# TRANSFORMING NON-CONTIGUOUS INSTRUCTION SPECIFIERS TO CONTIGUOUS INSTRUCTION SPECIFIERS

## BACKGROUND

**[0001]** The invention relates, in general, to emulation within a computing environment, and in particular, to emulation of specifiers within instructions.

**[0002]** Emulation imitates functions on a computer architecture, referred to as a target architecture. The target architecture differs from a computer architecture, referred to as a source architecture, for which the functions were defined. For instance, an instruction written for the z/Architecture provided by International Business Machines Corporation, Armonk, New York, may be translated and represented as one or more instructions of a different architecture, such as PowerPC, also offered by International Business Machines Corporation, or another architecture offered by International Business Machines Corporation or another company. These translated instructions perform the same or a similar function as the instruction being translated.

**[0003]** There are different types of emulation, including interpretation and translation. With interpretation, data representing an instruction is read, and as each instruction is decoded, it is executed. Each instruction is executed each time it is referenced. However, with translation, also referred to as binary translation or recompilation, sequences of instructions are translated from the instruction set of one computer architecture to the instruction set of another computer architecture.

**[0004]** There are multiple types of translation, including static translation and dynamic translation. In static translation, code of an instruction of the one architecture is converted to code that runs on the other architecture without previously executing the code. In contrast, in dynamic translation, at least a section of the code is executed and translated, and the result is placed in a cache for subsequent execution by a processor of the target computer architecture.

## SUMMARY

**[0005]** Shortcomings of the prior art are addressed and advantages are provided through the provision of a computer program product for transforming instruction specifiers of a computing environment. The computer program product includes a computer readable storage medium readable by a processing circuit and storing instructions for execution by the processing circuit for performing a method including: obtaining, by a processor, from a first instruction defined for a first computer architecture, a non-contiguous specifier, the non-contiguous specifier having a first portion and a second portion, wherein the obtaining includes obtaining the first portion from a first field of the instruction and the second portion from a second field of the instruction, the first field separate from the second field; generating a contiguous specifier using the first portion and the second portion, the generating using one or more rules based on the opcode of the first instruction; and using the contiguous specifier to indicate a resource to be used in execution of a second instruction, the second instruction defined for a second computer architecture different from the first computer architecture and emulating a function of the first instruction.

**[0006]** Methods and systems relating to one or more aspects of the present invention are also described and claimed herein. Further, services relating to one or more aspects of the present invention are also described and may be claimed herein.

**[0007]** Additional features and advantages are realized through the techniques of the present invention. Other embodiments and aspects of the invention are described in detail herein and are considered a part of the claimed invention.

## BRIEF DESCRIPTION OF THE DRAWINGS

**[0008]** Embodiments of the present invention will now be described, by way of example only, with reference to the accompanying drawings in which:

FIG. 1 depicts one example of a computing environment to incorporate and use one or more aspects of the present invention;

FIG. 2 depicts further details of the memory of FIG. 1, in accordance with an aspect of the present invention;

FIG. 3 depicts one embodiment of an overview of an emulation process that employs one or more of interpretation and translation;

FIG. 4 depicts one example of logic associated with the interpretation block referenced in FIG. 3;

FIG. 5 depicts one example of logic associated with the translation block referenced in FIG. 3;

FIG. 6 depicts another embodiment of an overview of an emulation process that employs one or more of interpretation and translation modified in accordance with an aspect of the present invention;

FIG. 7A depicts one example of logic associated with the interpretation block referenced in FIG. 6, in accordance with an aspect of the present invention;

FIG. 7B depicts one embodiment of the logic to transform a non-contiguous specifier to a contiguous specifier, in accordance with an aspect of the present invention;

FIG. 8 depicts one example of logic associated with the translation block referenced in FIG. 6, in accordance with an aspect of the present invention;

FIG. 9A depicts one embodiment of transforming a non-contiguous specifier in a Vector Load instruction of one computer architecture to a contiguous specifier in a Load Vector Indexed instruction of another computer architecture, in accordance with an aspect of the present invention;

FIG. 9B depicts another example of the transformation of FIG. 9A, including allocation of a particular register to the contiguous specifier, in accordance with an aspect of the present invention;

FIG. 10 depicts one example of a register file, in accordance with an aspect of the present invention;

FIG. 11 depicts an example of transforming non-contiguous specifiers to contiguous specifiers in allocating to memory during emulation, in accordance with an aspect of the present invention;

FIG. 12 depicts one embodiment of a computer program product incorporating one or more aspects of the present invention;

FIG. 13 depicts one embodiment of a host computer system to incorporate and use one or more aspects of the present invention;

FIG. 14 depicts a further example of a computer system to incorporate and use one or more aspects of the present invention;

FIG. 15 depicts another example of a computer system comprising a computer network to incorporate and use one or more aspects of the present invention;

FIG. 16 depicts one embodiment of various elements of a computer system to incorporate and use one or more aspects of the present invention;

FIG. 17A depicts one embodiment of the execution unit of the computer system of FIG. 16 to incorporate and use one or more aspects of the present invention;

FIG. 17B depicts one embodiment of the branch unit of the computer system of FIG. 16 to incorporate and use one or more aspects of the present invention;

FIG. 17C depicts one embodiment of the load/store unit of the computer system of FIG. 16 to incorporate and use one or more aspects of the present invention; and

FIG. 18 depicts one embodiment of an emulated host computer system to incorporate and use one or more aspects of the present invention.

## DETAILED DESCRIPTION

**[0009]** In accordance with an aspect of the present invention, a technique is provided for facilitating emulation of instructions that include non-contiguous specifiers. A non-contiguous specifier specifies a resource of an instruction, such as a register, using multiple fields of the instruction. For example, multiple fields of the instruction (e.g., two fields) include bits that together designate a particular register to be used by the instruction.

**[0010]** In a particular aspect of the invention, a technique is provided for transforming non-contiguous specifiers of instructions defined in one computer system architecture (e.g., the z/Architecture offered by International Business Machines Corporation) to contiguous specifiers usable by instructions defined in another computer system architecture (e.g., the PowerPC architecture offered by International Business Machines Corporation). The instructions defined in the another computer system architecture emulate the instructions defined for the one computer system architecture.

**[0011]** One embodiment of a computing environment providing emulation is described with reference to FIG. 1. In one example, a computing environment 100 includes, for instance, a native central processing unit 102, a memory 104, and one or more input/output devices and/or interfaces 106 coupled to one another via, for example, one or more buses 108 and/or other connections. As examples, computing environment 100 may include a PowerPC processor, a pSeries server or an xSeries server offered by International Business Machines Corporation, Armonk, New York; an HP Superdome with Intel Itanium II processors offered by Hewlett Packard Co., Palo Alto, California; and/or other machines based on architectures offered by International Business Machines Corporation, Hewlett Packard, Intel, Oracle, or others.

**[0012]** Native central processing unit 102 includes one or more native registers 110, such as one or more general purpose registers and/or one or more special purpose registers used during processing within the environment. These registers include information that represents the state of the environment at any particular point in time.

**[0013]** Moreover, native central processing unit 102 executes instructions and code that are stored in memory 104. In one particular example, the central processing unit executes emulator code 112 stored in memory 104. This code enables the processing environment configured in one architecture to emulate another architecture. For instance, emulator code 112 allows machines based on architectures other than the z/Architecture, such as PowerPC processors, pSeries servers, xSeries servers, HP Superdome servers or others, to emulate the z/Architecture and to execute software and instructions developed based on the z/Architecture.

**[0014]** Further details relating to emulator code 112 are described with reference to FIG. 2. Guest instructions 200 comprise software instructions (e.g., machine instructions) that were developed to be executed in an architecture other than that of native CPU 102. For example, guest instructions 200 may have been designed to execute on a z/Architecture processor, but instead, are being emulated on native CPU 102, which may be, for example, a PowerPC processor or other type of processor. In one example, emulator code 112 includes an instruction fetching unit 202 to obtain one or more guest instructions 200 from memory

104, and to optionally provide local buffering for the instructions obtained. It also includes an instruction translation routine 204 to determine the type of guest instruction that has been obtained and to translate the guest instruction into one or more corresponding native instructions 206. This translation includes, for instance, identifying the function to be performed by the guest instruction (e.g., via the opcode) and choosing the native instruction(s) to perform that function.

**[0015]** Further, emulator 112 includes an emulation control routine 210 to cause the native instructions to be executed. Emulation control routine 210 may cause native CPU 102 to execute a routine of native instructions that emulate one or more previously obtained guest instructions and, at the conclusion of such execution, return control to the instruction fetch routine to emulate the obtaining of the next guest instruction or a group of guest instructions. Execution of the native instructions 206 may include loading data into a register from memory 104; storing data back to memory from a register; or performing some type of arithmetic or logic operation, as determined by the translation routine.

**[0016]** Each routine is, for instance, implemented in software, which is stored in memory and executed by native central processing unit 102. In other examples, one or more of the routines or operations are implemented in firmware, hardware, software or some combination thereof. The registers of the emulated processor may be emulated using registers 110 of the native CPU or by using locations in memory 104. In embodiments, guest instructions 200, native instructions 206 and emulator code 112 may reside in the same memory or may be disbursed among different memory devices.

**[0017]** As used herein, firmware includes, e.g., the microcode, millicode and/or macrocode of the processor. It includes, for instance, the hardware-level instructions and/or data structures used in implementation of higher level machine code. In one embodiment, it includes, for instance, proprietary code that is typically delivered as microcode that includes trusted software or microcode specific to the underlying hardware and controls operating system access to the system hardware.

**[0018]** In one example, a guest instruction 200 that is obtained, translated and executed is one or more of the instructions described herein. The instruction, which is of one architecture (e.g., the z/Architecture) is fetched from memory, translated and represented as a sequence of native instructions 206 of another architecture (e.g., PowerPC, pSeries, xSeries, Intel, etc.). These native instructions are then executed.

**[0019]** Further details regarding emulation are described with reference to FIGs. 3-5. In particular, FIG. 3 depicts one embodiment of an overview of an emulation process that employs one or more of interpretation and translation; FIG. 4 depicts one embodiment of the logic associated with interpretation referenced in FIG. 3 (Technique 2000); and FIG. 5 depicts one embodiment of the logic associated with binary translation referenced in FIG. 3 (Technique 3000). In this particular example, instructions written for the z/Architecture are being translated to PowerPC instructions. However, the same techniques are applicable for emulation from the z/Architecture to other target architectures; from other source architectures to the PowerPC architecture; and/or from other source architectures to other target architectures.

**[0020]** Referring to FIG. 3, during emulation, an instruction, referred to as instruction X, is obtained and interpreted, as described in further detail with reference to FIG. 4, STEP 300. Various statistics relating to the interpreted instruction are updated, STEP 302, and then processing proceeds to the next instruction, which becomes instruction X in the logic, STEP 304. A determination is made as to whether this next instruction has a previous translated entry point, INQUIRY 306. If not, a further determination is made as to whether this next instruction has been seen N (e.g., 15) times, INQUIRY 308. That is, has this instruction been seen often enough in order to optimize execution by, for instance, performing just-in-time (JIT) compilation of the code, which provides an entry point for subsequent use. If this instruction has not been seen N times, such as 15 times, then processing continues with STEP 300. Otherwise, processing continues with forming a group of instructions and translating the group of instructions from one architecture to another architecture, STEP 310. One example of performing this translation is described with reference to FIG. 5. Subsequent to forming and translating the group, the group is executed, STEP 312, and processing continues to STEP 304.

**[0021]** Returning to INQUIRY 306, if there is an existing translated entry point for the instruction, processing continues with executing the group at the entry point, STEP 312.

**[0022]** Further details relating to interpreting an instruction (Technique 2000) are described with reference to FIG. 4. Initially, an instruction at the next program counter (PC) address is read, STEP 400. This instruction is analyzed, and opcode, register and immediate fields are extracted, STEP 402. Then, a branch is performed to the code that emulates behavior corresponding to the extracted opcode, STEP 404. The emulated code is then performed, STEP 406.

**[0023]** Further details regarding translating instructions within a group (Technique 3000) are described with reference to FIG. 5. Initially, an instruction in a pre-defined group of instructions is read, STEP 500. In one example, the group can be formed using a variety of ways. In accordance with one embodiment, a group is formed to encompass a single path of execution along a most likely path. In another embodiment, a group is formed to encompass one of the last previous execution paths, or the current execution path, based on the emulated architecture's state. In another embodiment, all branches are assumed not taken. In yet another embodiment, multiple paths are included in a group, such as all paths starting from the group starting point. In another embodiment, all instructions up to and including the first branch are added to a group (i.e., a group corresponds to a straight line piece of code also commonly known as a "basic block"). In each embodiment, a decision has to be made as to when and where to end a group. In one embodiment, a group is ended after a fixed number of instructions. In another embodiment, the group is ended after a cumulative probability of reaching an instruction is below a given threshold. In some embodiments, a group is stopped immediately when a stopping condition has been reached. In another set of embodiments, a group is stopped only at a well-defined "stopping point", e.g., a defined instruction, a specific group start alignment, or other condition.

**[0024]** Thereafter, the instruction is analyzed, and opcode, register and immediate fields are extracted from the instruction, STEP 502. Next, an internal representation of the extracted information is provided, STEP 504. This internal representation is a format of the

extracted information that is used by the processor (e.g., compiler or translator) to optimize decoding, register allocation, and/or other tasks associated with translating the instruction.

