instructions RESTORE VR and SAVE VR, the vector in-use bit associated with the specified vector-register pair is zero, or
- For the instruction SAVE CHANGED VR, the vector change bit associated with the specified vector-register pair is zero.

**Programming Note:** Interruptible nonvector instructions, such as MOVE LONG, permit access exceptions to be recognized no more than 2K byte locations beyond the location of the byte being processed, which permits access exceptions for a maximum of four operand pages, two for each operand. This is in addition to access exceptions during instruction fetching of up to four pages when the instruction is the target of EXECUTE. Interruptible vector instructions permit access exceptions to be recognized for up to eight operand pages, in addition to a possible four instruction pages. The eight operand pages are not necessarily contiguous.

**Exponent-Overflow Exception**

If, during execution of a MULTIPLY AND ACCUMULATE, MULTIPLY AND ADD, or MULTIPLY AND SUBTRACT instruction, the multiplication of an element pair results in an exponent overflow, only the multiplication part of the unit of operation is completed, and the addition or subtraction part is not performed. The unit of operation is completed by placing the overflowed product, as defined for the corresponding scalar floating-point multiply instruction, in the result location.

**Exponent-Underflow Exception**

If, during execution of a MULTIPLY AND ACCUMULATE, MULTIPLY AND ADD, or MULTIPLY AND SUBTRACT instruction, the multiplication of an element pair results in an exponent underflow, no interruption occurs, regardless of the value of the exponent-underflow mask in the PSW. In this case, a true zero is added in place of the product, and the operation continues.

**Floating-Point-Divide Exception**

When a floating-point-divide exception is recognized during execution of a vector floating-point DIVIDE instruction, the unit of operation is inhibited.

**Specification Exception**

Specification exceptions are recognized for the following causes in addition to the causes listed in the section “Specification Exception” of Chapter 6, “Interruptions,” of *IBM 370-XA Principles of Operation and IBM System/370 Principles of Operation.*

- An invalid vector-register number is designated by a VR field of a vector instruction.
- The stride of an instruction in the QST or VST format is specified to be in the same general register as the storage address.
- The third operand of an instruction in the QST format is specified to be in the same general register as the storage address.
- The instruction RESTORE VSR attempts to load values into the vector-status register that are
  - Other than all zeros in bits 0-14,
  - Greater than the section size in the vector-count field (bits 16-31), or
  - Greater than the section size in the vector-interruption-index field (bits 32-47).
- The instruction RESTORE VR, SAVE CHANGED VR, or SAVE VR specifies a number in the element-number field that is equal to or greater than the section size, or a number in the VR-pair field that is other than an even number from 0 to 14.
- The instruction EXTRACT ELEMENT or LOAD ELEMENT specifies an element number in the second operand that is equal to or greater than the section size.
Unnormalized-Operand Exception

An unnormalized-operand exception is recognized when, in a vector floating-point divide or multiply operation, a source-operand element has a nonzero fraction with a leftmost hexadecimal digit of zero. The vector floating-point instructions which may cause an unnormalized-operand exception to be recognized are DIVIDE, MULTIPLY, MULTIPLY AND ACCUMULATE, MULTIPLY AND ADD, and MULTIPLY AND SUBTRACT.

The unnormalized-operand exception is recognized for one operand element even when there is another operand that is zero, except that the floating-point-divide exception, which takes precedence, is recognized instead when the zero element is the divisor of a vector DIVIDE instruction.

The unit of operation is inhibited.

The instruction-length code is 2.

The unnormalized-operand exception is indicated by a program-interruption code of XX1E hex (or XX9E hex if a concurrent PER event is indicated), where XX is the exception-extension code.

Vector-Operation Exception

A vector-operation exception is recognized when a vector-facility instruction is executed while bit 14 of control register 0 is zero on a CPU which has the vector facility installed and available. The vector-operation exception is also recognized when a vector-facility instruction is executed and the vector facility is not installed or available on this CPU, but the facility can be made available to the program either on this CPU or on another CPU in the configuration.

When a vector-facility instruction is executed, and the vector facility is not installed on any CPU which is or can be placed in the configuration, it depends on the model whether a vector-operation exception or an operation exception is recognized.

The operation is nullified when the vector-operation exception is recognized.

The instruction-length code is 2 or 3.

The vector-operation exception is indicated by a program-interruption code of 0019 hex (or 0099 hex if a concurrent PER event is indicated).

Programming Note: The definition permits a vector-operation exception to occur even when no CPU in the configuration has the vector facility installed. See the section "Vector-Operation Control" on page 2-6 for more information.

Priority of Vector Interruptions

Multiple program-interruption conditions for vector-facility instructions are recognized, one after another, according to the same priority rules as apply to other instructions, together with the following rules:

• The unnormalized-operand exception has the same priority with respect to the nonarithmetic exceptions as the other arithmetic exceptions which can occur for vector instructions (exponent overflow, exponent underflow, fixed-point overflow, floating-point divide, and significance).

When more than one arithmetic-exception condition is recognized at the same time, unnormalized operand takes precedence over the exponent-overflow and exponent-underflow exceptions; the floating-point-divide exception takes precedence over the unnormalized-operand exception.

• The vector-operation exception has the same priority as the operation exception; the two exceptions are mutually exclusive.

• An access exception caused by the operand of RESTORE VSR takes precedence over a specification exception caused by the same operand.

See also the section "Multiple Program-Interruption Conditions" in Chapter 6, "Inter­ruptions," of IBM 370-XA Principles of Operation and IBM System/370 Principles of Operation.

Program Switching

The following instructions are provided to save, restore, and clear the vector-facility registers when switching from one program to another. The instructions marked "privileged" are restricted to programs operating in the supervisor state.
Saving and restoring of the vector registers is further assisted by their associated vector in-use bits and vector change bits. When the vector in-use bit for a vector-register pair is zero, the saving and subsequent restoring of those registers are eliminated, thus reducing the program-switching time, because the registers are known to contain all zeros.

For programs operating in the supervisor state, the vector change bits may serve to reduce switching time still further by permitting the saving of a vector-register pair to be eliminated when its vector in-use bit is one but its vector change bit is zero. Although such a vector-register pair is in use, its contents are known not to have been changed if its vector change bit has remained zero since it was last restored from its save area; consequently, the previously saved information is still valid.

The vector change bits do not affect the restoring of vector registers and, therefore, do not help to reduce the restore time. When an interruption-handling portion of the control program restores previously saved registers, restoring the contents of a pair of vector registers is not considered a change. Hence, executing RESTORE VR in the supervisor state is defined not to alter the vector change bits. Executing RESTORE VR in the problem state, however, sets the vector change bit of the affected vector-register pair to one, so as to protect the integrity of its use by the control program.

Program Use of the Restore and Save Instructions

The instructions RESTORE VR, SAVE CHANGED VR, and SAVE VR are defined to be interruptible and to restore or save only a single pair of vector registers each time they are executed. When more than one vector-register pair is to be restored or saved, the appropriate instruction must be used in a programming loop as follows.

First, the even general register to be specified by the instruction should be set to the beginning of the save area for the vector registers, and the odd general register should be set to zeros. Then the restore or save instruction should be executed. It should be followed by a BRANCH ON CONDITION with a mask of 5 back to the restore or save instruction. This causes each vector-register pair, in turn, to be restored or saved if its vector in-use bit (or vector change bit for SAVE CHANGED VR) is one, or to be skipped if the bit is zero.

Restore Operations

To restore the vector-status register and the vector registers, the instruction RESTORE VSR should be executed before the above programming loop for RESTORE VR. A complete set of restore operations also includes RESTORE VMR and RESTORE VAC. RESTORE VAC should be the last restore instruction executed to avoid having the others advance the vector-activity count unnecessarily.

Save Operations

A complete set of save operations consists of the instruction SAVE VAC, followed by a loop that uses either SAVE VR or SAVE CHANGED VR, and then the instructions SAVE VMR and SAVE VSR.

SAVE VAC is executed first, so as to avoid having the vector-activity count advanced by the other save operations, especially at a time when no vector operations were performed since the last time that the registers were restored.

Programs running in either the problem state or the supervisor state may use the instruction SAVE VR in the loop to save the entire contents of all vector-register pairs for which the vector in-use bits are ones.

Alternatively, when a program using vector-facility instructions is interrupted and the vector registers are to be placed back into an area from which they were previously restored, an interruption handler in the supervisor state may use the privileged instruction SAVE CHANGED VR in the loop. SAVE VSR should be executed only after the vector registers have been saved, so that the vector change bits, which SAVE CHANGED VR sets to zeros, are saved as zeros.

SAVE VR should be used instead of SAVE CHANGED VR when the vector information is to be saved in an area which may not be the one from which the
vector registers were last restored. Thus, `SAVE VR` is the appropriate instruction for a machine-check-interruption handler.

**Clear Operations**

The instruction `CLEAR VR` may be used to clear all or selected pairs of vector registers and to make sure that the vector interruption index is set to zero.

`CLEAR VR` may be executed by the control program to ensure that all vector registers are cleared before turning over the vector facility to a new program requesting vector operations. It should also be executed by the vector program to clear a vector-register pair that is not needed again soon. Both measures serve to avoid unnecessary saving and restoring.

When a vector-register pair has been cleared by means of `CLEAR VR`, and the corresponding vector in-use bit is zero, all elements in those registers contain zeros. The zero elements in a cleared register are valid operands. Such use of a cleared vector register or register pair as a source of all zeros does not set the associated vector in-use bit to one. One or more individual elements of a cleared vector-register pair may be replaced by an instruction such as `LOAD ELEMENT`, but as soon as any element in either or both registers of the pair has been changed, its vector in-use bit and vector change bit are set to ones, and the register pair is no longer considered cleared. The vector registers are considered to have been changed even when the value loaded is all zeros.

The instruction `RESTORE VSR` also clears a vector-register pair when it finds that the associated vector in-use bit is one and must be set to zero.

When either `CLEAR VR` or `RESTORE VSR` finds a vector in-use bit that is already zero, the instruction does not clear the vector-register pair again. If either instruction is interrupted and later reexecuted, instruction execution is resumed from the beginning, but the instruction skips over registers that were cleared before the interruption and have remained cleared.

---

**Save-Area Requirements**

To make programs that save and restore registers of the vector facility model-independent, the sizes and addresses of the save areas should be computed at execution time using the current section size, as obtained by the instruction `STORE VECTOR PARAMETERS`.

Figure 2-8 shows the save-area sizes and the boundary alignment for `RESTORE VR`, `SAVE CHANGED VR`, and `SAVE VR` as a function of the section size. Boundary alignment requires that the address of a vector-register save area be a multiple of the integral boundary shown in the second column (8 times the section size). The save-area size is given as the number of bytes required to save all 16 vector registers; when fewer consecutive vector registers are to be saved, this area may be reduced correspondingly. The figure also shows the vector-mask register (VMR), which requires 4Z bits (Z/2 bytes), where Z is the section size; the VMR save area has no alignment requirement.

<table>
<thead>
<tr>
<th>Section Size (Z)</th>
<th>Vector Registers (8Z)</th>
<th>Bytes for 16 VRs (64Z)</th>
<th>Bytes for Vector-Mask Register (Z/2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>64</td>
<td>512</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>128</td>
<td>1,024</td>
<td>8</td>
</tr>
<tr>
<td>32</td>
<td>256</td>
<td>2,048</td>
<td>16</td>
</tr>
<tr>
<td>64</td>
<td>512</td>
<td>4,096</td>
<td>32</td>
</tr>
<tr>
<td>128</td>
<td>1,024</td>
<td>8,192</td>
<td>64</td>
</tr>
<tr>
<td>256</td>
<td>2,048</td>
<td>16,384</td>
<td>128</td>
</tr>
<tr>
<td>512</td>
<td>4,096</td>
<td>32,768</td>
<td>256</td>
</tr>
</tbody>
</table>

Figure 2-8. Save-Area Requirements

**Relationship to Other Facilities**

**Program-Event Recording (PER)**

The following PER events are recognized for instructions of the vector facility:

- Instruction fetching
- Storage alteration

Whether PER general-register-alteration events are recognized for vector-facility instructions is undefined.
When the net count is zero for IC- or IM-class instructions, when the vector count is zero for NC-class instructions, or when all active bits in the vector-mask registers are zeros for the STORE MATCHED instruction, no PER storage-alteration events are recognized.

When an interruptible vector instruction is interrupted and PER storage alteration applies to storage locations corresponding to vector elements that are due to be changed by the instruction beyond the point of interruption, PER storage alteration is indicated if any such storage change actually occurred and may be indicated even if such a change did not occur. PER storage alteration is only recognized if no access exception exists for such locations at the time that the instruction is executed.

Vector-Store Operations

As for nonvector instructions, the processing of vector-facility instructions generally appears to a program running on the same CPU to follow the conceptual sequence: The execution of one instruction appears to precede the execution of the following instruction, the processing of one vector element appears to precede the processing of the following vector element, and an interruption takes place between instructions or between units of operation of interruptible instructions. As discussed below, however, this conceptual sequence is not necessarily observed by programs on other CPUs, by channel programs, or when vector-facility instructions are used to store into the instruction stream.

Storage-Operand Consistency

For all vector-facility instructions, multiple accesses may be made to all or some of the bytes of a storage operand.

Thus, unlike instructions which make only single-access references, intermediate results of a vector-facility store instruction may be observed by channel programs and by other CPU programs accessing the same storage location concurrently.

When an interruptible store-type vector instruction is interrupted and its execution is later resumed, a store performed by the instruction before its interruption may be repeated when execution is resumed.

(See the section "Storage-Operand Consistency" in Chapter 5, "Program Execution," of IBM 370-XA Principles of Operation and IBM System/370 Principles of Operation.)

Storing into Instruction Stream

When a vector-facility instruction is executed that causes storing into a location from which subsequent instructions have been prefetched, the copies of the prefetched instructions are not necessarily changed. (See the section "Instruction Fetching" in Chapter 5, "Program Execution," of IBM 370-XA Principles of Operation and IBM System/370 Principles of Operation for a complete list of functions which cause all copies of prefetched instructions to be discarded.)

Resets

In regard to the operation of the vector facility, CPU reset terminates execution of the current vector instruction and any manual operation. Pending machine-check-interruption conditions affecting the vector facility and check-stop states are cleared. All copies of prefetched vector-facility instructions or operands are discarded.

Initial CPU reset performs the functions of CPU reset mentioned above and initializes the vector-control bit, bit 14 of control register 0, to zero.

The registers of the vector facility (vector-status register, vector-mask register, vector-activity count, and all vector registers) are cleared to zero by clear reset and power-on reset.

Machine-Check Handling

Two bits of the machine-check-interruption code are associated with the vector facility: vector-facility failure and vector-facility source. The vector-facility-failure bit indicates to the program that vector-facility instructions should no longer be used. The vector-facility-source bit is a modifier to instruction-processing damage, which indicates that the vector facility is the error source.

These bits may be set to ones regardless of whether the vector-control bit, bit 14 of control register 0, is one or zero.
Vector-Facility Failure

Bit 6 (VF) of the machine-check-interruption code, when one, indicates that the vector facility has failed to such an extent that the service processor has made the facility not available.

This bit is not meaningful when system damage, bit 0 of the machine-check-interruption code, is one.

Vector-facility failure is a repressible condition, which has no subclass mask.

Vector-Facility Source

Bit 13 (vs) of the machine-check-interruption code, when one, indicates that the vector facility is the source of the reported machine-check condition. Vector-facility source is reported together with instruction-processing damage. When this bit is one, the contents of vector-facility registers may have been damaged or may contain incorrect information with no preserved error.

This bit is not meaningful when vector-facility failure, bit 6, is one.

Validation of Vector-Facility Registers

The following procedure can be used to validate the registers associated with the vector facility. The program should first execute RESTORE VSR, specifying all vector in-use bits as ones. This validates the vector-status register by setting it without first inspecting the previous contents. The program should then execute RESTORE VAC, RESTORE VMR, and RESTORE VR to load and validate the vector-activity count, the vector-mask register, and the vector registers.

Programming Notes:

1. When a vector-facility-failure condition is indicated, the program should stop using any functions associated with the vector facility. Thus, no vector-facility instructions should be executed; the vector-control bit, bit 14 of control register 0, should be set or remain set to zero; and the registers associated with the vector facility should not be validated or saved.

2. Although the purpose of the vector-facility-source bit is to indicate that the vector facility is the source of the instruction-processing damage, it is possible in some situations that the bit may be set to one when failures have occurred both in the vector facility and in other parts of the CPU.

3. Since a vector-facility-source condition may imply that vector-facility registers have been damaged, the registers should be validated before further use is attempted. If the vector-control bit is zero, it must be set to one to perform the validation.

4. The instruction RESTORE VR is the only instruction which validates the vector registers, and then only if their vector in-use bits are ones. In particular, the instruction CLEAR VR should not be used for validation, because this instruction may be implemented for performance reasons such that the registers are not actually cleared unless the program subsequently attempts to load or modify them. With this design, when the program next loads the vector register following a CLEAR VR instruction, only those elements which are not loaded, if any, are actually cleared at that time. Except for the possible effect on machine-check handling, this implementation gives the same results as if the instruction actually cleared the registers.
This chapter describes the instructions of the vector facility. When the operation on each element of a vector is the same as for a counterpart scalar instruction, the vector-instruction description does not repeat these details. The complete definition in these cases can be obtained from the description of the scalar instruction in IBM 370-XA Principles of Operation and IBM System/370 Principles of Operation.

Summary lists of the vector-facility instructions and their mnemonics, formats, and operation codes are contained in Appendix B. These lists also indicate the exceptional conditions in operand designations, data, or results that cause a program interruption.

Condition-code settings are summarized in Appendix C.

In many cases, several related vector operations are described under a single name. For example, MULTIPLY in the QST format is described as follows:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR1,QR2,RS2(RT2) [QST]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>QR1</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMS</td>
<td>'A4A2'</td>
<td>Binary</td>
</tr>
<tr>
<td>VMDS</td>
<td>'A492'</td>
<td>Long</td>
</tr>
<tr>
<td>VMES</td>
<td>'A482'</td>
<td>Short multiplier and multiplicand, long product</td>
</tr>
</tbody>
</table>
This figure is a “shorthand” representation for three different instructions, one binary and two floating-point multiply instructions. It replaces the following set of three figures:

VMS \( VR_1,GR_3,RS_2(RT_2) \) 
\[ \text{[QST, Binary operands]} \]
\[ 'A4A2' \]
\[ \begin{array}{cccc}
0 & 16 & 20 & 24 & 28 & 31 \\
\end{array} \]

VMDS \( VR_1,FR_3,RS_2(RT_2) \) 
\[ \text{[QST, Long operands]} \]
\[ 'A492' \]
\[ \begin{array}{cccc}
0 & 16 & 20 & 24 & 28 & 31 \\
\end{array} \]

VMES \( VR_1,FR_3,RS_2(RT_2) \) 
\[ \text{[QST, Short multiplier and multiplicand, long product]} \]
\[ 'A482' \]
\[ \begin{array}{cccc}
0 & 16 & 20 & 24 & 28 & 31 \\
\end{array} \]

Thus, the term “Binary” under the heading “Operands” for the first instruction indicates that the vector elements are 32-bit signed binary integers, that the scalar operand is taken from a general register, and that the operation on each element pair is performed in the same manner as the scalar MUL­TIPLY instruction described in Chapter 7, “General Instructions,” of IBM 370-XA Principles of Operation and IBM System/370 Principles of Operation.

Likewise, the terms “Short” or “Long” under the heading “Operands” for the second and third instructions indicate that the vector elements are floating-point numbers in the short or long floating-point format, respectively, that the scalar operand is taken from a floating-point register, and that the operation on each element pair is performed in the same manner as the corresponding scalar MUL­TIPLY instruction described in Chapter 9, “Floating-Point Instructions,” of IBM 370-XA Principles of Operation and IBM System/370 Principles of Operation.

Except for the new suffixes Q and S, which indicate scalar-vector operations, each mnemonic for a vector instruction is generally the same as the mnemonic for the counterpart scalar instruction prefixed with a v.

For several of the load and store instructions, the same instruction is used for vectors in the short floating-point format and in the 32-bit binary-integer or logical format. Separate mnemonics are assigned to the short and binary-logical formats for programming convenience, but the op codes for the two mnemonics are the same when the function is the same.

**Programming Note:** Programming notes in this section, as well as the examples in Appendix A, assume normal execution of vector instructions. In particular, they assume that the program does not alter the vector interruption index, so that each interruptible vector instruction begins its operation on the first element or element pair with the vector interruption index set to zero. If the instruction is interrupted for a cause other than an arithmetic exception, and if its execution is subsequently resumed, the vector interruption index and all other parameters are assumed to have been restored to the value they had at the time of interruption, so that the result is the same as if the interruption had not occurred.

### ACCUMULATE

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR_1,RS_2(RT_2)</th>
<th>VST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>16 20 24 28 31</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>VACD 'A417' Long operand and sum</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VACER 'A407' Short operand, long sum</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR_1,VR_2</th>
<th>[VV]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>16 24 28 31</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VACDR 'A517' Long operand and sum</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VACER 'A507' Short operand, long sum</th>
</tr>
</thead>
</table>

Partial sums of the elements of the second-operand vector are accumulated by adding the second-operand elements to the contents of element positions 0 to \( p-1 \) of the first operand. The partial-sum number \( p \) depends on the model.

The operation proceeds in an ascending sequence of element numbers. The \( I \)-th element of the second operand is added to the first-operand element at a position which is the remainder of dividing \( I \) by \( p \), where \( I \) varies from \( X \) to \( C-1 \), \( X \) is...
the initial vector interruption index (normally zero),
and C is the vector count. The operation accumu­
lates $C - X$ elements of the second operand.

Thus, second-operand elements $0, p, 2p, \ldots$ are
accumulated into position $0$ of the first operand;
second-operand elements $1, p+1, 2p+1, \ldots$ are
accumulated into position $1$; and so forth. The
contents of first-operand element positions above
$p-1$ remain unchanged.

Every addition is performed in the same manner as
for the scalar ADD NORMALIZED (ADR) instruction,
where the second-operand elements for VACE and
VACER are extended on the right with 32 zeros,
except that the condition code is not set.

A specification exception is recognized when the
VR1 field designates an invalid register number. In
the VST format, a specification exception is also
recognized when the second operand is not desig­
nated on an integral boundary, or when the RT2
field is nonzero and designates the same general
register as the RS2 field.

ACCUMULATE is a class-1M instruction. It is inter­
ruptible, the vector count and vector interruption
index determine the number of elements processed,
and element selection is affected by both the
vector-mask mode and the vector-mask register.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2 in VST format)
- Exponent overflow (with exception-extension code)
- Exponent underflow (with exception-extension code)
- Operation
- Significance (with exception-extension code)
- Specification
- Vector operation

**Programming Notes:**

1. ACCUMULATE is used, together with ZERO
   PARTIAL SUMS and SUM PARTIAL SUMS, to
   produce the scalar sum of the elements of a
   vector in a manner similar to the example in
   Appendix A ("Sum of Products" on page A-3) of
   using MULTIPLY AND ACCUMULATE to
   produce a sum of products.

2. The short-format ACCUMULATE instructions
   (VACE and VACER) add floating-point vector
   elements in the short format to produce a
   floating-point sum in the long format. This
   creates a result of higher precision than would
   an equivalent loop with the scalar short-format
   ADD instructions (AE or AER, respectively),
   which produces a sum in the short format.

**ADD**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR1,QR3,RS2(RT2)</th>
<th>[QST]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>QR3 RT2 VR1 RS2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>16 20 24 28 31</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR1,QR3,VR2</th>
<th>[QV]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>QR3 /// VR1 VR2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>16 20 24 28 31</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR1,VR3,RS2(RT2)</th>
<th>[VST]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>VR3 RT2 VR1 RS2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>16 20 24 28 31</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR1,VR3,VR2</th>
<th>[VV]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>VR3 /// VR1 VR2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>16 20 24 28 31</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAS</td>
<td>'A4A0'</td>
<td>Binary</td>
</tr>
<tr>
<td>VADS</td>
<td>'A490'</td>
<td>Long</td>
</tr>
<tr>
<td>VAES</td>
<td>'A480'</td>
<td>Short</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAQ</td>
<td>'A5A0'</td>
<td>Binary</td>
</tr>
<tr>
<td>VADQ</td>
<td>'A590'</td>
<td>Long</td>
</tr>
<tr>
<td>VAEQ</td>
<td>'A580'</td>
<td>Short</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>VA</td>
<td>'A420'</td>
<td>Binary</td>
</tr>
<tr>
<td>VAD</td>
<td>'A410'</td>
<td>Long</td>
</tr>
<tr>
<td>VAE</td>
<td>'A400'</td>
<td>Short</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR</td>
<td>'A520'</td>
<td>Binary</td>
</tr>
<tr>
<td>VADR</td>
<td>'A510'</td>
<td>Long</td>
</tr>
<tr>
<td>VAER</td>
<td>'A500'</td>
<td>Short</td>
</tr>
</tbody>
</table>

Element by element, the second-operand vector is
added to the third operand, and the result is placed
in the first-operand location.

The operation is performed on each pair of ele­
ments in the same manner as the corresponding
scalar operation, except that the condition code is not set. For floating-point operands, the scalar equivalent is ADD NORMALIZED.

A specification exception is recognized when a VR or QR field designates an invalid register number. In the QST and VST formats, a specification exception is recognized when the second operand is not designated on an integral boundary, or when the RT2 field is nonzero and designates the same general register as the RS2 field. For the VAS instruction, a specification exception is also recognized when the QR3 field designates the same general register as the RS2 field.

ADD is a class-1M instruction. It is interruptible, the vector count and vector interruption index determine the number of elements processed, and element selection is affected by both the vector-mask mode and the vector-mask register.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2 in QST and VST formats)
- Exponent overflow (with exception-extension code; floating-point operands only)
- Exponent underflow (with exception-extension code; floating-point operands only)
- Fixed-point overflow (with exception-extension code; binary operands only)
- Operation
- Significance (with exception-extension code; floating-point operands only)
- Specification
- Vector operation

### AND

VNS VR1,GR3,RS2(RT2) [QST]

```
  'A4A4'  GR3  RT2  VR1  RS2
  0       16  20  24  28  31
```

Element by element, the AND of the second and third operands is placed in the first-operand location.

The operation is performed on each pair of 32-bit elements in the same manner as the corresponding scalar operation, except that the condition code is not set.

For the VN and VNS instructions, a specification exception is recognized when the second operand is not designated on an integral boundary, or when the RT2 field is nonzero and designates the same general register as the RS2 field. For the VNS instruction, a specification exception is also recognized when the GR3 field designates the same general register as the RS2 field.

AND is a class-1M instruction. It is interruptible, the vector count and vector interruption index determine the number of elements processed, and element selection is affected by both the vector-mask mode and the vector-mask register.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2 in QST and VST formats)
- Operation
- Specification
- Vector operation
AND TO VMR

VNVM  RS2  [VS]

'A684'  / / / / / / / / RS2
0  16  28  31

The **AND** of the second-operand bit vector and of the active bits of the vector-mask register (VMR) is placed in the vector-mask register. Bits beyond the active bits are set to zeros.

**AND TO VMR** is a class-NC instruction. It is not interruptible, the vector count determines the number of bits processed, and bit selection is affected by neither the vector-mask mode nor the vector-mask register. The vector interruption index is not used and remains unchanged.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2)
- Operation
- Vector operation

CLEAR VR

VRCL  D2(B2)  [S]

'A6C5'  B2  D2
0  16  20  31

The specified pairs of vector registers are cleared, the associated vector in-use bits and vector change bits are set to zeros, and the vector interruption index is set to zero.

The second-operand address is not used to address storage. Instead, bits 24-31 of the second-operand address, called the second-operand bits, control which vector registers are cleared. The eight second-operand bits are associated with the eight even-numbered vector-register pairs from 0 to 14, and with the corresponding vector in-use bits and vector change bits. The leftmost bits of the address are ignored.

The vector interruption index is set to zero first, after which the eight second-operand bits are examined in any order. If a second-operand bit and the corresponding vector in-use bit are both ones, all element positions of the associated pair of vector registers are cleared to zeros; the corresponding vector in-use bits and vector change bits are then set to zeros. If a second-operand bit or the corresponding vector in-use bit is zero, the associated registers and bits remain unchanged.

If the instruction is interrupted before the operation is completed, the instruction address in the current psw identifies this instruction. If the interrupted instruction is then reexecuted, vector-register pairs, which were cleared and had their vector in-use bits and vector change bits set to zeros, are not cleared again, provided that their vector in-use bits are still zeros.

CLEAR VR is a class-Iz instruction. It is interruptible, the section size determines the number of elements processed, and element selection is affected by neither the vector-mask mode nor the vector-mask register. The vector count is not used and remains unchanged. The vector interruption index is set.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Operation
- Vector operation

**Programming Notes:**

1. When a vector in-use bit is zero, execution time of **CLEAR VR** is saved because the corresponding vector-register pair is already cleared, and the instruction does not clear those registers again.

2. **CLEAR VR** with a zero operand (VRCL 0) merely sets the vector interruption index to zero.

COMPARE

Mnemonic  M1,QR3,RS2(RT2)  [QST]

Op Code  QR3  RT2  M1  RS2
0  16  20  24  28  31

**Mnemonic**  **Op Code**  **Operands**
- VCS  'A4A8'  Binary
- VCDS  'A498'  Long
- VCES  'A488'  Short

Chapter 3. Vector-Facility Instructions  3-5
The third operand is compared with the second-operand vector, element by element. The corresponding bit in the vector-mask register is set to one or zero, depending on the comparison result and on the value of a modifier in bits 24-26 of the instruction.

The comparison is algebraic and is performed on each element pair in the same manner as the corresponding scalar operation, except for the way in which the result is indicated. The condition code is not set; instead, a single result bit is set in the vector-mask register for each element pair. The value of the result bit is selected from one of the modifier bits according to the comparison of the third-operand element with the second-operand element, as follows:

<table>
<thead>
<tr>
<th>Result of Comparison</th>
<th>Modifier Bit Whose Value Is Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>operands equal</td>
<td>M0 (bit 24)</td>
</tr>
<tr>
<td>_operand 3 low</td>
<td>M1 (bit 25)</td>
</tr>
<tr>
<td>Operand 3 high</td>
<td>M2 (bit 26)</td>
</tr>
</tbody>
</table>

Modifier bit M3, bit 27 of the instruction, is ignored.

Bits in the vector-mask register which do not correspond to elements being compared remain unchanged.