**[0025]** Further, a determination is made as to whether there is another instruction in the group to be translated, INQUIRY 506. If so, then processing continues with STEP 500. Otherwise, processing continues with optimizing the internal representation, STEP 508, allocating one or more registers for the group of instructions, STEP 510, and generating code that emulates the instructions in the group, STEP 512.

**[0026]** While the above interpretation and translation procedures provide for emulation of an instruction defined in one architecture to one or more instructions defined in another architecture, advancements may be made in the emulation of instructions that use non-contiguous specifiers. For instance, in accordance with an aspect of the present invention, improvements in emulation techniques are provided to address the situation in which a register operand of an instruction is designated by multiple fields of the instruction.

**[0027]** One type of instruction that uses non-contiguous specifiers is vector instructions that are part of a vector facility, provided in accordance with an aspect of the present invention. In many vector instructions, the register field does not include all of the bits needed to designate a register to be used by the instruction, but instead, another field is used along with the register field to designate a register. This other field is referred to herein as an RXB field.

**[0028]** The RXB field, also referred to as the register extension bit, is, for instance, a four bit field (bits 0-3) that includes the most significant bit for each of the vector register designated operands of a vector instruction. Bits for register designations not specified by the instruction are to be reserved and set to zero.

**[0029]** In one example, the RXB bits are defined as follows:

**[0030]** 0 - Most significant bit for the first vector register designation of the instruction.

**[0031]** 1 - Most significant bit for the second vector register designation of the instruction, if any.

**[0032]** 2 - Most significant bit for the third vector register designation of the instruction, if any.

**[0033]** 3 - Most significant bit for the fourth vector register designation of the instruction, if any.

**[0034]** Each bit is set to zero or one by, for instance, the assembler depending on the register number. For instance, for registers 0-15, the bit is set to 0; for registers 16-31, the bit is set to 1, etc.

**[0035]** In one embodiment, each RXB bit is an extension bit for a particular location in an instruction that includes one or more vector registers. For instance, in one or more vector instructions, bit 0 of RXB is an extension bit for location 8-11, which is assigned to e.g., $V_1$; bit 1 of RXB is an extension bit for location 12-15, which is assigned to, e.g., $V_2$; and so forth.

**[0036]** In a further embodiment, the RXB field includes additional bits, and more than one bit is used as an extension for each vector or location.

**[0037]** In accordance with an aspect of the present invention, techniques are provided for transforming non-contiguous operand specifiers into contiguous specifiers. Once transformed, the contiguous specifiers are used without any regard to the non-contiguous specifiers.

**[0038]** One embodiment of the logic to emulate instructions that use non-contiguous specifiers is described with reference to FIGs. 6-8. In particular, FIG. 6 depicts an overview of an emulation process including one or more of interpretation and translation of instructions that include non-contiguous specifiers; FIG. 7A depicts one embodiment of interpretation (Technique 6000), including interpretation of non-contiguous specifiers; FIG.

7B depicts one embodiment of transforming a non-contiguous specifier to a contiguous specifier; and FIG. 8 depicts one embodiment of translation (Technique 7000), including translation of non-contiguous specifiers.

**[0039]** Referring initially to FIG. 6, an overview of an emulation process is provided. This overview is similar to the overview provided in FIG. 3, except STEP 600 uses Technique 6000 described with reference to FIG. 7A, instead of Technique 2000 referenced in STEP 300; and STEP 610 uses Technique 7000 described with reference to FIG. 8, instead of Technique 3000 referenced in STEP 310. Since the overview is described above with reference to FIG. 3, it is not repeated here; instead, the discussion proceeds to the logic of FIG. 7A.

**[0040]** Referring to FIG. 7A, STEPS 700, 702, 704 and 706 are similar to STEPs 400, 402, 404 and 406, respectively, of FIG. 4, and therefore, are not described again; however, STEPs 703 and 705 are described. With STEP 703, in accordance with an aspect of the present invention, a contiguous specifier (also referred to herein as a contiguous index) is generated from a non-contiguous specifier. Further details regarding generation of a contiguous specifier from a non-contiguous specifier are described with reference to FIG. 7B.

**[0041]** Referring to FIG. 7B, in one embodiment, initially, the non-contiguous specifier is obtained, STEP 750. This includes, for instance, determining from the opcode that the instruction has a non-contiguous specifier and determining which fields of the instruction are used to designate the non-contiguous specifier. For instance, a portion of the opcode specifies a format of the instruction and this format indicates to the processor that the instruction has at least one non-contiguous specifier and it further specifies the fields used to designate the non-contiguous specifier. These fields are then read to obtain the data (e.g., bits) in those fields. For instance, in many vector instructions, location 8-11 of the instruction (e.g., $V_1$) specifies a plurality of bits (e.g., 4) used to designate a vector register, and an RXB field of the instruction includes one or more additional bits used to designate the particular vector register. These bits are obtained in this step.

**[0042]** Subsequent to obtaining the non-contiguous specifier (e.g., bits from register field $V_1$ and bit(s) from RXB), one or more rules are used to combine the portions of the non-contiguous specifier to create the contiguous specifier, STEP 752. The one or more rules depend, for instance, on the format of the instruction as specified by the opcode of the instruction. In a particular example in which the opcode indicates an RXB field, the one or more rules include using the RXB bit(s) associated with the register operand as the most significant bit(s) for the bits specified in the register field. For instance, the RXB field has, in one embodiment, 4 bits and each bit corresponds to a register operand. For instance, bit 0 corresponds to the first register operand, bit 1 corresponds to the second register operand, and so forth. So, the bit corresponding to the register operand is extracted and used to form the contiguous specifier. For example, if 0010 binary is specified in the first operand register field and 1000 binary is specified in the RXB field, the value of the bit associated with the first operand, bit 0, in this example, is concatenated to 0010. Therefore, the contiguous specifier is 10010 (register 18), in this example.

**[0043]** The generated contiguous specifier is then used as if it was the specifier provided in the instruction, STEP 754.

**[0044]** Thereafter, returning to FIG. 7A, a branch is performed to code that emulates the behavior corresponding to the opcode, STEP 704. Further, the contiguous index is used to manage the homogenized architecture resource without regard to the non-contiguous specifier, STEP 705. That is, the contiguous register specifier is used, as if there was no non-contiguous specifier. Each contiguous specifier indicates a register to be used by the emulation code. Thereafter, the emulation code is performed, STEP 706.

**[0045]** Further details regarding translation including the transforming of non-contiguous specifiers to contiguous specifiers (referred to as Technique 7000) are described with reference to FIG. 8. In one embodiment, STEPS 800, 802, 804, 806, 808, 810 and 812 are similar to STEPS 500, 502, 504, 506, 508, 510, and 512, respectively, of FIG. 5, and therefore, are not described here with reference to FIG. 8. However, in accordance with an aspect of the present invention, further steps are performed in order to transform a non-contiguous specifier of an instruction of a source architecture to a contiguous specifier of an

instruction of a target architecture. The instruction of the target architecture emulates a function of the instruction of the source architecture.

**[0046]** For example, in STEP 803, a contiguous specifier is generated from a non-contiguous specifier. As described above with reference to FIG. 7B, this includes obtaining the non-contiguous specifier from the instruction to be emulated, and using one or more rules to create the contiguous specifier from the non-contiguous specifier. In one embodiment, the opcode of the instruction that has the non-contiguous specifier indicates, at least implicitly by its format, that the instruction includes a non-contiguous specifier. For instance, the format of the instruction is indicated by one or more bits of the opcode (e.g., the first two bits), and based on the format, the processor (e.g., the compiler, translator, emulator of the processor) understands that this instruction includes a non-contiguous specifier, in which part of the specifier of a resource, such as a register, is included in one field of the instruction and one or more other parts of the specifier are located in one or more other fields of the instruction.

**[0047]** The opcode, as an example, also provides an indication to the processor of the one or more rules used to generate the contiguous specifier from the non-contiguous specifier. For instance, the opcode may indicate that a particular instruction is a vector register instruction, and thus, has an RXB field. Therefore, the processor accesses information (e.g., rules stored in memory or external storage) that indicates for an instruction with an RXB field, the RXB field provides the most significant bit for its corresponding register field. The rules specify, for instance, that to generate the contiguous field, the bits of the register field are combined with the one or more bits of the RXB field associated with the particular register operand

**[0048]** Subsequent to generating the contiguous specifier, the contiguous specifier is used without regard to the non-contiguous specifier. For instance, in STEP 808, the code is optimized using the contiguous specifier without regard to the non-contiguous specifier. Similarly, one or more registers are allocated using the contiguous specifier and without regard to the non-contiguous specifier, STEP 810. Yet further, in STEP 812, the emulated code is generated without regard to the non-contiguous specifier and using the allocation

performed in STEP 810. That is, in these steps, there is no indication that the contiguous specifier was generated from a non-contiguous specifier. The non-contiguous specifier is ignored.

**[0049]** Further details regarding translating a non-contiguous specifier to a contiguous specifier are described with reference to the examples in FIGs. 9A, 9B and 11. Referring initially to FIG. 9A, a Vector Load (VL) instruction 900 is depicted. In one example, the Vector Load instruction includes opcode fields 902a (e.g., bits 0-7), 902b (e.g., bits 40-47) indicating a vector load operation; a vector register field 904 (e.g., bits 8-11) used to designate a vector register ($V_1$); an index field ($X_2$) 906 (e.g., bits 12-15); a base field ($B_2$) 908 (e.g., bits 16-19); a displacement field ($D_2$) 910 (e.g., bits 20-31); and an RXB field 912 (e.g., bits 36-39). Each of the fields 904-912 in one example is separate and independent from the opcode field(s). Further, in one embodiment they are separate and independent from one another; however in other embodiments, more than one field may be combined. Further information on the use of these fields is described below.

**[0050]** In one example, selected bits (e.g., the first two bits of the opcode designated by opcode field 902a) specify the length and format of the instruction. In this particular example, the length is three half-words and the format is a vector register-and-index storage operation with an extended opcode field. The vector ($V_1$) field, along with its corresponding extension bit specified by RXB, designates a vector register (i.e., a non-contiguous specifier). In particular, for vector registers, the register containing the operand is specified using, for instance, a four-bit field of the register field with the addition of its register extension bit (RXB) as the most significant bit. For instance, if the four bit field in $V_1$ is 0010 binary and the extension bit for this operand is 1 binary, then the 5-bit field is 10010 binary, indicating register number 18 (in decimal).

**[0051]** The subscript number associated with the field of the instruction denotes the operand to which the field applies. For instance, the subscript number 1 associated with $V_1$ denotes the first operand, and so forth. This is used to determine which bit of the RXB field is combined with the register field. The register operand is one register in length, which is for instance, 128 bytes. In one example, in a vector register-and-index storage operation

instruction, the contents of general registers designated by the $X_2$ and $B_2$ fields are added to the contents of the $D_2$ field to form the second operand address. The displacement, $D_2$, for the Vector Load instruction is treated as a 12-bit unsigned integer, in one example.

**[0052]** In this example, since $V_1$ is the first operand, the leftmost location (e.g., bit 0) of the RXB is associated with this operand. Therefore, the value located in the leftmost location is combined with the value in the $V_1$ register field to generate the contiguous specifier, as described herein

**[0053]** In accordance with an aspect of the present invention, the Vector Load instruction 900, which is defined, for instance, in the z/Architecture, is emulated into a Load Vector Indexed instruction 950 defined, for instance, in the PowerPC architecture. Although, in this example, the z/Architecture is the source architecture and PowerPC is the target architecture, this is only one example. Many other architectures may be used for one or both of the source and target architectures.

**[0054]** Each architecture has associated with it particular registers that it may use. For instance, in the z/Architecture, there are 32 vector registers and other types of registers can map to a quadrant of the vector registers. As an example, as shown in FIG. 10, if there is a register file 1000 that includes 32 vector registers 1002 and each register is 128 bits in length, then 16 floating point registers 1004, which are 64 bits in length, can overlay the vector registers. Thus, as an example, when floating point register 2 is modified, then vector register 2 is also modified. Other mappings for other types of registers are also possible.

**[0055]** Similarly, the PowerPC or other target architecture has a set of registers assigned to it. This set of registers may be different or the same as the set of registers allocated to the source architecture. The target register may have more or less registers available for a particular type of instruction. For instance, in the example depicted in FIG. 9A, the Vector Load instruction and the Load Vector Indexed instruction have 32 vector registers available to it. Again, other examples are possible.

**[0056]** As indicated by the opcode, the Vector Load instruction includes a non-contiguous specifier, which in this example, is represented in the $V_1$ and RXB fields. These non-contiguous fields are combined to create a contiguous index in the Load Vector Indexed instruction 950. This contiguous specifier is indicated in VRT field 954 of instruction 950. In this particular example, as shown in the code VL v18, 0(0, gr5), the vector register being specified is register 18. This register is specified in the instruction by the non-contiguous specifier provided by the $V_1$ field and the RXB field. In this example, the V1 field includes a value of 2 (0010 binary) and the RXB field includes a value of 8 (1000 binary). Based on the pre-defined rules, since $V_1$ is the first operand, the leftmost bit (1) of 1000 is concatenated with the bits in the V1 field (0010) to produce a contiguous specifier of 10010, which is the value 18 in decimal.

**[0057]** As shown at reference numeral 956, a representation of 18 is placed in the VRT field of the Load Vector Indexed instruction, which corresponds to the register field ($V_1$) of the Vector Load instruction. For completeness, the RA and RB fields of instruction 950 correspond to the $X_2$ and $B_2$, respectively, of instruction 900. The $D_2$ field of instruction 900 has no corresponding field in instruction 950; and the opcode fields of instruction 900 correspond to the opcode fields of instruction 950.