A specification exception is recognized when a VR or QR field designates an invalid register number. In the QST and VST formats, a specification exception is recognized when the second operand is not designated on an integral boundary, or when the RT2 field is nonzero and designates the same general register as the RS2 field. For the VCS instruction, a specification exception is also recognized when the QR3 field designates the same general register as the RS2 field.

COMPARE is a class-Ic instruction. It is interruptible, the vector count and vector interruption index determine the number of elements processed, and element selection is affected by neither the vector-mask mode nor the vector-mask register.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2 in QST and VST formats)
- Operation
- Specification
- Vector operation

**Programming Notes:**
1. To obtain ones in the resultant bit vector when a desired comparison condition is found for an element of operand 3, the modifier bits should be specified as follows:

<table>
<thead>
<tr>
<th>Modifier Bits</th>
<th>Result Is One</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0 M1 M2 M3</td>
<td>If Operand-3 Comparison Is</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>— (always 0)</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>High</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>Low</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>Not equal</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>Equal</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>Not low</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>Not high</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>Any (always 1)</td>
</tr>
</tbody>
</table>

3-6 IBM System/370 Vector Operations
2. The modifier bits of the vector COMPAR E instruction correspond to the condition codes of the scalar COMPAR E instruction when an element of vector operand 3 is the same as the scalar operand 1 and the corresponding element of vector operand 2 is the same as the scalar operand 2. Thus, the value of the leftmost three bits of the mask field of the BRANCH ON CONDITION instruction, which causes branching when used to test the condition code of the scalar COMPAR E, is the same as the modifier value of the vector COMPAR E instruction, which sets a vector-mask bit to one for the same comparison condition.

3. The comparison instructions are the only ones which both modify the vector-mask register and are interruptible. They do not change those bits in the vector-mask register which lie beyond the last bit processed. This contrasts with the noninterruptible instructions which load or perform logical operations on the vector-mask register; they set to zeros all bits which lie beyond the last bit processed.

4. Unlike the related arithmetic and logical vector instructions, the comparison instructions are not executed under control of the vector-mask mode.

**COMPLEMENT VMR**

VCVM [RRE]

| 'A641' | //////////////////////////////////////////////////////////////////// |
| 0     | 16    | 31 |

The active bits of the vector-mask register (VMR) are complemented. Bits beyond the active bits of the vector-mask register are set to zeros.

**Program Exceptions:**

- Operation
- Vector operation

**COUNT LEFT ZEROS IN VMR**

VCZVM GR1 [RRE]

| 'A642' | //////////////////////////////////////////////////////////////////// |
| 0     | 16    | 24  | 28  | 31 |

Selected bits among the active bits of the vector-mask register (VMR) are counted, and the count is added to the contents of the general register designated by GR1. For the COUNT LEFT ZEROS IN VMR instruction, the selected bits are the zero bits to the left of the leftmost one bit. For the COUNT ONES IN VMR instruction, the selected bits are the one bits.

The general-register contents are treated as a 32-bit unsigned binary integer. Any carry out of the leftmost bit of the sum is ignored; there is no overflow indication.

Condition code 0, 1, or 3 is set according to whether the active bits are all zeros, mixed zeros and ones, or all ones. When the vector count is zero, the general register is not altered, and condition code 0 is set.

**COUNT ONES IN VMR**

VCOVN GR1 [RRE]

| 'A643' | //////////////////////////////////////////////////////////////////// |
| 0     | 16    | 24  | 28  | 31 |

Selected bits among the active bits of the vector-mask register (VMR) are counted, and the count is added to the contents of the general register designated by GR1. For the COUNT LEFT ZEROS IN VMR instruction, the selected bits are the zero bits to the left of the leftmost one bit. For the COUNT ONES IN VMR instruction, the selected bits are the one bits.

The general-register contents are treated as a 32-bit unsigned binary integer. Any carry out of the leftmost bit of the sum is ignored; there is no overflow indication.

Condition code 0, 1, or 3 is set according to whether the active bits are all zeros, mixed zeros and ones, or all ones. When the vector count is zero, the general register is not altered, and condition code 0 is set.

**Resulting Condition Code:**

- 0: Active bits all zeros
- 1: Active bits mixed zeros and ones
- 2: --
- 3: Active bits all ones
Program Exceptions:
• Operation
• Vector operation

Programming Note: When only the condition-code result of COUNT LEFT ZEROS IN VMR or COUNT ONES IN VMR is required, but not the actual bit counts, the instruction TEST VMR may be used instead.

DIVIDE

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR1,FR3,RS2(RT2) [QST]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>FR3 RT2 VR1 RS2</td>
</tr>
<tr>
<td>0</td>
<td>16 20 24 28 31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>VDDS</td>
<td>'A493'</td>
<td>Long</td>
</tr>
<tr>
<td>VDES</td>
<td>'A483'</td>
<td>Short</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR1,FR3,VR2 [QV]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>FR3 ///// VR1 VR2</td>
</tr>
<tr>
<td>0</td>
<td>16 20 24 28 31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>VDDQ</td>
<td>'A593'</td>
<td>Long</td>
</tr>
<tr>
<td>VDEQ</td>
<td>'A583'</td>
<td>Short</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR1,VR3,RS2(RT2) [VST]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>VR3 RT2 VR1 RS2</td>
</tr>
<tr>
<td>0</td>
<td>16 20 24 28 31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>VDD</td>
<td>'A413'</td>
<td>Long</td>
</tr>
<tr>
<td>VDE</td>
<td>'A463'</td>
<td>Short</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR1,VR3,VR2 [VV]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>VR3 ///// VR1 VR2</td>
</tr>
<tr>
<td>0</td>
<td>16 20 24 28 31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>VDDR</td>
<td>'A513'</td>
<td>Long</td>
</tr>
<tr>
<td>VDER</td>
<td>'A563'</td>
<td>Short</td>
</tr>
</tbody>
</table>

Element by element, the third operand is divided by the second-operand vector, and the result is placed in the first-operand location.

The operation is performed on each pair of elements in the same manner as the corresponding scalar operation, except for two changes. When the fraction part of a divisor element is zero, so that a floating-point-divide exception is recognized, the unit of operation is inhibited. Also, the operands are not first normalized; when one or both of the source-operand elements have a nonzero fraction with a leftmost hexadecimal digit of zero, an unnormalized-operand exception is recognized, and the unit of operation is inhibited.

The floating-point-divide exception takes precedence over the unnormalized-operand exception, and both take precedence over the exponent overflow and exponent underflow exceptions.

A specification exception is recognized when a VR or QR field designates an invalid register number. In the QST and VST formats, a specification exception is recognized when the second operand is not designated on an integral boundary, or when the RT2 field is nonzero and designates the same general register as the RS2 field.

DIVIDE is a class-1M instruction. It is interruptible, the vector count and vector interruption index determine the number of elements processed, and element selection is affected by both the vector-mask mode and the vector-mask register.

Condition Code: The code remains unchanged.

Program Exceptions:
• Access (fetch, operand 2 in QST and VST formats)
• Exponent overflow (with exception-extension code)
• Exponent underflow (with exception-extension code)
• Floating-point divide (with exception-extension code)
• Operation
• Specification
• Unnormalized operand (with exception-extension code)
• Vector operation

Programming Notes:
1. The QST and QV formats provide for dividing a scalar operand by a vector. The operation of dividing a vector by a scalar can usually be replaced by the (generally faster) operation of multiplying the vector operand by the reciprocal of the scalar operand.
2. An unnormalized-operand exception is recognized whenever a divisor element is unnorma-
lized, even if the corresponding dividend element is zero.

**EXCLUSIVE OR**

\[
\text{VXS} \quad \text{VR}_1, \text{GR}_3, \text{RS}_2 (\text{RT}_2) \quad [\text{QST}]
\]

<table>
<thead>
<tr>
<th>'A4A6'</th>
<th>GR$_3$</th>
<th>RT$_2$</th>
<th>VR$_1$</th>
<th>RS$_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

\[
\text{VXQ} \quad \text{VR}_1, \text{GR}_3, \text{VR}_2 \quad [\text{QV}]
\]

<table>
<thead>
<tr>
<th>'A5A6'</th>
<th>GR$_3$</th>
<th>VR$_1$</th>
<th>VR$_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
<td>24</td>
</tr>
</tbody>
</table>

\[
\text{VX} \quad \text{VR}_1, \text{VR}_3, \text{RS}_2 (\text{RT}_2) \quad [\text{VST}]
\]

<table>
<thead>
<tr>
<th>'A426'</th>
<th>VR$_3$</th>
<th>RT$_2$</th>
<th>VR$_1$</th>
<th>RS$_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

\[
\text{VXR} \quad \text{VR}_1, \text{VR}_3, \text{VR}_2 \quad [\text{VV}]
\]

<table>
<thead>
<tr>
<th>'A526'</th>
<th>VR$_3$</th>
<th>VR$_1$</th>
<th>VR$_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
<td>24</td>
</tr>
</tbody>
</table>

Element by element, the **EXCLUSIVE OR** of the second and third operands is placed in the first-operand location.

The operation is performed on each pair of 32-bit elements in the same manner as the corresponding scalar operation, except that the condition code is not set.

For the **vxs** instructions, a specification exception is recognized when the second operand is not designated on an integral boundary, or when the **RT2** field is nonzero and designates the same general register as the **RS2** field. For the **vxs** instruction, a specification exception is also recognized when the **GR3** field designates the same general register as the **RS2** field.

**EXCLUSIVE OR** is a class-1M instruction. It is interruptible, the vector count and vector interruption index determine the number of elements processed, and element selection is affected by both the vector-mask mode and the vector-mask register.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2 in **QST** and **VST** formats)
- Operation
- Specification
- Vector operation

**EXCLUSIVE OR TO VMR**

\[
\text{VXNM} \quad \text{RS}_2 \quad [\text{VS}]
\]

| 'A686' | VR$_1$ | VR$_2$ | VR$_3$ | VR$_4$ | VR$_5$ | VR$_6$ | VR$_7$ | VR$_8$ | VR$_9$ | VR$_10$ | VR$_11$ | VR$_12$ | VR$_13$ | VR$_14$ | VR$_15$ | VR$_16$ | VR$_17$ | VR$_18$ | VR$_19$ | VR$_20$ | VR$_21$ | VR$_22$ | VR$_23$ | VR$_24$ | VR$_25$ | VR$_26$ | VR$_27$ | VR$_28$ | VR$_29$ | VR$_30$ | VR$_31$ |
|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| 0      | 16     | 20     | 24     | 28     | 31     |

The **EXCLUSIVE OR** of the second-operand bit vector and of the active bits of the vector-mask register (VMR) is placed in the vector-mask register. Bits beyond the active bits are set to zeros.

**EXCLUSIVE OR TO VMR** is a class-NC instruction. It is not interruptible, the vector count determines the number of bits processed, and bit selection is affected by neither the vector-mask mode nor the vector-mask register. The vector interruption index is not used and remains unchanged.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2)
- Operation
- Vector operation

**EXTRACT ELEMENT**

\[
\text{Mnemonic} \quad \text{VR}_1, \text{QR}_3, \text{GR}_2 \quad [\text{VR}]
\]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>QR$_3$</th>
<th>VR$_1$</th>
<th>GR$_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
<td>24</td>
</tr>
</tbody>
</table>

**Mnemonic**

- VXEL 'A629' Binary or logical
- VXELD 'A619' Long
- VXELE 'A609' Short

The element from the vector register or vector-register pair designated by **VR1**, which has the element number contained in the general register designated by **GR2**, is placed in the general or floating-point register designated by **QR3**.

**Condition Code:** The code remains unchanged.
The element number is a 32-bit unsigned binary integer which must be less than the section size.

For VXELE, the rightmost 32 bits of the floating-point register designated by QR3 remain unchanged.

For VXEL, if the GR2 and QR3 fields designate the same general register, the element number is obtained from that register before it is replaced by the specified vector element.

A specification exception is recognized when the VR1 or QR3 field designates an invalid register number, or when the element number is equal to or greater than the section size.

EXTRACT ELEMENT is a class-N1 instruction. It is not interruptible, one element is processed, and its execution is affected by neither the vector-mask mode nor the vector-mask register. The vector count and vector interruption index are not used and remain unchanged.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Operation
- Specification
- Vector operation

**EXTRACT VCT**

<table>
<thead>
<tr>
<th>VXVC</th>
<th>GR1 [RRE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A644'</td>
<td>[ ] [ ] GR1 [ ] [ ]</td>
</tr>
</tbody>
</table>

0 16 24 28 31

The vector count, with 16 zeros appended on the left, is placed in the general register designated by GR1.

EXTRACT VCT is a class-N0 instruction. It is not interruptible, no elements are processed, and its execution is affected by neither the vector-mask mode nor the vector-mask register. The vector count remains unchanged. The vector interruption index is not used and remains unchanged.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Operation
- Vector operation

**EXTRACT VECTOR MASK MODE**

<table>
<thead>
<tr>
<th>VXVMM</th>
<th>GR1 [RRE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A646'</td>
<td>[ ] [ ] GR1 [ ] [ ]</td>
</tr>
</tbody>
</table>

0 16 24 28 31

Bits 16-31 of the general register designated by GR1 are set to the value of bits 0-15 of the vector-status register. Thus, bit 31 of the general register indicates the current setting of the vector-mask mode. Bits 0-15 of the general register are set to zeros.

EXTRACT VECTOR MASK MODE is a class-N0 instruction. It is not interruptible, no elements are processed, and its execution is affected by the vector-mask mode but not by the vector-mask register. The vector count and vector interruption index are not used and remain unchanged.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Operation
- Vector operation

**Programming Note:** The program should not rely on bits 16-30 of the general register being set to zeros. Those bits correspond to unassigned bits of the vector-status register, which are reserved for possible future use.

**LOAD**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR1,QR2 [QV]</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR1</td>
<td>QR2 [ ] [ ] [ ]</td>
</tr>
</tbody>
</table>

0 16 20 24 28 31

**Mnemonic** | **Op Code** | **Operands** |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VLQ</td>
<td>'A5A9'</td>
<td>Binary or logical</td>
</tr>
<tr>
<td>VLDQ</td>
<td>'A599'</td>
<td>Long</td>
</tr>
<tr>
<td>VLEQ</td>
<td>'A589'</td>
<td>Short</td>
</tr>
</tbody>
</table>
Mnemonic VRl,RS2(RT2) [VST]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>RT2</th>
<th>VRl</th>
<th>RS2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
<td>24</td>
</tr>
</tbody>
</table>

Mnemonic Op Code Operands
VL 'A409' Binary or logical
VLD 'A419' Long
VLE 'A409' Short

Mnemonic VRl,VR2 [VV]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>VRl</th>
<th>VR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

Mnemonic Op Code Operands
VLR 'AS09' Binary or logical
VLDR 'AS09' Long
VLER 'AS09' Short

Element by element, the second operand is placed unchanged in consecutive first-operand locations.

A specification exception is recognized when a VR or QR field designates an invalid register number. In the VST format, a specification exception is also recognized when the second operand is not designated on an integral boundary, or when the RT2 field is nonzero and designates the same general register as the RS2 field.

LOAD is a class-1C instruction. It is interruptible, the vector count and vector interruption index determine the number of elements processed, and element selection is affected by neither the vector-mask mode nor the vector-mask register.

Condition Code: The code remains unchanged.

Program Exceptions:
- Access (fetch, operand 2 in VST format)
- Operation
- Specification
- Vector operation

LOAD BIT INDEX

VLBIX VRl,GR3,D2(B2) [RSE]

<table>
<thead>
<tr>
<th>E428</th>
<th>GR3</th>
<th>VRl</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

Bit by bit, the second operand is converted from a bit vector to a vector of element numbers, the result vector is placed in the first-operand location, and the number of elements in the result vector is placed in the vector count.

The result-vector elements are 32-bit signed binary integers, which give the positions of the one bits in the second operand, relative to the starting address of the second operand and in sequence from left to right. No result-vector elements are generated for zero bits.

The GR3 field must designate an even register number to specify an even-odd pair of general registers. The registers contain a bit index and a bit count, as follows:

<table>
<thead>
<tr>
<th>GR3 Bit Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GR3+1 Bit Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
</tr>
</tbody>
</table>

Both are treated as 32-bit signed binary integers. The bit index identifies the first bit of the second operand to be processed. The bit count gives the number of bits to be processed. If the bit count is zero or less than zero, no bits are processed. Upon completion or interruption of the instruction, the bit index identifies the next bit to be processed, and the bit count, if greater than zero, gives the number of bits remaining.

The address of the byte location containing the current bit to be processed is the sum, modulo the address size, of the second-operand address and of a number obtained by shifting bits 0-28 of the current bit index right by three bit positions, with bits equal to bit 0 being shifted into the leftmost three bit positions (without changing the contents of the general register). The rightmost three bits of the current bit index designate the bit within the byte.

Chapter 3. Vector-Facility Instructions 3-11
Execution of the instruction consists of a repetition of the following procedure:

The current value of the vector interruption index is placed in the vector count. Then, if the vector count is equal to the section size, or if the bit count is zero or less than zero, the vector interruption index is set to zero, and instruction execution is completed. Otherwise, the second-operand bit designated by the current bit index is selected. If the selected bit is one, the value of the bit index is placed in the first-operand element location designated by the vector interruption index, and the vector interruption index is then incremented by one. Next, regardless of the value of the selected bit, one is added algebraically to the bit index, and one is subtracted from the bit count. The procedure is then repeated.

Execution of the instruction may be interrupted, but only upon return to the starting point of the repetitive procedure.

When 31-bit addressing is in effect, incrementing the bit index beyond the value $2^{31}$ may cause an overflow, which is not signaled to the program. The result of incrementing the bit index beyond $2^{31}$ is undefined.

A specification exception is recognized when the GRJ field designates an invalid register number.

The B2 field should not designate the same general register as either of the pair of registers designated by the GR3 field. The result fields (bit count, bit index, condition code, vector count, vector interruption index, and vector register) are undefined if B2 is nonzero and $B2 = GR3$ or $B2 = GR3 + 1$.

LOAD BIT INDEX is a class-IG instruction. It is interruptible, a general register and the vector interruption index determine the number of elements processed, and element selection is affected by neither the vector-mask mode nor the vector-mask register. The vector count is set by the instruction.

**Resulting Condition Code:**

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Vector count zero; bit count zero</td>
</tr>
<tr>
<td>1</td>
<td>Vector count zero; bit count less than zero</td>
</tr>
<tr>
<td>2</td>
<td>Vector count equal to section size; bit count greater than zero</td>
</tr>
<tr>
<td>3</td>
<td>Vector count greater than zero; bit count zero or less than zero</td>
</tr>
</tbody>
</table>

**Program Exceptions:**

- Access (fetch, operand 2)
- Operation
- Specification
- Vector operation

**Programming Notes:**

1. Example of LOAD BIT INDEX:

   Bit Positions: 012345678
   Bit Vector: 010001101
   Result Vector: 1 5 6 8

2. The bit index in the even register should normally be set to zero by the program before entering a sectioning loop that contains the instruction. An initial nonzero value may be useful to shorten a bit vector that would otherwise contain a large number of leading zeros.

3. Assuming normal use of the instruction with the vector interruption index initially set to zero, LOAD BIT INDEX sets the vector count to the number of result elements generated. The vector count is then available to control subsequent vector instructions.

   If condition code 2 is set, the vector count has been set to the section size; a full section of element numbers has been loaded by the instruction, and more bits remain to be processed. If condition code 3 is set, the vector count has been set to a value equal to or less than the section size; the last or only section of element numbers has been loaded, and no more bits remain to be processed. If condition code 0 or 1 is set, the vector count is zero, and there were no bits to be processed and no element numbers to be loaded.

4. If all bits in the second operand are zeros, no result elements are generated, and the vector count is set to the initial vector interruption index, which normally is zero. This may also occur for the last pass through a sectioning loop using this instruction, if the number of one bits in the second operand happens to be a multiple of the section size, thus generating one or more full sections, with the remainder of the second operand containing only zero bits. Subsequent vector instructions will still function correctly, because no elements are processed when the vector count is zero.

5. The effect on the result fields of specifying the same general register for the base register of the second operand and for the bit index or bit
count is unpredictable; it may depend on the model, on the occurrence of asynchronous interruptions such as I/O, or on other events that are not under the direct control of the program.

6. Programs using extremely large values of the bit index when 31-bit addressing is in effect must limit those values so that they cannot exceed $2^{31}-1$, which corresponds to a byte location of $2^{28}-1$ relative to the second-operand address. Allowing the instruction to increment the bit index to the next value may or may not cause overflow; the next byte location might be either $2^{28}$ or $-2^{28}$ relative to the second-operand address. The result may not be repeatable from one instruction execution to the next.

When 24-bit addressing is in effect, byte addresses in storage are computed modulo $2^{24}$, so that the possibility of overflow at a bit index of $2^{31}-1$ does not affect the resultant address.

7. Figure 3-1 is a summary of the operation.
LOAD COMPLEMENT

Mnemonic | VR1,VR2 | [VV]
---|---|---
| Op Code | VR1 | VR2 |

16 24 28 31

Mnemonic | Op Code | Operands
---|---|---
VLCR | 'A562' | Binary
VLCDR | 'A552' | Long
VLCE | 'A542' | Short

Element by element, the second-operand vector is placed in the first-operand location with the opposite sign. For VLCR, each result element is the two’s complement of the corresponding source element. For VLCDR and VLCE, each result element is the corresponding source element with the sign bit inverted.

The operation is performed on each element in the same manner as the corresponding scalar operation, except that the condition code is not set.

A specification exception is recognized when a VR field designates an invalid register number.

LOAD COMPLEMENT is a class-1M instruction. It is interruptible, the vector count and vector interruption index determine the number of elements processed, and element selection is affected by both the vector-mask mode and the vector-mask register.

Condition Code: The code remains unchanged.

Program Exceptions:
- Fixed-point overflow (with exception-extension code; VLCR only)
- Operation
- Specification
- Vector operation

LOAD ELEMENT

Mnemonic | VR1,QR3,GR2 | [VR]
---|---|---
| Op Code | QR3 | VR1 | GR2 |

0 16 20 24 28 31

Mnemonic | Op Code | Operands
---|---|---
VLEL | 'A628' | Binary or logical
VLELD | 'A618' | Long
VLELE | 'A608' | Short

The element in the vector register or vector-register pair designated by VR1, which has the element number contained in the general register designated by GR2, is replaced by the scalar operand in the general or floating-point register designated by QR3.

The element number is a 32-bit unsigned binary integer which must be less than the section size.

A specification exception is recognized when the VR1 or QR3 field designates an invalid register number, or when the element number is equal to or greater than the section size.

LOAD ELEMENT is a class-N1 instruction. It is not interruptible, one element is processed, and its execution is affected by neither the vector-mask mode nor the vector-mask register. The vector count and vector interruption index are not used and remain unchanged.

Condition Code: The code remains unchanged.

Program Exceptions:
- Operation
- Specification
- Vector operation

LOAD EXPANDED

Mnemonic | VR1,RS2(RT2) | [VST]
---|---|---
| Op Code | RT2 | VR1 | RS2 |

0 16 20 24 28 31

Mnemonic | Op Code | Operands
---|---|---
VLY | 'A40B' | Binary or logical
VLYD | 'A41B' | Long
VLYE | 'A40B' | Short

3-14 IBM System/370 Vector Operations
Element by element, successive elements of the second-operand vector are placed unchanged in the element locations of the first operand that correspond to ones in the active bits of the vector-mask register. Element locations of the first operand that correspond to zeros in the active bits of the vector-mask register remain unchanged, and there are no corresponding second-operand locations in storage.

A specification exception is recognized when the VR1 field designates an invalid register number, when the second operand is not designated on an integral boundary, or when the RT2 field is nonzero and designates the same general register as the RS2 field.

When the active bits of the vector-mask register are all zeros, no access exceptions are recognized for the storage location specified by the second operand.

LOAD EXPANDED is a class-Ic instruction. It is interruptible, the vector count and vector interruption index determine the number of elements processed, and element selection is affected by the vector-mask register but not by the vector-mask mode.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2)
- Operation
- Specification
- Vector operation

**Programming Notes:**
1. The number of vector elements which are loaded from storage and the amount by which the address in the general register designated by RS2 is updated correspond to the number of ones among the active bits of the vector-mask register.
2. The operation performed by LOAD EXPANDED is the opposite of STORE COMPRESSED.

---

**LOAD HALFWORD**

<table>
<thead>
<tr>
<th>VLH</th>
<th>VR1,RS2(RT2)</th>
<th>[VST]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1A429</td>
<td>//\</td>
<td>RT2</td>
</tr>
</tbody>
</table>

0 16 20 24 28 31

Element by element, the second operand is extended from a vector of 16-bit signed binary integers to a vector of 32-bit signed binary integers, and the result is placed in consecutive first-operand locations.

Each second-operand element is two bytes in length. The element is extended upon loading to 32 bits by setting each of the 16 leftmost bit positions of the first-operand element equal to the sign bit of the second-operand element.

A specification exception is recognized when the second operand is not designated on a halfword boundary, or when the RT2 field is nonzero and designates the same general register as the RS2 field.

LOAD HALFWORD is a class-Ic instruction. It is interruptible, the vector count and vector interruption index determine the number of elements processed, and element selection is affected by neither the vector-mask mode nor the vector-mask register.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2)
- Operation
- Specification
- Vector operation
LOAD INDIRECT

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR1, VR3, D2(B2)</th>
<th>[RSE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>VR3///</td>
<td>VR1///</td>
</tr>
<tr>
<td>0</td>
<td>16 20 24 28 32 36 47</td>
<td></td>
</tr>
</tbody>
</table>

Element by element, the third operand is used to select elements of the second-operand vector in storage and place them unchanged in the element positions of the first operand which correspond to those of the third operand.

The third operand is a vector of 32-bit signed binary integers. The address of each second-operand element is computed as the sum of the second-operand origin and the offset obtained from each element of the third operand, as follows.

The second-operand origin is generated from the base-address (B2) and displacement (D2) fields using the normal rules of address generation. The offset is obtained by shifting the current third-operand element to the left by two bits (for VLI or VLIE) or three bits (for VLID), with zeros appended on the right. The origin and offset are added. The rightmost 31 or 24 bits of the sum, depending on the address size, are used as the storage address. The second-operand element is fetched from that address and loaded into the first-operand location at the same element position as that from which the third-operand element was obtained.

During the shift and addition operations, any carries or shifts into or out of the unused bit positions on the left are ignored.

A specification exception is recognized when the VR1 field designates an invalid register number, or when the second operand is not designated on an integral boundary.

LOAD INDIRECT is a class-IC instruction. It is interruptible, the vector count and vector interruption index determine the number of elements processed, and element selection is affected by neither the vector-mask mode nor the vector-mask register.

Condition Code: The code remains unchanged.

Program Exceptions:
- Access (fetch, operand 2)
- Operation
- Specification
- Vector operation

Programming Note: LOAD INDIRECT is used to load a vector by indirect element selection. The instruction fetches elements from storage in the following sequence of addresses: \( A + w \times E(0), A + w \times E(1), A + w \times E(2), \ldots \), where \( A \) is the origin of the vector in storage, \( w \) is the width of each element, and \( E(0), E(1), E(2), \ldots \) are the element numbers contained in positions 0, 1, 2, \ldots of the vector register designated by the VR3 field of the instruction.

The origin is \( A = (B2) + D2 \), where \( (B2) \) represents the contents of the base register designated by the B2 field, and \( D2 \) is the displacement designated by the D2 field.

The element width \( w \) is 4 for VLI or VLIE and 8 for VLID. The storage elements are loaded successively into element positions 0, 1, 2, \ldots of the target register designated by VR1.

LOAD INTEGER VECTOR

<table>
<thead>
<tr>
<th>VLINT</th>
<th>VR1, RS2(RT2)</th>
<th>[VST]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A42A'</td>
<td>/////</td>
<td>RT2</td>
</tr>
<tr>
<td>0</td>
<td>16 20 24 28 31</td>
<td></td>
</tr>
</tbody>
</table>

Element by element, a vector of uniformly spaced integers, as specified by the second-operand designation, is placed in consecutive first-operand locations.

If the vector interruption index \( X \) is less than the vector count, the contents of the general register designated by RS2 replace element \( X \) of the first operand (normally \( X = 0 \) at the start). Then, the contents of that general register are incremented by adding the contents of the general register designated by RT2 (the stride), both being treated as 32-bit binary integers. Any overflow during the addition is ignored. The vector interruption index \( X \) is then incremented by one.

These steps are repeated for each successive first-operand element until incrementing \( X \) causes it to
equal the vector count. The vector interruption index is then set to zero.

The general register designated by RT₂ remains unchanged. If the RT₂ field of the instruction is zero, general register 0 is not used for the increment; instead, the increment is +1, so that consecutive integers are loaded.

A specification exception is recognized when the RT₂ field is nonzero and designates the same general register as the RS₂ field.

LOAD INTEGER VECTOR is a class-Ic instruction. It is interruptible, the vector count and vector interruption index determine the number of elements processed, and element selection is affected by neither the vector-mask mode nor the vector-mask register.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Operation
- Specification
- Vector operation

**Programming Note:** The operation resembles the generation of storage addresses for QST- and VST-format instructions, except that the element size w is 1, no storage references for operands take place, no access exceptions for operands are recognized, and all 32 bits of both general registers participate in the operation. The result is independent of the address size.

Performing a LOAD INTEGER VECTOR operation also resembles the execution of a loop using the nonvector instruction LOAD ADDRESS. They differ in that LOAD INTEGER VECTOR does not depend on the address size; it does not set to zeros the leftmost one or eight bit positions. LOAD INTEGER VECTOR can generate negative numbers, which LOAD ADDRESS cannot.

### LOAD MATCHED

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR₁,QR₂</th>
<th>[QV]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>QR₂</td>
<td>VR₁</td>
</tr>
<tr>
<td>0</td>
<td>16 20 24 28 31</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLMQ</td>
<td>'A5AA'</td>
<td>Binary or logical</td>
</tr>
<tr>
<td>VLMDQ</td>
<td>'A59A'</td>
<td>Long</td>
</tr>
<tr>
<td>VLMEQ</td>
<td>'A58A'</td>
<td>Short</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR₁,RS₂(RT₂)</th>
<th>[VST]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>RT₂</td>
<td>VR₁</td>
</tr>
<tr>
<td>0</td>
<td>16 20 24 28 31</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLM</td>
<td>'A40A'</td>
<td>Binary or logical</td>
</tr>
<tr>
<td>VLMD</td>
<td>'A41A'</td>
<td>Long</td>
</tr>
<tr>
<td>VLME</td>
<td>'A40A'</td>
<td>Short</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR₁,VR₂</th>
<th>[VV]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>VR₁</td>
<td>VR₂</td>
</tr>
<tr>
<td>0</td>
<td>16 24 28 31</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLMR</td>
<td>'A59A'</td>
<td>Binary or logical</td>
</tr>
<tr>
<td>VLMDR</td>
<td>'A51A'</td>
<td>Long</td>
</tr>
<tr>
<td>VLMER</td>
<td>'A50A'</td>
<td>Short</td>
</tr>
</tbody>
</table>

Element by element, elements of the second operand corresponding to ones in the active bits of the vector-mask register are placed unchanged in the corresponding element locations of the first operand. Elements of the second operand corresponding to zeros in the active bits of the vector-mask register are not loaded, and the corresponding element locations of the first operand remain unchanged.

A specification exception is recognized when a VR or QR field designates an invalid register number. In the VST format, a specification exception is also recognized when the second operand is not designated on an integral boundary, or when the RT₂ field is nonzero and designates the same general register as the RS₂ field.

No access exceptions are recognized for elements of the second operand which correspond to zeros in the active bits of the vector-mask register; however, the general register designated by the RS₂ field is updated for each of those elements.

Chapter 3. Vector-Facility Instructions 3-17
LOAD MATCHED is a class-1C instruction. It is interruptible, the vector count and vector interruption index determine the number of elements processed, and element selection is affected by the vector-mask register but not by the vector-mask mode.

Condition Code: The code remains unchanged.

Program Exceptions:
- Access (fetch, operand 2 in VST format)
- Operation
- Specification
- Vector operation

Programming Notes:
1. The instructions LOAD and LOAD MATCHED, in corresponding formats, perform the same function on those elements which correspond to ones in the active bits of the vector-mask register; that is, each such element is loaded from the same storage location into the same vector-register position. LOAD MATCHED differs in that elements in storage corresponding to zeros in the active bits of the vector-mask register are skipped.

2. LOAD, LOAD EXPANDED, and LOAD MATCHED, in corresponding formats, perform the same function when all active bit positions of the vector-mask register contain ones.

LOAD NEGATIVE

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR1,VR2</th>
<th>[VV]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>VR1</td>
<td>VR2</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

Element by element, the negative of the absolute value of the second-operand vector is placed in the first-operand location.

The operation is performed on each element in the same manner as the corresponding scalar operation, except that the condition code is not set.

A specification exception is recognized when a VR field designates an invalid register number.

LOAD POSITIVE

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR1,VR2</th>
<th>[VV]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>VR1</td>
<td>VR2</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

Element by element, the absolute value of the second-operand vector is placed in the first-operand location.

The operation is performed on each element in the same manner as the corresponding scalar operation, except that the condition code is not set.

A specification exception is recognized when a VR field designates an invalid register number.
LOAD VCT AND UPDATE

If the operand in the general register designated by the GR field is greater than zero, the vector count (VCT) is replaced by the lesser of the section size and the operand. If the operand is zero or less than zero, the vector count is set to zero. The general register is then updated by subtracting the new vector count from the register contents.

The register contents are treated as a 32-bit signed binary integer. The vector count and section size are treated as 16-bit unsigned binary integers.

LOAD VCT AND UPDATE is a class-No instruction. It is not interruptible, no elements are processed, and its execution is affected by neither the vector-mask mode nor the vector-mask register. The vector count is set. The vector interruption index is not used and remains unchanged.

Resulting Condition Code:

0 Vector count zero; register result zero
1 Vector count zero; register result less than zero
2 Vector count equal to section size; register result greater than zero
3 Vector count greater than zero; register result zero

Program Exceptions:

- Operation
- Vector operation

Programming Notes:

1. LOAD VCT AND UPDATE may be used at the start of a sectioning loop to determine the number of vector elements to be processed during each pass through the loop. Before entering the loop, the program initializes the general-register operand to the total number of elements in the vector. The end of the loop may simply be a BRANCH ON CONDITION instruction, if the condition code has not been changed since the start of the loop, or the branch may be preceded by LOAD AND TEST specifying the general register as both the first and second operand.

If LOAD VCT AND UPDATE sets condition code 2, the vector count has been set to the section size; a full section of vector elements are to be processed, and more remain to be processed. If it sets condition code 3, the vector count has a value equal to or less than the section size, and the last or only section is to be processed. If it sets condition code 0 or 1, the vector count is zero, and there are no vector elements to be processed.

2. If LOAD AND TEST is used instead at the end of the loop, condition code 2 simply indicates that the general register contents are greater than zero, and there are more elements to be processed. Any other condition code means that there are no more elements.

3. The general-register operand remains greater than zero at the end of instruction execution only if condition code 2 is set. For the other condition codes, the final register contents are zero or negative.

LOAD VCT FROM ADDRESS

If the second-operand-address value is greater than zero, the vector count (VCT) is replaced by the lesser of the section size and the address value. If the second-operand-address value is zero or less than zero, the vector count is set to zero.

If the B2 field of the instruction is not zero, the second-operand-address value is formed by adding the contents of the general register designated by the B2 field and the contents of the 12-bit D2 field of the instruction. All 32 bits in the general register designated by the B2 field participate in the addition, which is independent of the address size. The result of the addition is used as the operand itself and not to address storage. It is treated as a 32-bit signed binary integer.

If the B2 field of the instruction is zero, general register 0 is not used; instead, the address value consists of the D2 field with 20 zero bits appended on the left.
No storage references for operands take place, and the address value is not inspected for boundary alignment or access exceptions.

LOAD VCT FROM ADDRESS is a class-N0 instruction. It is not interruptible, no elements are processed, and its execution is affected by neither the vector-mask mode nor the vector-mask register. The vector count is set. The vector interruption index is not used and remains unchanged.

Resulting Condition Code:

0  Vector count zero; second-operand address zero
1  Vector count zero; second-operand address less than zero
2  Vector count equal to section size; second-operand address greater than section size
3  Vector count greater than zero; second-operand address less than or equal to section size and greater than zero

Program Exceptions:

- Operation
- Vector operation

Programming Note: LOAD VCT FROM ADDRESS may be used to set the vector count to the section size by specifying a B2 field of zero and placing a value greater than 511 in the D2 field.

LOAD VMR

VLVM RS2 [VS]

'A680' ///////// RS2
0 16 28 31

The second-operand bit vector replaces the active bits of the vector-mask register (VMR). Bits beyond the active bits are set to zeros.

LOAD VMR is a class-NC instruction. It is not interruptible, the vector count determines the number of bits processed, and bit selection is affected by neither the vector-mask mode nor the vector-mask register. The vector interruption index is not used and remains unchanged.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2)
- Operation
- Vector operation

LOAD VMR COMPLEMENT

VLVM RS2 [VS]

'A681'///////// RS2
0 16 28 31

The complement of the bits from the second-operand bit vector replaces the active bits of the vector-mask register (VMR). Bits beyond the active bits are set to zeros.

LOAD VMR COMPLEMENT is a class-NC instruction. It is not interruptible, the vector count determines the number of bits processed, and bit selection is affected by neither the vector-mask mode nor the vector-mask register. The vector interruption index is not used and remains unchanged.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2)
- Operation
- Vector operation

LOAD ZERO

Mnemonic VR1 [VV]

Op Code ///////// VR1 ///
0 16 24 28 31

Mnemonic Op Code Operands
VLZB 'A50B' Binary or logical
VLZD 'A51B' Long
VLZER 'A50B' Short

The first-operand vector is set to zero. Only element positions numbered less than the vector count are set to zero. Any element positions numbered equal to or greater than the vector count remain unchanged.

A specification exception is recognized when the VR1 field designates an invalid register number.
LOAD ZERO is a class-IC instruction. It is interruptible, the vector count and vector interruption index determine the number of element positions set to zero, and element selection is affected by neither the vector-mask mode nor the vector-mask register.

Condition Code: The code remains unchanged.

Program Exceptions:
- Operation
- Specification
- Vector operation

Programming Note: The instruction LOAD ZERO is equivalent to LOAD (VLQ, VLDQ, or VLEQ) with an implied scalar source operand of zero. It provides the fastest way to set a vector register to zero.

MAXIMUM ABSOLUTE

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR1,FR3,GR2</th>
<th>[VR]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>FR3 /////</td>
<td>VR1</td>
</tr>
<tr>
<td>0</td>
<td>16 20 24 28 31</td>
<td></td>
</tr>
</tbody>
</table>

Mnemonic Op Code Operands
VMXAD 'A612' Long
VMXAE 'A602' Short

MAXIMUM SIGNED

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR1,FR3,GR2</th>
<th>[VR]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>FR3 /////</td>
<td>VR1</td>
</tr>
<tr>
<td>0</td>
<td>16 20 24 28 31</td>
<td></td>
</tr>
</tbody>
</table>

Mnemonic Op Code Operands
VMXSD 'A610' Long
VMXSE 'A600' Short

MINIMUM SIGNED

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR1,FR3,GR2</th>
<th>[VR]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>FR3 /////</td>
<td>VR1</td>
</tr>
<tr>
<td>0</td>
<td>16 20 24 28 31</td>
<td></td>
</tr>
</tbody>
</table>

Mnemonic Op Code Operands
VMNSD 'A611' Long
VMNSE 'A601' Short

The scalar third operand and all first-operand vector elements are compared to determine the maximum or minimum value, which replaces the third operand. The instruction MAXIMUM ABSO LUTE compares absolute values to select the maximum. The instructions MAXIMUM SIGNED and MINIMUM SIGNED compare signed values to select the maximum or minimum, respectively.

The comparison of each pair of absolute or signed operand values is performed in the same manner as the scalar floating-point COMPARE instruction for the same format, except that the result is the selection of one element of the pair instead of a condition-code setting.

The scalar third operand is compared with each element of the first operand in turn to determine the selected (maximum absolute, maximum signed, or minimum signed) value. If the comparison is unequal and the first-operand element is the selected value, the first-operand element replaces the third operand; otherwise, no change takes place. The operation then continues with the next element of the first operand in the sequence of element numbers.

The GR2 field must be zero or even. When nonzero, it designates an even-odd pair of general registers. The contents of the odd general register are treated as a 32-bit unsigned binary integer, which is incremented by one after each first-operand element has been processed; any carry out of bit position 0 is ignored. Each time a new selected value replaces the third operand, the current contents of the odd general register, before it is incremented, are placed in the even general register.

When the GR2 field is zero, the action associated with the general registers is not performed, and their contents remain unchanged.

For VMXAE, VMXSE, and VMNSE, the rightmost 32 bits of the floating-point register designated by FR3 remain unchanged.

A specification exception is recognized when the VR1, GR2, or FR3 field designates an invalid register number.

MAXIMUM ABSOLUTE, MAXIMUM SIGNED, and MINIMUM SIGNED are class-IC instructions. They are interruptible, the vector count and vector interruption index determine the number of elements processed, and element selection is affected by both the vector-mask mode and the vector-mask register. When the vector-mask mode is on, no selection
takes place for first-operand elements corresponding to zero mask bits: the third operand and the even general register remain unchanged. However, when the GR₂ field is nonzero, the odd general register is incremented by one for every first-operand element, regardless of the mode and mask bits.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Operation
- Specification
- Vector operation

**Programming Notes:**

1. Because the current third operand is compared with every element of the first operand, including element 0, these instructions can be used in a sectioning loop to find the selected value of a vector of any length. Before starting the first, or only, section, the program should initialize the third operand as follows.

   MAXIMUM ABSOLUTE: zero
   MAXIMUM SIGNED: largest negative value
   MINIMUM SIGNED: largest positive value

2. If the GR₂ field is not zero, and the program initializes both of the specified pair of general registers to zero before executing the instruction, the even register will contain the number of the selected element, counting from the start (element 0) of the first section. If no element was selected, the even register will retain its initial contents. The odd register will contain the cumulative number of elements processed. When the first operand contains two or more elements that could equally qualify as the selected element, the instruction selects the first one.

3. Since the element values are floating-point numbers, the rules for floating-point comparison apply, and two or more elements with different bit patterns may satisfy the test for maximum or minimum value. For example, elements with zero fractions compare equal even though their sign and characteristic may differ. (See also the programming notes for the **COMPARE** instruction in Chapter 9, “Floating-Point Instructions,” of **IBM 370-XA Principles of Operation** and **IBM System/370 Principles of Operation**.)

---

**MULTIPLY**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR₁,QR₁,RS₁(RT₁)</th>
<th>[QST]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>QR₁</td>
<td>RT₁</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR₁,QR₁,RS₁(RT₁)</th>
<th>[QV]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>QR₁</td>
<td>///</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR₁,QR₁,RS₁(RT₁)</th>
<th>[VST]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>VR₁</td>
<td>RT₁</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR₁,QR₁,RS₁(RT₁)</th>
<th>[VV]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>VR₁</td>
<td>///</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

Element by element, the product of the second operand and the third operand is placed in the first-operand location. The operation is performed on each pair of elements in the same manner as the corresponding scalar operation, except for the following differences:

- For binary operands, the third-operand designation may be any register number. Each element of the third operand is a 32-bit signed binary integer, as is each element of the second operand. The first-operand location is a
vector-register pair, which receives product elements consisting of 64-bit signed binary integers.

- For floating-point operands, the operands are not first normalized. When one or both of the source-operand elements have a nonzero fraction with a leftmost hexadecimal digit of zero, an unnormalized-operand exception is recognized, and the unit of operation is inhibited.

A specification exception is recognized when a VR or QR field designates an invalid register number. In the QST and VST formats, a specification exception is recognized when the second operand is not designated on an integral boundary, or when the RT2 field is nonzero and designates the same general register as the RS2 field. For the VMS instruction, a specification exception is also recognized when the QR3 field designates the same general register as the RS2 field.

**MULTIPLY** is a class-1M instruction. It is interruptible, the vector count and vector interruption index determine the number of elements processed, and element selection is affected by both the vector-mask mode and the vector-mask register.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**

- Access (fetch, operand 2 in QST and VST formats)
- Exponent overflow (with exception-extension code; floating-point operands only)
- Exponent underflow (with exception-extension code; floating-point operands only)
- Operation
- Specification
- Unnormalized operand (with exception-extension code; floating-point operands only)
- Vector operation

**MULTIPLY AND ACCUMULATE**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR1, VR3, RS2(RT2) [VST]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>VR1</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMCD</td>
<td>'A416'</td>
<td>Short multiplier and multiplicand; long first operand, product, and sum</td>
</tr>
<tr>
<td>VMCE</td>
<td>'A406'</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR1, VR3, VR2 [VV]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>VR1</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMCDR</td>
<td>'A516'</td>
<td>Short multiplier and multiplicand; long first operand, product, and sum</td>
</tr>
<tr>
<td>VMCER</td>
<td>'A506'</td>
<td></td>
</tr>
</tbody>
</table>

Partial sums of the products of corresponding elements of the second and third operands are accumulated by adding the products to the contents of element positions 0 to p−1 of the first operand. The partial-sum number p depends on the model.

The operation proceeds in an ascending sequence of element numbers. The product of the lth elements of the second and third operands is added to the first-operand element at a position which is the remainder of dividing l by p, where l varies from X to C−1, X is the initial vector interruption index (normally zero), and C is the vector count. The operation accumulates C-X element products.

Thus, the products formed from second- and third-operand elements 0, p, 2p, ..., are accumulated into position 0 of the first operand; products from elements l, p+1, 2p+1, ..., are accumulated into position l; etc. The contents of first-operand element positions above p−1 remain unchanged.

Every multiplication is performed in the same manner as the corresponding scalar floating-point, short or long, **MULTIPLY** instruction, except that the operand elements are not first normalized. Every addition is performed in the same manner as the scalar instruction **ADD NORMALIZED** (ADR), except that the condition code is not set.

When one or both of a pair of second- and third-operand elements have a nonzero fraction with a
leftmost hexadecimal digit of zero, an unnormalized-operand exception is recognized, and the unit of operation is inhibited.

If the multiplication of an element pair results in an exponent underflow, a true zero is used in place of the product in the addition operation, and no exception is recognized. If the multiplication results in an exponent overflow, the product replaces the corresponding partial-sum element, and an exponent overflow is recognized. Exceptions in the addition are recognized in the same manner as for the scalar instruction ADD NORMALIZED (ADR).

A specification exception is recognized when a VR field designates an invalid register number. In the VST format, a specification exception is also recognized when the second operand is not designated on an integral boundary, or when the RT2 field is nonzero and designates the same general register as the RS2 field.

MULTIPLY AND ACCUMULATE is a class-1M instruction. It is interruptible, the vector count and vector interruption index determine the number of elements processed, and element selection is affected by both the vector-mask mode and the vector-mask register.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2 in VST format)
- Exponent overflow (with exception-extension code)
- Exponent underflow (with exception-extension code)
- Operation
- Significance (with exception-extension code)
- Specification
- Unnormalized operand (with exception-extension code)
- Vector operation

### MULTIPLY AND ADD

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR1, FR3, RS2(RT2) [QST]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>FR3</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMADS</td>
<td>'A494'</td>
<td>Long</td>
</tr>
<tr>
<td>VMAES</td>
<td>'A484'</td>
<td>Short multiplier and multiplicand; long first operand, product, and sum</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR1, FR3, VR2 [QV]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>FR3</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMADQ</td>
<td>'A594'</td>
<td>Long</td>
</tr>
<tr>
<td>VMAEQ</td>
<td>'A584'</td>
<td>Short multiplier and multiplicand; long first operand, product, and sum</td>
</tr>
</tbody>
</table>

### MULTIPLY AND SUBTRACT

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR1, FR3, RS2(RT2) [QST]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>FR3</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMADS</td>
<td>'A495'</td>
<td>Long</td>
</tr>
<tr>
<td>VMAES</td>
<td>'A485'</td>
<td>Short multiplier and multiplicand; long first operand, product and difference</td>
</tr>
</tbody>
</table>

3-24 IBM System/370 Vector Operations
### Mnemonic VR1,FR2,VR2 [QV]

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>FR2</th>
<th>VR1</th>
<th>VR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRMSDQ</td>
<td>'A595'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VMSEQ</td>
<td>'A585'</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRMSDQ</td>
<td>'A415'</td>
<td>Long</td>
</tr>
<tr>
<td>VMSEQ</td>
<td>'A405'</td>
<td>Short multiplier and multiplicand; long first operand, product and difference</td>
</tr>
</tbody>
</table>

### Mnemonic VR1,VR2,RS2(RT2) [VST]

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>VR3</th>
<th>RT2</th>
<th>VR1</th>
<th>RS2</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRMSDQ</td>
<td>'A415'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VMSEQ</td>
<td>'A405'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Element by element, the third operand is multiplied by the second-operand vector, and the product is added to, or subtracted from, the first-operand vector. The sum or difference is placed in the first-operand location.

Every multiplication is performed in the same manner as the corresponding scalar floating-point, short or long, MULTIPLY instruction, except that the operand elements are not first normalized. Every addition or subtraction is performed in the same manner as the scalar instruction ADD NORMALIZED (ADR) or SUBTRACT NORMALIZED (SDR), respectively, except that the condition code is not set.

When one or both of a pair of second- and third-operand elements have a nonzero fraction with a leftmost hexadecimal digit of zero, an unnormalized-operand exception is recognized, and the unit of operation is inhibited.

If the multiplication of an element pair results in an exponent underflow, a true zero is used in place of the product in the addition or subtraction operation, and no exception is recognized. If the multiplication of an element pair results in an exponent overflow, the corresponding product replaces the first-operand element, and an exponent overflow is recognized. Exceptions in the addition or subtraction are recognized in the same manner as for the scalar instruction ADD NORMALIZED (ADR) or SUBTRACT NORMALIZED (SDR), respectively.

A specification exception is recognized when a VR or FR field designates an invalid register number. In the QST and VST formats, a specification exception is also recognized when the second operand is not designated on an integral boundary, or when the RT2 field is nonzero and designates the same general register as the RS2 field.

MULTIPLY AND ADD and MULTIPLY AND SUBTRACT are class-IM instructions. They are interruptible, the vector count and vector interruption index determine the number of elements processed, and element selection is affected by both the vector-mask mode and the vector-mask register.

### Condition Code: The code remains unchanged.

### Program Exceptions:

- Access (fetch, operand 2 in QST and VST formats)
- Exponent overflow (with exception-extension code)
- Exponent underflow (with exception-extension code)
- Operation
- Significance (with exception-extension code)
- Specification
- Unnormalized operand (with exception-extension code)
- Vector operation

### Programming Notes:

1. The MULTIPLY AND ADD and MULTIPLY AND SUBTRACT operations may be summarized as:
   \[ op1 = op1 \pm op1 \times op2 \]

2. If the constant 1.0 is placed in the third-operand location, MULTIPLY AND ADD (VMABS or VMABEQ) and MULTIPLY AND SUBTRACT (VMSES or VMSEQ) may be used to add (subtract) a vector in the short format to (from) a vector in the long format.
Element by element, the OR of the second and third operands is placed in the first-operand location.

The operation is performed on each pair of 32-bit elements in the same manner as the corresponding scalar operation, except that the condition code is not set.

For the vo and vos instructions, a specification exception is recognized when the second operand is not designated on an integral boundary, or when the RT2 field is nonzero and designates the same general register as the RS2 field. For the vos instruction, a specification exception is also recognized when the GRJ field designates the same general register as the RS2 field.

OR is a class-IM instruction. It is interruptible, the vector count and vector interruption index determine the number of bits processed, and bit selection is affected by neither the vector-mask mode nor the vector-mask register. The vector interruption index is not used and remains unchanged.

Condition Code: The code remains unchanged.

Program Exceptions:
- Access (fetch, operand 2)
- Operation
- Vector operation

The OR of the second-operand bit vector and of the active bits of the vector-mask register (VMR) is placed in the vector-mask register. Bits beyond the active bits are set to zeros.

OR TO VMR is a class-NC instruction. It is not interruptible, the vector count determines the number of bits processed, and bit selection is affected by neither the vector-mask mode nor the vector-mask register. The vector interruption index is not used and remains unchanged.

Condition Code: The code remains unchanged.

Program Exceptions:
- Access (fetch, operand 2)
- Operation
- Vector operation

RESTORE VAC

Bits 8-63 of the vector-activity count (VAC) are replaced by bits 8-63 of the doubleword designated by the second-operand address; bits 0-7 of the VAC are set to zeros. Execution of this instruction does not increment the vector-activity count and leaves the loaded value unchanged.

The operand must be designated on a doubleword boundary; otherwise, a specification exception is recognized.
RESTORE VAC is a class-NO instruction. It is not interruptible, no elements are processed, and its execution is affected by neither the vector-mask mode nor the vector-mask register. The vector count and vector interruption index are not used and remain unchanged.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2)
- Operation
- Privileged operation
- Specification
- Vector operation

RESTORE VMR

<table>
<thead>
<tr>
<th>VMRRS</th>
<th>D2(B2)</th>
<th>[S]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A6C3'</td>
<td>B2</td>
<td>D2</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

The second operand replaces the entire contents of the vector-mask register (VMR).

The length of the second operand is 4Z bits (Z/2 bytes), where Z is the section size. The contents of only the first Z bits are necessarily fetched and placed in the VMR; additional bits may or may not be fetched from the second operand, and access exceptions may or may not be recognized for that portion of the operand.

RESTORE VMR is a class-NZ instruction. It is not interruptible, the section size determines the number of bits processed, and bit selection is affected by neither the vector-mask mode nor the vector-mask register. The vector count and vector interruption index are not used and remain unchanged.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2)
- Operation
- Vector operation

RESTORE VR

<table>
<thead>
<tr>
<th>VRRS</th>
<th>GR1</th>
<th>[RRE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A648'</td>
<td>///</td>
<td>GR1</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

If the vector in-use bit associated with a specified pair of vector registers is one, the contents of those vector registers are replaced by consecutive doublewords from a storage area called the save area of the vector-register pair. If the vector in-use bit is zero, the vector registers remain unchanged. In either case, the address of the save area is incremented to the location of the save area of the next pair of vector registers.

The GR1 field must designate an even register number to specify an even-odd pair of general registers. The odd general register contains two 16-bit unsigned binary integers as follows: bits 0-15 contain an element number, which designates the location of the first element pair in the vector-register (VR) pair designated by bits 16-31. The even general register contains a save-area address, which identifies the storage location of the element pair specified by the odd general register.

Graphically, the general-register contents may be represented as follows:

<table>
<thead>
<tr>
<th>Element Number</th>
<th>VR Pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
</tr>
</tbody>
</table>

Depending on the address size, the rightmost 31 or 24 bits of the contents of the even general register are used as the save-area address. When the general register is updated to the address of the next location, the leftmost one or eight bit positions, respectively, of the general register are set to zeros.

If the instruction is interrupted, the save-area-address and element-number fields have been updated to indicate the next element to be processed in the current save area and vector registers.

At the completion of the instruction, the save-area-address field is updated to the storage location of the next pair of vector registers, the element-number field is set to zero, and the VR-pair field is
incremented by 2. If vector-register pair 14 was just restored, the VR-pair field is set to 16, and the save-area-address field is set to the next address following the end of the save area of vector-register pair 14.

At the start of execution, the VR-pair field must be an even number from 0 to 14, and the element-number field must be less than the section size; also, whether or not the storage location will be accessed, the starting address of the save area for the current VR pair must be on a boundary which is a multiple of 8 times the section size.

The starting addresses of the save areas for the current and next pair of vector registers are given in the following formulas:

\[
\begin{align*}
SAC &= SAF - 8 \times ENF \\
SAN &= SAC + 8 \times SS
\end{align*}
\]

evaluated modulo the address size, where:

- **ENF**: Contents of the element-number field at the beginning of the operation (normally zero)
- **SAC**: Starting address of save area for the current VR pair
- **SAF**: Contents of the save-area-address field at the beginning of the operation
- **SAN**: Starting address of save area for the next VR pair
- **SS**: Section size

If the vector in-use bit examined was associated with vector-register pair 14 and 15, condition code 0 or 2 is set according to whether the bit was zero or one, respectively. If the vector in-use bit examined was associated with any other register pair, condition code 1 or 3 is set according to whether the bit was zero or one, respectively.

When the CPU is in the problem state, and the vector in-use bit of the specified pair of vector registers is one, execution of this instruction sets the vector change bit of the vector-register pair to one; execution in the supervisor state does not alter the vector change bits.

A specification exception is recognized when at the start of execution:

- The GR1 field designates an odd register number.
- The starting address of the save area is not a multiple of 8 times the section size.

**RESTORE VR** is a class-1Z instruction. It is interruptible, the section size and element-number field determine the number of elements processed, and element selection is affected by neither the vector-mask mode nor the vector-mask register. The vector count and vector interruption index are not used and remain unchanged.

### Resulting Condition Code:

0 VRs 14 and 15 examined and not restored
1 VR pair other than 14 and 15 examined and not restored
2 VRs 14 and 15 restored
3 VR pair other than 14 and 15 restored

**Program Exceptions:**

- Access (fetch, save-area location)
- Operation
- Specification
- Vector operation

**Programming Note:** See the section “Program Use of the Restore and Save Instructions” on page 2-27 for a discussion of the use of the instructions RESTORE VR, SAVE CHANGED VR, and SAVE VR.

### RESTORE VSR

<table>
<thead>
<tr>
<th>VSRRS</th>
<th>( D_3 (B_2) )</th>
<th>[S]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A6C2'</td>
<td>B_2</td>
<td>D_2</td>
</tr>
</tbody>
</table>

When the CPU is in the problem state, the vector in-use bit of the specified pair of vector registers is one, execution of this instruction sets the vector change bit of the vector-register pair to one; execution in the supervisor state does not alter the vector change bits.

A specification exception is recognized when at the start of execution:

- The GR1 field designates an odd register number.
- The starting address of the save area is not a multiple of 8 times the section size.

The vector in-use bits, bits 48-55 of the vector-status register, and the vector change bits, bits 56-63 of the register, are set in pairs sequentially from left to right, a vector in-use bit being set together with the corresponding vector change bit.

If the second operand specifies that a vector in-use bit is to be set to one, it is set to one. The setting of the corresponding vector change bit depends on
whether the instruction is executed in the supervisor or problem state. If the vector in-use bit is set to one while in the supervisor state, the vector change bit is set to the value specified by the second operand. If the vector in-use bit is set to one while in the problem state, the vector change bit is set to one, ignoring the second operand.

If the second operand specifies that a vector in-use bit is to be set to zero, the old setting of the vector in-use bit is first tested before it is changed. If the old setting was one, all element positions of the associated pair of vector registers are cleared to zeros, and both the vector in-use bit and the corresponding vector change bit are then set to zeros. If the old setting was zero, both the vector in-use bit and the corresponding vector change bit are simply set to zeros.

A specification exception is recognized if any of the following is true:

- The second operand is not designated on a doubleword boundary.
- The value to be placed in bit positions 0-14 of the vector-status register is not all zeros.
- The value to be placed in the vector count, bits 16-31 of the vector-status register, is greater than the section size.
- The value to be placed in the vector interruption index, bits 32-47 of the vector-status register, is greater than the section size.

SAVE CHANGED VR

VR SVC GR1 [RRE]

A specification exception is recognized when at the start of execution:

- The GR1 field designates an odd register number.
- The starting address of the save area is not a multiple of 8 times the section size.
- The element number is equal to or greater than the section size.
- The VR-pair field contains other than an even number from 0 to 14.

Condition Code: The code remains unchanged.

Program Exceptions:
- Access (fetch, operand 2)
- Operation
- Specification
- Vector operation
The vector count and vector interruption index are not used and remain unchanged.

Resulting Condition Code:
0 VRs 14 and 15 examined and not saved
1 VR pair other than 14 and 15 examined and not saved
2 VRs 14 and 15 saved
3 VR pair other than 14 and 15 saved

Program Exceptions:
- Access (store, save-area location)
- Operation
- Privileged operation
- Specification
- Vector operation

Programming Notes:
1. The operation is the same as for SAVE VR, except that the instruction is privileged, the vector change bit takes the place of the vector in-use bit, and the vector change bit is set to zero after a vector-register pair is saved. The effect is that a vector-register pair is saved only if it has been loaded or modified since the last use of SAVE CHANGED VR designating this pair.

   If the vector in-use bit is zero, the vector change bit is also zero, so that neither instruction will perform a save operation.