**[0058]** A further example is depicted in FIG. 9B. In this example, as with the example depicted in FIG. 9A, the non-contiguous specifier ($V_1$, RXB) of instruction 900 is being transformed into a contiguous specifier (VRT) of instruction 950. However, in this example, the register allocated for instruction 950 does not have the same number as the transformed contiguous specifier; instead, the contiguous specifier is mapped to a different register. For instance, in the example in FIG. 9A, the non-contiguous specifier references register 18, as does the contiguous specifier. That is, there is a one for one mapping. However, in FIG. 9B, the non-contiguous specifier of 18 is transformed into a contiguous specifier of 18, but then, the 18 of the contiguous specifier is mapped to a different register, such as register 7 (see reference number 980). That is, register 18 in the source architecture maps to register 7 in the target architecture, in this particular example. Such a mapping is pre-defined and accessible to the processor.

**[0059]** A yet further example is depicted in FIG. 11. In this example, instead of allocating to a register during emulation, as in FIGs. 9A and 9B, allocation is to memory. In this example, an instruction, VLR, is used to move the contents of one vector register, VR 18, to another vector register, VR 24. However, in this example, it is assumed that the register file is not large enough to include these vector registers, so memory is used, instead. That is, there is a contiguous portion of memory that stores a plurality of vectors as an array. The array starts at an address, rvbase, at which the first register, e.g., register 0, is stored; and then, the next register is stored at an offset, e.g., 16 bytes, from rvbase; and the third register is stored at the offset from the second register, and so forth. Thus, in this example, register 18 is at an offset 288 from rvbase, and register 24 is at an offset 384 from rvbase.

**[0060]** In this example, there are two non-contiguous specifiers ($V_1$, RXB; and $V_2$, RXB). Thus, two contiguous specifiers are generated. For instance, since $V_1$ is the first operand, the first contiguous specifier is generated by a concatenation of the bits in $V_1$ with bit 0 of RXB. Since $V_1$ includes 1000 in binary (8 decimal) and RXB includes 1100 in binary (12 decimal), the first contiguous specifier is formed by concatenating 1 (from bit 0 of RXB) with 1000 (from $V_1$) providing 11000 (24 in decimal). Similarly, the second contiguous specifier is generated by a concatenation of 0010 (2 in decimal for $V_2$) and 1 (from bit 1 of RXB) providing 10010 (18 in decimal). Since these registers are within memory, vector register 24 is at an offset 384 from rvbase, and vector register 18 is at an offset 288 from rvbase. These values are shown in FIG. 11 at 1102, 1104, respectively.

**[0061]** The pseudo-code on the right of FIG. 11 and the instructions on the left describe moving a contiguous number of bytes that correspond to a vector register at a vector offset at 18 (which corresponds to a byte offset at 288) to a vector offset at 24 (which corresponds to a byte offset at 384). In particular, a load immediate (LI) loads a value of 288 into rtemp1, and then a vector load is performed at an address provided by rvbase plus the offset in rtemp1, and the value is stored in a temporary vector register, vtemp2. Then, the next load immediate loads 384 into rtemp1, and a store back out to memory is performed at a location that corresponds to the address plus offset in vector register 24 (e.g., offset 288).

**[0062]** Although various examples are described above, many other examples and variations are possible. Additional information regarding vector instructions and use of the RXB field is described in a patent application co-filed herewith, entitled "Instruction to Load Data Up to A Specified Memory Boundary Indicated by the Instruction," U.S. Serial No._____, (IBM Docket No. POU920120030US1), Jonathan D. Bradbury et al., which is hereby incorporated herein by reference in its entirety.

**[0063]** Further, various architectures are mentioned herein. One embodiment of the z/Architecture is described in an IBM® publication entitled, "z/Architecture Principles of Operation," IBM® Publication No. SA22-7832-08, Ninth Edition, August, 2010, which is hereby incorporated herein by reference in its entirety. IBM® and Z/ARCHITECTURE® are registered trademarks of International Business Machines Corporation, Armonk, New York, USA. Other names used herein may be registered trademarks, trademarks, or product names of International Business Machines Corporation or other companies. Additionally, one embodiment of the Power Architecture is described in "Power ISA™ Version 2.06 Revision B," International Business Machines Corporation, July 23, 2010, which is hereby incorporated herein by reference in its entirety. POWER ARCHITECTURE® is a registered trademark of International Business Machines Corporation. Yet further, one embodiment of an Intel architecture is described in "Intel® 64 and IA-32 Architectures Developer's Manual: Vol. 2B, Instructions Set Reference, A-L," Order Number 253666-041US, December 2011, and "Intel® 64 and IA-32 Architectures Developer's Manual: Vol. 2B, Instructions Set Reference, M-Z," Order Number 253667-041US, December 2011, each of which is hereby incorporated herein by reference in its entirety. Intel® is a registered trademark of Intel Corporation, Santa Clara, California.

**[0064]** Described in detail herein is a technique for transforming non-contiguous specifiers of an instruction defined for one system architecture to contiguous specifiers for an instruction defined for another system architecture. Previous architecture emulation has not successfully addressed the emulation of systems with non-contiguous specifiers, and particularly non-contiguous register specifiers, in either fixed or variable width instruction sets. However, in accordance with an aspect of the present invention, a technique is provided to extend prior emulators to handle non-contiguous specifiers. The technique

includes, for instance, reading non-contiguous specifiers, generating a contiguous index from a non-contiguous specifier, and using the contiguous index to access a homogeneous resource or represent a homogeneous resource.

**[0065]** In a further embodiment, in accordance with a JIT implementation, a contiguous index is used to perform allocation decisions, optionally representing a resource accessed by a non-contiguous specifier by a non-contiguous/non-homogeneous resource, but not reflective of partitioning by non-contiguous specifier boundaries, but by optimization decisions. That is, in one embodiment, an instruction defined for one architecture has at least one non-contiguous specifier for at least one resource, and that at least one non-contiguous specifier is transformed to at least one contiguous specifier. That at least one contiguous specifier is used to select at least one resource for an instruction of another architecture to use. The instruction of the other architecture, however, uses non-contiguous specifiers. Thus, the at least one contiguous specifier for the at least one selected resource is then transformed into at least one non-contiguous specifier for use by the instruction of the second architecture. In one embodiment, this is performed by an emulator.

**[0066]** In one embodiment, an emulator is provided for emulating instruction execution of a first computer architecture instruction set on a processor designed to a second computer architecture. The emulator includes, for instance, fetching instructions of an application by the emulation program; interpreting the opcode of the instructions in order to select an emulation module for emulating the instructions; determining from the opcode that the instructions employ non-contiguous register fields; combining non-contiguous register fields of the instruction to form a combined register field; and using the combined register field by instructions of the emulation module, in order to emulate the instructions.

**[0067]** Further, in one embodiment, the register space includes a sub-section, and the first computer architecture instruction set includes first instructions having register fields for only accessing the sub-section, and second instructions having non-contiguous register fields for accessing all of the register space.

**[0068]** In one embodiment, the RXB field is in the same location for all the instructions using the RXB field. The RXB bits are bit significant in that bit 36 of the RXB field is used to extend bits 8-11 of the instruction; bit 37 of RXB is used to extend bits 12-15; bit 38 of RXB is used to extend bits 16-19; and bit 39 of RXB is used to extend bits 32-35, as examples. Further, the decision to use a bit of the RXB as an extension bit is opcode dependent (e.g., $R_1$ vs. $V_1$). Moreover, non-contiguous specifiers can use fields other than RXB fields.

**[0069]** Herein, memory, main memory, storage and main storage are used interchangeably, unless otherwise noted explicitly or by context.

**[0070]** Additional details relating to the vector facility, including examples of instructions, are provided as part of this Detailed Description further below.

**[0071]** As will be appreciated by one skilled in the art, one or more aspects of the present invention may be embodied as a system, method or computer program product. Accordingly, one or more aspects of the present invention may take the form of an entirely hardware embodiment, an entirely software embodiment (including firmware, resident software, micro-code, etc.) or an embodiment combining software and hardware aspects that may all generally be referred to herein as a "circuit," "module" or "system". Furthermore, one or more aspects of the present invention may take the form of a computer program product embodied in one or more computer readable medium(s) having computer readable program code embodied thereon.

**[0072]** Any combination of one or more computer readable medium(s) may be utilized. The computer readable medium may be a computer readable storage medium. A computer readable storage medium may be, for example, but not limited to, an electronic, magnetic, optical, electromagnetic, infrared or semiconductor system, apparatus, or device, or any suitable combination of the foregoing. More specific examples (a non-exhaustive list) of the computer readable storage medium include the following: an electrical connection having one or more wires, a portable computer diskette, a hard disk, a random access memory (RAM), a read-only memory (ROM), an erasable programmable read-only memory

(EPROM or Flash memory), an optical fiber, a portable compact disc read-only memory (CD-ROM), an optical storage device, a magnetic storage device, or any suitable combination of the foregoing. In the context of this document, a computer readable storage medium may be any tangible medium that can contain or store a program for use by or in connection with an instruction execution system, apparatus, or device.

**[0073]** Referring now to FIG. 12, in one example, a computer program product 1200 includes, for instance, one or more non-transitory computer readable storage media 1202 to store computer readable program code means or logic 1204 thereon to provide and facilitate one or more aspects of the present invention.

**[0074]** Program code embodied on a computer readable medium may be transmitted using an appropriate medium, including but not limited to wireless, wireline, optical fiber cable, RF, etc., or any suitable combination of the foregoing.

**[0075]** Computer program code for carrying out operations for one or more aspects of the present invention may be written in any combination of one or more programming languages, including an object oriented programming language, such as Java, Smalltalk, C++ or the like, and conventional procedural programming languages, such as the "C" programming language, assembler or similar programming languages. The program code may execute entirely on the user's computer, partly on the user's computer, as a stand-alone software package, partly on the user's computer and partly on a remote computer or entirely on the remote computer or server. In the latter scenario, the remote computer may be connected to the user's computer through any type of network, including a local area network (LAN) or a wide area network (WAN), or the connection may be made to an external computer (for example, through the Internet using an Internet Service Provider).

**[0076]** One or more aspects of the present invention are described herein with reference to flowchart illustrations and/or block diagrams of methods, apparatus (systems) and computer program products according to embodiments of the invention. It will be understood that each block of the flowchart illustrations and/or block diagrams, and combinations of blocks in the flowchart illustrations and/or block diagrams, can be

implemented by computer program instructions. These computer program instructions may be provided to a processor of a general purpose computer, special purpose computer, or other programmable data processing apparatus to produce a machine, such that the instructions, which execute via the processor of the computer or other programmable data processing apparatus, create means for implementing the functions/acts specified in the flowchart and/or block diagram block or blocks.

**[0077]** These computer program instructions may also be stored in a computer readable medium that can direct a computer, other programmable data processing apparatus, or other devices to function in a particular manner, such that the instructions stored in the computer readable medium produce an article of manufacture including instructions which implement the function/act specified in the flowchart and/or block diagram block or blocks.

**[0078]** The computer program instructions may also be loaded onto a computer, other programmable data processing apparatus, or other devices to cause a series of operational steps to be performed on the computer, other programmable apparatus or other devices to produce a computer implemented process such that the instructions which execute on the computer or other programmable apparatus provide processes for implementing the functions/acts specified in the flowchart and/or block diagram block or blocks.

**[0079]** The flowchart and block diagrams in the figures illustrate the architecture, functionality, and operation of possible implementations of systems, methods and computer program products according to various embodiments of one or more aspects of the present invention. In this regard, each block in the flowchart or block diagrams may represent a module, segment, or portion of code, which comprises one or more executable instructions for implementing the specified logical function(s). It should also be noted that, in some alternative implementations, the functions noted in the block may occur out of the order noted in the figures. For example, two blocks shown in succession may, in fact, be executed substantially concurrently, or the blocks may sometimes be executed in the reverse order, depending upon the functionality involved. It will also be noted that each block of the block diagrams and/or flowchart illustration, and combinations of blocks in the block diagrams

and/or flowchart illustration, can be implemented by special purpose hardware-based systems that perform the specified functions or acts, or combinations of special purpose hardware and computer instructions.

**[0080]** In addition to the above, one or more aspects of the present invention may be provided, offered, deployed, managed, serviced, etc. by a service provider who offers management of customer environments. For instance, the service provider can create, maintain, support, etc. computer code and/or a computer infrastructure that performs one or more aspects of the present invention for one or more customers. In return, the service provider may receive payment from the customer under a subscription and/or fee agreement, as examples. Additionally or alternatively, the service provider may receive payment from the sale of advertising content to one or more third parties.

**[0081]** In one aspect of the present invention, an application may be deployed for performing one or more aspects of the present invention. As one example, the deploying of an application comprises providing computer infrastructure operable to perform one or more aspects of the present invention.

**[0082]** As a further aspect of the present invention, a computing infrastructure may be deployed comprising integrating computer readable code into a computing system, in which the code in combination with the computing system is capable of performing one or more aspects of the present invention.

**[0083]** As yet a further aspect of the present invention, a process for integrating computing infrastructure comprising integrating computer readable code into a computer system may be provided. The computer system comprises a computer readable medium, in which the computer medium comprises one or more aspects of the present invention. The code in combination with the computer system is capable of performing one or more aspects of the present invention.

**[0084]** Although various embodiments are described above, these are only examples. For example, computing environments of other architectures can incorporate and use one or

more aspects of the present invention. Further, vectors of other sizes or other registers may be used, and changes to the instructions may be made without departing from the spirit of the present invention. Additionally, other instructions may be used in the processing. Further, one or more aspects of the invention relating to transforming non-contiguous specifiers to contiguous specifiers may be used in other contexts. Further, the specifiers may be for other than registers. Other changes are also possible.

**[0085]** Further, other types of computing environments can benefit from one or more aspects of the present invention. As an example, a data processing system suitable for storing and/or executing program code is usable that includes at least two processors coupled directly or indirectly to memory elements through a system bus. The memory elements include, for instance, local memory employed during actual execution of the program code, bulk storage, and cache memory which provide temporary storage of at least some program code in order to reduce the number of times code must be retrieved from bulk storage during execution.

**[0086]** Input/Output or I/O devices (including, but not limited to, keyboards, displays, pointing devices, DASD, tape, CDs, DVDs, thumb drives and other memory media, etc.) can be coupled to the system either directly or through intervening I/O controllers. Network adapters may also be coupled to the system to enable the data processing system to become coupled to other data processing systems or remote printers or storage devices through intervening private or public networks. Modems, cable modems, and Ethernet cards are just a few of the available types of network adapters.

**[0087]** Referring to FIG. 13, representative components of a Host Computer system 5000 to implement one or more aspects of the present invention are portrayed. The representative host computer 5000 comprises one or more CPUs 5001 in communication with computer memory (i.e., central storage) 5002, as well as I/O interfaces to storage media devices 5011 and networks 5010 for communicating with other computers or SANs and the like. The CPU 5001 is compliant with an architecture having an architected instruction set and architected functionality. The CPU 5001 may have dynamic address translation (DAT) 5003 for transforming program addresses (virtual addresses) into real addresses of memory.

A DAT typically includes a translation lookaside buffer (TLB) 5007 for caching translations so that later accesses to the block of computer memory 5002 do not require the delay of address translation. Typically, a cache 5009 is employed between computer memory 5002 and the processor 5001. The cache 5009 may be hierarchical having a large cache available to more than one CPU and smaller, faster (lower level) caches between the large cache and each CPU. In some implementations, the lower level caches are split to provide separate low level caches for instruction fetching and data accesses. In one embodiment, an instruction is fetched from memory 5002 by an instruction fetch unit 5004 via a cache 5009. The instruction is decoded in an instruction decode unit 5006 and dispatched (with other instructions in some embodiments) to instruction execution unit or units 5008. Typically several execution units 5008 are employed, for example an arithmetic execution unit, a floating point execution unit and a branch instruction execution unit. The instruction is executed by the execution unit, accessing operands from instruction specified registers or memory as needed. If an operand is to be accessed (loaded or stored) from memory 5002, a load/store unit 5005 typically handles the access under control of the instruction being executed. Instructions may be executed in hardware circuits or in internal microcode (firmware) or by a combination of both.

**[0088]** As noted, a computer system includes information in local (or main) storage, as well as addressing, protection, and reference and change recording. Some aspects of addressing include the format of addresses, the concept of address spaces, the various types of addresses, and the manner in which one type of address is translated to another type of address. Some of main storage includes permanently assigned storage locations. Main storage provides the system with directly addressable fast-access storage of data. Both data and programs are to be loaded into main storage (from input devices) before they can be processed.

**[0089]** Main storage may include one or more smaller, faster-access buffer storages, sometimes called caches. A cache is typically physically associated with a CPU or an I/O processor. The effects, except on performance, of the physical construction and use of distinct storage media are generally not observable by the program.

**[0090]** Separate caches may be maintained for instructions and for data operands. Information within a cache is maintained in contiguous bytes on an integral boundary called a cache block or cache line (or line, for short). A model may provide an EXTRACT CACHE ATTRIBUTE instruction which returns the size of a cache line in bytes. A model may also provide PREFETCH DATA and PREFETCH DATA RELATIVE LONG instructions which effects the prefetching of storage into the data or instruction cache or the releasing of data from the cache.

**[0091]** Storage is viewed as a long horizontal string of bits. For most operations, accesses to storage proceed in a left-to-right sequence. The string of bits is subdivided into units of eight bits. An eight-bit unit is called a byte, which is the basic building block of all information formats. Each byte location in storage is identified by a unique nonnegative integer, which is the address of that byte location or, simply, the byte address. Adjacent byte locations have consecutive addresses, starting with 0 on the left and proceeding in a left-to-right sequence. Addresses are unsigned binary integers and are 24, 31, or 64 bits.

**[0092]** Information is transmitted between storage and a CPU or a channel subsystem one byte, or a group of bytes, at a time. Unless otherwise specified, in, for instance, the z/Architecture, a group of bytes in storage is addressed by the leftmost byte of the group. The number of bytes in the group is either implied or explicitly specified by the operation to be performed. When used in a CPU operation, a group of bytes is called a field. Within each group of bytes, in, for instance, the z/Architecture, bits are numbered in a left-to-right sequence. In the z/Architecture, the leftmost bits are sometimes referred to as the "high-order" bits and the rightmost bits as the "low-order" bits. Bit numbers are not storage addresses, however. Only bytes can be addressed. To operate on individual bits of a byte in storage, the entire byte is accessed. The bits in a byte are numbered 0 through 7, from left to right (in, e.g., the z/Architecture). The bits in an address may be numbered 8-31 or 40-63 for 24-bit addresses, or 1-31 or 33-63 for 31-bit addresses; they are numbered 0-63 for 64-bit addresses. Within any other fixed-length format of multiple bytes, the bits making up the format are consecutively numbered starting from 0. For purposes of error detection, and in preferably for correction, one or more check bits may be transmitted with each byte or with a group of bytes. Such check bits are generated automatically by the machine and cannot be

directly controlled by the program. Storage capacities are expressed in number of bytes. When the length of a storage-operand field is implied by the operation code of an instruction, the field is said to have a fixed length, which can be one, two, four, eight, or sixteen bytes. Larger fields may be implied for some instructions. When the length of a storage-operand field is not implied but is stated explicitly, the field is said to have a variable length. Variable-length operands can vary in length by increments of one byte (or with some instructions, in multiples of two bytes or other multiples). When information is placed in storage, the contents of only those byte locations are replaced that are included in the designated field, even though the width of the physical path to storage may be greater than the length of the field being stored.

**[0093]** Certain units of information are to be on an integral boundary in storage. A boundary is called integral for a unit of information when its storage address is a multiple of the length of the unit in bytes. Special names are given to fields of 2, 4, 8, and 16 bytes on an integral boundary. A halfword is a group of two consecutive bytes on a two-byte boundary and is the basic building block of instructions. A word is a group of four consecutive bytes on a four-byte boundary. A doubleword is a group of eight consecutive bytes on an eight-byte boundary. A quadword is a group of 16 consecutive bytes on a 16-byte boundary. When storage addresses designate halfwords, words, doublewords, and quadwords, the binary representation of the address contains one, two, three, or four rightmost zero bits, respectively. Instructions are to be on two-byte integral boundaries. The storage operands of most instructions do not have boundary-alignment requirements.

**[0094]** On devices that implement separate caches for instructions and data operands, a significant delay may be experienced if the program stores into a cache line from which instructions are subsequently fetched, regardless of whether the store alters the instructions that are subsequently fetched.

**[0095]** In one embodiment, the invention may be practiced by software (sometimes referred to licensed internal code, firmware, micro-code, milli-code, pico-code and the like, any of which would be consistent with one or more aspects the present invention). Referring to FIG. 13, software program code which embodies one or more aspects of the present

invention may be accessed by processor 5001 of the host system 5000 from long-term storage media devices 5011, such as a CD-ROM drive, tape drive or hard drive. The software program code may be embodied on any of a variety of known media for use with a data processing system, such as a diskette, hard drive, or CD-ROM. The code may be distributed on such media, or may be distributed to users from computer memory 5002 or storage of one computer system over a network 5010 to other computer systems for use by users of such other systems.

**[0096]** The software program code includes an operating system which controls the function and interaction of the various computer components and one or more application programs. Program code is normally paged from storage media device 5011 to the relatively higher-speed computer storage 5002 where it is available for processing by processor 5001. The techniques and methods for embodying software program code in memory, on physical media, and/or distributing software code via networks are well known and will not be further discussed herein. Program code, when created and stored on a tangible medium (including but not limited to electronic memory modules (RAM), flash memory, Compact Discs (CDs), DVDs, Magnetic Tape and the like is often referred to as a "computer program product". The computer program product medium is typically readable by a processing circuit preferably in a computer system for execution by the processing circuit.

**[0097]** FIG. 14 illustrates a representative workstation or server hardware system in which one or more aspects of the present invention may be practiced. The system 5020 of FIG. 14 comprises a representative base computer system 5021, such as a personal computer, a workstation or a server, including optional peripheral devices. The base computer system 5021 includes one or more processors 5026 and a bus employed to connect and enable communication between the processor(s) 5026 and the other components of the system 5021 in accordance with known techniques. The bus connects the processor 5026 to memory 5025 and long-term storage 5027 which can include a hard drive (including any of magnetic media, CD, DVD and Flash Memory for example) or a tape drive for example. The system 5021 might also include a user interface adapter, which connects the microprocessor 5026 via the bus to one or more interface devices, such as a keyboard 5024,

a mouse 5023, a printer/scanner 5030 and/or other interface devices, which can be any user interface device, such as a touch sensitive screen, digitized entry pad, etc. The bus also connects a display device 5022, such as an LCD screen or monitor, to the microprocessor 5026 via a display adapter.

**[0098]** The system 5021 may communicate with other computers or networks of computers by way of a network adapter capable of communicating 5028 with a network 5029. Example network adapters are communications channels, token ring, Ethernet or modems. Alternatively, the system 5021 may communicate using a wireless interface, such as a CDPD (cellular digital packet data) card. The system 5021 may be associated with such other computers in a Local Area Network (LAN) or a Wide Area Network (WAN), or the system 5021 can be a client in a client/server arrangement with another computer, etc. All of these configurations, as well as the appropriate communications hardware and software, are known in the art.

**[0099]** FIG. 15 illustrates a data processing network 5040 in which one or more aspects of the present invention may be practiced. The data processing network 5040 may include a plurality of individual networks, such as a wireless network and a wired network, each of which may include a plurality of individual workstations 5041, 5042, 5043, 5044. Additionally, as those skilled in the art will appreciate, one or more LANs may be included, where a LAN may comprise a plurality of intelligent workstations coupled to a host processor.

**[00100]** Still referring to FIG. 15, the networks may also include mainframe computers or servers, such as a gateway computer (client server 5046) or application server (remote server 5048 which may access a data repository and may also be accessed directly from a workstation 5045). A gateway computer 5046 serves as a point of entry into each individual network. A gateway is needed when connecting one networking protocol to another. The gateway 5046 may be preferably coupled to another network (the Internet 5047 for example) by means of a communications link. The gateway 5046 may also be directly coupled to one or more workstations 5041, 5042, 5043, 5044 using a communications link. The gateway

computer may be implemented utilizing an IBM eServer™ System z server available from International Business Machines Corporation.

**[00101]** Referring concurrently to FIG. 14 and FIG. 15, software programming code which may embody one or more aspects of the present invention may be accessed by the processor 5026 of the system 5020 from long-term storage media 5027, such as a CD-ROM drive or hard drive. The software programming code may be embodied on any of a variety of known media for use with a data processing system, such as a diskette, hard drive, or CD-ROM. The code may be distributed on such media, or may be distributed to users 5050, 5051 from the memory or storage of one computer system over a network to other computer systems for use by users of such other systems.

**[00102]** Alternatively, the programming code may be embodied in the memory 5025, and accessed by the processor 5026 using the processor bus. Such programming code includes an operating system which controls the function and interaction of the various computer components and one or more application programs 5032. Program code is normally paged from storage media 5027 to high-speed memory 5025 where it is available for processing by the processor 5026. The techniques and methods for embodying software programming code in memory, on physical media, and/or distributing software code via networks are well known and will not be further discussed herein. Program code, when created and stored on a tangible medium (including but not limited to electronic memory modules (RAM), flash memory, Compact Discs (CDs), DVDs, Magnetic Tape and the like is often referred to as a "computer program product". The computer program product medium is typically readable by a processing circuit preferably in a computer system for execution by the processing circuit.

**[00103]** The cache that is most readily available to the processor (normally faster and smaller than other caches of the processor) is the lowest (L1 or level one) cache and main store (main memory) is the highest level cache (L3 if there are 3 levels). The lowest level cache is often divided into an instruction cache (I-Cache) holding machine instructions to be executed and a data cache (D-Cache) holding data operands.

**[00104]** Referring to FIG.16, an exemplary processor embodiment is depicted for processor 5026. Typically one or more levels of cache 5053 are employed to buffer memory blocks in order to improve processor performance. The cache 5053 is a high speed buffer holding cache lines of memory data that are likely to be used. Typical cache lines are 64, 128 or 256 bytes of memory data. Separate caches are often employed for caching instructions than for caching data. Cache coherence (synchronization of copies of lines in memory and the caches) is often provided by various "snoop" algorithms well known in the art. Main memory storage 5025 of a processor system is often referred to as a cache. In a processor system having 4 levels of cache 5053, main storage 5025 is sometimes referred to as the level 5 (L5) cache since it is typically faster and only holds a portion of the non-volatile storage (DASD, tape etc) that is available to a computer system. Main storage 5025 "caches" pages of data paged in and out of the main storage 5025 by the operating system.