2. See the section "Program Use of the Restore and Save Instructions" on page 2-27 for a discussion of the use of the instructions RESTORE VR, SAVE CHANGED VR, and SAVE VR.

SAVE VAC

```
VACSV D2(B2) [S]

\[ \begin{array}{c|c|c}
| 'A6CA' | B2 | D2 \\
|---|---|---|
| 0 | 16 | 20 | 31 \\
\end{array} \]
```

The current value of the vector-activity count (VAC) is stored at the doubleword designated by the second-operand address. Execution of this instruction does not increment the vector-activity count and leaves its value unchanged.

SAVE VMR

```
VMRSV D2(B2) [S]

\[ \begin{array}{c|c|c}
| 'A6C1' | B2 | D2 \\
|---|---|---|
| 0 | 16 | 20 | 31 \\
\end{array} \]
```

The contents of the entire vector-mask register (VMR) are placed unchanged in storage at the second-operand location.

The length of the second operand is 4Z bits (Z/2 bytes), where Z is the section size. Only the first Z bits of the result are defined to be the VMR contents; the remaining 3Z bits of the result are undefined, and storing of that part of the result may or may not take place.

SAVE VMR is a class-N0 instruction. It is not interruptible, the section size determines the number of bits processed, and bit selection is affected by neither the vector-mask mode nor the vector-mask register. The vector count and vector interruption index are not used and remain unchanged.

Condition Code: The code remains unchanged.

Program Exceptions:
- Access (store, operand 2)
- Operation
- Privileged operation
- Specification
- Vector operation
SAVE VR

VRSV  GR1  [RRE]

<table>
<thead>
<tr>
<th>'A64A'</th>
<th>/ / / / / /</th>
<th>GR1</th>
<th>/ / / /</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

If the vector in-use bit associated with a specified pair of vector registers is one, the contents of those vector registers are placed in consecutive doublewords of a storage area called the save area of the vector-register pair. If the vector in-use bit is zero, the vector registers are not stored. In either case, the address of the save area is incremented to the location of the save area of the next pair of vector registers.

The operand parameters, their updating, and the condition-code setting are the same as for the instruction RESTORE VR.

A specification exception is recognized when at the start of execution:

- The GR1 field designates an odd register number.
- The starting address of the save area is not a multiple of 8 times the section size.
- The element number is equal to or greater than the section size.
- The VR-pair field contains other than an even number from 0 to 14.

SAVE VR is a class-I2 instruction. It is interruptible, the section size and element-number field determine the number of elements processed, and element selection is affected by neither the vector-mask mode nor the vector-mask register. The vector count and vector interruption index are not used and remain unchanged.

Resulting Condition Code:

0  VRs 14 and 15 examined and not saved
1  VR pair other than 14 and 15 examined and not saved
2  VRs 14 and 15 saved
3  VR pair other than 14 and 15 saved

Program Exceptions:

- Access (store, save-area location)
- Operation
- Specification
- Vector operation

Programming Note: See the section "Program Use of the Restore and Save Instructions" on page 2-27 for a discussion of the use of the instructions RESTORE VR, SAVE CHANGED VR, and SAVE VR.

SAVE VSR

VSRSV  D2(B2)  [S]

<table>
<thead>
<tr>
<th>'A6C0'</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

The contents of the vector-status register (VSR) are placed in storage at the doubleword location designated by the second-operand address, except that, when the CPU is in the problem state, the value of the vector change bits stored by the instruction is undefined.

A specification exception is recognized when the second operand is not designated on a doubleword boundary.

SAVE VSR is a class-N0 instruction. It is not interruptible, no elements are processed, and its execution is affected by neither the vector-mask mode nor the vector-mask register. The vector count and vector interruption index are not used and remain unchanged.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (store, operand 2)
- Operation
- Specification
- Vector operation
SET VECTOR MASK MODE

VSVM M D2 (B2) [S]

<table>
<thead>
<tr>
<th>'A6C6'</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

The vector-mask mode is set on or off, depending on whether the rightmost bit, bit 31, of the second-operand address is one or zero, respectively. The second-operand address is not used to address data, and all address bits other than bit 31 are ignored.

SET VECTOR MASK MODE is a class-No instruction. It is not interruptible, no elements are processed, and its execution is not affected by the vector-mask register. The vector-mask mode is set. The vector count and vector interruption index are not used and remain unchanged.

Condition Code: The code remains unchanged.

Program Exceptions:
- Operation
- Vector operation

SHIFT LEFT SINGLE LOGICAL

VSLL VR1, VR3, D2 (B2) [RSE]

<table>
<thead>
<tr>
<th>'E425'</th>
<th>VR3</th>
<th>VR1</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

One by one, the elements in the third-operand vector are shifted left (VSLL) or right (VSRL) by the number of bits specified by the second-operand address, and the result is placed in the first-operand location.

The operation is performed on each element in the same manner as the corresponding scalar operation.

SHIFT RIGHT SINGLE LOGICAL and SHIFT LEFT SINGLE LOGICAL are class-IM instructions. They are interruptible, the vector count and vector interruption index determine the number of elements processed, and element selection is affected by both the vector-mask mode and the vector-mask register.

Condition Code: The code remains unchanged.

Program Exceptions:
- Access (store, operand 2)
- Operation
- Specification
- Vector operation
STORE COMPRESSED

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>VR₁,RS₂(RT₂)</th>
<th>[VST]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>0</td>
<td>16</td>
</tr>
</tbody>
</table>

Element by element, elements of the first-operand vector corresponding to ones in the active bits of the vector-mask register are placed unchanged in storage at successive element locations of the second operand.

First-operand elements corresponding to zeros in the active bits of the vector-mask register are skipped, and there are no corresponding element locations of the second operand. If the active bits of the vector-mask register are all zeros, no access exceptions are recognized for the storage location specified by the second operand, the change bits for the storage operand remain unchanged, and no PER event for storage alteration is indicated.

A specification exception is recognized when the VR₁ field designates an invalid register number, when the second operand is not designated on an integral boundary, or when the RT₂ field is nonzero and designates the same general register as the RS₂ field.

STORE COMPRESSED is a class-1C instruction. It is interruptible, the vector count and vector interruption index determine the number of elements processed, and element selection is affected by the vector-mask register but not by the vector-mask mode.

Condition Code: The code remains unchanged.

Program Exceptions:
- Access (store, operand 2)
- Operation
- Specification
- Vector operation

Programming Notes:
1. The number of vector elements which are stored and the amount by which the address in the general register designated by RS₂ is updated correspond to the number of ones among the active bits of the vector-mask register.
2. The operation performed by STORE COMPRESSED is the opposite of LOAD EXPANDED.

STORE HALFWORD

<table>
<thead>
<tr>
<th>VSTH</th>
<th>VR₁,RS₂(RT₂)</th>
<th>[VST]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A420'</td>
<td>0</td>
<td>16</td>
</tr>
</tbody>
</table>

Element by element, the rightmost 16 bits of each first-operand vector element are placed unchanged in storage at the second-operand location.

A specification exception is recognized when the second operand is not designated on a halfword boundary, or when the RT₂ field is nonzero and designates the same general register as the RS₂ field.

STORE HALFWORD is a class-1C instruction. It is interruptible, the vector count and vector interruption index determine the number of elements processed, and element selection is affected by neither the vector-mask mode nor the vector-mask register.

Condition Code: The code remains unchanged.

Program Exceptions:
- Access (store, operand 2)
- Operation
- Specification
- Vector operation
STORE INDIRECT

Element by element, the third operand is used to select element locations of the second operand in storage, at which elements of the first-operand vector are placed. The element positions of the first operand correspond to those of the third operand.

The method of selecting elements of each operand is the same as for LOAD INDIRECT, the amount of left shift of the third-operand elements being two bits for VSTI or VSTIE and three bits for VSTID. The selected first-operand elements are stored at the specified second-operand locations.

A specification exception is recognized when the VR1 field designates an invalid register number, when the second operand is not designated on an integral boundary, or when the RT2 field is nonzero and designates the same general register as the RS2 field.

اقتراحي للمستخدم: لا تحدث أي صيغة معينة بالعсловية في تلك التعليمات في حالة البداية...

STORE INDIRECT is a class-Ic instruction. It is interruptible, the vector count and vector interruption index determine the number of elements processed, and element selection is affected by neither the vector-mask mode nor the vector-mask register.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (store, operand 2)
- Operation
- Specification
- Vector operation

**Programming Note:** STORE INDIRECT, which is the opposite of LOAD INDIRECT, is used to store a vector by indirect element selection. See also the programming note under LOAD INDIRECT.

STORE MATCHED

Element by element, elements of the first-operand vector corresponding to ones in the active bits of the vector-mask register are placed unchanged in storage at the corresponding element locations of the second operand. Elements of the first operand corresponding to zeros in the active bits of the vector-mask register are not stored, and the corresponding second-operand locations in storage remain unchanged.

A specification exception is recognized when the VR1 field designates an invalid register number, when the second operand is not designated on an integral boundary, or when the RT2 field is nonzero and designates the same general register as the RS2 field.

No access exceptions and PER storage-alteration events are recognized for elements of the second operand which correspond to zeros in the active bits of the vector-mask register, and the corresponding change bits remain unchanged; however, the general register designated by the RS2 field is updated for each of those elements.

STORE MATCHED is a class-Ic instruction. It is interruptible, the vector count and vector interruption index determine the number of elements processed, and element selection is affected by the vector-mask register but not by the vector-mask mode.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (store, operand 2)
- Operation
- Specification
- Vector operation
Programming Notes:

1. The instructions STORE and STORE MATCHED, in corresponding formats, perform the same function on those elements which correspond to ones in the active bits of the vector-mask register; that is, each such element is copied from the same vector-register position into the same storage location. STORE MATCHED differs in that storage locations remain unchanged for elements which correspond to zero bits.

2. STORE, STORE COMPRESSED, and STORE MATCHED, in corresponding formats, perform the same function when all active bit positions of the vector-mask register contain ones.

STORE VECTOR PARAMETERS

<table>
<thead>
<tr>
<th>VSTVP</th>
<th>D2(B2)</th>
<th>[S]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A6C8'</td>
<td>B2</td>
<td>D2</td>
</tr>
</tbody>
</table>

The 16-bit section size and the 16-bit partial-sum number are placed in storage in the left and right half, respectively, of the word at the location designated by the second-operand address.

A specification exception is recognized when the second operand is not designated on a word boundary.

STORE VECTOR PARAMETERS is a class-NO instruction. It is not interruptible, no elements are processed, and its execution is affected by neither the vector-mask mode nor the vector-mask register. The vector count and vector interruption index are not used and remain unchanged.

Condition Code: The code remains unchanged.

Program Exceptions:
- Access (store, operand 2)
- Operation
- Specification
- Vector operation

STORE VMR

<table>
<thead>
<tr>
<th>VSTVM</th>
<th>RS2</th>
<th>[VS]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A682'</td>
<td>/ / / / / / / /</td>
<td>RS2</td>
</tr>
</tbody>
</table>

0 16 28 31

The contents of the active-bit positions of the vector-mask register are stored as a bit vector at the second-operand location.

When the vector count is not a multiple of 8, zeros are stored for any bits in the last byte which are to the right of the last bit specified by the vector count.

When the vector count is zero, no bits are stored. No access exceptions are recognized for the second operand, the change bits for the operand remain unchanged, and PER storage-alteration events are not indicated.

STORE VMR is a class-NC instruction. It is not interruptible, the vector count determines the number of bits processed, and bit selection is affected by neither the vector-mask mode nor the vector-mask register. The vector interruption index is not used and remains unchanged.

Condition Code: The code remains unchanged.

Program Exceptions:
- Access (store, operand 2)
- Operation
- Vector operation

SUBTRACT

<table>
<thead>
<tr>
<th>VR1,QR3,RS2(RT2)</th>
<th>[QST]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>QR3</td>
</tr>
<tr>
<td>0 16 20 24 28 31</td>
<td></td>
</tr>
</tbody>
</table>

Mnemonic Op Code 28 31
- VSS 'A4A1' Binary
- VSDS 'A491' Long
- VSES 'A481' Short

Chapter 3. Vector-Facility Instructions 3-35
Element by element, the second-operand vector is subtracted from the third operand, and the result is placed in the first-operand location.

The operation is performed on each pair of elements in the same manner as the corresponding scalar operation, except that the condition code is not set. For floating-point operands, the scalar equivalent is SUBTRACT NORMALIZED.

A specification exception is recognized when a VR or QR field designates an invalid register number. In the QST and VST formats, a specification exception is recognized when the second operand is not designated on an integral boundary, or when the RT2 field is nonzero and designates the same general register as the RS2 field. For the VSS instruction, a specification exception is also recognized when the QR3 field designates the same general register as the RS2 field.

SUBTRACT is a class-1M instruction. It is interruptible, the vector count and vector interruption index determine the number of elements processed, and element selection is affected by both the vector-mask mode and the vector-mask register.
A specification exception is recognized when the VR1 or FR2 field designates an invalid register number.

SUM PARTIAL SUMS is a class-IP instruction. It is interruptible, the partial-sum number and vector interruption index determine the number of elements processed, and element selection is affected by neither the vector-mask mode nor the vector-mask register. The vector count is not used and remains unchanged.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Operation
- Vector operation

**Programming Note:** An example of the use of SUM PARTIAL SUMS is given in Appendix A (see "Sum of Products" on page A-3).

### TEST VMR

VTVM

<table>
<thead>
<tr>
<th>'A640'</th>
<th>//////////////</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

The active bits of the vector-mask register are tested, and condition code 0, 1, or 3 is set according to whether those bits are all zeros, mixed zeros and ones, or all ones.

When the vector count is zero, condition code 0 is set.

TEST VMR is a class-NC instruction. It is not interruptible, the vector count determines the number of bits processed, and bit selection is affected by neither the vector-mask mode nor the vector-mask register. The vector interruption index is not used and remains unchanged.

**Resulting Condition Code:**
- 0: Active bits all zeros
- 1: Active bits mixed zeros and ones
- 2: --
- 3: Active bits all ones

**Program Exceptions:**
- Operation
- Vector operation

**Programming Note:** The instruction TEST VMR performs the testing portion of the instructions COUNT LEFT ZEROS IN VMR and COUNT ONES IN VMR. It may be used to distinguish the all-zeros and all-ones conditions when the exact count is not required.

### ZERO PARTIAL SUMS

VZPSD VR1 [VR]

<table>
<thead>
<tr>
<th>'A61B'</th>
<th>//////////////</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16 24 28 31</td>
</tr>
</tbody>
</table>

Partial-sum element locations of the vector-register pair designated by VR1 are set to zero.

The operation begins with setting to zero element $X$ of the first operand, where $X$ is the initial vector interruption index (normally zero). It proceeds in an ascending sequence of element numbers by successively setting to zero $p-X$ first-operand elements, where $p$ is the model-dependent partial-sum number. The last one is element $p-1$. The vector interruption index is then set to zero.

If the initial vector interruption index $X$ is equal to or greater than $p$, the vector-register contents and the associated vector in-use bit and vector change bit remain unchanged. The vector interruption index is then set to zero, and instruction execution is completed.

A specification exception is recognized if the VR1 field designates an invalid register number.

ZERO PARTIAL SUMS is a class-IP instruction. It is interruptible, the partial-sum number and vector interruption index determine the number of elements processed, and element selection is affected by neither the vector-mask mode nor the vector-mask register. The vector count is not used and remains unchanged.
**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Operation
- Specification

- Vector operation

**Programming Note:** An example of the use of ZERO PARTIAL SUMS is given in Appendix A (see "Sum of Products" on page A-3).
Appendix A. Instruction-Use Examples

Operations on Full Vectors ............... A-1
Contiguous Vectors ....................... A-1
Vectors with Stride ...................... A-2
Vector and Scalar Operands .............. A-2
Sum of Products .......................... A-3
Compare and Swap Vector Elements ....... A-3
Conditional Arithmetic ................. A-4
Exception Avoidance ..................... A-4

Add to Magnitude .......................... A-4
Operations on Sparse Vectors .......... A-4
Full Added to Sparse to Give Full ...... A-5
Sparse Added to Sparse to Give Sparse ... A-5
Floating-Point-Vector Conversions ...... A-6
Fixed Point to Floating Point .......... A-6
Floating Point to Fixed Point .......... A-6

This appendix contains a number of simple examples of the use of vector instructions.

Every example has a sectioning loop, so that vectors of any length can be handled, independent of the section size. The first example illustrates sectioning in some detail; the others use the same or a similar technique.

The examples are written in assembler language. Register operands are indicated symbolically with a prefix G, F, or V to identify more clearly whether an operand refers to a general register, floating-point register, or vector register, respectively.

Comments are written to the right of the instruction or on separate lines that begin with an asterisk (*).

Operations on Full Vectors

The following examples illustrate operations on full vectors, where both zero and nonzero elements are represented in storage. Vectors in storage are accessed by sequential addressing.

The first three examples use three different methods of controlling the sectioning loop.

Contiguous Vectors

Two contiguous vectors A and B in storage are added, and the result is stored in contiguous vector C. The number of elements in each is specified by N. All vectors are in the long floating-point format.

\[ C = A + B \]

Assuming, for purposes of illustration, a vector-section size of 8 and a vector length of 20, the above program would process three sections in turn (two full sections of eight elements and one partial section of four elements) before ending the loop. One section of A and one section of B are added in vector-register pair 0 and 1. The result is stored in a section of C, as illustrated below:

Since all vectors are stored contiguously, the stride for the three vector instructions VLD, VAD, and
VSTD is set to one by specifying a value of zero in the RT2 subfield. This may be done in the assembler language either by placing a zero inside the parentheses of the stride subfield, as in:

Mnemonic VR1,VR3,RS2(0)

or by omitting the subfield, including the parentheses, altogether:

Mnemonic VR1,VR3,RS2

Each of these instructions automatically updates the storage address in the designated general register to the value that will be needed for the next time, if any, around the loop.

The BRANCH ON CONDITION (BC) instruction tests the condition code set by VLVCU, because none of the intervening instructions change the condition code. If an instruction setting the condition code had intervened, the instruction "LTR GO,GO" inserted before the BC instruction would test the contents of GRO; BC would test for condition code 2 in either case.

The following figure shows the condition-code setting (CC), the vector count (vCT), and the contents of the general registers at the start, before executing the first VLVCU instruction, and at the end of each loop thereafter.

<table>
<thead>
<tr>
<th>Loop</th>
<th>CC</th>
<th>VCT</th>
<th>GR0</th>
<th>GR1</th>
<th>GR2</th>
<th>GR3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>1</td>
<td>20</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>End 1</td>
<td>2</td>
<td>8</td>
<td>12</td>
<td>12</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>End 2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>12</td>
<td>12</td>
<td>128</td>
</tr>
<tr>
<td>End 3</td>
<td>3</td>
<td>0</td>
<td>160</td>
<td>160</td>
<td>160</td>
<td></td>
</tr>
</tbody>
</table>

**Vectors with Stride**

This example modifies the previous example in four ways. All vector elements are in the short floating-point format. The result of the addition is returned to the storage location of vector B. Vector B is assumed to be stored with a stride T. Finally, a BC instruction which tests for the end of the loop is placed immediately after the VLVCU instruction, and the loop is closed with an unconditional branch. This method, which could be used if additional instructions were to change the condition code later in the loop, allows the loop to be bypassed when the initial vector count is zero.

(Note, however, that the previous loop control also works with a vector count of zero, because no elements would be processed if vector instructions were executed with a zero vector count.)

\[ B = A + B \]

\[ L G0,N \quad \text{Vector length to GR0} \]
\[ LA G1,A \quad \text{Address of A to GR1} \]
\[ LA G2,B \quad \text{Address of B to GR2} \]
\[ LR G3,G2 \quad \text{Copy address in GR3} \]
\[ L G4,T \quad \text{Stride for B to GR4} \]
\[ LP VLVCU G0 \quad \text{Load VCT, update GR0} \]
\[ BC 12,NXT \quad \text{Exit loop if VCT=0} \]
\[ VLE V0,G1 \quad \text{Load section of A} \]
\[ VAE V0, G0, G2(64) \quad \text{Add section of B} \]
\[ L G4,N \quad \text{Vector length to GR4} \]
\[ LD F0,S \quad \text{Load S into FR0} \]
\[ VLE V0, G1 \quad \text{Load section of A} \]
\[ VAE V0, G0, G2(64) \quad \text{Add section of B} \]
\[ VSTD V0, G3(64) \quad \text{Store result in B} \]
\[ VSTD V0, G3 \quad \text{Store result in B} \]
\[ VSTD V0, G3 \quad \text{Add section of B} \]
\[ VLE V0, G1 \quad \text{Exit loop if VCT=0} \]
\[ L G4,N \quad \text{Vector length to GR4} \]
\[ LD F0,S \quad \text{Load S into FR0} \]

**Vector and Scalar Operands**

This example illustrates the use of both vector and scalar operands. It also shows how the three-operand arithmetic vector instructions can sometimes be used to avoid a separate vector-load instruction. A third loop-control method is used here.

\[ A \text{ and } B \text{ are vectors of length } N, \text{ and } S \text{ is a scalar. } \]
\[ A \text{ and } B \text{ are vectors of length } N, \text{ and } S \text{ is a scalar. All are in the long floating-point format.} \]
\[ B = A \times (S-A) \]

\[ L G1,A \quad \text{Address of A to GR1} \]
\[ LR G2,G1 \quad \text{Copy address in GR2} \]
\[ LA G3,B \quad \text{Address of B to GR3} \]
\[ L G4,N \quad \text{Vector length to GR4} \]
\[ LD F0,S \quad \text{Load S into FR0} \]
\[ VSTD V0, G3 \quad \text{Store result in B} \]
\[ VSTD V0, G3 \quad \text{Add section of B} \]
\[ VSTD V0, G3 \quad \text{Store result in B} \]

A-2 IBM System/370 Vector Operations
The VSDS instruction subtracts vector A in storage from the scalar S. VMD multiplies the result by vector A, again from its storage location. VSTD stores the product as B. There are two VLVCU loop-control instructions, one before entry into the loop and one at the end.

Note that the QST-format arithmetic instruction (VSDS) saves a separate load instruction at the expense of having to access storage twice for the same vector section A. Depending on the model, a separate load instruction followed by QV-format arithmetic instructions may be more efficient in some circumstances, particularly when the stride is greater than one.

Note further that the QST-format instructions are defined such that VSDS subtracts a vector from a scalar \((S-V)\). Subtracting a scalar from a vector \((V-S)\) can be done conveniently by first changing the sign of the scalar and then adding, using VADS. Similarly, the VDDS instruction divides a scalar by a vector \((S/V)\). Division of a vector by a scalar \((V/S)\) can be performed by first taking the reciprocal of the scalar and then multiplying, using VMDS. (The same comment applies to the corresponding QV-format instructions.)

**Sum of Products**

The use of MULTIPLY and ACCUMULATE and related instructions is illustrated by computing the inner product of a row vector A, taken from a matrix of dimensions \(I\) by \(J\), and a column vector B, taken from another matrix of dimensions \(J\) by \(K\). Each matrix is assumed to be stored in column order. Therefore, row vector A has a stride \(I\) and a length \(J\), and column vector B is contiguous and has the same length \(J\). The inner product of the two vectors is a scalar value that is the sum of the element-by-element products of vectors A and B; it is stored at address C.

\[
* \quad C = \text{SUM} \left( A \ast B \right) \\
* 
\]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VZPSD V0</td>
<td>Zero partial sums</td>
</tr>
<tr>
<td>LP VLVCU G0</td>
<td>Load VCI, update GR0</td>
</tr>
<tr>
<td>VLD V2,G1(G2)</td>
<td>Row A section to VR2</td>
</tr>
<tr>
<td>VMCD V0,V2,G3</td>
<td>Multiply by column B partial sums to VR0</td>
</tr>
<tr>
<td>BC 2,LP</td>
<td>Branch back if GR0&gt;0</td>
</tr>
<tr>
<td>SDR F0,F0</td>
<td>Clear FR0 to zero</td>
</tr>
<tr>
<td>VSPSD V0,F0</td>
<td>Scalar sum to FR0</td>
</tr>
<tr>
<td>STD F0,C</td>
<td>Store scalar sum</td>
</tr>
</tbody>
</table>

First the VZPSD instruction clears the partial-sum locations in VR0 to zero. Then the sectioning loop accumulates partial sums: The VLD instruction loads a section of row A (with stride) into VR2. The VMCD instruction multiplies the elements of row A in VR2 by elements of column B in storage (without stride) and accumulates \(p\) partial sums in VR0; the number \(p\) depends on the model.

After the sectioning loop is ended and all partial sums have been accumulated in VR0, FR0 is cleared by means of SDR, and the \(p\) partial sums are then added to FR0 by use of the VSPSD instruction. The scalar sum is stored in C by STD.

Note that the program is independent of the vector-section size and the number of partial sums, both of which depend on the model, because the instructions VZPSD, VLVCU, VMCD, and VSPSD take care of these dependencies automatically.

**Compare and Swap Vector Elements**

Two vectors A and B, both of length \(N\), are to be compared and their elements swapped so that vector A will have the smaller element of each pair and vector B the larger. The elements are 32-bit signed binary integers and stored contiguously.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L G0,N</td>
<td>Vector length to GR0</td>
</tr>
<tr>
<td>LA G1,A</td>
<td>Address of A to GR1</td>
</tr>
<tr>
<td>LR G2,G1</td>
<td>Copy address in GR2</td>
</tr>
<tr>
<td>LA G3,B</td>
<td>Address of B to GR3</td>
</tr>
<tr>
<td>LR G4,G3</td>
<td>Copy address in GR4</td>
</tr>
<tr>
<td>LP VLVCU G0</td>
<td>Load VCI, update GR0</td>
</tr>
<tr>
<td>VL V0,G1</td>
<td>Section of A to VR0</td>
</tr>
<tr>
<td>VL V1,G3</td>
<td>Section of B to VR1</td>
</tr>
<tr>
<td>VCR 2,V0,V1</td>
<td>Check where A&gt;B</td>
</tr>
<tr>
<td>VSTM V0,G4</td>
<td>Store greater in B</td>
</tr>
<tr>
<td>VSTM V1,G2</td>
<td>Store lesser in A</td>
</tr>
<tr>
<td>BC 2,LP</td>
<td>Branch back if GR0&gt;0</td>
</tr>
</tbody>
</table>
Conditional Arithmetic

Exception Avoidance

One use of conditional arithmetic in the vector-mask mode is to bypass vector elements which would cause an exception during the arithmetic operation and to provide a predetermined alternate result for those elements. The example divides two vectors $A$ and $B$. The divisor $B$ is tested for zeros. By using the vector-mask mode, no division is performed for zero divisor elements, thus avoiding a disruptive floating-point divide exception; the corresponding elements in result vector $C$ are set to the maximum positive value $MP$. All floating-point numbers are in the long format.

In this example, performing the arithmetic conditionally requires two extra vector instructions inside the sectioning loop.

\[
C = A / B
\]

\* 
L G0,N Vector length to GR0
LA G1,A Address of A to GR1
LA G2,B Address of B to GR2
LR G3,G2 Copy address in GR3
LA G4,C Address of C to GR4
SDR F0,F0 Clear FR0 to zero
LD F2,MP Load max. positive number $MP$ in FR2
VSVMM 1 Vector-mask mode on
LP VLVCU G0 Load VCT, update GR0
VCDS 6,F0,G2 Compare section of B not equal to zero
VLDO V0,F2 Load MP in all elem.
VLD V2,G1 Load section of A positions of VR0
VDD V0,V2,G3 Conditionally divide $A$ by section of $B$
VSTD V0,G4 Store section in C
BC 2,LP Branch back if GR0>0
VSVMM 0 Set mask mode off

Add to Magnitude

Another use of conditional arithmetic is to perform addition to the magnitude of a vector regardless of signs. This may be illustrated by rounding a vector $V$ of length $N$, consisting of floating-point numbers in the short format, to integer values. First, 0.5 is added to the magnitude of each element. Then, the digits to the right of the implied radix point are truncated. The rounded vector $R$ remains in the short floating-point format.

Let $H$ and $Z$ be constants with the following hexadecimal formats and values:

$H = 40 \ 80 \ 00 \ 00 = 0.5$
$Z = 47 \ 00 \ 00 \ 00 = 0$ (unnormalized)

$H$ is the value which is to be added to or subtracted from each vector element, depending on its sign.

The constant $Z$ is an unnormalized zero with such a characteristic that its addition to a short floating-point number having a smaller characteristic forces that number to be shifted to the right, placing the units digit in the guard-digit position. This causes any digits to the right of the implied radix point to be truncated and the result to be normalized. Any number with an equal or larger characteristic has no significant digits to the right of the implied radix point and remains unchanged.

\* 
R = ROUND(V)

\* 
L G0,N Vector length to GR0
LA G1,V Address of V to GR1
LA G2,R Address of R to GR2
SDR F0,F0 Clear FR0 to zero
LE F2,H Load $H$ into FR2
LNER F4,F2 Load $-H$ into FR4
LE F6,Z Load Z into FR6
LP VLVCU G0 Load VCT, update GR0
VLE V0,G1 Load section of V
VSVMM 1 Vector-mask mode on
VCEQ 12,F0,V0 Compare; set mask to one where $0 \leq V$
VAEQ V0,F2,V0 Add 0.5 under mask
VCVM Complement mask bits
VAEQ V0,F4,V0 Add -0.5 under mask
VSVMM 0 Vector-mask mode off
VAEQ V0,F6,V0 Add Z
VSTE V0,G2 Store section of R
BC 2,LP Branch back if GR0>0

Add to Magnitude

A variation of this rounding technique is incorporated in a later example of floating-point to fixed-point conversion.

Operations on Sparse Vectors

This section gives some examples of operating on sparse vectors, where only nonzero elements are directly represented in storage.
When many vector elements are zero, considerable storage may be saved by using a dense representation containing only those elements which are nonzero. The resulting nonzero elements can be stored in contiguous locations along with a bit vector indicating the nonzero values in the corresponding full vector. A full vector can be converted to such a dense vector by performing a not-equal comparison of the vector to a scalar zero and using the resulting bit vector as a mask in a STORE COMPRESSED instruction.

For use in the following examples, assume two vectors \( A \) and \( B \). The full vectors are 10 elements in length; elements 0, 2, 5, 6, 7, and 9 of vector \( A \) are nonzero; and elements 2, 4, 5, and 7 of vector \( B \) are nonzero. The figures show the full vectors, the result of a not-equal comparison to zero, and the dense vectors for \( A \) and \( B \).

**Full Vector A (AF):**

\[ \begin{array}{cccccccccc}
A_0 & A_1 & A_2 & A_3 & A_4 & A_5 & A_6 & A_7 & A_8 & A_9 \\
\end{array} \]

**Result of comparing \( A \neq 0 \) (mask AM):**

\[ \begin{array}{cccccccccc}
1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
\end{array} \]

**Dense Vector A (AD):**

\[ \begin{array}{cccccccccc}
A_0 & A_2 & A_5 & A_6 & A_7 & A_9 \\
\end{array} \]

**Full Vector B (BF):**

\[ \begin{array}{cccccccccc}
B_0 & B_1 & B_2 & B_3 & B_4 & B_5 & B_6 & B_7 & B_8 & B_9 \\
\end{array} \]

**Result of comparing \( B \neq 0 \) (mask BM):**

\[ \begin{array}{cccccccccc}
0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
\end{array} \]

**Dense Vector B (BD):**

\[ \begin{array}{cccccccccc}
B_2 & B_4 & B_5 & B_7 \\
\end{array} \]

**Full Added to Sparse to Give Full**

This example shows the addition of elements of full vector \( BF \), which correspond to nonzero elements of vector \( A \), to dense vector \( AD \). The result elements are replaced in \( BF \). The length of the full vectors is \( N \), which is also the number of bits in the mask.

**Sparse Added to Sparse to Give Sparse**

The following example adds dense vectors \( AD \) and \( BD \) to obtain dense vector \( CD \). The mask for \( CD \) is obtained by ORing the mask for \( AD \) with the mask for \( BD \), using the instruction OR TO VMR.

**Instructions**

- \( \text{LA} \; G1, AD \) Address of \( AD \) to \( GR1 \)
- \( \text{SR} \; G2,G2 \) Clear bit index in \( GR2 \)
- \( \text{L} \; G3,N \) Bit count \( N \) to \( GR3 \)
- \( \text{LP} \; \text{VLBIX} \; V0,G2,AM \) Convert mask \( AM \) to element numbers in VR0
- \( \text{VLID} \; V2,V0,BF \) Load \( BF \) indirectly
- \( \text{VAD} \; V2,V2,G1 \) Add \( AD \) contiguously
- \( \text{VSTID} \; V2,V0,BF \) Store indirectly
- \( \text{BC} \; 2,LP \) Branch back if \( GR3 \neq 0 \)

The \( \text{VLBIX} \) instruction converts the bit mask \( AM \) to a vector of element numbers, using the general-register pair \( GR2 \) and \( GR3 \) as the bit index and bit count. This instruction creates up to a full section of element numbers in \( VR0 \) and places the corresponding vector count in \( VCT \) for use by subsequent vector instructions. \( GR2 \) and \( GR3 \) are updated for the next pass through the loop. \( \text{VLID} \) uses the generated element numbers to select elements of full \( BF \) to correspond to all the elements of dense \( AD \), which are added together by the instruction \( \text{VAD} \). \( \text{VSTID} \) then stores the results back into the same elements of \( BF \). The \( \text{BC} \) instruction tests the condition code set by \( \text{VLBIX} \) and branches back if there are more bits to be processed.
Floating-Point-Vector Conversions

The conversion techniques illustrated here are similar to the scalar examples in *IBM 370-XA Principles of Operation* and *IBM System/370 Principles of Operation*, which may be consulted for more details. The methods differ, however, because of different characteristics of the vector-instruction set.

Fixed Point to Floating Point

Assume a vector $K$ of length $N$ in storage, the elements of which are 32-bit signed binary integers. The elements are to be converted to floating-point numbers in the long format, and the result is to be stored as vector $W$.

Assume a floating-point constant $C$ in storage with the following hexadecimal format and value:

$$C = CE \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 = -2^{31}$$

This is an unnormalized floating-point number in the long format with the characteristic 4E, which is the proper characteristic for a right-aligned, unnormalized integer.

$$L \ G0,N \ Vector\ length\ to\ GR0$$
$$LA \ G1,K \ Address\ of\ K\ to\ GR1$$
$$LA \ G2,W \ Address\ of\ W\ to\ GR2$$
$$LD \ F0,C \ Load\ C\ into\ FR0$$
$$LP \ VLVCU \ G0 \ Load\ VCT,\ update\ GR0$$
$$VL \ V1,G1 \ Load\ K\ into\ VR1$$
$$VLCE\ V1,V1 \ K + 2^{31}$$
$$VLEQ \ V0,F0 \ V = -(K + 2^{31})$$
$$VSDQ \ V0,F0,V0 \ W = -2^{31} - V$$
$$VSTD \ V0,G2 \ Store\ W$$
$$BC \ 2,LP \ Branch\ back\ if\ GR0>0$$

Inside the sectioning loop, the VLCER instruction (**Load Complement** in short floating-point format) inverts the sign bit, bit 0, of each element in VR1, without altering bits 1-31. Considering these elements still as signed binary integers, the operation is equivalent to adding $2^{31}$ to each, ignoring overflow, which changes all elements into positive numbers in the range 0 to $2^{32} - 1$. The VLEQ instruction places the left half of the constant $C$ into each element position of VR0, which has the effect of converting the contents of VR1 to a vector $V$ of negative unnormalized floating-point numbers in the long format, occupying VR0 and VR1.

The next instruction, VSDQ, subtracts $V$ from the entire constant $C$, which is equivalent to subtracting $2^{31}$ from the original elements, thus restoring them to the range $-2^{31}$ to $2^{31} - 1$. The elements are normalized during this operation.

The next example presents an alternate program, the loop of which is shorter by one vector instruction.

$$L \ G0,N \ Vector\ length\ to\ GR0$$
$$LA \ G1,K \ Address\ of\ K\ to\ GR1$$
$$LA \ G2,W \ Address\ of\ W\ to\ GR2$$
$$LD \ F0,C \ Load\ C\ into\ FR0$$
$$LP \ VLVCU \ G0 \ Load\ VCT,\ update\ GR0$$
$$VLDQ \ V0,F0 \ Load\ C\ into\ VR0,\ VR1$$
$$VX \ V1,V1,G1 \ V = -(K+2^{31})$$
$$VSDQ \ V0,F0,V0 \ W = -2^{31} - V$$
$$VSTD \ V0,G2 \ Store\ W$$
$$BC \ 2,LP \ Branch\ back\ if\ GR0>0$$

The VLDQ instruction loads the entire constant $C$ into VR0 and VR1. Then, the VX instruction fetches the elements of $K$ from storage and exclusive ORs them into VR1, which contained a left-most one followed by 31 zeros. This inverts the sign bit, as did VLCER in the previous example. The rest of the program is the same.

Floating Point to Fixed Point

This example combines conversion from floating to fixed point with a variation of the rounding technique shown in a previous example.

* Start of range test

$$L \ G0,N \ Vector\ length\ to\ GR0$$
$$LA \ G1,W \ Address\ of\ W\ to\ GR1$$
$$LR \ G2,G1 \ Copy\ address\ to\ GR2$$
$$LD \ F0,L \ FR0: upper\ limit\ L$$
$$LNDR \ F2,F0 \ FR2: lower\ limit\ -L$$
$$LP1 \ VLVCU \ G0 \ Load\ VCT,\ update\ GR0$$
$$VCDS \ 12,F0,G0 \ Compare\ L\ and\ W;\ set$$
$$* \ mask\ bit\ to\ one\ when$$
$$* \ L\ is\ equal\ or\ low$$
$$VTVM \ Test\ mask\ bits$$
$$BC \ 5,OVFLO \ Exit\ if\ any\ ones$$
$$VCDS \ 2,F2,G2 \ Compare\ -L\ and\ W;\ set$$
$$* \ mask\ bit\ to\ one\ when$$
$$* \ when\ -L\ is\ high$$
$$VTVM \ Test\ mask\ bits$$
$$BC \ 5,OVFLO \ Exit\ if\ any\ ones$$
$$LTR \ G0,G0 \ Test\ residual\ count$$
$$BC \ 2,LP1 \ Branch\ back\ if\ GR0>0$$

* Start of conversion with rounding

$$L \ G0,N \ Vector\ length\ to\ GR0$$
$$LA \ G1,W \ Address\ of\ W\ to\ GR1$$
$$LA \ G2,K \ Address\ of\ K\ to\ GR2$$
LD  F0,G  Load  G  into FR0
LD  F2,H  Load  H  into FR2
LD  F4,M  Load  M  into FR4
LP2  VLVCU  G0  Load  VCT,  update  GR0
VADS  V0,F2,G1  Add  0.5  to  W  section
VSVMM  1  Vector-mask  mode  on
VCDQ  2,F2,V0  Compare;  set  mask  to
*  one  where  0.5>W
  VADQ  V0,F4,V0  Add  -1.0  under  mask
  VSVMM  0  Set  mask  mode  off
  VADQ  V0,F0,V0  Add  2^{53}
VST  V1,G2  Store  K  from  VR1
BC  2,LP2  Branch  back  if  GR0>0

Assume  a  vector  W  of  length  N  in  storage,  the  elements  of  which  are  floating-point  numbers  in  the
long  format.  Assume  this  vector  is  to  be  converted  to  a  vector  of  signed  binary  integers,  and  the  result
is  to  be  stored  as  vector  K.  Assume  floating-point  constants  in  storage  with  the  following  names,  hexa­decimal  formats,  and  values:

\begin{align*}
L &= 48 \ 80 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \ = 2^{31} \\
G &= 4F \ 82 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \ = 2^{53} \\
H &= 40 \ 80 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \ = 0.5 \\
M &= C1 \ 10 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \ = -1.0
\end{align*}

L  is  the  upper  limit  of  the  range  of  numbers  which,  after  truncation  of  the  fractional  part,  are  representable  as  signed  binary  integers.  Vector  W  is  com­pared  with  this  limit  in  a  separate  sectioning  loop  before  conversion  is  started,  so  that  nothing  is
stored  if  any  element  of  W  is  out  of  range.  This  comparison  loop  can  be  omitted  if  all  elements  are
known  to  be  within  range.

H  and  M  are  the  constants  0.5  and  -1.0,  respectively.  Rounding  is  accomplished  by  first  adding
0.5  unconditionally  to  vector  W,  and  then  adding  -1.0  conditionally  where  the  elements  are  now  less
than  0.5,  which  is  equivalent  to  subtracting  0.5  from  all  initially  negative  elements.

The  constant  G  is  chosen  such  that  its  addition  to  a  number  within  the  representable  range  forces  that
number  to  be  shifted  to  the  right,  with  the  units  digit  in  the  guard-digit  position,  and  the  result  to  be
normalized  to  the  left  by  one  digit  position.  This  causes  any  fraction  part  to  be  truncated,  leaving  the
rounded  integer  part  in  the  right  half  of  the  vector-register  pair.
Appendix B. Lists Of Instructions

The following figures list the vector instructions by name, mnemonic, and op code.

Explanation of Symbols in "Characteristics" Column

A Access exceptions
C Condition code is set
EO Exponent-overflow exception
EU Exponent-underflow exception
FK Floating-point-divide exception
IC Class-IC instruction; interruptible; vector count and vector interruption index determine number of elements processed; does not depend on vector-mask mode
IF Fixed-point-overflow exception
IG Class-IG instruction; interruptible; general register, vector interruption index, and section size determine number of elements processed; sets vector count; does not depend on vector-mask mode
IM Class-IM instruction; interruptible; vector count and vector interruption index determine number of elements processed; depends on vector-mask mode
IP Class-IP instruction; interruptible; partial-sum number and vector interruption index determine number of elements processed; does not depend on vector-mask mode
IZ Class-IZ instruction; interruptible; vector-section size determines number of elements processed; does not depend on vector-mask mode
J Arithmetic exception; exception-extension code is stored
LS Significance exception
NC Class-NC instruction; not interruptible; vector count determines number of elements processed; does not depend on vector-mask mode
NZ Class-NZ instruction; not interruptible; vector-section size determines number of elements processed; does not depend on vector-mask mode
N0 Class-N0 instruction; not interruptible; no vector elements processed; does not depend on vector-mask mode
N1 Class-N1 instruction; not interruptible; one vector element processed; does not depend on vector-mask mode
P Privileged-operation exception
QST QST instruction format
QV QV instruction format
R* PER general-register-alteration event may or may not be recognized
RRE RRE instruction format
RSE RSE instruction format
S S instruction format
SP Specification exception
ST PER storage-alteration event
U Unnormalized-operand exception
VB Sets vector in-use bit and vector change bit
VE Vector facility and vector-operation exception
VH Sets vector change bit
VR VR instruction format
VS VS instruction format
VST VST instruction format
VV Leaves vector change bit unaltered

Notes

1 Same op code as for short; separate mnemonic for programming convenience
2 Execution differs in problem state and supervisor state
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCUMULATE (long)</td>
<td>VACD</td>
<td>VST VE A SP J EU EO LS IM VB</td>
<td>R* A417</td>
</tr>
<tr>
<td>ACCUMULATE (long)</td>
<td>VACDR</td>
<td>VV VE A SP J EU EO LS IM VB</td>
<td>A517</td>
</tr>
<tr>
<td>ACCUMULATE (short to long)</td>
<td>VACE</td>
<td>VST VE A SP J EU EO LS IM VB</td>
<td>A447</td>
</tr>
<tr>
<td>ACCUMULATE (short to long)</td>
<td>VACER</td>
<td>VV VE A SP J EU EO LS IM VB</td>
<td>A587</td>
</tr>
<tr>
<td>ADD (binary)</td>
<td>VAQ</td>
<td>QV VE J IF IM VB</td>
<td>A5A0</td>
</tr>
<tr>
<td>ADD (binary)</td>
<td>VAS</td>
<td>QST VE A SP J IF IM VB</td>
<td>A520</td>
</tr>
<tr>
<td>ADD (long)</td>
<td>VAD</td>
<td>VST VE A SP J EU EO LS IM VB</td>
<td>A4A0</td>
</tr>
<tr>
<td>ADD (long)</td>
<td>VAQ</td>
<td>QV VE J EU EO LS IM VB</td>
<td>A590</td>
</tr>
<tr>
<td>ADD (long)</td>
<td>VADR</td>
<td>VV VE J EU EO LS IM VB</td>
<td>A518</td>
</tr>
<tr>
<td>ADD (long)</td>
<td>VADS</td>
<td>QST VE A SP J EU EO LS IM VB</td>
<td>A498</td>
</tr>
<tr>
<td>ADD (short)</td>
<td>VAEQ</td>
<td>QV VE J EU EO LS IM VB</td>
<td>A508</td>
</tr>
<tr>
<td>ADD (short)</td>
<td>VAES</td>
<td>QST VE A SP J EU EO LS IM VB</td>
<td>A488</td>
</tr>
<tr>
<td>AND</td>
<td>VN</td>
<td>VST VE A SP J EU EO LS IM VB</td>
<td>A4A4</td>
</tr>
<tr>
<td>AND</td>
<td>VNQ</td>
<td>QV VE IM VB</td>
<td>A524</td>
</tr>
<tr>
<td>AND</td>
<td>VNR</td>
<td>VV VE IM VB</td>
<td>A524</td>
</tr>
<tr>
<td>AND</td>
<td>VNS</td>
<td>QST VE A SP IM VB</td>
<td>A4A4</td>
</tr>
<tr>
<td>AND TO VMR</td>
<td>VNVM</td>
<td>VS VE A NC</td>
<td>R* A684</td>
</tr>
<tr>
<td>CLEAR VR</td>
<td>VRCL</td>
<td>S VE IZ VB</td>
<td>A6C5</td>
</tr>
<tr>
<td>COMPAR (binary)</td>
<td>VC</td>
<td>VST VE A SP IC</td>
<td>R* A428</td>
</tr>
<tr>
<td>COMPAR (binary)</td>
<td>VCQ</td>
<td>QV VE IC</td>
<td>A5A8</td>
</tr>
<tr>
<td>COMPAR (binary)</td>
<td>VCR</td>
<td>VV VE IC</td>
<td>A528</td>
</tr>
<tr>
<td>COMPAR (binary)</td>
<td>VCS</td>
<td>QST VE A SP IC</td>
<td>R* A4A8</td>
</tr>
<tr>
<td>COMPAR (long)</td>
<td>VCD</td>
<td>VST VE A SP IC</td>
<td>R* A418</td>
</tr>
<tr>
<td>COMPAR (long)</td>
<td>VCDQ</td>
<td>QV VE IC</td>
<td>A598</td>
</tr>
<tr>
<td>COMPAR (long)</td>
<td>VCDR</td>
<td>VV VE IC</td>
<td>A518</td>
</tr>
<tr>
<td>COMPAR (long)</td>
<td>VCDQ</td>
<td>QST VE A SP IC</td>
<td>R* A498</td>
</tr>
<tr>
<td>COMPAR (short)</td>
<td>VCE</td>
<td>VST VE A SP IC</td>
<td>R* A408</td>
</tr>
<tr>
<td>COMPAR (short)</td>
<td>VCEQ</td>
<td>QV VE IC</td>
<td>A588</td>
</tr>
<tr>
<td>COMPAR (short)</td>
<td>VCR</td>
<td>VV VE IC</td>
<td>A508</td>
</tr>
<tr>
<td>COMPAR (short)</td>
<td>VCES</td>
<td>QST VE A SP IC</td>
<td>R* A488</td>
</tr>
<tr>
<td>COMPLEMENT VMR</td>
<td>VCVM</td>
<td>RRE VE NC</td>
<td>A641</td>
</tr>
<tr>
<td>COUNT LEFT ZEROS IN VMR</td>
<td>VCVZVM</td>
<td>RRE C VE NC</td>
<td>R* A642</td>
</tr>
<tr>
<td>COUNT ONES IN VMR</td>
<td>VCDVM</td>
<td>RRE C VE NC</td>
<td>R* A643</td>
</tr>
<tr>
<td>DIVIDE (long)</td>
<td>VDD</td>
<td>VST VE A SP J U EU EO FK IM VB</td>
<td>R* A413</td>
</tr>
<tr>
<td>DIVIDE (long)</td>
<td>VDDQ</td>
<td>QV VE J U EU EO FK IM VB</td>
<td>A593</td>
</tr>
<tr>
<td>DIVIDE (long)</td>
<td>VDDR</td>
<td>VV VE J U EU EO FK IM VB</td>
<td>A513</td>
</tr>
</tbody>
</table>

Figure B-1 (Part 1 of 5). Instructions Arranged by Name

B-2 IBM System/370 Vector Operations
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIVIDE (long)</td>
<td>VDDS (long)</td>
<td>J U E U E0 FK IM VB</td>
<td>A493</td>
</tr>
<tr>
<td>DIVIDE (short)</td>
<td>VDE (short)</td>
<td>J U E U E0 FK IM VB</td>
<td>A403</td>
</tr>
<tr>
<td>DIVIDE (short)</td>
<td>VDER (short)</td>
<td>J U E U E0 FK IM VB</td>
<td>A583</td>
</tr>
<tr>
<td>DIVIDE (short)</td>
<td>VDES (short)</td>
<td>J U E U E0 FK IM VB</td>
<td>A503</td>
</tr>
<tr>
<td>DIVIDE (short)</td>
<td>VDIVPES (long)</td>
<td>J U E U E0 FK IM VB</td>
<td>A483</td>
</tr>
<tr>
<td>EXCLUSIVE OR</td>
<td>VX (long)</td>
<td>IM VB R*</td>
<td>A426</td>
</tr>
<tr>
<td>EXCLUSIVE OR</td>
<td>VXQ (short)</td>
<td>IM VB R*</td>
<td>A546</td>
</tr>
<tr>
<td>EXCLUSIVE OR</td>
<td>VXSR (long)</td>
<td>IM VB R*</td>
<td>A526</td>
</tr>
<tr>
<td>EXCLUSIVE OR</td>
<td>VXSM (short)</td>
<td>IM VB R*</td>
<td>A446</td>
</tr>
<tr>
<td>EXCLUSIVE OR TO VMR</td>
<td>VXVM (long)</td>
<td>NC R*</td>
<td>A686</td>
</tr>
<tr>
<td>EXTRACT ELEMENT (binary)</td>
<td>VXSEL (long)</td>
<td>VR VE N1 R*</td>
<td>A629</td>
</tr>
<tr>
<td>EXTRACT ELEMENT (long)</td>
<td>VXELSE (short)</td>
<td>VR VE N1 R*</td>
<td>A619</td>
</tr>
<tr>
<td>EXTRACT VCT</td>
<td>VXVC (short)</td>
<td>IM VB R*</td>
<td>A644</td>
</tr>
<tr>
<td>EXTRACT VECTOR MASK MODE</td>
<td>VXVMM (long)</td>
<td>RRE VE N0 R*</td>
<td>A646</td>
</tr>
<tr>
<td>LOAD (binary)</td>
<td>VLD (short)</td>
<td>IC VB R*</td>
<td>A419</td>
</tr>
<tr>
<td>LOAD (binary)</td>
<td>VLDQ (long)</td>
<td>IC VB R*</td>
<td>A409</td>
</tr>
<tr>
<td>LOAD (binary)</td>
<td>VLDQ (long)</td>
<td>IC VB R*</td>
<td>A599</td>
</tr>
<tr>
<td>LOAD (long)</td>
<td>VLD (short)</td>
<td>IC VB R*</td>
<td>A409</td>
</tr>
<tr>
<td>LOAD (long)</td>
<td>VLD (short)</td>
<td>IC VB R*</td>
<td>A599</td>
</tr>
<tr>
<td>LOAD (long)</td>
<td>VLD (long)</td>
<td>IC VB R*</td>
<td>A519</td>
</tr>
<tr>
<td>LOAD COMPLEMENT (binary)</td>
<td>VLCR (short)</td>
<td>VV VE J IF IM VB</td>
<td>A562</td>
</tr>
<tr>
<td>LOAD COMPLEMENT (long)</td>
<td>VLCDR (long)</td>
<td>VV VE SP</td>
<td>A552</td>
</tr>
<tr>
<td>LOAD COMPLEMENT (short)</td>
<td>VLCER (short)</td>
<td>VV VE SP</td>
<td>A542</td>
</tr>
<tr>
<td>LOAD ELEMENT (binary)</td>
<td>VLEL (short)</td>
<td>VR VE N1 VB</td>
<td>A628</td>
</tr>
<tr>
<td>LOAD ELEMENT (long)</td>
<td>VLELD (long)</td>
<td>VR VE N1 VB</td>
<td>A618</td>
</tr>
<tr>
<td>LOAD ELEMENT (short)</td>
<td>VLELE (short)</td>
<td>VR VE N1 VB</td>
<td>A608</td>
</tr>
<tr>
<td>LOAD EXPANDED (binary)</td>
<td>VLY (short)</td>
<td>VV VE SP</td>
<td>A408</td>
</tr>
<tr>
<td>LOAD EXPANDED (long)</td>
<td>VLYD (short)</td>
<td>VV VE SP</td>
<td>A418</td>
</tr>
<tr>
<td>LOAD EXPANDED (short)</td>
<td>VLYE (short)</td>
<td>VV VE SP</td>
<td>A408</td>
</tr>
<tr>
<td>LOAD HALFWORD</td>
<td>VLNH (short)</td>
<td>VV VE SP</td>
<td>A429</td>
</tr>
<tr>
<td>LOAD INDIRECT (binary)</td>
<td>VLI (short)</td>
<td>RSE VE A SP</td>
<td>E409</td>
</tr>
<tr>
<td>LOAD INDIRECT (long)</td>
<td>VLI (short)</td>
<td>RSE VE A SP</td>
<td>E410</td>
</tr>
<tr>
<td>LOAD INDIRECT (short)</td>
<td>VLI (short)</td>
<td>RSE VE A SP</td>
<td>E400</td>
</tr>
<tr>
<td>LOAD INTEGER VECTOR</td>
<td>VLIINT (short)</td>
<td>VV VE SP</td>
<td>A422</td>
</tr>
<tr>
<td>LOAD MATCHED (binary)</td>
<td>VLM (short)</td>
<td>VV VE SP</td>
<td>A40A</td>
</tr>
</tbody>
</table>

Figure B-1 (Part 2 of 5). Instructions Arranged by Name
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD MATCHED (binary)</td>
<td>VLMQ</td>
<td>QV VE</td>
<td>IC VB</td>
</tr>
<tr>
<td>LOAD MATCHED (binary)¹</td>
<td>VLMR</td>
<td>VV VE</td>
<td>IC VB</td>
</tr>
<tr>
<td>LOAD MATCHED (long)</td>
<td>VLMQ</td>
<td>VST VE A SP</td>
<td>IC VB</td>
</tr>
<tr>
<td>LOAD MATCHED (long)</td>
<td>VLMQ</td>
<td>QV VE SP</td>
<td>IC VB</td>
</tr>
<tr>
<td>LOAD MATCHED (long)</td>
<td>VLMR</td>
<td>VV VE SP</td>
<td>IC VB</td>
</tr>
<tr>
<td>LOAD MATCHED (short)</td>
<td>VLMR</td>
<td>VST VE A SP</td>
<td>IC VB</td>
</tr>
<tr>
<td>LOAD MATCHED (short)</td>
<td>VLMR</td>
<td>QV VE SP</td>
<td>IC VB</td>
</tr>
<tr>
<td>LOAD MATCHED (short)</td>
<td>VLMR</td>
<td>VV VE</td>
<td>IC VB</td>
</tr>
<tr>
<td>LOAD NEGATIVE (binary)</td>
<td>VLMDR</td>
<td>VV VE</td>
<td>IM VB</td>
</tr>
<tr>
<td>LOAD NEGATIVE (long)</td>
<td>VLMDR</td>
<td>VV VE SP</td>
<td>IM VB</td>
</tr>
<tr>
<td>LOAD NEGATIVE (short)</td>
<td>VLMDR</td>
<td>VV VE</td>
<td>IM VB</td>
</tr>
<tr>
<td>LOAD POSITIVE (binary)</td>
<td>VLMPR</td>
<td>VV VE</td>
<td>IM VB</td>
</tr>
<tr>
<td>LOAD POSITIVE (long)</td>
<td>VLMDR</td>
<td>VV VE SP</td>
<td>IM VB</td>
</tr>
<tr>
<td>LOAD POSITIVE (short)</td>
<td>VLMCR</td>
<td>VV VE</td>
<td>IM VB</td>
</tr>
<tr>
<td>LOAD VCT AND UPDATE</td>
<td>VLVCA</td>
<td>S C VE</td>
<td>NO</td>
</tr>
<tr>
<td>LOAD VCT FROM ADDRESS</td>
<td>VLVCA</td>
<td>S C VE</td>
<td>NO</td>
</tr>
<tr>
<td>LOAD VNR</td>
<td>VLMR</td>
<td>VS VE A</td>
<td>NC</td>
</tr>
<tr>
<td>LOAD VNR COMPLEMENT</td>
<td>VLMCR</td>
<td>VS VE A</td>
<td>NC</td>
</tr>
<tr>
<td>LOAD ZERO (binary)²</td>
<td>VLMER</td>
<td>VV VE</td>
<td>IC VB</td>
</tr>
<tr>
<td>LOAD ZERO (long)</td>
<td>VLMER</td>
<td>VV VE SP</td>
<td>IC VB</td>
</tr>
<tr>
<td>LOAD ZERO (short)</td>
<td>VLMER</td>
<td>VV VE</td>
<td>IC VB</td>
</tr>
<tr>
<td>MAXIMUM ABSOLUTE (long)</td>
<td>VMXAD</td>
<td>VR VE SP</td>
<td>IM</td>
</tr>
<tr>
<td>MAXIMUM ABSOLUTE (short)</td>
<td>VMXAD</td>
<td>VR VE SP</td>
<td>IM</td>
</tr>
<tr>
<td>MAXIMUM SIGNED (long)</td>
<td>VMXSD</td>
<td>VR VE SP</td>
<td>IM</td>
</tr>
<tr>
<td>MAXIMUM SIGNED (short)</td>
<td>VMXSD</td>
<td>VR VE SP</td>
<td>IM</td>
</tr>
<tr>
<td>MINIMUM SIGNED (long)</td>
<td>VMNSD</td>
<td>VR VE SP</td>
<td>IM</td>
</tr>
<tr>
<td>MINIMUM SIGNED (short)</td>
<td>VMNSE</td>
<td>VR VE SP</td>
<td>IM</td>
</tr>
<tr>
<td>MULTIPLY (binary)</td>
<td>VM</td>
<td>VST VE A SP</td>
<td>IM VB</td>
</tr>
<tr>
<td>MULTIPLY (binary)</td>
<td>VMQ</td>
<td>QV VE SP</td>
<td>IM VB</td>
</tr>
<tr>
<td>MULTIPLY (binary)</td>
<td>VMS</td>
<td>VV VE SP</td>
<td>IM VB</td>
</tr>
<tr>
<td>MULTIPLY (binary)</td>
<td>VMS</td>
<td>QST VE A SP</td>
<td>IM VB</td>
</tr>
<tr>
<td>MULTIPLY (long)</td>
<td>VMD</td>
<td>VST VE A SP</td>
<td>J U EU EO IM VB</td>
</tr>
<tr>
<td>MULTIPLY (long)</td>
<td>VMD</td>
<td>QV VE SP</td>
<td>J U EU EO IM VB</td>
</tr>
<tr>
<td>MULTIPLY (long)</td>
<td>VMDR</td>
<td>VV VE SP</td>
<td>J U EU EO IM VB</td>
</tr>
<tr>
<td>MULTIPLY (long)</td>
<td>VMDS</td>
<td>QST VE A SP</td>
<td>J U EU EO IM VB</td>
</tr>
<tr>
<td>MULTIPLY (short to long)</td>
<td>VME</td>
<td>VST VE A SP</td>
<td>J U EU EO IM VB</td>
</tr>
<tr>
<td>MULTIPLY (short to long)</td>
<td>VMEQ</td>
<td>QV VE SP</td>
<td>J U EU EO IM VB</td>
</tr>
<tr>
<td>MULTIPLY (short to long)</td>
<td>VMER</td>
<td>VV VE SP</td>
<td>J U EU EO IM VB</td>
</tr>
<tr>
<td>MULTIPLY (short to long)</td>
<td>VMES</td>
<td>QST VE A SP</td>
<td>J U EU EO IM VB</td>
</tr>
<tr>
<td>MULTIPLY AND ACCUMULATE (long)</td>
<td>VMCD</td>
<td>VST VE A SP</td>
<td>J U EU EO LS IM VB</td>
</tr>
</tbody>
</table>