**[00105]** A program counter (instruction counter) 5061 keeps track of the address of the current instruction to be executed. A program counter in a z/Architecture processor is 64 bits and can be truncated to 31 or 24 bits to support prior addressing limits. A program counter is typically embodied in a PSW (program status word) of a computer such that it persists during context switching. Thus, a program in progress, having a program counter value, may be interrupted by, for example, the operating system (context switch from the program environment to the operating system environment). The PSW of the program maintains the program counter value while the program is not active, and the program counter (in the PSW) of the operating system is used while the operating system is executing. Typically, the program counter is incremented by an amount equal to the number of bytes of the current instruction. RISC (Reduced Instruction Set Computing) instructions are typically fixed length while CISC (Complex Instruction Set Computing) instructions are typically variable length. Instructions of the IBM z/Architecture are CISC instructions having a length of 2, 4 or 6 bytes. The Program counter 5061 is modified by either a context switch operation or a branch taken operation of a branch instruction for example. In a context switch operation, the current program counter value is saved in the program status word along with other state information about the program being executed (such as condition codes), and a new program counter value is loaded pointing to an instruction of a new program module to be executed. A branch taken operation is performed in order to permit

the program to make decisions or loop within the program by loading the result of the branch instruction into the program counter 5061.

**[00106]** Typically an instruction fetch unit 5055 is employed to fetch instructions on behalf of the processor 5026. The fetch unit either fetches "next sequential instructions", target instructions of branch taken instructions, or first instructions of a program following a context switch. Modern Instruction fetch units often employ prefetch techniques to speculatively prefetch instructions based on the likelihood that the prefetched instructions might be used. For example, a fetch unit may fetch 16 bytes of instruction that includes the next sequential instruction and additional bytes of further sequential instructions.

**[00107]** The fetched instructions are then executed by the processor 5026. In an embodiment, the fetched instruction(s) are passed to a dispatch unit 5056 of the fetch unit. The dispatch unit decodes the instruction(s) and forwards information about the decoded instruction(s) to appropriate units 5057, 5058, 5060. An execution unit 5057 will typically receive information about decoded arithmetic instructions from the instruction fetch unit 5055 and will perform arithmetic operations on operands according to the opcode of the instruction. Operands are provided to the execution unit 5057 preferably either from memory 5025, architected registers 5059 or from an immediate field of the instruction being executed. Results of the execution, when stored, are stored either in memory 5025, registers 5059 or in other machine hardware (such as control registers, PSW registers and the like).

**[00108]** A processor 5026 typically has one or more units 5057, 5058, 5060 for executing the function of the instruction. Referring to FIG. 17A, an execution unit 5057 may communicate with architected general registers 5059, a decode/dispatch unit 5056, a load store unit 5060, and other 5065 processor units by way of interfacing logic 5071. An execution unit 5057 may employ several register circuits 5067, 5068, 5069 to hold information that the arithmetic logic unit (ALU) 5066 will operate on. The ALU performs arithmetic operations such as add, subtract, multiply and divide as well as logical function such as and, or and exclusive-or (XOR), rotate and shift. Preferably the ALU supports specialized operations that are design dependent. Other circuits may provide other architected facilities 5072 including condition codes and recovery support logic for example.

Typically the result of an ALU operation is held in an output register circuit 5070 which can forward the result to a variety of other processing functions. There are many arrangements of processor units, the present description is only intended to provide a representative understanding of one embodiment.

**[00109]** An ADD instruction for example would be executed in an execution unit 5057 having arithmetic and logical functionality while a floating point instruction for example would be executed in a floating point execution having specialized floating point capability. Preferably, an execution unit operates on operands identified by an instruction by performing an opcode defined function on the operands. For example, an ADD instruction may be executed by an execution unit 5057 on operands found in two registers 5059 identified by register fields of the instruction.

**[00110]** The execution unit 5057 performs the arithmetic addition on two operands and stores the result in a third operand where the third operand may be a third register or one of the two source registers. The execution unit preferably utilizes an Arithmetic Logic Unit (ALU) 5066 that is capable of performing a variety of logical functions such as Shift, Rotate, And, Or and XOR as well as a variety of algebraic functions including any of add, subtract, multiply, divide. Some ALUs 5066 are designed for scalar operations and some for floating point. Data may be Big Endian (where the least significant byte is at the highest byte address) or Little Endian (where the least significant byte is at the lowest byte address) depending on architecture. The IBM z/Architecture is Big Endian. Signed fields may be sign and magnitude, 1's complement or 2's complement depending on architecture. A 2's complement number is advantageous in that the ALU does not need to design a subtract capability since either a negative value or a positive value in 2's complement requires only an addition within the ALU. Numbers are commonly described in shorthand, where a 12 bit field defines an address of a 4,096 byte block and is commonly described as a 4 Kbyte (Kilo-byte) block, for example.

**[00111]** Referring to FIG. 17B, branch instruction information for executing a branch instruction is typically sent to a branch unit 5058 which often employs a branch prediction algorithm such as a branch history table 5082 to predict the outcome of the branch before

other conditional operations are complete. The target of the current branch instruction will be fetched and speculatively executed before the conditional operations are complete. When the conditional operations are completed the speculatively executed branch instructions are either completed or discarded based on the conditions of the conditional operation and the speculated outcome. A typical branch instruction may test condition codes and branch to a target address if the condition codes meet the branch requirement of the branch instruction, a target address may be calculated based on several numbers including ones found in register fields or an immediate field of the instruction for example. The branch unit 5058 may employ an ALU 5074 having a plurality of input register circuits 5075, 5076, 5077 and an output register circuit 5080. The branch unit 5058 may communicate with general registers 5059, decode dispatch unit 5056 or other circuits 5073, for example.

**[00112]** The execution of a group of instructions can be interrupted for a variety of reasons including a context switch initiated by an operating system, a program exception or error causing a context switch, an I/O interruption signal causing a context switch or multi-threading activity of a plurality of programs (in a multi-threaded environment), for example. Preferably a context switch action saves state information about a currently executing program and then loads state information about another program being invoked. State information may be saved in hardware registers or in memory for example. State information preferably comprises a program counter value pointing to a next instruction to be executed, condition codes, memory translation information and architected register content. A context switch activity can be exercised by hardware circuits, application programs, operating system programs or firmware code (microcode, pico-code or licensed internal code (LIC)) alone or in combination.

**[00113]** A processor accesses operands according to instruction defined methods. The instruction may provide an immediate operand using the value of a portion of the instruction, may provide one or more register fields explicitly pointing to either general purpose registers or special purpose registers (floating point registers for example). The instruction may utilize implied registers identified by an opcode field as operands. The instruction may utilize memory locations for operands. A memory location of an operand may be provided by a register, an immediate field, or a combination of registers and immediate field as

exemplified by the z/Architecture long displacement facility wherein the instruction defines a base register, an index register and an immediate field (displacement field) that are added together to provide the address of the operand in memory for example. Location herein typically implies a location in main memory (main storage) unless otherwise indicated.

**[00114]** Referring to FIG. 17C, a processor accesses storage using a load/store unit 5060. The load/store unit 5060 may perform a load operation by obtaining the address of the target operand in memory 5053 and loading the operand in a register 5059 or another memory 5053 location, or may perform a store operation by obtaining the address of the target operand in memory 5053 and storing data obtained from a register 5059 or another memory 5053 location in the target operand location in memory 5053. The load/store unit 5060 may be speculative and may access memory in a sequence that is out-of-order relative to instruction sequence, however the load/store unit 5060 is to maintain the appearance to programs that instructions were executed in order. A load/store unit 5060 may communicate with general registers 5059, decode/dispatch unit 5056, cache/memory interface 5053 or other elements 5083 and comprises various register circuits, ALUs 5085 and control logic 5090 to calculate storage addresses and to provide pipeline sequencing to keep operations in-order. Some operations may be out of order but the load/store unit provides functionality to make the out of order operations to appear to the program as having been performed in order, as is well known in the art.

**[00115]** Preferably addresses that an application program "sees" are often referred to as virtual addresses. Virtual addresses are sometimes referred to as "logical addresses" and "effective addresses". These virtual addresses are virtual in that they are redirected to physical memory location by one of a variety of dynamic address translation (DAT) technologies including, but not limited to, simply prefixing a virtual address with an offset value, translating the virtual address via one or more translation tables, the translation tables preferably comprising at least a segment table and a page table alone or in combination, preferably, the segment table having an entry pointing to the page table. In the z/Architecture, a hierarchy of translation is provided including a region first table, a region second table, a region third table, a segment table and an optional page table. The performance of the address translation is often improved by utilizing a translation lookaside

buffer (TLB) which comprises entries mapping a virtual address to an associated physical memory location. The entries are created when the DAT translates a virtual address using the translation tables. Subsequent use of the virtual address can then utilize the entry of the fast TLB rather than the slow sequential translation table accesses. TLB content may be managed by a variety of replacement algorithms including LRU (Least Recently used).

**[00116]** In the case where the processor is a processor of a multi-processor system, each processor has responsibility to keep shared resources, such as I/O, caches, TLBs and memory, interlocked for coherency. Typically, "snoop" technologies will be utilized in maintaining cache coherency. In a snoop environment, each cache line may be marked as being in any one of a shared state, an exclusive state, a changed state, an invalid state and the like in order to facilitate sharing.

**[00117]** I/O units 5054 (FIG. 16) provide the processor with means for attaching to peripheral devices including tape, disc, printers, displays, and networks for example. I/O units are often presented to the computer program by software drivers. In mainframes, such as the System z from IBM®, channel adapters and open system adapters are I/O units of the mainframe that provide the communications between the operating system and peripheral devices.

**[00118]** Further, other types of computing environments can benefit from one or more aspects of the present invention. As an example, an environment may include an emulator (e.g., software or other emulation mechanisms), in which a particular architecture (including, for instance, instruction execution, architected functions, such as address translation, and architected registers) or a subset thereof is emulated (e.g., on a native computer system having a processor and memory). In such an environment, one or more emulation functions of the emulator can implement one or more aspects of the present invention, even though a computer executing the emulator may have a different architecture than the capabilities being emulated. As one example, in emulation mode, the specific instruction or operation being emulated is decoded, and an appropriate emulation function is built to implement the individual instruction or operation.

**[00119]** In an emulation environment, a host computer includes, for instance, a memory to store instructions and data; an instruction fetch unit to fetch instructions from memory and to optionally, provide local buffering for the fetched instruction; an instruction decode unit to receive the fetched instructions and to determine the type of instructions that have been fetched; and an instruction execution unit to execute the instructions. Execution may include loading data into a register from memory; storing data back to memory from a register; or performing some type of arithmetic or logical operation, as determined by the decode unit. In one example, each unit is implemented in software. For instance, the operations being performed by the units are implemented as one or more subroutines within emulator software.

**[00120]** More particularly, in a mainframe, architected machine instructions are used by programmers, usually today "C" programmers, often by way of a compiler application. These instructions stored in the storage medium may be executed natively in a z/Architecture IBM® Server, or alternatively in machines executing other architectures. They can be emulated in the existing and in future IBM® mainframe servers and on other machines of IBM® (e.g., Power Systems servers and System x® Servers). They can be executed in machines running Linux on a wide variety of machines using hardware manufactured by IBM®, Intel®, AMD™, and others. Besides execution on that hardware under a z/Architecture, Linux can be used as well as machines which use emulation by Hercules, UMX, or FSI (Fundamental Software, Inc), where generally execution is in an emulation mode. In emulation mode, emulation software is executed by a native processor to emulate the architecture of an emulated processor.

**[00121]** The native processor typically executes emulation software comprising either firmware or a native operating system to perform emulation of the emulated processor. The emulation software is responsible for fetching and executing instructions of the emulated processor architecture. The emulation software maintains an emulated program counter to keep track of instruction boundaries. The emulation software may fetch one or more emulated machine instructions at a time and convert the one or more emulated machine instructions to a corresponding group of native machine instructions for execution by the native processor. These converted instructions may be cached such that a faster conversion

can be accomplished. Notwithstanding, the emulation software is to maintain the architecture rules of the emulated processor architecture so as to assure operating systems and applications written for the emulated processor operate correctly. Furthermore, the emulation software is to provide resources identified by the emulated processor architecture including, but not limited to, control registers, general purpose registers, floating point registers, dynamic address translation function including segment tables and page tables for example, interrupt mechanisms, context switch mechanisms, Time of Day (TOD) clocks and architected interfaces to I/O subsystems such that an operating system or an application program designed to run on the emulated processor, can be run on the native processor having the emulation software.

**[00122]** A specific instruction being emulated is decoded, and a subroutine is called to perform the function of the individual instruction. An emulation software function emulating a function of an emulated processor is implemented, for example, in a "C" subroutine or driver, or some other method of providing a driver for the specific hardware as will be within the skill of those in the art after understanding the description of the preferred embodiment. Various software and hardware emulation patents including, but not limited to U.S. Letters Patent No. 5,551,013, entitled "Multiprocessor for Hardware Emulation", by Beausoleil et al.; and U.S. Letters Patent No. 6,009,261, entitled "Preprocessing of Stored Target Routines for Emulating Incompatible Instructions on a Target Processor", by Scalzi et al; and U.S. Letters Patent No. 5,574,873, entitled "Decoding Guest Instruction to Directly Access Emulation Routines that Emulate the Guest Instructions", by Davidian et al; and U.S. Letters Patent No. 6,308,255, entitled "Symmetrical Multiprocessing Bus and Chipset Used for Coprocessor Support Allowing Non-Native Code to Run in a System", by Gorishek et al; and U.S. Letters Patent No. 6,463,582, entitled "Dynamic Optimizing Object Code Translator for Architecture Emulation and Dynamic Optimizing Object Code Translation Method", by Lethin et al; and U.S. Letters Patent No. 5,790,825, entitled "Method for Emulating Guest Instructions on a Host Computer Through Dynamic Recompilation of Host Instructions", by Eric Traut, each of which is hereby incorporated herein by reference in its entirety; and many others, illustrate a variety of known ways to achieve emulation of an instruction format architected for a different machine for a target machine available to those skilled in the art.

**[00123]** In FIG. 18, an example of an emulated host computer system 5092 is provided that emulates a host computer system 5000' of a host architecture. In the emulated host computer system 5092, the host processor (CPU) 5091 is an emulated host processor (or virtual host processor) and comprises an emulation processor 5093 having a different native instruction set architecture than that of the processor 5091 of the host computer 5000'. The emulated host computer system 5092 has memory 5094 accessible to the emulation processor 5093. In the example embodiment, the memory 5094 is partitioned into a host computer memory 5096 portion and an emulation routines 5097 portion. The host computer memory 5096 is available to programs of the emulated host computer 5092 according to host computer architecture. The emulation processor 5093 executes native instructions of an architected instruction set of an architecture other than that of the emulated processor 5091, the native instructions obtained from emulation routines memory 5097, and may access a host instruction for execution from a program in host computer memory 5096 by employing one or more instruction(s) obtained in a sequence & access/decode routine which may decode the host instruction(s) accessed to determine a native instruction execution routine for emulating the function of the host instruction accessed. Other facilities that are defined for the host computer system 5000' architecture may be emulated by architected facilities routines, including such facilities as general purpose registers, control registers, dynamic address translation and I/O subsystem support and processor cache, for example. The emulation routines may also take advantage of functions available in the emulation processor 5093 (such as general registers and dynamic translation of virtual addresses) to improve performance of the emulation routines. Special hardware and off-load engines may also be provided to assist the processor 5093 in emulating the function of the host computer 5000'.

**[00124]** The terminology used herein is for the purpose of describing particular embodiments only and is not intended to be limiting of the invention. As used herein, the singular forms "a", "an" and "the" are intended to include the plural forms as well, unless the context clearly indicates otherwise. It will be further understood that the terms "comprises" and/or "comprising", when used in this specification, specify the presence of stated features, integers, steps, operations, elements, and/or components, but do not preclude the presence or addition of one or more other features, integers, steps, operations, elements, components and/or groups thereof.

**[00125]** The corresponding structures, materials, acts, and equivalents of all means or step plus function elements in the claims below, if any, are intended to include any structure, material, or act for performing the function in combination with other claimed elements as specifically claimed. The description of one or more aspects of the present invention has been presented for purposes of illustration and description, but is not intended to be exhaustive or limited to the invention in the form disclosed. Many modifications and variations will be apparent to those of ordinary skill in the art without departing from the scope and spirit of the invention. The embodiment was chosen and described in order to best explain the principles of the invention and the practical application, and to enable others of ordinary skill in the art to understand the invention for various embodiments with various modifications as are suited to the particular use contemplated.

## Chapter 23. Vector String Instructions

### Vector String Facility

⋮

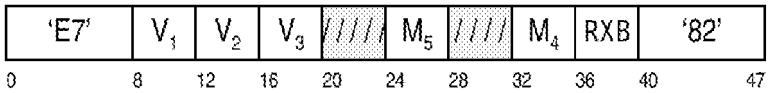
### Instructions

Unless otherwise specified all operands are vector-register operands. A "V" in the assembler syntax designates a vector operand.

| Name | Mnemonic | Characteristics | | | | | | | | Opcode | Page |
|---|---|---|---|---|---|---|---|---|---|---|---|
| VECTOR FIND ANY EQUAL | VFAE | VRR-b | C* | VF | $o^9$ | SP | Dv | | | E782 | 23-1 |
| VECTOR FIND ELEMENT EQUAL | VFEE | VRR-b | C* | VF | $o^9$ | SP | Dv | | | E780 | 23-2 |
| VECTOR FIND ELEMENT NOT EQUAL | VFENE | VRR-b | C* | VF | $o^9$ | SP | Dv | | | E781 | 23-3 |
| VECTOR STRING RANGE COMPARE | VSTRC | VRR-d | C* | VF | $o^9$ | SP | Dv | | | E78A | 23-4 |

### VECTOR FIND ANY EQUAL

VFAE $V_1,V_2,V_3,M_4,M_5$ [VRR-b]




Proceeding from left to right, every unsigned binary integer element of the second operand is compared for equality with each unsigned binary integer element of the third operand and optionally zero if the Zero Search flag is set in the $M_5$ field.

If the Result Type (RT) flag in the $M_5$ field is zero, then for each element in the second operand that matches any element in the third operand, or optionally zero, the bit positions of the corresponding element in the first operand are set to ones, otherwise they are set to zero. If the Result Type (RT) flag in the $M_5$ field is one, then the byte index of the leftmost element in the second operand that matches an element in the third operand or zero is stored in byte seven of the first operand.

Each instruction has an Extended Mnemonic section which describe recommended extended mnemonics and their corresponding machine assembler syntax.

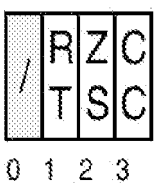
**Programming Note:** For all instructions that optionally set the condition code, performance may be degraded if the condition code is set.

If the result Type (RT) flag in the $M_5$ field is one and no bytes are found to be equal, or zero if the zero search flag is set, an index equal to the number of bytes in the vector is stored in byte seven of the first operand.

The $M_4$ field specifies the element size control (ES). The ES control specifies the size of the elements in the vector register operands. If a reserved value is specified, a specification exception is recognized.

0 - Byte
1 - Halfword
2 - Word
3-15 - Reserved

The $M_5$ field has the following format:



The bits of the $M_5$ field are defined as follows:

• ***Result Type (RT):*** If zero, each resulting element is a mask of all range comparisons on that element. If one, a byte index is stored into byte seven of the first operand and zeros are stored in all other elements.
• ***Zero Search (ZS):*** If one, each element of the second operand is also compared to zero.
• ***Condition Code Set (CC):*** If zero, the condition code is not set and remains unchanged. If one, the condition code is set as specified in the following section.

**Special Conditions**

A specification exception is recognized and no other action is taken if any of the following occurs:

1. The M4 field contains a value from 3-15.
2. Bit 0 of the M5 field are not zero.

***Resulting Condition Code:***

If the CC flag is zero, the code remains unchanged.

If the CC flag is one, the code is set as follows:

0 If the ZS-bit is set, there were no matches in a lower indexed element than zero in the second operand.

1 Some elements of the second operand match at least one element in the third operand

2 All elements of the second operand matched at least one element in the third operand

3 No elements in the second operand match any elements in the third operand

***Program Exceptions:***

1 Data with DXC FE, Vector Register

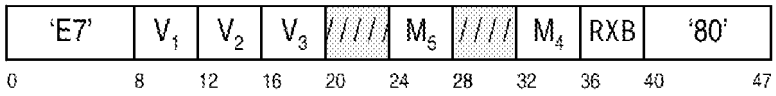
- Operation if the vector-extension facility is not installed
- Specification (Reserved ES value)
- Transaction Constraint

***Extended Mnemonics:***

| | | |
|---|---|---|
| VFAEB | $V_1,V_2,V_3,M_5$ | VFAE $V_1,V_2,V_3,0,M_5$ |
| VFAEH | $V_1,V_2,V_3,M_5$ | VFAE $V_1,V_2,V_3,1,M_5$ |
| VFAEF | $V_1,V_2,V_3,M_5$ | VFAE $V_1,V_2,V_3,2,M_5$ |
| VFAEBS | $V_1,V_2,V_3,M_5$ | VFAE $V_1,V_2,V_3,0,(M_5 \mid X'1')$ |
| VFAEHS | $V_1,V_2,V_3,M_5$ | VFAE $V_1,V_2,V_3,1,(M_5 \mid X'1')$ |
| VFAEFS | $V_1,V_2,V_3,M_5$ | VFAE $V_1,V_2,V_3,2,(M_5 \mid X'1')$ |
| VFAEZB | $V_1,V_2,V_3,M_5$ | VFAE $V_1,V_2,V_3,0,(M_5 \mid X'2')$ |
| VFAEZH | $V_1,V_2,V_3,M_5$ | VFAE $V_1,V_2,V_3,1,(M_5 \mid X'2')$ |
| VFAEZF | $V_1,V_2,V_3,M_5$ | VFAE $V_1,V_2,V_3,2,(M_5 \mid X'2')$ |
| VFAEZBS | $V_1,V_2,V_3,M_5$ | VFAE $V_1,V_2,V_3,0,(M_5 \mid X'3')$ |
| VFAEZHS | $V_1,V_2,V_3,M_5$ | VFAE $V_1$,V2,V3,1,($M_5$ \| X'3') |
| VFAEZFS | $V_1,V_2,V_3,M_5$ | VFAE $V_1,V_2,V_3,2,(M_5 \mid X'3')$ |

**VECTOR FIND ELEMENT EQUAL**

VFEE $V_1,V_2,V_3,M_4,M_5$ [VRR-b]]




Proceeding from left to right, the unsigned binary integer elements of the second operand are compared with the corresponding unsigned binary integer elements of the third operand. If two elements are equal, the byte index of the first byte of the leftmost equal element is placed in byte seven of the first operand. Zeros are stored in the remaining bytes of the first

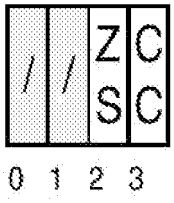
operand. If no bytes are found to be equal, or zero if the zero compare is set, then an index equal to the number of bytes in the vector is stored in byte seven of the first operand. Zeros are stored in the remaining bytes.

If the Zero Search (ZS) bit is set in the $M_5$ field, then each element in the second operand is also compared for equality with zero. If a zero element is found in the second operand before any other elements of the second and third operands are found to be equal, the byte index of the first byte of the element found to be zero is stored in byte seven the first operand and zeros are stored in all other byte locations. If the Condition Code Set (CC) flag is one, then the condition code is set to zero.

The $M_4$ field specifies the element size control (ES). The ES control specifies the size of the elements in the vector register operands. If a reserved value is specified, a specification exception is recognized.

0 - Byte
1 - Halfword
2 - Word
3-15 – Reserved

The $M_5$ field has the following format:




The bits of the $M_5$ field are defined as follows:

• ***Reserved:*** Bits 0-1 are reserved and must be zero. Otherwise, a specification exception is recognized.
• ***Zero Search (ZS):*** If one, each element of the second operand is also compared to zero.

• ***Condition Code Set (CC):*** If zero, the condition code remains unchanged. If one, the condition code is set as specified in the following section.

**Special Conditions**

A specification exception is recognized and no other action is taken if any of the following occurs:

1. The $M_4$ field contains a value from 3-15.
2. Bits 0-1 of the M5 field are not zero.

***Resulting Condition Code:***

If bit 3 of the $M_5$ field is set to one, the code is set as follows:

0 If the zero compare bit is set, comparison detected a zero element in the second operand in an element with a smaller index than any equal comparisons.

1 Comparison detected a match between the second and third operands in some element. If the zero compare bit is set, this match occurred in an element with an index less than or equal to the zero comparing element.

2 --

3 No elements compared equal.

If bit 3 of the $M_5$ field is zero, the code remains unchanged.

***Program Exceptions:***

• Data with DXC FE, Vector Register
• Operation if the vector-extension facility is not installed
• Specification (Reserved ES value)
• Transaction Constraint

***Extended Mnemonics:***

| | | |
|---|---|---|
| VFEEB | $V_1,V_2,V_3,M_5$ | VFEE $V_1,V_2,V_3,0,M_5$ |

| | | |
|---|---|---|
| VFEEH | $V_1,V_2,V_3,M_5$ | VFEE $V_1,V_2,V_3,1,M_5$ |
| VFEEF | $V_1,V_2,V_3,M_5$ | VFEE $V_1,V_2,V_3,0,(M_5 \mid X'1')$ |
| VFEEHS | $V_1,V_2,V_3,M_5$ | VFEE $V_1,V_2,V_3,1,(M_5 \mid X'1')$ |
| VFEEFS | $V_1,V_2,V_3,M_5$ | VFEE $V_1,V_2,V_3,2,(M_5 \mid X'1')$ |
| VFEEZB | $V_1,V_2,V_3,M_5$ | VFEE $V_1,V_2,V_3,0,(M_5 \mid X'2')$ |
| VFEEZH | $V_1,V_2,V_3,M_5$ | VFEE $V_1,V_2,V_3,1,(M_5 \mid X'2')$ |
| VFEEZF | $V_1,V_2,V_3,M_5$ | VFEE $V_1,V_2,V_3,2,(M_5 \mid X'2')$ |
| VFEEZBS | $V_1,V_2,V_3,M_5$ | VFEE $V_1,V_2,V_3,0,(M_5 \mid X'3')$ |
| VFEEZHS | $V_1,V_2,V_3,M_5$ | VFEE $V_1,V_2,V_3,1,(M_5 \mid X'3')$ |
| VFEEZFS | $V_1,V_2,V_3,M_5$ | VFEE $V_1,V_2,V_3,2,(M_5 \mid X'3')$ |

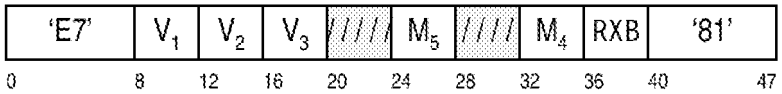
**Programming Notes:**

1. A byte index is always stored into the first operand for any element size. For example, if the element size was set to halfword and the 2nd indexed halfword compared equal, then a byte index of 4 would be stored.