Figure B-1 (Part 3 of 5). Instructions Arranged by Name
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULTIPLY AND ACCUMULATE (long)</td>
<td>VMCOD</td>
<td>J U EU EO LS IM VB</td>
<td>A516</td>
</tr>
<tr>
<td>MULTIPLY AND ACCUMULATE (short)</td>
<td>VMCE</td>
<td>J U EU EO LS IM VB</td>
<td>A406</td>
</tr>
<tr>
<td>MULTIPLY AND ACCUMULATE (long)</td>
<td>VMCE</td>
<td>J U EU EO LS IM VB</td>
<td>A506</td>
</tr>
<tr>
<td>MULTIPLY AND ADD (long)</td>
<td>VMAD</td>
<td>J U EU EO LS IM VB</td>
<td>A414</td>
</tr>
<tr>
<td>MULTIPLY AND ADD (long)</td>
<td>VMADQ</td>
<td>J U EU EO LS IM VB</td>
<td>A594</td>
</tr>
<tr>
<td>MULTIPLY AND ADD (long)</td>
<td>VMADS</td>
<td>J U EU EO LS IM VB</td>
<td>A494</td>
</tr>
<tr>
<td>MULTIPLY AND ADD (long)</td>
<td>VMAE</td>
<td>J U EU EO LS IM VB</td>
<td>A404</td>
</tr>
<tr>
<td>MULTIPLY AND ADD (short to long)</td>
<td>VMAES</td>
<td>J U EU EO LS IM VB</td>
<td>A484</td>
</tr>
<tr>
<td>MULTIPLY AND SUBTRACT (long)</td>
<td>VMSD</td>
<td>J U EU EO LS IM VB</td>
<td>A415</td>
</tr>
<tr>
<td>MULTIPLY AND SUBTRACT (long)</td>
<td>VMSDQ</td>
<td>J U EU EO LS IM VB</td>
<td>A595</td>
</tr>
<tr>
<td>MULTIPLY AND SUBTRACT (short to long)</td>
<td>VMSQ</td>
<td>J U EU EO LS IM VB</td>
<td>A495</td>
</tr>
<tr>
<td>MULTIPLY AND SUBTRACT (short to long)</td>
<td>VMSQ</td>
<td>J U EU EO LS IM VB</td>
<td>A405</td>
</tr>
<tr>
<td>OR</td>
<td>VO</td>
<td>IM VB R*</td>
<td>A425</td>
</tr>
<tr>
<td>OR</td>
<td>VOQ</td>
<td>IM VB R*</td>
<td>A5A5</td>
</tr>
<tr>
<td>OR</td>
<td>VOR</td>
<td>IM VB R*</td>
<td>A525</td>
</tr>
<tr>
<td>OR</td>
<td>VOS</td>
<td>IM VB R*</td>
<td>A445</td>
</tr>
<tr>
<td>OR TO VMR</td>
<td>VOV</td>
<td>NC R*</td>
<td>A685</td>
</tr>
<tr>
<td>RESTORE VAC</td>
<td>VACRS</td>
<td>P NO</td>
<td>A6CB</td>
</tr>
<tr>
<td>RESTORE VR</td>
<td>VMRRS</td>
<td>NZ</td>
<td>A6C3</td>
</tr>
<tr>
<td>RESTORE VR</td>
<td>VRRS</td>
<td>2 IZ VU R*</td>
<td>A648</td>
</tr>
<tr>
<td>RESTORE VSR</td>
<td>VSRRS</td>
<td>2 IZ VB</td>
<td>A6C2</td>
</tr>
<tr>
<td>SAVE CHANGED VR</td>
<td>VSRVC</td>
<td>P IZ VH R* ST</td>
<td>A649</td>
</tr>
<tr>
<td>SAVE VAC</td>
<td>VACSV</td>
<td>P NO ST</td>
<td>A6CA</td>
</tr>
<tr>
<td>SAVE VR</td>
<td>VMRSV</td>
<td>NZ ST</td>
<td>A6C1</td>
</tr>
<tr>
<td>SAVE VR</td>
<td>VRSV</td>
<td>IZ R* ST</td>
<td>A64A</td>
</tr>
<tr>
<td>SAVE VSR</td>
<td>VSRVS</td>
<td>NO ST</td>
<td>A6C8</td>
</tr>
<tr>
<td>SET VECTOR MASK MODE</td>
<td>VSVMM</td>
<td>NO</td>
<td>A6C6</td>
</tr>
<tr>
<td>SHIFT LEFT SINGLE LOGICAL</td>
<td>VSLL</td>
<td>IM VB</td>
<td>E425</td>
</tr>
<tr>
<td>SHIFT RIGHT SINGLE LOGICAL</td>
<td>VSRL</td>
<td>IM VB</td>
<td>E424</td>
</tr>
<tr>
<td>STORE (binary)</td>
<td>VST</td>
<td>IC R* ST</td>
<td>A40D</td>
</tr>
<tr>
<td>STORE (long)</td>
<td>VSTD</td>
<td>IC R* ST</td>
<td>A41D</td>
</tr>
<tr>
<td>STORE (short)</td>
<td>VSTE</td>
<td>IC R* ST</td>
<td>A40D</td>
</tr>
<tr>
<td>STORE COMPRESSED (binary)</td>
<td>VSTK</td>
<td>IC R* ST</td>
<td>A40F</td>
</tr>
<tr>
<td>STORE COMPRESSED (long)</td>
<td>VSTKD</td>
<td>IC R* ST</td>
<td>A41F</td>
</tr>
<tr>
<td>STORE COMPRESSED (short)</td>
<td>VSTKE</td>
<td>IC R* ST</td>
<td>A40F</td>
</tr>
<tr>
<td>STORE HALFWORD</td>
<td>VSTH</td>
<td>IC R* ST</td>
<td>A42D</td>
</tr>
<tr>
<td>STORE INDIRECT (binary)</td>
<td>VSTI</td>
<td>IC ST</td>
<td>E401</td>
</tr>
</tbody>
</table>

Figure B-1 (Part 4 of 5). Instructions Arranged by Name
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORE INDIRECT (long)</td>
<td>VSTID</td>
<td>RSE VE A SP</td>
<td>IC</td>
</tr>
<tr>
<td>STORE INDIRECT (short)</td>
<td>VSTIE</td>
<td>RSE VE A SP</td>
<td>IC</td>
</tr>
<tr>
<td>STORE MATCHED (binary)</td>
<td>VSTM</td>
<td>VST VE A SP</td>
<td>IC</td>
</tr>
<tr>
<td>STORE MATCHED (long)</td>
<td>VSTMD</td>
<td>VST VE A SP</td>
<td>IC</td>
</tr>
<tr>
<td>STORE MATCHED (short)</td>
<td>VSTME</td>
<td>VST VE A SP</td>
<td>IC</td>
</tr>
<tr>
<td>STORE VECTOR PARAMETERS</td>
<td>VSTVP</td>
<td>S VE A SP</td>
<td>NO</td>
</tr>
<tr>
<td>STORE VMR</td>
<td>VSTVM</td>
<td>VS VE A</td>
<td>NC</td>
</tr>
<tr>
<td>SUBTRACT (binary)</td>
<td>VS</td>
<td>VST VE A SP</td>
<td>J IF</td>
</tr>
<tr>
<td>SUBTRACT (binary)</td>
<td>VSQ</td>
<td>QV VE</td>
<td>J IF</td>
</tr>
<tr>
<td>SUBTRACT (binary)</td>
<td>VSR</td>
<td>VV VE</td>
<td>J IF</td>
</tr>
<tr>
<td>SUBTRACT (binary)</td>
<td>VSS</td>
<td>QST VE A SP</td>
<td>J IF</td>
</tr>
<tr>
<td>SUBTRACT (long)</td>
<td>VSD</td>
<td>VST VE A SP</td>
<td>J EU EO LS IM VB R* A411</td>
</tr>
<tr>
<td>SUBTRACT (long)</td>
<td>VSDQ</td>
<td>QV VE SP</td>
<td>J EU EO LS IM VB A591</td>
</tr>
<tr>
<td>SUBTRACT (long)</td>
<td>VSDR</td>
<td>VV VE SP</td>
<td>J EU EO LS IM VB A511</td>
</tr>
<tr>
<td>SUBTRACT (long)</td>
<td>VSDS</td>
<td>QST VE A SP</td>
<td>J EU EO LS IM VB R* A491</td>
</tr>
<tr>
<td>SUBTRACT (short)</td>
<td>VSE</td>
<td>VST VE A SP</td>
<td>J EU EO LS IM VB R* A401</td>
</tr>
<tr>
<td>SUBTRACT (short)</td>
<td>VSEQ</td>
<td>QV VE SP</td>
<td>J EU EO LS IM VB A581</td>
</tr>
<tr>
<td>SUBTRACT (short)</td>
<td>VSER</td>
<td>VV VE</td>
<td>J EU EO LS IM VB A501</td>
</tr>
<tr>
<td>SUBTRACT (short)</td>
<td>VSES</td>
<td>QST VE A SP</td>
<td>J EU EO LS IM VB R* A481</td>
</tr>
<tr>
<td>SUM PARTIAL SUMS (long)</td>
<td>VSPSD</td>
<td>VR VE</td>
<td>J EU EO LS IP A61A</td>
</tr>
<tr>
<td>TEST VMR</td>
<td>VTM</td>
<td>RRE C VE</td>
<td>NC</td>
</tr>
<tr>
<td>ZERO PARTIAL SUMS (long)</td>
<td>VZPSD</td>
<td>VR VE</td>
<td>SP</td>
</tr>
</tbody>
</table>

Figure B-1 (Part 5 of 5). Instructions Arranged by Name
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>VA</td>
<td>ADD  (binary)</td>
<td>VST VE A SP J IF IM VB R*</td>
<td>A420</td>
</tr>
<tr>
<td>VACD</td>
<td>ACCUMULATE (long)</td>
<td>VST VE A SP J EU EO LS IM VB R*</td>
<td>A417</td>
</tr>
<tr>
<td>VACDR</td>
<td>ACCUMULATE (long)</td>
<td>VV VE SP J EU EO LS IM VB</td>
<td>A517</td>
</tr>
<tr>
<td>VACE</td>
<td>ACCUMULATE (short to long)</td>
<td>VST VE A SP J EU EO LS IM VB R*</td>
<td>A407</td>
</tr>
<tr>
<td>VACER</td>
<td>ACCUMULATE (short to long)</td>
<td>VV VE SP J EU EO LS IM VB</td>
<td>A507</td>
</tr>
<tr>
<td>VACRS</td>
<td>RESTORE VAC</td>
<td>S VE A SP P N0</td>
<td>A6CB</td>
</tr>
<tr>
<td>VACSV</td>
<td>SAVE VAC</td>
<td>S VE A SP P N0 ST</td>
<td>A6CA</td>
</tr>
<tr>
<td>VAD</td>
<td>ADD (long)</td>
<td>VQ VE J EU EO LS IM VB R*</td>
<td>A410</td>
</tr>
<tr>
<td>VADOQ</td>
<td>ADD (long)</td>
<td>VV VE J EU EO LS IM VB</td>
<td>A509</td>
</tr>
<tr>
<td>VADR</td>
<td>ADD (long)</td>
<td>VV VE J EU EO LS IM VB</td>
<td>A510</td>
</tr>
<tr>
<td>VAQ</td>
<td>ADD (binary)</td>
<td>QV VE J IF IM VB</td>
<td>A500</td>
</tr>
<tr>
<td>VAR</td>
<td>ADD (binary)</td>
<td>VV VE J IF IM VB</td>
<td>A529</td>
</tr>
<tr>
<td>VAS</td>
<td>ADD (binary)</td>
<td>QST VE A SP J IF IM VB R*</td>
<td>A408</td>
</tr>
<tr>
<td>VC</td>
<td>COMPARE (binary)</td>
<td>VST VE A SP IC R*</td>
<td>A428</td>
</tr>
<tr>
<td>VCD</td>
<td>COMPARE (binary)</td>
<td>VST VE A SP IC R*</td>
<td>A418</td>
</tr>
<tr>
<td>VCDQ</td>
<td>COMPARE (long)</td>
<td>QV VE SP IC</td>
<td>A598</td>
</tr>
<tr>
<td>VCDR</td>
<td>COMPARE (long)</td>
<td>VV VE SP IC</td>
<td>A518</td>
</tr>
<tr>
<td>VCDQ</td>
<td>COMPARE (long)</td>
<td>QV VE SP IC</td>
<td>A598</td>
</tr>
<tr>
<td>VCE</td>
<td>COMPARE (short)</td>
<td>VST VE A SP IC R*</td>
<td>A408</td>
</tr>
<tr>
<td>VCEQ</td>
<td>COMPARE (short)</td>
<td>QV VE SP IC</td>
<td>A588</td>
</tr>
<tr>
<td>VCR</td>
<td>COMPARE (short)</td>
<td>VV VE A SP IC</td>
<td>A528</td>
</tr>
<tr>
<td>VCS</td>
<td>COMPARE (binary)</td>
<td>QST VE A SP IC R*</td>
<td>A408</td>
</tr>
<tr>
<td>VCSV</td>
<td>COMPLEMENT VMR</td>
<td>RRE VE NC</td>
<td>A641</td>
</tr>
<tr>
<td>VCSV</td>
<td>COUNT LEFT ZEROS IN VMR</td>
<td>RRE C VE NC</td>
<td>A642</td>
</tr>
<tr>
<td>VDD</td>
<td>DIVIDE (long)</td>
<td>VST VE J EU EO FK IM VB R*</td>
<td>A413</td>
</tr>
<tr>
<td>VDDQ</td>
<td>DIVIDE (long)</td>
<td>QV VE J EU EO FK IM VB</td>
<td>A593</td>
</tr>
<tr>
<td>VDDR</td>
<td>DIVIDE (long)</td>
<td>VV VE J EU EO FK IM VB</td>
<td>A513</td>
</tr>
<tr>
<td>VDDS</td>
<td>DIVIDE (long)</td>
<td>QST VE J EU EO FK IM VB R*</td>
<td>A493</td>
</tr>
<tr>
<td>VDE</td>
<td>DIVIDE (short)</td>
<td>VST VE J EU EO FK IM VB R*</td>
<td>A483</td>
</tr>
<tr>
<td>VDEQ</td>
<td>DIVIDE (short)</td>
<td>QV VE J EU EO FK IM VB</td>
<td>A583</td>
</tr>
<tr>
<td>VDER</td>
<td>DIVIDE (short)</td>
<td>VV VE J EU EO FK IM VB</td>
<td>A563</td>
</tr>
</tbody>
</table>

Figure B-2 (Part 1 of 5). Instructions Arranged by Mnemonic
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>VDES</td>
<td>DIVIDE (short)</td>
<td>QST VE A SP J U EU E0 FK IM VB R*</td>
<td>A483</td>
</tr>
<tr>
<td>VL</td>
<td>LOAD (binary)</td>
<td>VST VE A SP IC VB</td>
<td>A409</td>
</tr>
<tr>
<td>VLBIX</td>
<td>LOAD BIT INDEX</td>
<td>RSE C VE A SP IG VB</td>
<td>E428</td>
</tr>
<tr>
<td>VLCDR</td>
<td>LOAD COMPLEMENT (long)</td>
<td>VV VE SP IM VB</td>
<td>A552</td>
</tr>
<tr>
<td>VLCER</td>
<td>LOAD COMPLEMENT (short)</td>
<td>VV VE IM VB</td>
<td>A542</td>
</tr>
<tr>
<td>VLCR</td>
<td>LOAD COMPLEMENT (binary)</td>
<td>VV VE J IF IM VB</td>
<td>A562</td>
</tr>
<tr>
<td>VLCVM</td>
<td>LOAD VMR COMPLEMENT</td>
<td>VS VE A NC R*</td>
<td>A681</td>
</tr>
<tr>
<td>VLD</td>
<td>LOAD (long)</td>
<td>VST VE A SP IC VB</td>
<td>A419</td>
</tr>
<tr>
<td>VLDQ</td>
<td>LOAD (long)</td>
<td>QV VE SP IC VB</td>
<td>A599</td>
</tr>
<tr>
<td>VLDR</td>
<td>LOAD (long)</td>
<td>VV VE SP IC VB</td>
<td>A519</td>
</tr>
<tr>
<td>VLE</td>
<td>LOAD (short)</td>
<td>VST VE A SP IC VB</td>
<td>A409</td>
</tr>
<tr>
<td>VLEL</td>
<td>LOAD ELEMENT (binary)</td>
<td>VR VE SP N1 VB</td>
<td>A628</td>
</tr>
<tr>
<td>VLELD</td>
<td>LOAD ELEMENT (long)</td>
<td>VR VE SP N1 VB</td>
<td>A618</td>
</tr>
<tr>
<td>VLELE</td>
<td>LOAD ELEMENT (short)</td>
<td>VR VE SP N1 VB</td>
<td>A608</td>
</tr>
<tr>
<td>VLEQ</td>
<td>LOAD (short)</td>
<td>QV VE SP IC VB</td>
<td>A589</td>
</tr>
<tr>
<td>VLER</td>
<td>LOAD (short)</td>
<td>VV VE IC VB R*</td>
<td>A599</td>
</tr>
<tr>
<td>VLM</td>
<td>LOAD MATCHED (binary)</td>
<td>VST VE A SP IC VB</td>
<td>A429</td>
</tr>
<tr>
<td>VLMQ</td>
<td>LOAD MATCHED (binary)</td>
<td>QV VE IC VB A5AA</td>
<td></td>
</tr>
<tr>
<td>VLMR</td>
<td>LOAD MATCHED (binary)</td>
<td>VV VE IC VB A50A</td>
<td></td>
</tr>
<tr>
<td>VLNDR</td>
<td>LOAD MATCHED (short)</td>
<td>VST VE A SP IC VB</td>
<td>A40A</td>
</tr>
<tr>
<td>VLMQ</td>
<td>LOAD MATCHED (short)</td>
<td>QV VE IC VB A50A</td>
<td></td>
</tr>
<tr>
<td>VLPR</td>
<td>LOAD MATCHED (binary)</td>
<td>VV VE JC IF IM VB</td>
<td>A560</td>
</tr>
<tr>
<td>VLO</td>
<td>LOAD (binary)</td>
<td>QV VE IC VB A599</td>
<td></td>
</tr>
<tr>
<td>VLR</td>
<td>LOAD (binary)</td>
<td>VV VE IC VB A509</td>
<td></td>
</tr>
<tr>
<td>VLVCA</td>
<td>LOAD VCT FROM ADDRESS</td>
<td>S C VE NO</td>
<td>A6C4</td>
</tr>
<tr>
<td>VLVCU</td>
<td>LOAD VCT AND UPDATE</td>
<td>RRE C VE NO R*</td>
<td>A645</td>
</tr>
</tbody>
</table>

Figure B-2 (Part 2 of 5). Instructions Arranged by Mnemonic
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLVM</td>
<td>LOAD VMR</td>
<td>VS VE A</td>
<td>NC R* A680</td>
</tr>
<tr>
<td>VLY</td>
<td>LOAD EXPANDED (binary)</td>
<td>VST VE A SP IC VB R* A40B</td>
<td></td>
</tr>
<tr>
<td>VLYD</td>
<td>LOAD EXPANDED (long)</td>
<td>VST VE A SP IC VB R* A41B</td>
<td></td>
</tr>
<tr>
<td>VLYE</td>
<td>LOAD EXPANDED (short)</td>
<td>VST VE A SP IC VB R* A40B</td>
<td></td>
</tr>
<tr>
<td>VLZDR</td>
<td>LOAD ZERO (long)</td>
<td>VV VE IC VB</td>
<td>A51B</td>
</tr>
<tr>
<td>VLY</td>
<td>LOAD ZERO (short)</td>
<td>VV VE IC VB</td>
<td>A50B</td>
</tr>
<tr>
<td>VLYR</td>
<td>LOAD ZERO (binary)</td>
<td>VV VE IC VB</td>
<td>A50B</td>
</tr>
<tr>
<td>VM</td>
<td>MULTIPLY (binary)</td>
<td>VST VE A SP J EU EO LS IM VB R* A422</td>
<td></td>
</tr>
<tr>
<td>VMAD</td>
<td>MULTIPLY AND ADD (long)</td>
<td>VST VE A SP J EU EO LS IM VB R* A414</td>
<td></td>
</tr>
<tr>
<td>VMADO</td>
<td>MULTIPLY AND ADD (long)</td>
<td>QST VE A SP J EU EO LS IM VB R* A594</td>
<td></td>
</tr>
<tr>
<td>VMADS</td>
<td>MULTIPLY AND ADD (long)</td>
<td>QST VE A SP J EU EO LS IM VB R* A494</td>
<td></td>
</tr>
<tr>
<td>VMAE</td>
<td>MULTIPLY AND ADD (short to long)</td>
<td>VST VE A SP J EU EO LS IM VB R* A404</td>
<td></td>
</tr>
<tr>
<td>VMADS</td>
<td>MULTIPLY AND ADD (short to long)</td>
<td>QST VE A SP J EU EO LS IM VB R* A584</td>
<td></td>
</tr>
<tr>
<td>VMAES</td>
<td>MULTIPLY AND ADD (short to long)</td>
<td>QST VE A SP J EU EO LS IM VB R* A484</td>
<td></td>
</tr>
<tr>
<td>VMCD</td>
<td>MULTIPLY AND ACCUMULATE (long)</td>
<td>VST VE A SP J EU EO LS IM VB R* A416</td>
<td></td>
</tr>
<tr>
<td>VMCE</td>
<td>MULTIPLY AND ACCUMULATE (short to long)</td>
<td>VST VE A SP J EU EO LS IM VB R* A406</td>
<td></td>
</tr>
<tr>
<td>VMCDR</td>
<td>MULTIPLY AND ACCUMULATE (long)</td>
<td>VV VE A SP J EU EO LS IM VB R* A516</td>
<td></td>
</tr>
<tr>
<td>VMCE</td>
<td>MULTIPLY AND ACCUMULATE (short to long)</td>
<td>VV VE A SP J EU EO LS IM VB R* A506</td>
<td></td>
</tr>
<tr>
<td>VMER</td>
<td>MULTIPLY (long)</td>
<td>VV VE A SP J EU EO IM VB R* A512</td>
<td></td>
</tr>
<tr>
<td>VMD</td>
<td>MULTIPLY (long)</td>
<td>VV VE A SP J EU EO IM VB R* A412</td>
<td></td>
</tr>
<tr>
<td>VMDR</td>
<td>MULTIPLY (long)</td>
<td>QST VE A SP J EU EO IM VB R* A492</td>
<td></td>
</tr>
<tr>
<td>VMDS</td>
<td>MULTIPLY (long)</td>
<td>QST VE A SP J EU EO IM VB R* A506</td>
<td></td>
</tr>
<tr>
<td>VME</td>
<td>MULTIPLY (short to long)</td>
<td>VST VE A SP J EU EO IM VB R* A402</td>
<td></td>
</tr>
<tr>
<td>VMEQ</td>
<td>MULTIPLY (short to long)</td>
<td>QV VE SP J EU EO IM VB R* A502</td>
<td></td>
</tr>
<tr>
<td>VMER</td>
<td>MULTIPLY (short to long)</td>
<td>VV VE A SP J EU EO IM VB R* A502</td>
<td></td>
</tr>
<tr>
<td>VMES</td>
<td>MULTIPLY (short to long)</td>
<td>QST VE A SP J EU EO IM VB R* A482</td>
<td></td>
</tr>
<tr>
<td>VMNSD</td>
<td>MINIMUM SIGNED (long)</td>
<td>VR VE SP IM R* A611</td>
<td></td>
</tr>
<tr>
<td>VMRS</td>
<td>MINIMUM SIGNED (short)</td>
<td>VR VE SP IM R* A601</td>
<td></td>
</tr>
<tr>
<td>VMQ</td>
<td>MULTIPLY (binary)</td>
<td>QV VE SP IM VB A5A2</td>
<td></td>
</tr>
<tr>
<td>VMR</td>
<td>MULTIPLY (binary)</td>
<td>VV VE SP IM VB A522</td>
<td></td>
</tr>
<tr>
<td>VMRRS</td>
<td>RESTORE VMR</td>
<td>S VE A NZ</td>
<td>A6C3</td>
</tr>
<tr>
<td>VMRSV</td>
<td>SAVE VMR</td>
<td>S VE A NZ</td>
<td>A6C1</td>
</tr>
<tr>
<td>VMS</td>
<td>MULTIPLY (binary)</td>
<td>QST VE A SP IM VB R* A4A2</td>
<td></td>
</tr>
<tr>
<td>VMSD</td>
<td>MULTIPLY AND SUBTRACT (long)</td>
<td>VST VE A SP J EU EO LS IM VB R* A415</td>
<td></td>
</tr>
<tr>
<td>VMODQ</td>
<td>MULTIPLY AND SUBTRACT (long)</td>
<td>QV VE SP J EU EO LS IM VB R* A595</td>
<td></td>
</tr>
<tr>
<td>VMSDS</td>
<td>MULTIPLY AND SUBTRACT (long)</td>
<td>QST VE A SP J EU EO LS IM VB R* A495</td>
<td></td>
</tr>
<tr>
<td>VMSE</td>
<td>MULTIPLY AND SUBTRACT (short to long)</td>
<td>QST VE A SP J EU EO LS IM VB R* A405</td>
<td></td>
</tr>
<tr>
<td>VMSQ</td>
<td>MULTIPLY AND SUBTRACT (short to long)</td>
<td>QV VE SP J EU EO LS IM VB R* A585</td>
<td></td>
</tr>
<tr>
<td>VMSES</td>
<td>MULTIPLY AND SUBTRACT (short to long)</td>
<td>QST VE A SP J EU EO LS IM VB R* A485</td>
<td></td>
</tr>
<tr>
<td>VMXAD</td>
<td>MAXIMUM ABSOLUTE (long)</td>
<td>VR VE SP IM R* A612</td>
<td></td>
</tr>
</tbody>
</table>

Figure B-2 (Part 3 of 5). Instructions Arranged by Mnemonic

Appendix B. Lists Of Instructions B-9
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMXAE</td>
<td>MAXIMUM ABSOLUTE (short)</td>
<td>VR VE SP</td>
<td>IM R* A602</td>
</tr>
<tr>
<td>VMXSD</td>
<td>MAXIMUM SIGNED (long)</td>
<td>VR VE SP</td>
<td>IM R* A610</td>
</tr>
<tr>
<td>VMXSE</td>
<td>MAXIMUM SIGNED (short)</td>
<td>VR VE SP</td>
<td>IM R* A608</td>
</tr>
<tr>
<td>VN AND</td>
<td>VST VE A SP</td>
<td>IM VB</td>
<td>A424</td>
</tr>
<tr>
<td>VNQ AND</td>
<td>QV VE</td>
<td>IM VB</td>
<td>A5A4</td>
</tr>
<tr>
<td>VNR AND</td>
<td>VV VE</td>
<td>IM VB</td>
<td>A524</td>
</tr>
<tr>
<td>VNS AND</td>
<td>QST VE A SP</td>
<td>IM VB</td>
<td>A4A4</td>
</tr>
<tr>
<td>VNVM AND TO VMR</td>
<td>VS VE A</td>
<td>NC</td>
<td>R* A684</td>
</tr>
<tr>
<td>VO OR</td>
<td>VST VE A SP</td>
<td>IM VB</td>
<td>A425</td>
</tr>
<tr>
<td>VQQ OR</td>
<td>QV VE</td>
<td>IM VB</td>
<td>A5A5</td>
</tr>
<tr>
<td>VOR OR</td>
<td>VV VE</td>
<td>IM VB</td>
<td>A525</td>
</tr>
<tr>
<td>VOS OR</td>
<td>QST VE A SP</td>
<td>IM VB</td>
<td>A4A5</td>
</tr>
<tr>
<td>VOVN OR TO VMR</td>
<td>VS VE A</td>
<td>NC</td>
<td>R* A685</td>
</tr>
<tr>
<td>VRLC CLEAR VR</td>
<td>S VE</td>
<td>IZ VB</td>
<td>A6C5</td>
</tr>
<tr>
<td>VRSS RESTORE VR</td>
<td>RRE C VE A SP 2</td>
<td>IZ VU R*</td>
<td>A648</td>
</tr>
<tr>
<td>VRSV SAVE VR</td>
<td>RRE C VE A SP</td>
<td>IZ R* ST</td>
<td>A6A4</td>
</tr>
<tr>
<td>VRSC SAVE CHANGED VR</td>
<td>RRE C VE A SP P</td>
<td>IZ VH R* ST</td>
<td>A649</td>
</tr>
<tr>
<td>VS SUBTRACT (binary)</td>
<td>VST VE A SP J IF</td>
<td>IM VB R*</td>
<td>A421</td>
</tr>
<tr>
<td>VSO SUBTRACT (long)</td>
<td>VST VE A SP J EU EO LS IM VB</td>
<td>R* A411</td>
<td></td>
</tr>
<tr>
<td>VSDQ SUBTRACT (long)</td>
<td>QV VE</td>
<td>J EU EO LS IM VB</td>
<td>A591</td>
</tr>
<tr>
<td>VSDR SUBTRACT (long)</td>
<td>VV VE SP</td>
<td>J EU EO LS IM VB</td>
<td>A511</td>
</tr>
<tr>
<td>VSDS SUBTRACT (long)</td>
<td>QST VE A SP</td>
<td>J EU EO LS IM VB</td>
<td>A491</td>
</tr>
<tr>
<td>VSE SUBTRACT (short)</td>
<td>VST VE A SP J EU EO LS IM VB</td>
<td>R* A401</td>
<td></td>
</tr>
<tr>
<td>VSEQ SUBTRACT (short)</td>
<td>QV VE SP</td>
<td>J EU EO LS IM VB</td>
<td>A581</td>
</tr>
<tr>
<td>VSER SUBTRACT (short)</td>
<td>VV VE</td>
<td>J EU EO LS IM VB</td>
<td>A581</td>
</tr>
<tr>
<td>VSES SUBTRACT (short)</td>
<td>QST VE A SP</td>
<td>J EU EO LS IM VB</td>
<td>R* A481</td>
</tr>
<tr>
<td>VSSL SHIFIT LEFT SINGLE LOGICAL</td>
<td>RSE VE</td>
<td>IM VB</td>
<td>A425</td>
</tr>
<tr>
<td>VVPSD SUM PARTIAL SUMS (long)</td>
<td>VR VE SP</td>
<td>J EU EO LS IP</td>
<td>A61A</td>
</tr>
<tr>
<td>VSQ SUBTRACT (binary)</td>
<td>QV VE</td>
<td>J IF IM VB</td>
<td>A5A1</td>
</tr>
<tr>
<td>VSR SHIFIT RIGHT SINGLE LOGICAL</td>
<td>RSE VE</td>
<td>IM VB</td>
<td>E424</td>
</tr>
<tr>
<td>VSR SUBTRACT (binary)</td>
<td>VV VE</td>
<td>J IF IM VB</td>
<td>A521</td>
</tr>
<tr>
<td>VSRSS RESTORE VSR</td>
<td>S VE A SP 2</td>
<td>IZ VB</td>
<td>A6C2</td>
</tr>
<tr>
<td>VSRSV SAVE VSR</td>
<td>S VE A SP 2</td>
<td>NO ST</td>
<td>A6C0</td>
</tr>
<tr>
<td>VSS SUBTRACT (binary)</td>
<td>QST VE A SP J IF</td>
<td>IM VB R*</td>
<td>A4A1</td>
</tr>
<tr>
<td>VST STORE (binary)</td>
<td>VST VE A SP</td>
<td>IC R* ST</td>
<td>A40D</td>
</tr>
<tr>
<td>VSTD STORE (long)</td>
<td>VST VE A SP</td>
<td>IC</td>
<td>R* ST A41D</td>
</tr>
<tr>
<td>VSTE STORE (short)</td>
<td>VST VE A SP</td>
<td>IC</td>
<td>R* ST A40D</td>
</tr>
<tr>
<td>VSTH STORE HALFWORD</td>
<td>VST VE A SP</td>
<td>IC</td>
<td>R* ST A42D</td>
</tr>
<tr>
<td>VSTI STORE INDIRECT (binary)</td>
<td>RSE VE A SP</td>
<td>IC</td>
<td>ST E401</td>
</tr>
<tr>
<td>VSTID STORE INDIRECT (long)</td>
<td>RSE VE A SP</td>
<td>IC</td>
<td>ST E411</td>
</tr>
</tbody>
</table>