2. The third operand should not contain elements with a value of zero. If the third operand does contain a zero and it matches with a zero element in the second operand before any other equal comparisons, condition code one is set regardless of the zero compare bit setting.

**VECTOR FIND ELEMENT NOT EQUAL**

VFENE $V_1,V_2,V_3,M_4,M_5$ [VRR-b]




Proceeding from left to right, the unsigned binary integer elements of the second operand are compared with the corresponding unsigned binary integer elements of the third operand. If two elements are not equal, the byte index of the left-most non-equal element is placed in byte seven of the first operand and zeros are stored to all other bytes. If the Condition Code Set (CC) bit in the $M_5$ field is set to one, the condition code is set to indicate which operand

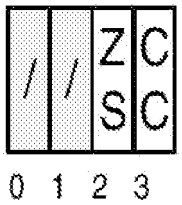
was greater. If all elements were equal, then abyte index equal to the vector size is placed in byte seven of the first operand and zeros are placed in all other byte locations. If the CC bit is one, condition code three is set.

If the zero search (ZS) bit is set in the $M_5$ field, each element in the second operand is also compared for equality with zero. If a zero element is found in the second operand before any other element of the second operand are found to be unequal, the byte index of the first byte of the element fount to be zero is stored in byte seven of the first operand. Zeros are stored in all other bytes and condition code 0 is set.

The $M_4$ field specifies the element size control (ES). The ES control specifies the size of the elements in the vector register operands. If a reserved value is specified, a specification exception is recognized.

0 - Byte
1 - Halfword
2 - Word
3-15 – Reserved

The $M_5$ field has the following format:




The bits of the $M_5$ field are defined as follows:

- ***Zero Search (ZS):*** If one, each element of the second operand is also compared to zero.
- ***Condition Code Set (CC):*** If zero, the condition code is not set and remains unchanged. If one, the condition code is set as specified in the following section.

**Special Conditions**

A specification exception is recognized and no other action is taken if any of the following occurs:

1. The $M_4$ field contains a value from 3-15.
2. Bits 0-1 of the $M_5$ field are not zero.

***Resulting Condition Code:***

If bit 3 of the $M_5$ field is set to one, the code is set as follows:

0 If the zero, compare bit is set, comparison detected a zero element in both operands in a lower indexed element than any unequal compares

1 An element mismatch was detected and the element in VR2 is less than the element in VR3

2 An element mismatch was detected and the element in VR2 is greater than the element in VR3

3 All elements compared equal, and if the zero compare bit is set, no zero elements were found in the second operand.

If bit 3 of the $M_5$ field is zero, the code remains unchanged.

***Program Exceptions:***

- Data with DXC FE, Vector Register
- Operation if the vector-extension facility is not installed
- Specification (Reserved ES value)
- Transaction Constraint

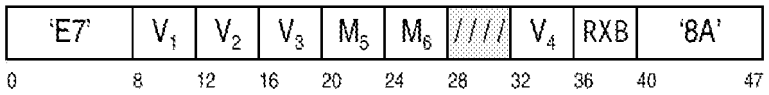
***Extended Mnemonics:***

| | | |
|---|---|---|
| VFENEB | $V_1,V_2,V_3,M_5$ | VFENE $V_1,V_2,V_3,0,M_5$ |
| VFENEH | $V_1,V_2,V_3,M_5$ | VFENE $V_1,V_2,V_3,1,M_5$ |

| | | |
|---|---|---|
| VFENEF | $V_1,V_2,V_3,M_5$ | VFENE $V_1,V_2,V_3,2,M_5$ |
| VFENEBS | $V_1,V_2,V_3,M_5$ | VFENE $V_1,V_2,V_3,0,(M_5 \mid X'1')$ |
| VFENEHS | $V_1,V_2,V_3,M_5$ | VFENE $V_1,V_2,V_3,1,(M_5 \mid X'1')$ |
| VFENEFS | $V_1,V_2,V_3,M_5$ | VFENE $V_1,V_2,V_3,2,(M_5 \mid X'1')$ |
| VFENEZB | $V_1,V_2,V_3,M_5$ | VFENE $V_1,V_2,V_3,0,(M_5 \mid X'2')$ |
| VFENEZH | $V_1,V_2,V_3,M_5$ | VFENE $V_1,V_2,V_3,1,(M_5 \mid X'2')$ |
| VFENEZF | $V_1,V_2,V_3,M_5$ | VFENE V1,V2,V3,2,($M_5$ \| X'2') |
| VFENEZBS | $V_1,V_2,V_3,M_5$ | VFENE $V_1,V_2,V_3,0,(M_5 \mid X'3')$ |
| VFENEZHS | $V_1,V_2,V_3,M_5$ | VFENE $V_1,V_2,V_3,1,(M_5 \mid X'3')$ |
| VFENEZFS | $V_1,V_2,V_3,M_5$ | VFENE $V_1,V_2,V_3,2,(M_5 \mid X'3')$ |

**VECTOR STRING RANGE COMPARE**

VSTRC $V_1,V_2,V_3,V_4,M_5,M_6$ [VRR-d]




Proceeding from left to right, the unsigned binary integer elements in the second operand are compared to ranges of values defined by even-odd pairs of elements in the third and fourth operands. The combined with control values from the fourth operand define the range of comparisons to be performed. If an element matches any of the ranges specified by the third and fourth operands, it is considered to be a match.

If the Result Type (RT) flag in the $M_6$ field is zero, the bit positions of the element in the first operand corresponding to the element being compared in the second operand are set to one if the element matches any of the ranges, otherwise they are set to zero.

If the Result Type (RT) flag in the M6 field is set to one, the byte index of the first element in the second operand that matches any of the ranges specified by the third and fourth operands or a zero comparison, if the ZS flag is set to one, is placed in byte seven of the first operand and zeros are stored in the remaining bytes. If no elements match, then an index

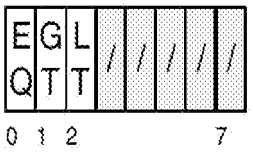
equal to the number of bytes in a vector is placed in byte seven of the first operand and zeros are stored in the remaining bytes.

The Zero Search (ZS) flag in the $M_6$ field, if set to one, will add a comparison to zero of the second operand elements to the ranges provided by the third and fourth operands. If a zero comparison in a lower indexed element than any other true comparison, then the condition code is set to zero.

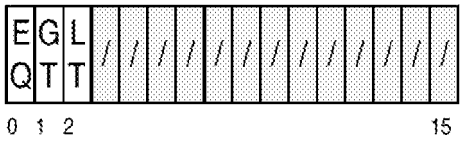
The operands contain elements of the size specified by the Element Size control in the $M_5$ field.

The fourth operand elements have the following format:

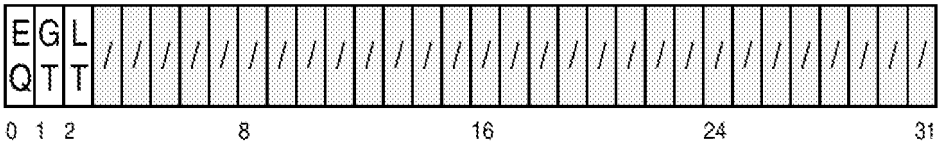
If ES equals 0:




If ES equals 1:




If ES equals 2:




The bits in the fourth operand elements are defined as follows:

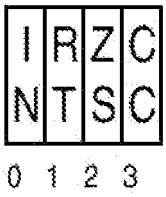
- ***Equal (EQ):*** When one a comparison for equality is made.
- ***Greater Than (GT):*** When one a greater than comparison is performed.
- ***Less Than (LT):*** When one a less than comparison is performed.
- All other bits are reserved and should be zero to ensure future compatibility.

The control bits may be used in any combination. If none of the bits are set, the comparison will always produce a false result. If all of the bits are set, the comparison will always produce a true result.

The $M_5$ field specifies the element size control (ES). The ES control specifies the size of the elements in the vector register operands. If a reserved value is specified, a specification exception is recognized.

0 - Byte
1 - Halfword
2 - Word
3-15 - Reserved

The $M_6$ field has the following format:




The bits of the $M_6$ field are defined as follows:

- ***Invert Result (IN):*** If zero, the comparison proceeds with the pair of values in the control vector. If one, the result of the pairs of the comparisons in the ranges are inverted.
- ***Result Type (RT):*** If zero, each resulting element is a mask of all range comparisons on that element. If one, an index is stored into byte seven of the first operand. Zeroes are stored in the remaining bytes.
- ***Zero Search (ZS):*** If one, each element of the second operand is also compared to zero.
- ***Condition Code Set (CC):*** If zero, the condition code is not set and remains unchanged. If one, the condition code is set as specified in the following section.

**Special Conditions**

A specification exception is recognized and no other action is taken if any of the following occurs:

1. The $M_4$ field contains a value from 3-15.

***Resulting Condition Code:***

| | |
|---|---|
| 0 | If ZS=1 and a zero is found in a lower indexed element than any compare |
| 1 | Comparison found |
| 2 | -- |
| 3 | No comparison found |

***Program Exceptions:***

- Data with DXC FE, Vector Register
- Operation if the vector-extension facility is not installed
- Specification (Reserved ES value)
- Transaction Constraint

***Extended Mnemonics:***

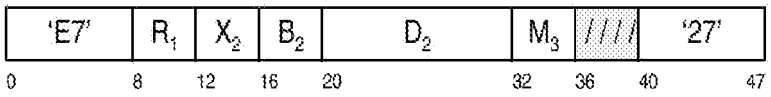
| | | |
|---|---|---|
| VSTRCB | $V_1,V_2,V_3,V_4,M_6$ | VSTRC $V_1,V_2,V_3,V_4,0,M_6$ |
| VSTRCH | $V_1,V_2,V_3,V_4,M_6$ | VSTRC $V_1,V_2,V_3,V_4,1,M_6$ |
| VSTRCF | $V_1,V_2,V_3,V_4,M_6$ | VSTRC $V_1,V_2,V_3,V_4,2,M_6$ |
| VSTRCBS | $V_1,V_2,V_3,V_4,M_6$ | VSTRC $V_1,V_2,V_3,V_4,0,(M_6 \mid X'1')$ |
| VSTRCHS | $V_1,V_2,V_3,V_4,M_6$ | VSTRC $V_1,V_2,V_3,V_4,1,(M_6 \mid X'1')$ |
| VSTRCFS | $V_1,V_2,V_3,V_4,M_6$ | VSTRC $V_1,V_2,V_3,V_4,2,(M_6 \mid X'1')$ |
| VSTRCZB | $V_1,V_2,V_3,V_4,M_6$ | VSTRC $V_1,V_2,V_3,V_4,0,(M_6 \mid X'2')$ |
| VSTRCZH | $V_1,V_2,V_3,V_4,M_6$ | VSTRC $V_1,V_2,V_3,V_4,1,(M_6 \mid X'2')$ |
| VSTRCZF | $V_1,V_2,V_3,V_4,M_6$ | VSTRC $V_1,V_2,V_3,V_4,2,(M_6 \mid X'2')$ |
| VSTRCZBS | $V_1,V_2,V_3,V_4,M_6$ | VSTRC $V_1,V_2,V_3,V_4,0,(M_6 \mid X'3')$ |
| VSTRCZHS | $V_1,V_2,V_3,V_4,M_6$ | VSTRC $V_1,V_2,V_3,V_4,1,(M_6 \mid X'3')$ |
| VSTRCZFS | $V_1,V_2,V_3,V_4,M_6$ | VSTRC $V_1,V_2,V_3,V_4,2,(M_6 \mid X'3')$ |

| | VR2 → | A | b | C | d | e | F | 1 | 2 |
|---|---|---|---|---|---|---|---|---|---|
| GE | A | T | T | T | T | T | T | F | F |
| LE | Z | T | F | T | F | F | T | F | F |
| GE | a | F | T | F | T | T | F | F | F |
| LE | c | T | T | T | F | F | T | T | T |
| LE | 4 | F | F | F | F | F | F | T | T |
| GE | 0 | T | T | T | T | T | T | T | T |
| EQ | d | F | F | F | T | F | F | F | F |
| EQ | d | F | F | F | T | F | F | F | F |
| VR4↑ | VR3 ↑ | | | | | | | | |
| IN=0 | VR1 (a)→ | FFFF | FFFF | FFFF | FFFF | 0000 | FFFF | FFFF | FFFF |
| IN=1 | VR1 (a)→ | 0000 | 0000 | 0000 | 0000 | FFFF | 0000 | 0000 | 0000 |
| IN=0 | VR1(b)→ | 0000 | 0000 | 0000 | 0000 | | | | |
| IN=1 | VR1(b)→ | 0000 | 0000 | 0000 | 0008 | | | | |
| | | | | | index | | | | |

*Figure 23-1.*
*ES=1,ZS=0*
*VR1(a) Results with RT=0*
*VR1(b) Results with RT=1*

## LOAD COUNT TO BLOCK BOUNDARY

LCBB $R_1,D_2(X_2,B_2),M_3$ [RXE]




A 32-bit unsigned binary integer containing the number of bytes possible to load from the second operand location without crossing a specified block boundary, capped at sixteen is placed in the first operand.

The displacement is treated as a 12-bit unsigned integer.

The second operand address is not used to address data.

The $M_3$ field specifies a code that is used to signal the CPU as to the block boundary size to compute the number of possible bytes loaded. If a reserved value is specified then a specification exception is recognized.

**Code Boundary**

0 64-Byte

1 128-Byte

2 256-Byte

3 512-Byte

4 1K-Byte

5 5 2K-Byte

6 4K-Byte

7-15 Reserved

***Resulting Condition Code:***

0 Operand one is sixteen

1 --

2 --

3 Operand one less than sixteen

***Resulting Condition Code:***

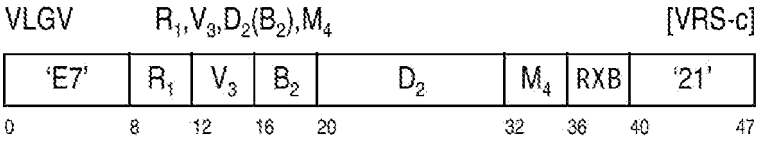
***Program Exceptions:***

• Operation if the vector-extension facility is not installed

• Specification

**Programming Note:** It is expected that LOAD COUNT TO BLOCK BOUNDARY will be used in conjunction with VECTOR LOAD TO BLOCK BOUNDARY to determine the number of bytes that were loaded.

**VECTOR LOAD GR FROM VR ELEMENT**



The element of the third operand of size specified by the ES value in the M4 field and indexed by the second operand address is placed in the first operand location. The third operand is a vector register. The first operand is a general register. If the index specified by the second operand address is greater than the highest numbered element in the third operand, of the specified element size, the data in the first operand is unpredictable.
If the vector register element is smaller than a doubleword, the element is right aligned in the 64-bit general register and zeros fill the remaining bits.

The second operand address is not used to address data; instead the rightmost 12 bits of the address are used to specify the index of an element within the second operand.

The $M_4$ field specifies the element size control (ES). The ES control specifies the size of the elements in the vector register operands. If a reserved value is specified, a specification exception is recognized.

0 - Byte
1 - Halfword
2 - Word
3 – Doubleword
4-15 – Reserved unchanged.

***Resulting Condition Code:*** The code is unchanged.

***Program Exceptions:***

- Data with DXC FE, Vector Register
- Operation if the vector-extension facility is not installed
- Specification (Reserved ES value)
- Transaction Constraint

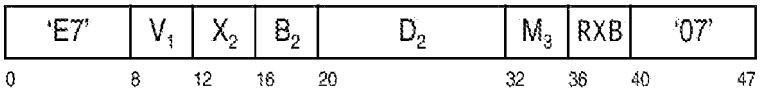
***Extended Mnemonics:***

| | | |
|---|---|---|
| VLGVB | $R_1,V_3,D_2(B2)$ | VLGV R1,V3,D2(B2),0 |

| | | |
|---|---|---|
| VLGVH | R1,V3,D2(B2) | VLGV R1,V3,D2(B2),1 |
| VLGVF | R1,V3,D2(B2) | VLGV R1,V3,D2(B2),2 |
| VLGVG | R1,V3,D2(B2) | VLGV R1,V3,D2(B2),3 |

## VECTOR LOAD TO BLOCK BOUNDARY

VLBB $V_1,D_2(X_2,B_2),M_3$ [VRX]




The first operand is loaded starting at the zero indexed byte element with bytes from the second operand. If a boundary condition is encountered, the rest of the first operand is unpredictable. Access exceptions are not recognized on bytes not loaded.

The displacement for VLBB is treated as a 12-bit unsigned integer.

The $M_3$ field specifies a code that is used to signal the CPU as to the block boundary size to load to. If a reserved value is specified, a specification exception is recognized.

**Code Boundary**

0 64-Byte
1 128-Byte
2 256-Byte
3 512-Byte
4 1K-Byte
5 2K-Byte
6 4K-Byte
7-15 Reserved

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

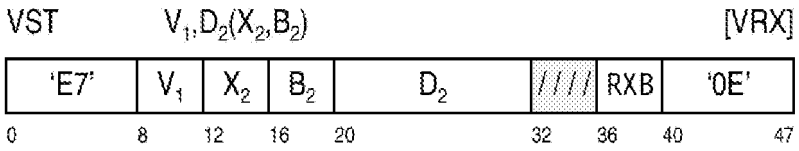
***Program Exceptions:***

- Access (fetch, operand 2)
- Data with DXC FE, Vector Register
- Operation if the vector-extension facility is not installed
- Specification (Reserved Block Boundary Code)
- Transaction Constraint

**Programming Notes:**

1. In certain circumstances data may be loaded past the block boundary. However, this will only occur if there are no access exceptions on that data.

**VECTOR STORE**



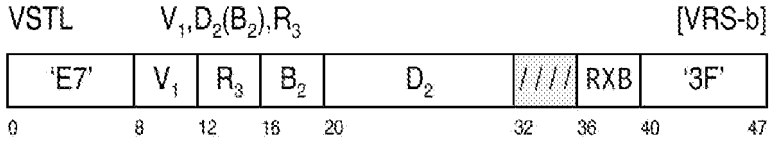

The 128-bit value in the first operand is stored to the storage location specified by the second operand. The displacement for VST is treated as a 12-bit unsigned integer.

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

***Program Exceptions:***

- Access (store, operand 2)
- Data with DXC FE, Vector Register
- Operation if the vector-extension facility is not installed
- Transaction Constraint

## VECTOR STORE WITH LENGTH




Proceeding from left to right, bytes from the first operand are stored at the second operand location. The general register specified third operand contains a 32-bit unsigned integer containing a value that represents the highest indexed byte to store. If the third operand contains a value greater than or equal to the highest byte index of the vector, all bytes of the first operand are stored.

Access exceptions are only recognized on bytes stored.
The displacement for VECTOR STORE WITH LENGTH is treated as a 12-bit unsigned integer.

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

***Program Exceptions:***

- Access (store, operand 2)
- Data with DXC FE, Vector Register
- Operation if the vector-extension facility is not installed
- Transaction Constraint

### RXB Description

All vector instructions have a field in bits 36-40 of the instruction labeled as RXB. This field contains the most significant bits for all of the vector register designated operands. Bits for register designations not specified by the instruction are reserved and should be set to zero; otherwise, the program may not operate compatibly in the future. The most significant bit is concatenated to the left of the four-bit register designation to create the five-bit vector register designation.

The bits are defined as follows:

0. Most significant bit for the vector register designation in bits 8-11 of the instruction.
1. Most significant bit for the vector register designation in bits 12-15 of the instruction.
2. Most significant bit for the vector register designation in bits 16-19 of the instruction.
3. Most significant bit for the vector register designation in bits 32-35 of the instruction.

**Vector Enablement Control**

The vector registers and instructions may only be used if both the vector enablement control (bit 46) and the AFP-register-control (bit 45) in control register zero are set to one. If the vector facility is installed and a vector instruction is executed without the enablement bits set, a data exception with DXC FE hex is recognized. If the vector facility is not installed, an operation exception is recognized.

## CLAIMS

1. A computer program product for transforming instruction specifiers of a computing environment, the computer program product comprising:

a computer readable storage medium readable by a processing circuit and storing instructions for execution by the processing circuit for performing a method comprising:

obtaining, by a processor, from a first instruction defined for a first computer architecture, a non-contiguous specifier, the non-contiguous specifier having a first portion and a second portion, wherein the obtaining comprises obtaining the first portion from a first field of the instruction and the second portion from a second field of the instruction, the first field separate from the second field;

generating a contiguous specifier using the first portion and the second portion, the generating using one or more rules based on the opcode of the first instruction; and

using the contiguous specifier to indicate a resource to be used in execution of a second instruction, the second instruction defined for a second computer architecture different from the first computer architecture and emulating a function of the first instruction.

2. The computer program product of claim 1, wherein the processor comprises an emulator, and wherein the first portion includes a first one or more bits, and the second portion includes a second one or more bits, and the generating comprises concatenating the second one or more bits with the first one or more bits to form the contiguous specifier, wherein the second one or more bits are the most significant bits of the contiguous specifier.

3. The computer program product of claim 2, wherein the first field has an operand position associated therewith, and the second one or more bits are a subset of a plurality of bits of the second field, and wherein the obtaining comprises selecting the second one or more bits from the plurality of bits of the second field based on the operand position of the first field.

4. The computer program product of claim 3, wherein the operand position of the first field is as a first operand, and wherein the second one or more bits are selected from a left-most location of the second field.

5. The computer program product of any preceding claim, wherein the first field consists of a register field, the second field consists of an extension field, the first portion consists of a plurality of bits from the register field, the second portion consists of a bit from the extension field in a location of the instruction corresponding to the register field, and the generating comprises concatenating the bit from the extension field with the bits from the register field to provide the contiguous specifier.

6. The computer program product of any preceding claim, wherein the using the contiguous specifier to indicate a resource includes using the contiguous specifier to map to a register to be used by the second instruction.

7. The computer program product of claim 6, wherein the register mapped to by the contiguous specifier has the same value as the contiguous specifier.

8. The computer program product of claim 6, wherein the register mapped to by the contiguous specifier has a different value from the contiguous specifier.

9. The computer program product of any preceding claim, wherein the first computer architecture includes an instruction set comprising first instructions having register fields to access a sub-section of a register space of the first computer architecture, and having second instructions having non-contiguous register fields for accessing the sub-section and remaining subsections of the register space, the first instructions precluded from accessing the remaining subsections.

10. The computer program product of claim 1, wherein the first field consists of a register field, the second field consists of an extension field, the first portion consists of a plurality of bits from the register field, the second portion consists of a bit from the extension field in a location of the instruction corresponding to the register field, and the generating

comprises concatenating the bit from the extension field with the bits from the register field to provide the contiguous specifier, and further comprising:

obtaining, by the processor, from the first instruction, another non-contiguous specifier, the another non-contiguous specifier having another first portion and another second portion, wherein the obtaining comprises obtaining the another first portion from another first field of the instruction and the another second portion from another bit of the extension field, the another first field separate from the first field and the extension field;

generating another contiguous specifier using the another first portion and the another bit, the generating using one or more rules based on the opcode of the first instruction; and

using the another contiguous specifier to indicate a resource to be used in execution of the second instruction.

11. A computer system for transforming instruction specifiers of a computing environment, the computer system comprising:

a memory; and

a processor in communications with the memory, wherein the computer system is configured to perform a method, said method comprising:

obtaining, by a processor, from a first instruction defined for a first computer architecture, a non-contiguous specifier, the non-contiguous specifier having a first portion and a second portion, wherein the obtaining comprises obtaining the first portion from a first field of the instruction and the second portion from a second field of the instruction, the first field separate from the second field;

generating a contiguous specifier using the first portion and the second portion, the generating using one or more rules based on the opcode of the first instruction; and

using the contiguous specifier to indicate a resource to be used in execution of a second instruction, the second instruction defined for a second computer architecture different from the first computer architecture and emulating a function of the first instruction.

12. The computer system of claim 11, wherein the processor comprises an emulator, and wherein the first portion includes a first one or more bits, and the second portion includes a

second one or more bits, and the generating comprises concatenating the second one or more bits with the first one or more bits to form the contiguous specifier, wherein the second one or more bits are the most significant bits of the contiguous specifier.

13. The computer system of claim 12, wherein the first field has an operand position associated therewith, and the second one or more bits are a subset of a plurality of bits of the second field, and wherein the obtaining comprises selecting the second one or more bits from the plurality of bits of the second field based on the operand position of the first field.

14. The computer system of claim 13, wherein the operand position of the first field is as a first operand, and wherein the second one or more bits are selected from a left-most location of the second field.

15. The computer system of any of claims 11 to 14, wherein the first field comprises a register field, the second field comprises an extension field, the first portion comprises a plurality of bits from the register field, the second portion comprises a bit from the extension field in a location corresponding to the register field, and the generating comprises concatenating the bit from the extension field with the bits from the register field to provide the contiguous specifier.

16. The computer system of any of claims 11 to 15, wherein the using the contiguous specifier to indicate a resource includes using the contiguous specifier to map to a register to be used by the second instruction.

17. The computer system of claim 16, wherein the register mapped to by the contiguous specifier has one of: the same value as the contiguous specifier or has a different value from the contiguous specifier.

18. A method of transforming instruction specifiers of a computing environment, the method comprising:

obtaining, by a processor, from a first instruction defined for a first computer architecture, a non-contiguous specifier, the non-contiguous specifier having a first portion

and a second portion, wherein the obtaining comprises obtaining the first portion from a first field of the instruction and the second portion from a second field of the instruction, the first field separate from the second field;

generating a contiguous specifier using the first portion and the second portion, the generating using one or more rules based on the opcode of the first instruction; and

using the contiguous specifier to indicate a resource to be used in execution of a second instruction, the second instruction defined for a second computer architecture different from the first computer architecture and emulating a function of the first instruction.

19. The method of claim 18, wherein the processor comprises an emulator, and wherein the first portion includes a first one or more bits, and the second portion includes a second one or more bits, and the generating comprises concatenating the second one or more bits with the first one or more bits to form the contiguous specifier, wherein the second one or more bits are the most significant bits of the contiguous specifier.

20. The method of claim 18 or claim 19, wherein the first field comprises a register field, the second field comprises an extension field, the first portion comprises a plurality of bits from the register field, the second portion comprises a bit from the extension field in a location corresponding to the register field, and the generating comprises concatenating the bit from the extension field with the bits from the register field to provide the contiguous specifier.