Figure B-2 (Part 4 of 5). Instructions Arranged by Mnemonic
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSTIE</td>
<td>STORE INDIRECT (short)</td>
<td>RSE VE A SP</td>
<td>IC ST E401</td>
</tr>
<tr>
<td>VSTK</td>
<td>STORE COMPRESSED (binary)¹</td>
<td>VST VE A SP</td>
<td>IC R* ST A40F</td>
</tr>
<tr>
<td>VSTKD</td>
<td>STORE COMPRESSED (long)</td>
<td>VST VE A SP</td>
<td>IC R* ST A41F</td>
</tr>
<tr>
<td>VSTKE</td>
<td>STORE COMPRESSED (short)</td>
<td>VST VE A SP</td>
<td>IC R* ST A46F</td>
</tr>
<tr>
<td>VSTM</td>
<td>STORE MATCHED (binary)¹</td>
<td>VST VE A SP</td>
<td>IC R* ST A40E</td>
</tr>
<tr>
<td>VSTMD</td>
<td>STORE MATCHED (long)</td>
<td>VST VE A SP</td>
<td>IC R* ST A41E</td>
</tr>
<tr>
<td>VSTMME</td>
<td>STORE MATCHED (short)</td>
<td>VST VE A SP</td>
<td>IC R* ST A40E</td>
</tr>
<tr>
<td>VSTVMD</td>
<td>STORE VMR</td>
<td>VS VE A</td>
<td>NC R* ST A682</td>
</tr>
<tr>
<td>VSTMVPM</td>
<td>STORE VECTOR PARAMETERS</td>
<td>S VE A SP</td>
<td>N0 ST A6C8</td>
</tr>
<tr>
<td>VSVMM</td>
<td>SET VECTOR MASK MODE</td>
<td>S VE A</td>
<td>N0 A6C6</td>
</tr>
<tr>
<td>VTVM</td>
<td>TEST VMR</td>
<td>RRE C VE</td>
<td>NC A640</td>
</tr>
<tr>
<td>VX</td>
<td>EXCLUSIVE OR</td>
<td>VST VE A SP</td>
<td>IM VB R* A426</td>
</tr>
<tr>
<td>VXEL</td>
<td>EXTRACT ELEMENT (binary)</td>
<td>VR VE SP</td>
<td>N1 R* A629</td>
</tr>
<tr>
<td>VXELD</td>
<td>EXTRACT ELEMENT (long)</td>
<td>VR VE SP</td>
<td>N1 A619</td>
</tr>
<tr>
<td>VXELDE</td>
<td>EXTRACT ELEMENT (short)</td>
<td>VR VE SP</td>
<td>N1 A609</td>
</tr>
<tr>
<td>VXQ</td>
<td>EXCLUSIVE OR</td>
<td>QV VE</td>
<td>IM VB A5A6</td>
</tr>
<tr>
<td>VXRE</td>
<td>EXCLUSIVE OR</td>
<td>VV VE</td>
<td>IM VB A526</td>
</tr>
<tr>
<td>VXSE</td>
<td>EXCLUSIVE OR</td>
<td>QST VE A SP</td>
<td>IM VB R* A4A6</td>
</tr>
<tr>
<td>VXVC</td>
<td>EXTRACT VCT</td>
<td>RRE VE</td>
<td>N0 R* A644</td>
</tr>
<tr>
<td>VXVMD</td>
<td>EXCLUSIVE OR TO VMR</td>
<td>VS VE A</td>
<td>NC R* A686</td>
</tr>
<tr>
<td>VXVMM</td>
<td>EXTRACT VECTOR MASK MODE</td>
<td>RRE VE</td>
<td>N0 R* A646</td>
</tr>
<tr>
<td>VXPSD</td>
<td>ZERO PARTIAL SUMS (long)</td>
<td>VR VE SP</td>
<td>IP VB A618</td>
</tr>
</tbody>
</table>

Figure  B-2  (Part 5 of 5). Instructions Arranged by Mnemonic
<table>
<thead>
<tr>
<th>Op Code</th>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>A400</td>
<td>ADD (short)</td>
<td>VAE</td>
<td>J EU EO LS IM VB R*</td>
</tr>
<tr>
<td>A401</td>
<td>SUBTRACT (short)</td>
<td>VSE</td>
<td>J EU EO LS IM VB R*</td>
</tr>
<tr>
<td>A402</td>
<td>MULTIPLY (short to long)</td>
<td>VME</td>
<td>J U EU EO IM VB R*</td>
</tr>
<tr>
<td>A403</td>
<td>DIVIDE (short)</td>
<td>VDE</td>
<td>J U EU EO FK IM VB R*</td>
</tr>
<tr>
<td>A404</td>
<td>MULTIPLY AND ADD (short to long)</td>
<td>VMAE</td>
<td>J U EU EO LS IM VB R*</td>
</tr>
<tr>
<td>A405</td>
<td>MULTIPLY AND SUBTRACT (short)</td>
<td>VMSE</td>
<td>J U EU EO LS IM VB R*</td>
</tr>
<tr>
<td>A406</td>
<td>MULTIPLY AND ACCUMULATE (long)</td>
<td>VMCE</td>
<td>J U EU EO LS IM VB R*</td>
</tr>
<tr>
<td>A407</td>
<td>ACCUMULATE (short to long)</td>
<td>VACE</td>
<td>J U EU EO LS IM VB R*</td>
</tr>
<tr>
<td>A408</td>
<td>COMPARE (short)</td>
<td>VCE</td>
<td>IC R*</td>
</tr>
<tr>
<td>A409</td>
<td>LOAD (binary)</td>
<td>VL</td>
<td>IC VB R*</td>
</tr>
<tr>
<td>A410</td>
<td>STORE (binary)</td>
<td>VST</td>
<td>IC R* ST</td>
</tr>
<tr>
<td>A411</td>
<td>SUBTRACT (long)</td>
<td>VSD</td>
<td>IC R* ST</td>
</tr>
<tr>
<td>A412</td>
<td>MULTIPLY (long)</td>
<td>VMD</td>
<td>IC R* ST</td>
</tr>
<tr>
<td>A413</td>
<td>DIVIDE (long)</td>
<td>VDD</td>
<td>IC R* ST</td>
</tr>
<tr>
<td>A414</td>
<td>MULTIPLY AND ADD (long)</td>
<td>VMAD</td>
<td>J U EU EO LS IM VB R*</td>
</tr>
<tr>
<td>A415</td>
<td>MULTIPLY AND SUBTRACT (long)</td>
<td>VMSD</td>
<td>J U EU EO LS IM VB R*</td>
</tr>
<tr>
<td>A416</td>
<td>MULTIPLY AND ACCUMULATE (long)</td>
<td>VMCD</td>
<td>J U EU EO LS IM VB R*</td>
</tr>
<tr>
<td>A417</td>
<td>ACCUMULATE (long)</td>
<td>VACD</td>
<td>J U EU EO LS IM VB R*</td>
</tr>
<tr>
<td>A418</td>
<td>COMPARE (long)</td>
<td>VCD</td>
<td>IC R*</td>
</tr>
<tr>
<td>A419</td>
<td>LOAD (long)</td>
<td>VLD</td>
<td>IC VB R*</td>
</tr>
<tr>
<td>A420</td>
<td>LOAD MATCHED (long)</td>
<td>VLMD</td>
<td>IC VB R*</td>
</tr>
<tr>
<td>A421</td>
<td>LOAD EXPANDED (long)</td>
<td>VLYD</td>
<td>IC VB R*</td>
</tr>
<tr>
<td>A422</td>
<td>STORE (long)</td>
<td>VSTD</td>
<td>IC R* ST</td>
</tr>
<tr>
<td>A423</td>
<td>STORE MATCHED (long)</td>
<td>VSTMD</td>
<td>IC R* ST</td>
</tr>
<tr>
<td>A424</td>
<td>STORE COMPRESSED (long)</td>
<td>VSTKD</td>
<td>IC R* ST</td>
</tr>
<tr>
<td>A425</td>
<td>ADD (binary)</td>
<td>VA</td>
<td>J IF IM VB R*</td>
</tr>
<tr>
<td>A426</td>
<td>SUBTRACT (binary)</td>
<td>VS</td>
<td>J IF IM VB R*</td>
</tr>
<tr>
<td>A427</td>
<td>MULTIPLY (binary)</td>
<td>VM</td>
<td>IM VB R*</td>
</tr>
<tr>
<td>A428</td>
<td>AND</td>
<td>VN</td>
<td>IM VB R*</td>
</tr>
<tr>
<td>Op Code</td>
<td>Name</td>
<td>Mnemonic</td>
<td>Characteristics</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>----------</td>
<td>-----------------</td>
</tr>
<tr>
<td>A425 OR</td>
<td>VO</td>
<td>VST</td>
<td>A SP IM VB R*</td>
</tr>
<tr>
<td>A426 EXCLUSIVE OR</td>
<td>VX</td>
<td>VST</td>
<td>A SP IM VB R*</td>
</tr>
<tr>
<td>A428 COMPARE (binary)</td>
<td>VC</td>
<td>VST</td>
<td>A SP IC R*</td>
</tr>
<tr>
<td>A429 LOAD HALFWORD</td>
<td>VSTH</td>
<td>VST</td>
<td>A SP IC VB R*</td>
</tr>
<tr>
<td>A430 LOAD INTEGER VECTOR</td>
<td>VSTINT</td>
<td>VST</td>
<td>A SP IC VB R*</td>
</tr>
<tr>
<td>A431 STORE HALFWORD</td>
<td>VSTH</td>
<td>VST</td>
<td>A SP IC R*</td>
</tr>
<tr>
<td>A480 ADD (short)</td>
<td>VAES</td>
<td>QST</td>
<td>A SP J EU EO LS IM VB R*</td>
</tr>
<tr>
<td>A481 SUBTRACT (short)</td>
<td>VSES</td>
<td>QST</td>
<td>A SP J EU EO LS IM VB R*</td>
</tr>
<tr>
<td>A482 COMPARE (short)</td>
<td>VMES</td>
<td>QST</td>
<td>A SP J EU EO IM VB R*</td>
</tr>
<tr>
<td>A432 LOAD HALFWORD</td>
<td>VSTH</td>
<td>VST</td>
<td>A SP IC R*</td>
</tr>
<tr>
<td>A433 COMPARE (short)</td>
<td>VSTH</td>
<td>VST</td>
<td>A SP IC R*</td>
</tr>
<tr>
<td>A484 MULTIPY (short)</td>
<td>VMAES</td>
<td>QST</td>
<td>A SP J EU EO LS IM VB R*</td>
</tr>
<tr>
<td>A485 MULTIPY AND ADD (short)</td>
<td>VMES</td>
<td>QST</td>
<td>A SP J EU EO LS IM VB R*</td>
</tr>
<tr>
<td>A486 LOAD (short)</td>
<td>VSTH</td>
<td>VST</td>
<td>A SP IC VB R*</td>
</tr>
<tr>
<td>A434 LOAD HALFWORD</td>
<td>VSTH</td>
<td>VST</td>
<td>A SP IC VB R*</td>
</tr>
<tr>
<td>A435 COMPARE (short)</td>
<td>VSTH</td>
<td>VST</td>
<td>A SP IC VB R*</td>
</tr>
<tr>
<td>A500 ADD (short)</td>
<td>VAER</td>
<td>VVE</td>
<td>J EU EO IM VB R*</td>
</tr>
<tr>
<td>A501 SUBTRACT (short)</td>
<td>VSER</td>
<td>VVE</td>
<td>J EU EO IM VB R*</td>
</tr>
<tr>
<td>A502 MULTIPY (short to long)</td>
<td>VMER</td>
<td>VVE</td>
<td>J EU EO IM VB R*</td>
</tr>
<tr>
<td>A503 COMPARE (short)</td>
<td>VACER</td>
<td>VVE</td>
<td>J EU EO IM VB R*</td>
</tr>
<tr>
<td>A504 MULTIPY (long)</td>
<td>VMDS</td>
<td>QST</td>
<td>A SP J EU EO IM VB R*</td>
</tr>
<tr>
<td>A505 DIVIDE (long)</td>
<td>VDDS</td>
<td>QST</td>
<td>A SP J EU EO FK IM VB R*</td>
</tr>
<tr>
<td>A506 MULTIPY AND ADD (long)</td>
<td>VMADS</td>
<td>QST</td>
<td>A SP J EU EO IM VB R*</td>
</tr>
<tr>
<td>A507 MULTIPY AND SUBTRACT (long)</td>
<td>VMDS</td>
<td>QST</td>
<td>A SP J EU EO IM VB R*</td>
</tr>
<tr>
<td>A508 COMPARE (long)</td>
<td>VCDOS</td>
<td>QST</td>
<td>A SP IC R*</td>
</tr>
<tr>
<td>A509 ADD (binary)</td>
<td>VAS</td>
<td>QST</td>
<td>A SP J IF IM VB R*</td>
</tr>
<tr>
<td>A510 SUBTRACT (binary)</td>
<td>VSS</td>
<td>QST</td>
<td>A SP J IF IM VB R*</td>
</tr>
<tr>
<td>A511 MULTIPY (binary)</td>
<td>VMS</td>
<td>QST</td>
<td>A SP IM VB R*</td>
</tr>
<tr>
<td>A512 AND</td>
<td>VNS</td>
<td>QST</td>
<td>A SP IM VB R*</td>
</tr>
<tr>
<td>A513 EXCLUSIVE OR</td>
<td>VOS</td>
<td>QST</td>
<td>A SP IM VB R*</td>
</tr>
<tr>
<td>A514 MULTIPY (binary)</td>
<td>VMOS</td>
<td>QST</td>
<td>A SP IM VB R*</td>
</tr>
<tr>
<td>A515 ADD (binary)</td>
<td>VAS</td>
<td>QST</td>
<td>A SP J IF IM VB R*</td>
</tr>
<tr>
<td>A516 SUBTRACT (binary)</td>
<td>VSS</td>
<td>QST</td>
<td>A SP J IF IM VB R*</td>
</tr>
<tr>
<td>A517 MULTIPY (binary)</td>
<td>VMS</td>
<td>QST</td>
<td>A SP IM VB R*</td>
</tr>
<tr>
<td>A518 AND</td>
<td>VNS</td>
<td>QST</td>
<td>A SP IM VB R*</td>
</tr>
<tr>
<td>A519 EXCLUSIVE OR</td>
<td>VOS</td>
<td>QST</td>
<td>A SP IM VB R*</td>
</tr>
<tr>
<td>A520 ADD (binary)</td>
<td>VAS</td>
<td>QST</td>
<td>A SP J IF IM VB R*</td>
</tr>
<tr>
<td>A521 SUBTRACT (binary)</td>
<td>VSS</td>
<td>QST</td>
<td>A SP J IF IM VB R*</td>
</tr>
<tr>
<td>A522 MULTIPY (binary)</td>
<td>VMS</td>
<td>QST</td>
<td>A SP IM VB R*</td>
</tr>
<tr>
<td>A523 AND</td>
<td>VNS</td>
<td>QST</td>
<td>A SP IM VB R*</td>
</tr>
<tr>
<td>A524 EXCLUSIVE OR</td>
<td>VOS</td>
<td>QST</td>
<td>A SP IM VB R*</td>
</tr>
<tr>
<td>A525 ADD (binary)</td>
<td>VAS</td>
<td>QST</td>
<td>A SP J IF IM VB R*</td>
</tr>
<tr>
<td>A526 SUBTRACT (binary)</td>
<td>VSS</td>
<td>QST</td>
<td>A SP J IF IM VB R*</td>
</tr>
<tr>
<td>A527 MULTIPY (binary)</td>
<td>VMS</td>
<td>QST</td>
<td>A SP IM VB R*</td>
</tr>
<tr>
<td>A528 AND</td>
<td>VNS</td>
<td>QST</td>
<td>A SP IM VB R*</td>
</tr>
<tr>
<td>A529 EXCLUSIVE OR</td>
<td>VOS</td>
<td>QST</td>
<td>A SP IM VB R*</td>
</tr>
<tr>
<td>A530 ADD (binary)</td>
<td>VAS</td>
<td>QST</td>
<td>A SP J IF IM VB R*</td>
</tr>
<tr>
<td>A531 SUBTRACT (binary)</td>
<td>VSS</td>
<td>QST</td>
<td>A SP J IF IM VB R*</td>
</tr>
<tr>
<td>A532 MULTIPY (binary)</td>
<td>VMS</td>
<td>QST</td>
<td>A SP IM VB R*</td>
</tr>
<tr>
<td>A533 AND</td>
<td>VNS</td>
<td>QST</td>
<td>A SP IM VB R*</td>
</tr>
</tbody>
</table>

Figure B-3 (Part 2 of 5). Instructions Arranged by Op Code
<table>
<thead>
<tr>
<th>Op Code</th>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>A510</td>
<td>ADD (long)</td>
<td>VADR</td>
<td>SP J EU EO LS IM VB</td>
</tr>
<tr>
<td>A511</td>
<td>SUBTRACT (long)</td>
<td>VSDR</td>
<td>SP J EU EO LS IM VB</td>
</tr>
<tr>
<td>A512</td>
<td>MULTIPLY (long)</td>
<td>VMDR</td>
<td>SP J U EU EO IM VB</td>
</tr>
<tr>
<td>A513</td>
<td>DIVIDE (long)</td>
<td>VDDR</td>
<td>SP J U EU EO FK IM VB</td>
</tr>
<tr>
<td>A516</td>
<td>MULTIPLY AND ACCUMULATE (long)</td>
<td>VMCDR</td>
<td>SP J U EU EO LS IM VB</td>
</tr>
<tr>
<td>A517</td>
<td>ACCUMULATE (long)</td>
<td>VACDR</td>
<td>SP J EU EO LS IM VB</td>
</tr>
<tr>
<td>A518</td>
<td>COMPARE (long)</td>
<td>VCDR</td>
<td>IC</td>
</tr>
<tr>
<td>A519</td>
<td>LOAD (long)</td>
<td>VLDR</td>
<td>IC VB</td>
</tr>
<tr>
<td>A51A</td>
<td>LOAD MATCHED (long)</td>
<td>VLMDR</td>
<td>IC VB</td>
</tr>
<tr>
<td>A51B</td>
<td>LOAD ZERO (long)</td>
<td>VLZDR</td>
<td>IC VB</td>
</tr>
<tr>
<td>A520</td>
<td>ADD (binary)</td>
<td>VAR</td>
<td>J IF IM VB</td>
</tr>
<tr>
<td>A521</td>
<td>SUBTRACT (binary)</td>
<td>VSR</td>
<td>J IF IM VB</td>
</tr>
<tr>
<td>A522</td>
<td>MULTIPLY (binary)</td>
<td>VMR</td>
<td>SP IM VB</td>
</tr>
<tr>
<td>A524</td>
<td>AND</td>
<td>VNR</td>
<td>IM VB</td>
</tr>
<tr>
<td>A525</td>
<td>OR</td>
<td>VOR</td>
<td>IM VB</td>
</tr>
<tr>
<td>A526</td>
<td>EXCLUSIVE OR</td>
<td>VXR</td>
<td>IM VB</td>
</tr>
<tr>
<td>A528</td>
<td>COMPARE (binary)</td>
<td>VCR</td>
<td>IC</td>
</tr>
<tr>
<td>A540</td>
<td>LOAD POSITIVE (short)</td>
<td>VLPER</td>
<td>IM VB</td>
</tr>
<tr>
<td>A541</td>
<td>LOAD NEGATIVE (short)</td>
<td>VLNER</td>
<td>IM VB</td>
</tr>
<tr>
<td>A542</td>
<td>LOAD COMPLEMENT (short)</td>
<td>VLCER</td>
<td>IM VB</td>
</tr>
<tr>
<td>A550</td>
<td>LOAD POSITIVE (long)</td>
<td>VLPDR</td>
<td>IM VB</td>
</tr>
<tr>
<td>A551</td>
<td>LOAD NEGATIVE (long)</td>
<td>VLNRD</td>
<td>IM VB</td>
</tr>
<tr>
<td>A552</td>
<td>LOAD COMPLEMENT (long)</td>
<td>VLDOR</td>
<td>IM VB</td>
</tr>
<tr>
<td>A560</td>
<td>LOAD POSITIVE (binary)</td>
<td>VLPB</td>
<td>IM VB</td>
</tr>
<tr>
<td>A561</td>
<td>LOAD NEGATIVE (binary)</td>
<td>VLRN</td>
<td>IM VB</td>
</tr>
<tr>
<td>A562</td>
<td>LOAD COMPLEMENT (binary)</td>
<td>VLCB</td>
<td>J IF IM VB</td>
</tr>
<tr>
<td>A580</td>
<td>ADD (short)</td>
<td>VAEQ</td>
<td>J EU EO LS IM VB</td>
</tr>
<tr>
<td>A581</td>
<td>SUBTRACT (short)</td>
<td>VSEQ</td>
<td>J EU EO LS IM VB</td>
</tr>
<tr>
<td>A582</td>
<td>MULTIPLY (short to long)</td>
<td>VMEQ</td>
<td>J U EU EO IM VB</td>
</tr>
<tr>
<td>A583</td>
<td>DIVIDE (short)</td>
<td>VDEQ</td>
<td>J U EU EO FK IM VB</td>
</tr>
<tr>
<td>A584</td>
<td>MULTIPLY AND ADD (short)</td>
<td>VMMAEQ</td>
<td>J U EU EO LS IM VB</td>
</tr>
<tr>
<td>A585</td>
<td>MULTIPLY AND SUBTRACT (to long)</td>
<td>VMSEQ</td>
<td>J U EU EO LS IM VB</td>
</tr>
<tr>
<td>A588</td>
<td>COMPARE (short)</td>
<td>VCEQ</td>
<td>IC</td>
</tr>
<tr>
<td>A589</td>
<td>LOAD (short)</td>
<td>VLEQ</td>
<td>IC VB</td>
</tr>
<tr>
<td>A58A</td>
<td>LOAD MATCHED (short)</td>
<td>VLMSEQ</td>
<td>IC VB</td>
</tr>
<tr>
<td>A590</td>
<td>ADD (long)</td>
<td>VADQ</td>
<td>J EU EO LS IM VB</td>
</tr>
<tr>
<td>A591</td>
<td>SUBTRACT (long)</td>
<td>VSDQ</td>
<td>J EU EO LS IM VB</td>
</tr>
<tr>
<td>A592</td>
<td>MULTIPLY (long)</td>
<td>VMDQ</td>
<td>J U EU EO IM VB</td>
</tr>
<tr>
<td>A593</td>
<td>DIVIDE (long)</td>
<td>VDDQ</td>
<td>J U EU EO FK IM VB</td>
</tr>
<tr>
<td>A594</td>
<td>MULTIPLY AND ADD (long)</td>
<td>VMADQ</td>
<td>J U EU EO LS IM VB</td>
</tr>
</tbody>
</table>

Figure B-3 (Part 3 of 5). Instructions Arranged by Op Code
<table>
<thead>
<tr>
<th>Op Code</th>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>A595</td>
<td>MULTIPLY AND SUBTRACT (long)</td>
<td>VMDOQ QV VE SP</td>
<td>J U EU EO LS IM VB</td>
</tr>
<tr>
<td>A598</td>
<td>COMPARE (long)</td>
<td>VCDQ QV VE SP</td>
<td>IC</td>
</tr>
<tr>
<td>A599</td>
<td>LOAD (long)</td>
<td>VLDQ QV VE SP</td>
<td>IC VB</td>
</tr>
<tr>
<td>A59A</td>
<td>LOAD MATCHED (long)</td>
<td>VLMDOQ QV VE SP</td>
<td>IC VB</td>
</tr>
<tr>
<td>A5A0</td>
<td>ADD (binary)</td>
<td>VAQ QV VE</td>
<td>J IF IM VB</td>
</tr>
<tr>
<td>A5A1</td>
<td>SUBTRACT (binary)</td>
<td>VSQ QV VE J IF</td>
<td>IM VB</td>
</tr>
<tr>
<td>A5A2</td>
<td>MULTIPLY (binary)</td>
<td>VMQ QV VE</td>
<td>IM VB</td>
</tr>
<tr>
<td>A5A4</td>
<td>AND</td>
<td>VNQ QV VE</td>
<td>IM VB</td>
</tr>
<tr>
<td>A5A5</td>
<td>OR</td>
<td>VOQ QV VE</td>
<td>IM VB</td>
</tr>
<tr>
<td>A5A6</td>
<td>EXCLUSIVE OR</td>
<td>VVOQ QV VE</td>
<td>IM VB</td>
</tr>
<tr>
<td>A5A8</td>
<td>COMPARE (binary)</td>
<td>VCQ QV VE</td>
<td>IC</td>
</tr>
<tr>
<td>A5A9</td>
<td>LOAD (binary)</td>
<td>VLQ QV VE</td>
<td>IC VB</td>
</tr>
<tr>
<td>A5AA</td>
<td>LOAD MATCHED (binary)</td>
<td>VLMQ QV VE</td>
<td>IC VB</td>
</tr>
<tr>
<td>A600</td>
<td>MAXIMUM SIGNED (short)</td>
<td>VMXSE VR VE SP</td>
<td>IM R*</td>
</tr>
<tr>
<td>A601</td>
<td>MINIMUM SIGNED (short)</td>
<td>VMSN VR VE SP</td>
<td>IM R*</td>
</tr>
<tr>
<td>A602</td>
<td>MAXIMUM ABSOLUTE (short)</td>
<td>VMXAE VR VE SP</td>
<td>IM R*</td>
</tr>
<tr>
<td>A608</td>
<td>LOAD ELEMENT (short)</td>
<td>VLELE VR VE SP</td>
<td>N1 VB</td>
</tr>
<tr>
<td>A609</td>
<td>EXTRACT ELEMENT (short)</td>
<td>VXLEVR VE SP</td>
<td>N1</td>
</tr>
<tr>
<td>A610</td>
<td>MAXIMUM SIGNED (long)</td>
<td>VMXSD VR VE SP</td>
<td>IM R*</td>
</tr>
<tr>
<td>A611</td>
<td>MINIMUM SIGNED (long)</td>
<td>VMNSD VR VE SP</td>
<td>IM R*</td>
</tr>
<tr>
<td>A612</td>
<td>MAXIMUM ABSOLUTE (long)</td>
<td>VMXAD VR VE SP</td>
<td>IM R*</td>
</tr>
<tr>
<td>A618</td>
<td>LOAD ELEMENT (long)</td>
<td>VLELD VR VE SP</td>
<td>N1 VB</td>
</tr>
<tr>
<td>A619</td>
<td>EXTRACT ELEMENT (long)</td>
<td>VXLELR VE SP</td>
<td>N1</td>
</tr>
<tr>
<td>A61A</td>
<td>SUM PARTIAL SUMS (long)</td>
<td>VSPSD VR VE SP J EU EO LS IP</td>
<td></td>
</tr>
<tr>
<td>A61B</td>
<td>ZERO PARTIAL SUMS (long)</td>
<td>VZPSD VR VE SP</td>
<td>IP VB</td>
</tr>
<tr>
<td>A628</td>
<td>LOAD ELEMENT (binary)</td>
<td>VLELVR VE SP</td>
<td>N1 VB</td>
</tr>
<tr>
<td>A629</td>
<td>EXTRACT ELEMENT (binary)</td>
<td>VXELVR VE SP</td>
<td>N1 R*</td>
</tr>
<tr>
<td>A640</td>
<td>TEST VMR</td>
<td>VTVM RRE C VE</td>
<td>NC</td>
</tr>
<tr>
<td>A641</td>
<td>COMPLEMENT VMR</td>
<td>VCVM RRE C VE</td>
<td>NC</td>
</tr>
<tr>
<td>A642</td>
<td>COUNT LEFT ZEROS IN VMR</td>
<td>VCVM RRE C VE</td>
<td>NC R*</td>
</tr>
<tr>
<td>A643</td>
<td>COUNT ONES IN VMR</td>
<td>VCOVM RRE C VE</td>
<td>NC R*</td>
</tr>
<tr>
<td>A644</td>
<td>EXTRACT VCT</td>
<td>VXXC RRE VE</td>
<td>NO R*</td>
</tr>
<tr>
<td>A645</td>
<td>LOAD VCT AND UPDATE</td>
<td>VLVCU RRE C VE</td>
<td>NO R*</td>
</tr>
<tr>
<td>A646</td>
<td>EXTRACT VECTOR MASK MODE</td>
<td>VXVMR RRE VE</td>
<td>NO R*</td>
</tr>
<tr>
<td>A648</td>
<td>RESTORE VR</td>
<td>VRRS RRE C VE A SP 2</td>
<td>IZ VU R*</td>
</tr>
<tr>
<td>A649</td>
<td>SAVE CHANGED VR</td>
<td>VRSVC RRE C VE A SP</td>
<td>IZ VH R* ST</td>
</tr>
<tr>
<td>A64A</td>
<td>SAVE VR</td>
<td>VRSV RRE C VE A SP</td>
<td>IZ R* ST</td>
</tr>
<tr>
<td>A689</td>
<td>LOAD VMR</td>
<td>VLVM VS VE A</td>
<td>NC R*</td>
</tr>
<tr>
<td>A681</td>
<td>LOAD VMR COMPLEMENT</td>
<td>VLVCNM VS VE A</td>
<td>NC R*</td>
</tr>
<tr>
<td>A682</td>
<td>STORE VMR</td>
<td>VSTVM VS VE A</td>
<td>NC R* ST</td>
</tr>
</tbody>
</table>

Figure B-3 (Part 4 of 5). Instructions Arranged by Op Code
<table>
<thead>
<tr>
<th>Op Code</th>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>A684</td>
<td>AND TO VMR</td>
<td>VNVM</td>
<td>VS VE A NC NC R*</td>
</tr>
<tr>
<td>A685</td>
<td>OR TO VMR</td>
<td>VOVM</td>
<td>VS VE A NC NC R*</td>
</tr>
<tr>
<td>A686</td>
<td>EXCLUSIVE OR TO VMR</td>
<td>VXVM</td>
<td>VS VE A NC NC R*</td>
</tr>
<tr>
<td>A6C0</td>
<td>SAVE VSR</td>
<td>VSRSV</td>
<td>S VE A SP 2 NO ST</td>
</tr>
<tr>
<td>A6C1</td>
<td>SAVE VMR</td>
<td>VMRSV</td>
<td>S VE A NZ ST</td>
</tr>
<tr>
<td>A6C2</td>
<td>RESTORE VSR</td>
<td>VSRRS</td>
<td>S VE A SP 2 IZ VB</td>
</tr>
<tr>
<td>A6C3</td>
<td>RESTORE VMR</td>
<td>VMRRS</td>
<td>S VE A NZ</td>
</tr>
<tr>
<td>A6C4</td>
<td>LOAD VCT FROM ADDRESS</td>
<td>VLYCA</td>
<td>S C VE NO</td>
</tr>
<tr>
<td>A6C5</td>
<td>CLEAR VR</td>
<td>VRCL</td>
<td>S VE IZ VB</td>
</tr>
<tr>
<td>A6C6</td>
<td>SET VECTOR MASK MODE</td>
<td>VSVM</td>
<td>S VE NO</td>
</tr>
<tr>
<td>A6C8</td>
<td>STORE VECTOR PARAMETERS</td>
<td>VSTVP</td>
<td>S VE A SP P NO ST</td>
</tr>
<tr>
<td>A6CA</td>
<td>SAVE VAC</td>
<td>VACSV</td>
<td>S VE A SP P NO ST</td>
</tr>
<tr>
<td>A6CB</td>
<td>RESTORE VAC</td>
<td>VACRS</td>
<td>S VE A SP P NO</td>
</tr>
<tr>
<td>E400</td>
<td>LOAD INDIRECT (binary)1</td>
<td>VLI</td>
<td>RSE VE A SP IC VB</td>
</tr>
<tr>
<td>E400</td>
<td>LOAD INDIRECT (short)</td>
<td>VLIIE</td>
<td>RSE VE A SP IC VB</td>
</tr>
<tr>
<td>E401</td>
<td>STORE INDIRECT (binary)1</td>
<td>VSTI</td>
<td>RSE VE A SP IC ST</td>
</tr>
<tr>
<td>E401</td>
<td>STORE INDIRECT (short)</td>
<td>VSTIE</td>
<td>RSE VE A SP IC ST</td>
</tr>
<tr>
<td>E410</td>
<td>LOAD INDIRECT (long)</td>
<td>VLDI</td>
<td>RSE VE A SP IC VB</td>
</tr>
<tr>
<td>E411</td>
<td>STORE INDIRECT (long)</td>
<td>VSTID</td>
<td>RSE VE A SP IC ST</td>
</tr>
<tr>
<td>E424</td>
<td>SHIFT RIGHT SINGLE LOGICAL</td>
<td>VSRL</td>
<td>RSE VE IM VB</td>
</tr>
<tr>
<td>E425</td>
<td>SHIFT LEFT SINGLE LOGICAL</td>
<td>VSLL</td>
<td>RSE VE IM VB</td>
</tr>
<tr>
<td>E428</td>
<td>LOAD BIT INDEX</td>
<td>VLBIX</td>
<td>RSE C VE A SP IG VB R*</td>
</tr>
</tbody>
</table>

Figure B-3 (Part 5 of 5). Instructions Arranged by Op Code
## Appendix C. Condition-Code Settings

This appendix lists the condition-code setting for vector instructions which set the condition code.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition Code</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT LEFT ZEROS IN VMR</td>
<td>All zeros</td>
<td>Zeros and ones</td>
<td>—</td>
<td>All ones</td>
<td>—</td>
</tr>
<tr>
<td>COUNT ONES IN VMR</td>
<td>All zeros</td>
<td>Zeros and ones</td>
<td>—</td>
<td>All ones</td>
<td>—</td>
</tr>
<tr>
<td>LOAD BIT INDEX</td>
<td>VCT = 0, Bit count = 0</td>
<td>VCT = 0, Bit count &lt; 0</td>
<td>—</td>
<td>VCT = sec.size, Bit count &gt; 0</td>
<td>—</td>
</tr>
<tr>
<td>LOAD VCT AND UPDATE</td>
<td>VCT = 0, New length = 0</td>
<td>VCT = 0, New length &lt; 0</td>
<td>VCT = sec.size, New length &gt; 0</td>
<td>VCT &gt; 0, New length = 0</td>
<td>—</td>
</tr>
<tr>
<td>LOAD VCT FROM ADDRESS</td>
<td>VCT = 0, Address = 0</td>
<td>VCT = 0, Address &lt; 0</td>
<td>VCT = sec.size, Address &gt; section size</td>
<td>VCT &gt; 0, Address ≤ section size</td>
<td>—</td>
</tr>
<tr>
<td>RESTORE VR</td>
<td>VR pair 14-15 not loaded</td>
<td>Other VR pair not loaded</td>
<td>VR pair 14-15 loaded</td>
<td>Other VR pair loaded</td>
<td>—</td>
</tr>
<tr>
<td>SAVE CHANGED VR</td>
<td>VR pair 14-15 not loaded</td>
<td>Other VR pair not loaded</td>
<td>VR pair 14-15 stored</td>
<td>Other VR pair stored</td>
<td>—</td>
</tr>
<tr>
<td>SAVE VR</td>
<td>VR pair 14-15 not loaded</td>
<td>Other VR pair not loaded</td>
<td>VR pair 14-15 stored</td>
<td>Other VR pair stored</td>
<td>—</td>
</tr>
<tr>
<td>TEST VMR</td>
<td>All zeros</td>
<td>Zeros and ones</td>
<td>—</td>
<td>All ones</td>
<td>—</td>
</tr>
</tbody>
</table>

Figure C-1. Summary of Condition-Code Settings
## Index

### A
- access exceptions for vector operands 2-24
- access of vectors in storage 2-8
- ACCUMULATE (VACD, VACDR, VACE, VACER) vector instructions 3-2
- active bits and elements 2-3
- activity count for vectors 2-5
- ADD (VA, VAD, VADQ, VADR, VADS, VAQ, VAR, VAR, VAS) vector instructions 3-3
  - examples A-1
- address generation 2-17
  - for LOAD INTEGER VECTOR 3-16
  - for LOAD/STORE INDIRECT 3-16
- address size 2-6
- address updating 2-8
  - in sectioning 2-11
- addressing mode (in 370-XA mode) 2-6
- alignment on storage boundary 2-8
- AND (VN) vector instructions 3-4
- AND TO VMR (VNVM) vector instruction 3-5
- architectural mode 2-6
- arithmetic (conditional) 2-11
  - examples A-4
- arithmetic exceptions 2-21
- arithmetic partial-completion bit 2-21
- arithmetic vectors 2-8
- availability of vector facility 2-6,2-26
  - effect of machine check on 2-30

### B
- binary integers 2-7
- bit count 3-11
- bit index 3-11
  - relation of to element number 2-10
- bit vector 2-10
- boundary alignment 2-8

### C
- change bits 2-5
  - in saving and restoring 2-27
- classes of vector instructions 2-12
- CLEAR VR (VRCL) vector instruction 3-5
- clearing of vector registers 2-28
- COMPARE (VC, VCD, VCDQ, VCDR, VCDS, VCE, VCEQ, VCEQ, VCEQ, VCQ, VCR, VCS) vector instructions 3-5
  - examples A-3,A-6
- compatibility of vector programs 1-1
- COMPLEMENT VMR (VCVM) vector instruction 3-7
- completion of unit of operation 2-21
- conceptual sequence of vector operations 2-19,2-29
- condition code
  - setting of 2-20
  - summary C-1
- use of in sectioning 2-11
- conditional vector arithmetic 2-11
  - examples A-4,A-6
- configuration of vector facility 2-6
- contiguous vectors 2-8
- access exceptions for 2-24
  - examples A-1
- control bit in control register 0 2-6
- effect of on machine check 2-29
- conversion
  - of bits to element numbers 2-10
  - of floating-point vectors A-6
- count
  - bit 3-11
  - net 2-15
  - vector 2-3
  - vector-activity 2-5
- COUNT LEFT ZEROS IN VMR (VCZVM) vector instruction 3-7
- COUNT ONES IN VMR (VCOVM) vector instruction 3-7

### D
- damage to vector facility 2-30
- data types 2-7
- DIVIDE (VDD, VDDQ, VDDR, VDDS, VDE, VDEQ, VDER, VDES) vector instructions 3-8
  - example A-4

### E
- element
  - indirect selection of 2-9
  - vector 2-2
- element number 2-2
  - relation of to bit index 2-10
- exception-extension code 2-21
- exceptions
  - access 2-24
  - arithmetic 2-21
  - avoidance of 2-4
  - exponent-overflow 2-21,2-25
  - exponent-underflow 2-21,2-25
  - fixed-point-overflow 2-7,2-21
  - floating-point-divide 2-21,2-25
  - operation 2-6
  - significance 2-21
  - specification 2-25
  - unnormalized-operand 2-21,2-26
  - vector-operation 2-6,2-26
- EXCLUSIVE OR (VX, VXQ, VXR, VXS) vector instructions 3-9
- EXCLUSIVE OR TO VMR (VXVM) vector instruction 3-9
- exponent-overflow exception 2-21,2-25
- exponent-underflow exception 2-21,2-25
extension code for exceptions 2-21
EXTRACT ELEMENT (VXEL, VXELD, VXEL, VXELX) vector instructions 3-9
EXTRACT VCT (VXVC) vector instruction 3-10
EXTRACT VECTOR MASK MODE (VXVMM) vector instruction 3-10

F

failure of vector facility 2-30
fields in vector-instruction formats 2-13
fixed-point-overflow exception 2-7,2-21
floating-point conversion (examples) A-6
floating-point-divide exception 2-21,2-25
floating-point register (in vector operations) 2-10
formats of vector instructions 2-13

G

general register (in vector operations) 2-10
avoiding GR0 for addresses 2-15

I

IC (vector-instruction class) 2-15
IG (vector-instruction class) 2-12
ILC (instruction-length code) 2-23
IM (vector-instruction class) 2-15
in-use bits 2-4
in saving and restoring 2-27
index
bit 3-11
vector interruption (see vector interruption index)
indirect element selection 2-9
load instruction for 3-16
store instruction for 3-34
inhibition of unit of operation 2-22
initialization 2-28
inner product (example) A-3
instruction-length code (ILC) 2-23
instructions (see vector-facility instructions)
interruptible vector instructions 2-19
interrupt
conditions for 2-24
effect of 2-23
of vector instructions 2-19
priority of 2-26
interrupt index (see vector interruption index)
invalid vector-register numbers 2-8
IP (vector-instruction class) 2-12
IZ (vector-instruction class) 2-12

L

length of vectors (see vector count)
LOAD (VL, VLD, VLQ, VLD, VLE, VLEQ, VLDR, VLQ, VLR) vector instructions 3-10
LOAD BIT INDEX (VLBIX) vector instruction 3-11

example A-5
LOAD COMPLEMENT (VLCDR, VLCER, VLCR) vector instructions 3-14
LOAD ELEMENT (VLEL, VLELD, VLELE) vector instructions 3-14
LOAD EXPANDED (VLY, VLYD, VLYE) vector instructions 3-14
example A-5
LOAD EXPANDED (VLY, VLYD, VLYE) vector instructions 3-14
example A-5
LOAD HALFWORD (VLH) vector instruction 3-15
LOAD INDIRECT (VLI, VLID, VLIE) vector instructions 3-16
example A-5
LOAD INTEGER VECTOR (VLINT) vector instruction 3-16
LOAD MATCHED (VLQ, VLMQ, VLMQ, VLMDR, VLME, VLMEQ, VLMER, VLMQ, VLMR) vector instructions 3-17
LOAD NEGATIVE (VLRD, VLRER, VLRN) vector instructions 3-18
LOAD POSITIVE (VLPDR, VLPER, VLPQ) vector instructions 3-18
LOAD VCT AND UPDATE (VLVCU) vector instruction 3-19
examples A-1
LOAD VCT FROM ADDRESS (VLVCA) vector instruction 3-19
LOAD VMR (VLVM) vector instruction 3-20
LOAD VMR COMPLEMENT (VLVCM) vector instruction 3-20
LOAD ZERO (VLZDR, VLZER, VLZQ) vector instructions 3-20
logical data 2-7
loop for sectioning 2-11

M

machine check 2-29
mask bits
bit vector for 2-10
register for 2-2
mask mode (see vector-mask mode)
MAXIMUM ABSOLUTE (VMXAD, VMXAE) vector instructions 3-21
MAXIMUM SIGNED (VMXSD, VMXSE) vector instructions 3-21
MINIMUM SIGNED (VMNSD, VMNSE) vector instructions 3-21
mode
addressing 2-6
architectural 2-6
vector-mask (see vector-mask mode)
model-dependent vector functions 1-2
MULTIPLY (VM, VMD, VMDQ, VMDR, VMDS, VME, VMEQ, VMER, VMES, VMQ, VMR, VMS) vector instructions 3-22
examples A-2
MULTIPLY AND ACCUMULATE (VMCD, VMCDR, VMCE, VMCM) vector instructions 3-23
example A-3
MULTIPLY AND ADD (VMAD, VMADQ, VMADS, VMAE, VMAEQ, VMAES) vector instructions 3-24

X-2 IBM System/370 Vector Operations
MULTIPLY AND SUBTRACT (VMSD, VMSDQ, VMSDS, VMSE, VMSEQ, VMSES) vector instructions 3-24
multiprocessing considerations 2-2

NC (vector-instruction class) 2-18
net count (of vector elements) 2-15
nullification of unit of operation 2-22
number of vector element 2-2
NZ (vector-instruction class) 2-12
N0 (vector-instruction class) 2-12
N1 (vector-instruction class) 2-12

operand parameters (for interruptible vector instruction) 2-20
operands for vector instructions 2-7
operation exception 2-6
OR (VO, VOQ, VOR, VOS) vector instructions 3-26
OR TO VMR (VOVM) vector instruction 3-26
overflow
fixed-point 2-7,2-21
floating-point 2-21,2-25

parameters
operand (for interruptible vector instruction) 2-20
vector 2-2
partial-sum number 2-2
partial sums
for ACCUMULATE 3-2
for MULTIPLY AND ACCUMULATE 3-23
for SUM PARTIAL SUMS 3-36
for ZERO PARTIAL SUMS 3-37
PER (program-event recording) 2-28
prefetching of instructions 2-29
priority of vector interruptions 2-26
program initialization 2-28
program-interruption conditions 2-24
program switching 2-26
PSW (program-status word) after interruption 2-23

QST instruction format 2-13
QV instruction format 2-13

register
vector-activity count 2-5
vector-mask (see vector-mask register) 2-3
vector-status 2-3
registers
floating-point 2-10

S instruction format 2-13
SAVE CHANGED VR (VRSVC) vector instruction 3-29
SAVE VAC (VACSV) vector instruction 3-30
SAVE VMR (VMRSV) vector instruction 3-30
SAVE VR (VRSV) vector instruction 3-31
SAVE VSR (VSRSV) vector instruction 3-31
saving of registers 2-27
scalar ill
scalar operands and registers 2-10
section size 2-2
sectioning 2-11
examples A-1
sequence of vector operations 2-19,2-29
sequential addressing of vector elements 2-8
SET VECTOR MASK MODE (VSVMM) vector instruction 3-32
examples A-4
SHIFT LEFT SINGLE LOGICAL (VSLL) vector instruction 3-32
SHIFT RIGHT SINGLE LOGICAL (VSR) vector instruction 3-32
signed binary integers 2-7
significance exception 2-21
source of machine check 2-30
specification exception 2-25
storage-operand consistency 2-29
STORE (VST, VSTD, VSTE) vector instructions 3-32
STORE COMPRESSED (VSTK, VSTKD, VSTKE) vector instructions 3-33
elements A-5
example A-5
STORE HALFWORD (VSTH) vector instruction 3-33
STORE INDIRECT (VSTI, VSTID, VSTIE) vector instructions 3-34
example A-5
STORE MATCHED (VSTM, VSTMD, VSTME) vector instructions 3-34
examples A-3
STORE VECTOR PARAMETERS (VSTVP) vector instruction 3-35
STORE VMR (VSTVP) vector instruction 3-35
storing into instruction stream 2-29
stride 2-8
examples A-2
in address generation 2-17
SUBTRACT (VS, VSD, VSDQ, VSDR, VSDS, VSE, VSEQ, VSER, VSES, VSQ, VSR, VSS) vector instructions 3-35
examples A-2
sum of products (example) A-3
SUM PARTIAL SUMS (VSPSD) vector instruction 3-36
example A-3
suppression of unit of operation 2-22

termination 2-21
TEST VMR (VfVM) vector instruction 3-37
three-operand instructions 2-15

units of operation 2-20
unnormalized-operand exception 2-21,2-26
unsigned binary integers 2-7
updating of vector addresses (see address updating)

VA (ADD) vector instruction 3-3
VAC (vector-activity count) 2-5
VACD (ACCUMULATE) vector instruction 3-2
VACDR (ACCUMULATE) vector instruction 3-2
VACE (ACCUMULATE) vector instruction 3-2
VACER (ACCUMULATE) vector instruction 3-2
VACRS (RESTORE VAC) vector instruction 3-26
VACSV (SAVE VAC) vector instruction 3-30
VAD (ADD) vector instruction 3-3
VADQ (ADD) vector instruction 3-3
VADR (ADD) vector instruction 3-3
VADS (ADD) vector instruction 3-3
valid vector-register numbers 2-8
validation of vector-facility registers 2-30
VAQ (ADD) vector instruction 3-3
VAR (ADD) vector instruction 3-3
VAS (ADD) vector instruction 3-3
VC (COMPARE) vector instruction 3-5
VCD (COMPARE) vector instruction 3-5
VCDQ (COMPARE) vector instruction 3-5
VCDR (COMPARE) vector instruction 3-5
VCDS (COMPARE) vector instruction 3-5
VCE (COMPARE) vector instruction 3-5
VEQ (COMPARE) vector instruction 3-5
VCR (COMPARE) vector instruction 3-5
VCES (COMPARE) vector instruction 3-5
VCOVM (COUNT ONES IN VMR) vector instruction 3-7
VQC (COMPARE) vector instruction 3-5
VCR (COMPARE) vector instruction 3-5
VCS (COMPARE) vector instruction 3-5
VCT (vector count) 2-3
VCM (COMPLEMENT VMR) vector instruction 3-7
VCZVM (COUNT LEFT ZEROS IN VMR) vector instruction 3-7
VDD (DIVIDE) vector instruction 3-8
VDDDQ (DIVIDE) vector instruction 3-8
VDDR (DIVIDE) vector instruction 3-8
VDDS (DIVIDE) vector instruction 3-8
VDE (DIVIDE) vector instruction 3-8
VDEQ (DIVIDE) vector instruction 3-8
VDER (DIVIDE) vector instruction 3-8
VDES (DIVIDE) vector instruction 3-8
valid i
of bits 2-10
section size for 2-2
vector-activity count (VAC) 2-5
vector change bits 2-5
for saving and restoring 2-27
vector-control bit 2-6
effect of on machine check 2-29
vector count (VCT) 2-3
vector element 2-2
vector facility 2-2
availability of 2-6,2-26
configuration of 2-6
vector-facility failure 2-30
vector-facility instructions 3-1
classes of 2-12
effect of interruption on 2-23
fields of 2-13
interruptible 2-19
prefetching of 2-29
storing into 2-29
summary of 2-15
three-operand 2-15
units of operation for 2-20
vector-facility registers 2-2
(see also vector-mask register, vector register)
validation of 2-30
vector-activity count 2-5
vector-status register 2-3
vector-facility source (of damage) 2-30
vector in-use bits 2-4
for saving and restoring 2-27
vector interruption index (VIX) 2-4
after interruption 2-24
vector length (see vector count)
vector machine check 2-29
vector-mask mode (VMM) 2-11
bit in vector-status register 2-3
examples of use A-4,A-6
vector-mask register (VMR) 2-2
vector-operation exception 2-6,2-26
vector register (VR) 2-2
valid numbers for 2-8
vector-status register (VSR) 2-3
VIX (see vector interruption index)
VL (LOAD) vector instruction 3-10
VLBIX (LOAD BIT INDEX) vector instruction 3-11
VLCDR (LOAD COMPLEMENT) vector instruction 3-14
VLCER (LOAD COMPLEMENT) vector instruction 3-14
VLCR (LOAD COMPLEMENT) vector instruction 3-14
VLCVM (LOAD VMR COMPLEMENT) vector instruction 3-20
VLD (LOAD) vector instruction 3-10
VLDQ (LOAD) vector instruction 3-10
VLDR (LOAD) vector instruction 3-10
VLE (LOAD) vector instruction 3-10
VLEL (LOAD ELEMENT) vector instruction 3-14
VLELD (LOAD ELEMENT) vector instruction 3-14
VLELE (LOAD ELEMENT) vector instruction 3-14
VLEQ (LOAD) vector instruction 3-10
VLER (LOAD) vector instruction 3-10
VLH (LOAD HALFWORD) vector instruction 3-15
VLID (LOAD INDIRECT) vector instruction 3-16
VLIE (LOAD INDIRECT) vector instruction 3-16
VLINT (LOAD INTEGER VECTOR) vector instruction 3-16
VLM (LOAD MATCHED) vector instruction 3-17
VLMQ (LOAD MATCHED) vector instruction 3-17
VLMDRQ (LOAD MATCHED) vector instruction 3-17
VLMCRQ (LOAD MATCHED) vector instruction 3-17
VLMER (LOAD MATCHED) vector instruction 3-17
VLMDRQ (LOAD MATCHED) vector instruction 3-17
VLMDQ (LOAD MATCHED) vector instruction 3-17
VLNDR (LOAD NEGATIVE) vector instruction 3-18
VLENR (LOAD NEGATIVE) vector instruction 3-18
VLNR (LOAD NEGATIVE) vector instruction 3-18
VLPDR (LOAD POSITIVE) vector instruction 3-18
VLPER (LOAD POSITIVE) vector instruction 3-18
VLPDR (LOAD POSITIVE) vector instruction 3-18
VLER (LOAD) vector instruction 3-10
VLQ (LOAD) vector instruction 3-10
VLQ (LOAD) vector instruction 3-10
VLQCA (LOAD VCT FROM ADDRESS) vector instruction 3-19
VLVCU (LOAD VCT AND UPDATE) vector instruction 3-19
VLM (LOAD VMR) vector instruction 3-20
VLY (LOAD EXPANDED) vector instruction 3-14
VLYD (LOAD EXPANDED) vector instruction 3-14
VLYE (LOAD EXPANDED) vector instruction 3-14
VLZDR (LOAD ZERO) vector instruction 3-20
VLZER (LOAD ZERO) vector instruction 3-20
VLZR (LOAD ZERO) vector instruction 3-20
VM (MULTIPLY) vector instruction 3-22
VMAD (MULTIPLY AND ADD) vector instruction 3-24
VMADC (MULTIPLY AND ADD) vector instruction 3-24
VMAC (MULTIPLY AND ADD) vector instruction 3-24
VMACE (MULTIPLY AND ADD) vector instruction 3-24
VMACD (MULTIPLY AND ADD) vector instruction 3-24
VMADD (MULTIPLY AND ADD) vector instruction 3-24
VMADDQ (MULTIPLY AND ADD) vector instruction 3-24
VMADSD (MULTIPLY AND ADD) vector instruction 3-24
VMAE (MULTIPLY AND ADD) vector instruction 3-24
VMAEQ (MULTIPLY AND ADD) vector instruction 3-24
VMAES (MULTIPLY AND ADD) vector instruction 3-24
VMCD (MULTIPLY AND ACCUMULATE) vector instruction 3-23
VMCDR (MULTIPLY AND ACCUMULATE) vector instruction 3-23
VMCE (MULTIPLY AND ACCUMULATE) vector instruction 3-23
VMCER (MULTIPLY AND ACCUMULATE) vector instruction 3-23
VMD (MULTIPLY) vector instruction 3-22
VMDQ (MULTIPLY) vector instruction 3-22
VMDR (MULTIPLY) vector instruction 3-22
VMDQ (MULTIPLY) vector instruction 3-22
VMDR (MULTIPLY) vector instruction 3-22
VMDS (MULTIPLY) vector instruction 3-22
VME (MULTIPLY) vector instruction 3-22
VMEQ (MULTIPLY) vector instruction 3-22
VMER (MULTIPLY) vector instruction 3-22
VMES (MULTIPLY) vector instruction 3-22
VM (see vector-mask mode)
VMNSD (MINIMUM SIGNED) vector instruction 3-21
VMNSEQ (MINIMUM SIGNED) vector instruction 3-21
VMQ (MULTIPLY) vector instruction 3-22
VMR (see vector-mask register)
VMRRS (RESTORE VMR) vector instruction 3-27
VMRSV (SAVE VMR) vector instruction 3-30
VMS (MULTIPLY) vector instruction 3-22
VMSE (MULTIPLY AND SUBTRACT) vector instruction 3-24
VMSE (MULTIPLY AND SUBTRACT) vector instruction 3-24
VMSE (MULTIPLY AND SUBTRACT) vector instruction 3-24
VMSEQ (MULTIPLY AND SUBTRACT) vector instruction 3-24
VMSES (MULTIPLY AND SUBTRACT) vector instruction 3-24
VMXAD (MAXIMUM ABSOLUTE) vector instruction 3-21
VMXAE (MAXIMUM ABSOLUTE) vector instruction 3-21
VMXSD (MAXIMUM ABSOLUTE) vector instruction 3-21
VMXSE (MAXIMUM SIGNED) vector instruction 3-21
VMXSE (MAXIMUM SIGNED) vector instruction 3-21
VN (AND) vector instruction 3-4
VNV (AND TO VMR) vector instruction 3-5
VO (OR) vector instruction 3-26
VOQ (OR) vector instruction 3-26
VOR (OR) vector instruction 3-26
VOS (OR) vector instruction 3-26
VOVM (OR TO VMR) vector instruction 3-26
VR (see vector register)
VR instruction format 2-13
VRCL (CLEAR VR) vector instruction 3-5
VR (see vector register)
VR instruction format 2-13
VR (see vector register)
VR instruction format 2-13
VSDS (SUBTRACT) vector instruction 3-35
VSE (SUBTRACT) vector instruction 3-35
VSEQ (SUBTRACT) vector instruction 3-35
VSER (SUBTRACT) vector instruction 3-35
VSES (SUBTRACT) vector instruction 3-35
VSL (SHIFT LEFT SINGLE LOGICAL) vector instruction 3-32
VSPPD (SUM PARTIAL SUMS) vector instruction 3-36
VSQ (SUBTRACT) vector instruction 3-35
VSR (SUBTRACT) vector instruction 3-35
VSR (vector-status register) 2-3
VSR (SHIFT RIGHT SINGLE LOGICAL) vector instruction 3-32
VSR (RESTORE VSR) vector instruction 3-28
VSRV (SAVE VSR) vector instruction 3-31
VSS (SUBTRACT) vector instruction 3-35
VSS (vector-section size) (see section size)
VST (STORE) vector instruction 3-32
VSTD (STORE) vector instruction 3-32
VSTE (STORE) vector instruction 3-32
VSTH (STORE HALFWORD) vector instruction 3-33
VSTI (STORE INDIRECT) vector instruction 3-34
VSTID (STORE INDIRECT) vector instruction 3-34
VSTIE (STORE INDIRECT) vector instruction 3-34
VSTK (STORE COMPRESSED) vector instruction 3-33
VSTKD (STORE COMPRESSED) vector instruction 3-33
VSTKE (STORE COMPRESSED) vector instruction 3-33
VSTM (STORE MATCHED) vector instruction 3-34
VSTMD (STORE MATCHED) vector instruction 3-34
VSTME (STORE MATCHED) vector instruction 3-34
VSTVM (STORE VMR) vector instruction 3-35
VSTVP (STORE VECTOR PARAMETERS) vector instruction 3-35
VSVM (SET VECTOR MASK MODE) vector instruction 3-32
VTVM (TEST VMR) vector instruction 3-37
VX (EXCLUSIVE OR) vector instruction 3-9
VXEL (EXTRACT ELEMENT) vector instruction 3-9
VXELD (EXTRACT ELEMENT) vector instruction 3-9
VXELE (EXTRACT ELEMENT) vector instruction 3-9
VXVMM (EXTRACT VECTOR MASK MODE) vector instruction 3-10
VZPSD (ZERO PARTIAL SUMS) vector instruction 3-37
ZERO PARTIAL SUMS (VZPSD) vector instruction 3-37
zero stride 2-8
IBM System/370
Vector Operations

Order No. SA22-7125-2

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Possible topics for comment are:
Clarity  Accuracy  Completeness  Organization  Coding  Retrieval  Legibility

If you wish a reply, give your name, company, mailing address, and date:

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

What is your occupation? ____________________________________________________

Number of latest Newsletter associated with this publication: ____________________

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the front cover or title page.)
Reader's Comment Form

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department E57
P.O. Box 390
Poughkeepsie, New York 12602

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES