

APPARATUS AND METHOD FOR VECTOR BROADCAST AND XORAND LOGICAL INSTRUCTION

Field of the Invention

[0001] Embodiments of the invention relate generally to the field of computer systems. More particularly, the embodiments of the invention relate to an apparatus and method for performing a vector broadcast and XORAND logical instruction within a computer processor.

Background

[0002] Certain types of applications often require the same operation to be performed on a large number of data items (referred to as "data parallelism"). Single Instruction Multiple Data (SIMD) refers to a type of instruction that causes a processor to perform an operation on multiple data items. SIMD technology is especially suited to processors that can logically divide the bits in a register into a number of fixed-sized data elements, each of which represents a separate value. For example, the bits in a 256-bit register may be specified as a source operand to be operated on as four separate 64-bit packed data elements (quad-word (Q) size data elements), eight separate 32-bit packed data elements (double word (D) size data elements), sixteen separate 16-bit packed data elements (word (W) size data elements), or thirty-two separate 8-bit data elements (byte (B) size data elements). This type of data is referred to as "packed" data type or a "vector" data type, and operands of this data type are referred to as packed data operands or vector operands. In other words, a packed data item or vector refers to a sequence of packed data elements, and a packed data operand or a vector operand is a source or destination operand of a SIMD instruction (also known as a packed data instruction or a vector instruction).

[0003] The SIMD technology, such as that employed by the Intel® Core™ processors having an instruction set including x86, MMX™, Streaming SIMD Extensions (SSE), SSE2, SSE3, SSE4.1, and SSE4.2 instructions, has enabled a significant improvement in application performance. An additional set of SIMD extensions, referred to the Advanced Vector Extensions (AVX) (AVX1 and AVX2) and using the Vector Extensions (VEX) coding scheme, has been released (see, e.g., see Intel® 64 and IA-32 Architectures Software Developers Manual, October 2011; and see Intel® Advanced Vector Extensions Programming Reference, June 2011). These AVX extensions have been further proposed to be extended to support 512-bit registers (AVX-512) using the Extended Vector Extensions (EVEX) coding scheme.

[0004] A challenge exists in multiplying a Boolean (bit) matrix with a Boolean vector.

Current implementations may in many cases require repeated Boolean addition (XOR) operations on the elements of a matrix. This wastes significant processor cycles and as a result the Boolean matrix multiplication operation is very slow. Thus, an increase in efficiency may be gained if such bit matrix multiplication operation were implemented in a way to reduce unnecessary calculations.

BRIEF DESCRIPTION OF THE DRAWINGS

[0005] **FIG. 1A** is a block diagram illustrating both an exemplary in-order pipeline and an exemplary register renaming, out-of-order issue/execution pipeline according to embodiments of the invention;

[0006] **FIG. 1B** is a block diagram illustrating both an exemplary embodiment of an in-order architecture core and an exemplary register renaming, out-of-order issue/execution architecture core to be included in a processor according to embodiments of the invention;

[0007] **FIG. 2** is a block diagram of a single core processor and a multicore processor with integrated memory controller and graphics according to embodiments of the invention;

[0008] **FIG. 3** illustrates a block diagram of a system in accordance with one embodiment of the present invention;

[0009] **FIG. 4** illustrates a block diagram of a second system in accordance with an embodiment of the present invention;

[0010] **FIG. 5** illustrates a block diagram of a third system in accordance with an embodiment of the present invention;

[0011] **FIG. 6** illustrates a block diagram of a system on a chip (SoC) in accordance with an embodiment of the present invention;

[0012] **FIG. 7** illustrates a block diagram contrasting the use of a software instruction converter to convert binary instructions in a source instruction set to binary instructions in a target instruction set according to embodiments of the invention;

[0013] **FIG. 8** is a block diagram illustrating a system 800 that is operable to perform an embodiment of a vector broadcast and XORAND logical instruction;

[0014] **FIG. 9** illustrates logic for performing a vector broadcast and XORAND logical instruction in accordance with one embodiment of the invention;

[0015] **FIG. 10** is a flow diagram of a method 1000 for a system operable to perform an embodiment of a vector broadcast and XORAND logical instruction;

[0016] **FIG. 11** illustrates exemplary pseudocode for logic operable to perform an embodiment of a vector broadcast and XORAND logical instruction;

[0017] **FIGS. 12A and 12B** are block diagrams illustrating a generic vector friendly instruction format and instruction templates thereof according to embodiments of the invention;

[0018] **FIGS. 13A-D** illustrate an exemplary specific vector friendly instruction format according to embodiments of the invention;

[0019] FIG. 14 is a block diagram of a register architecture according to one embodiment of the invention; and

[0020] FIGS. 15A-B illustrate a block diagram of a more specific exemplary in-order core architecture.

DETAILED DESCRIPTION

EXEMPLARY PROCESSOR ARCHITECTURES

[0021] **Figure 1A** is a block diagram illustrating both an exemplary in-order fetch, decode, retire pipeline and an exemplary register renaming, out-of-order issue/execution pipeline according to embodiments of the invention. **Figure 1B** is a block diagram illustrating both an exemplary embodiment of an in-order fetch, decode, retire core and an exemplary register renaming, out-of-order issue/execution architecture core to be included in a processor according to embodiments of the invention. The solid lined boxes in **Figures 1A-B** illustrate the in-order portions of the pipeline and core, while the optional addition of the dashed lined boxes illustrates the register renaming, out-of-order issue/execution pipeline and core.

[0022] In **Figure 1A**, a processor pipeline 100 includes a fetch stage 102, a length decode stage 104, a decode stage 106, an allocation stage 108, a renaming stage 110, a scheduling (also known as a dispatch or issue) stage 112, a register read/memory read stage 114, an execute stage 116, a write back/memory write stage 118, an exception handling stage 122, and a commit stage 124.

[0023] **Figure 1B** shows processor core 190 including a front end unit 130 coupled to an execution engine unit 150, and both are coupled to a memory unit 170. The core 190 may be a reduced instruction set computing (RISC) core, a complex instruction set computing (CISC) core, a very long instruction word (VLIW) core, or a hybrid or alternative core type. As yet another option, the core 190 may be a special-purpose core, such as, for example, a network or communication core, compression engine, coprocessor core, general purpose computing graphics processing unit (GPGPU) core, graphics core, or the like.

[0024] The front end unit 130 includes a branch prediction unit 132 coupled to an instruction cache unit 134, which is coupled to an instruction translation lookaside buffer (TLB) 136, which is coupled to an instruction fetch unit 138, which is coupled to a decode unit 140. The decode unit 140 (or decoder) may decode instructions, and generate as an output one or more micro-operations, micro-code entry points, microinstructions, other instructions, or other control signals, which are decoded from, or which otherwise reflect, or are derived from, the original instructions. The decode unit 140 may be implemented using various different mechanisms. Examples of suitable mechanisms include, but are not limited to, look-up tables, hardware implementations, programmable logic arrays (PLAs), microcode read only memories (ROMs), etc. In one embodiment, the core 190 includes a microcode ROM or other medium that stores microcode for certain macroinstructions (e.g., in decode unit 140 or otherwise within

the front end unit 130). The decode unit 140 is coupled to a rename/allocator unit 152 in the execution engine unit 150.

[0025] The execution engine unit 150 includes the rename/allocator unit 152 coupled to a retirement unit 154 and a set of one or more scheduler unit(s) 156. The scheduler unit(s) 156 represents any number of different schedulers, including reservations stations, central instruction window, etc. The scheduler unit(s) 156 is coupled to the physical register file(s) unit(s) 158. Each of the physical register file(s) units 158 represents one or more physical register files, different ones of which store one or more different data types, such as scalar integer, scalar floating point, packed integer, packed floating point, vector integer, vector floating point, status (e.g., an instruction pointer that is the address of the next instruction to be executed), etc. In one embodiment, the physical register file(s) unit 158 comprises a vector registers unit, a write mask registers unit, and a scalar registers unit. These register units may provide architectural vector registers, vector mask registers, and general purpose registers. The physical register file(s) unit(s) 158 is overlapped by the retirement unit 154 to illustrate various ways in which register renaming and out-of-order execution may be implemented (e.g., using a reorder buffer(s) and a retirement register file(s); using a future file(s), a history buffer(s), and a retirement register file(s); using a register maps and a pool of registers; etc.). The retirement unit 154 and the physical register file(s) unit(s) 158 are coupled to the execution cluster(s) 160. The execution cluster(s) 160 includes a set of one or more execution units 162 and a set of one or more memory access units 164. The execution units 162 may perform various operations (e.g., shifts, addition, subtraction, multiplication) and on various types of data (e.g., scalar floating point, packed integer, packed floating point, vector integer, vector floating point). While some embodiments may include a number of execution units dedicated to specific functions or sets of functions, other embodiments may include only one execution unit or multiple execution units that all perform all functions. The scheduler unit(s) 156, physical register file(s) unit(s) 158, and execution cluster(s) 160 are shown as being possibly plural because certain embodiments create separate pipelines for certain types of data/operations (e.g., a scalar integer pipeline, a scalar floating point/packed integer/packed floating point/vector integer/vector floating point pipeline, and/or a memory access pipeline that each have their own scheduler unit, physical register file(s) unit, and/or execution cluster – and in the case of a separate memory access pipeline, certain embodiments are implemented in which only the execution cluster of this pipeline has the memory access unit(s) 164). It should also be understood that where separate pipelines are used, one or more of these pipelines may be out-

of-order issue/execution and the rest in-order.

[0026] The set of memory access units 164 is coupled to the memory unit 170, which includes a data TLB unit 172 coupled to a data cache unit 174 coupled to a level 2 (L2) cache unit 176. In one exemplary embodiment, the memory access units 164 may include a load unit, a store address unit, and a store data unit, each of which is coupled to the data TLB unit 172 in the memory unit 170. The instruction cache unit 134 is further coupled to a level 2 (L2) cache unit 176 in the memory unit 170. The L2 cache unit 176 is coupled to one or more other levels of cache and eventually to a main memory.

[0027] By way of example, the exemplary register renaming, out-of-order issue/execution core architecture may implement the pipeline 100 as follows: 1) the instruction fetch 138 performs the fetch and length decoding stages 102 and 104; 2) the decode unit 140 performs the decode stage 106; 3) the rename/allocator unit 152 performs the allocation stage 108 and renaming stage 110; 4) the scheduler unit(s) 156 performs the schedule stage 112; 5) the physical register file(s) unit(s) 158 and the memory unit 170 perform the register read/memory read stage 114; the execution cluster 160 perform the execute stage 116; 6) the memory unit 170 and the physical register file(s) unit(s) 158 perform the write back/memory write stage 118; 7) various units may be involved in the exception handling stage 122; and 8) the retirement unit 154 and the physical register file(s) unit(s) 158 perform the commit stage 124.

[0028] The core 190 may support one or more instructions sets (e.g., the x86 instruction set (with some extensions that have been added with newer versions); the MIPS instruction set of MIPS Technologies of Sunnyvale, CA; the ARM instruction set (with optional additional extensions such as NEON) of ARM Holdings of Sunnyvale, CA), including the instruction(s) described herein. In one embodiment, the core 190 includes logic to support a packed data instruction set extension (e.g., AVX1, AVX2, and/or some form of the generic vector friendly instruction format (U=0 and/or U=1), described below), thereby allowing the operations used by many multimedia applications to be performed using packed data.

[0029] It should be understood that the core may support multithreading (executing two or more parallel sets of operations or threads), and may do so in a variety of ways including time sliced multithreading, simultaneous multithreading (where a single physical core provides a logical core for each of the threads that physical core is simultaneously multithreading), or a combination thereof (e.g., time sliced fetching and decoding and simultaneous multithreading thereafter such as in the Intel® Hyperthreading technology).

[0030] While register renaming is described in the context of out-of-order execution, it should

be understood that register renaming may be used in an in-order architecture. While the illustrated embodiment of the processor also includes separate instruction and data cache units 134/174 and a shared L2 cache unit 176, alternative embodiments may have a single internal cache for both instructions and data, such as, for example, a Level 1 (L1) internal cache, or multiple levels of internal cache. In some embodiments, the system may include a combination of an internal cache and an external cache that is external to the core and/or the processor. Alternatively, all of the cache may be external to the core and/or the processor.

[0031] **Figure 2** is a block diagram of a processor 200 that may have more than one core, may have an integrated memory controller, and may have integrated graphics according to embodiments of the invention. The solid lined boxes in **Figure 2** illustrate a processor 200 with a single core 202A, a system agent 210, a set of one or more bus controller units 216, while the optional addition of the dashed lined boxes illustrates an alternative processor 200 with multiple cores 202A-N, a set of one or more integrated memory controller unit(s) 214 in the system agent unit 210, and special purpose logic 208.

[0032] Thus, different implementations of the processor 200 may include: 1) a CPU with the special purpose logic 208 being integrated graphics and/or scientific (throughput) logic (which may include one or more cores), and the cores 202A-N being one or more general purpose cores (e.g., general purpose in-order cores, general purpose out-of-order cores, a combination of the two); 2) a coprocessor with the cores 202A-N being a large number of special purpose cores intended primarily for graphics and/or scientific (throughput); and 3) a coprocessor with the cores 202A-N being a large number of general purpose in-order cores. Thus, the processor 200 may be a general-purpose processor, coprocessor or special-purpose processor, such as, for example, a network or communication processor, compression engine, graphics processor, GPGPU (general purpose graphics processing unit), a high-throughput many integrated core (MIC) coprocessor (including 30 or more cores), embedded processor, or the like. The processor may be implemented on one or more chips. The processor 200 may be a part of and/or may be implemented on one or more substrates using any of a number of process technologies, such as, for example, BiCMOS, CMOS, or NMOS.

[0033] The memory hierarchy includes one or more levels of cache within the cores, a set or one or more shared cache units 206, and external memory (not shown) coupled to the set of integrated memory controller units 214. The set of shared cache units 206 may include one or more mid-level caches, such as level 2 (L2), level 3 (L3), level 4 (L4), or other levels of cache, a last level cache (LLC), and/or combinations thereof. While in one embodiment a ring based

interconnect unit 212 interconnects the integrated graphics logic 208, the set of shared cache units 206, and the system agent unit 210/integrated memory controller unit(s) 214, alternative embodiments may use any number of well-known techniques for interconnecting such units. In one embodiment, coherency is maintained between one or more cache units 206 and cores 202-A-N.

[0034] In some embodiments, one or more of the cores 202A-N are capable of multi-threading. The system agent 210 includes those components coordinating and operating cores 202A-N. The system agent unit 210 may include for example a power control unit (PCU) and a display unit. The PCU may be or include logic and components needed for regulating the power state of the cores 202A-N and the integrated graphics logic 208. The display unit is for driving one or more externally connected displays.

[0035] The cores 202A-N may be homogenous or heterogeneous in terms of architecture instruction set; that is, two or more of the cores 202A-N may be capable of execution the same instruction set, while others may be capable of executing only a subset of that instruction set or a different instruction set. In one embodiment, the cores 202A-N are heterogeneous and include both the “small” cores and “big” cores described below.

[0036] **Figures 3-6** are block diagrams of exemplary computer architectures. Other system designs and configurations known in the arts for laptops, desktops, handheld PCs, personal digital assistants, engineering workstations, servers, network devices, network hubs, switches, embedded processors, digital signal processors (DSPs), graphics devices, video game devices, set-top boxes, micro controllers, cell phones, portable media players, hand held devices, and various other electronic devices, are also suitable. In general, a huge variety of systems or electronic devices capable of incorporating a processor and/or other execution logic as disclosed herein are generally suitable.

[0037] Referring now to **Figure 3**, shown is a block diagram of a system 300 in accordance with one embodiment of the present invention. The system 300 may include one or more processors 310, 315, which are coupled to a controller hub 320. In one embodiment the controller hub 320 includes a graphics memory controller hub (GMCH) 390 and an Input/Output Hub (IOH) 350 (which may be on separate chips); the GMCH 390 includes memory and graphics controllers to which are coupled memory 340 and a coprocessor 345; the IOH 350 is couples input/output (I/O) devices 360 to the GMCH 390. Alternatively, one or both of the memory and graphics controllers are integrated within the processor (as described herein), the memory 340 and the coprocessor 345 are coupled directly to the processor 310, and

the controller hub 320 in a single chip with the IOH 350.

[0038] The optional nature of additional processors 315 is denoted in **Figure 3** with broken lines. Each processor 310, 315 may include one or more of the processing cores described herein and may be some version of the processor 200.

[0039] The memory 340 may be, for example, dynamic random access memory (DRAM), phase change memory (PCM), or a combination of the two. For at least one embodiment, the controller hub 320 communicates with the processor(s) 310, 315 via a multi-drop bus, such as a frontside bus (FSB), point-to-point interface such as QuickPath Interconnect (QPI), or similar connection 395.

[0040] In one embodiment, the coprocessor 345 is a special-purpose processor, such as, for example, a high-throughput MIC processor, a network or communication processor, compression engine, graphics processor, GPGPU, embedded processor, or the like. In one embodiment, controller hub 320 may include an integrated graphics accelerator.

[0041] There can be a variety of differences between the physical resources 310, 315 in terms of a spectrum of metrics of merit including architectural, microarchitectural, thermal, power consumption characteristics, and the like.

[0042] In one embodiment, the processor 310 executes instructions that control data processing operations of a general type. Embedded within the instructions may be coprocessor instructions. The processor 310 recognizes these coprocessor instructions as being of a type that should be executed by the attached coprocessor 345. Accordingly, the processor 310 issues these coprocessor instructions (or control signals representing coprocessor instructions) on a coprocessor bus or other interconnect, to coprocessor 345. Coprocessor(s) 345 accept and execute the received coprocessor instructions.

[0043] Referring now to **Figure 4**, shown is a block diagram of a first more specific exemplary system 400 in accordance with an embodiment of the present invention. As shown in **Figure 4**, multiprocessor system 400 is a point-to-point interconnect system, and includes a first processor 470 and a second processor 480 coupled via a point-to-point interconnect 450. Each of processors 470 and 480 may be some version of the processor 200. In one embodiment of the invention, processors 470 and 480 are respectively processors 310 and 315, while coprocessor 438 is coprocessor 345. In another embodiment, processors 470 and 480 are respectively processor 310 coprocessor 345.

[0044] Processors 470 and 480 are shown including integrated memory controller (IMC) units 472 and 482, respectively. Processor 470 also includes as part of its bus controller units point-

to-point (P-P) interfaces 476 and 478; similarly, second processor 480 includes P-P interfaces 486 and 488. Processors 470, 480 may exchange information via a point-to-point (P-P) interface 450 using P-P interface circuits 478, 488. As shown in **Figure 4**, IMCs 472 and 482 couple the processors to respective memories, namely a memory 432 and a memory 434, which may be portions of main memory locally attached to the respective processors.

[0045] Processors 470, 480 may each exchange information with a chipset 490 via individual P-P interfaces 452, 454 using point to point interface circuits 476, 494, 486, 498. Chipset 490 may optionally exchange information with the coprocessor 438 via a high-performance interface 439. In one embodiment, the coprocessor 438 is a special-purpose processor, such as, for example, a high-throughput MIC processor, a network or communication processor, compression engine, graphics processor, GPGPU, embedded processor, or the like.

[0046] A shared cache (not shown) may be included in either processor or outside of both processors, yet connected with the processors via P-P interconnect, such that either or both processors' local cache information may be stored in the shared cache if a processor is placed into a low power mode.

[0047] Chipset 490 may be coupled to a first bus 416 via an interface 496. In one embodiment, first bus 416 may be a Peripheral Component Interconnect (PCI) bus, or a bus such as a PCI Express bus or another third generation I/O interconnect bus, although the scope of the present invention is not so limited.

[0048] As shown in **Figure 4**, various I/O devices 414 may be coupled to first bus 416, along with a bus bridge 418 which couples first bus 416 to a second bus 420. In one embodiment, one or more additional processor(s) 415, such as coprocessors, high-throughput MIC processors, GPGPU's, accelerators (such as, e.g., graphics accelerators or digital signal processing (DSP) units), field programmable gate arrays, or any other processor, are coupled to first bus 416. In one embodiment, second bus 420 may be a low pin count (LPC) bus. Various devices may be coupled to a second bus 420 including, for example, a keyboard and/or mouse 422, communication devices 427 and a storage unit 428 such as a disk drive or other mass storage device which may include instructions/code and data 430, in one embodiment. Further, an audio I/O 424 may be coupled to the second bus 420. Note that other architectures are possible. For example, instead of the point-to-point architecture of **Figure 4**, a system may implement a multi-drop bus or other such architecture.

[0049] Referring now to **Figure 5**, shown is a block diagram of a second more specific exemplary system 500 in accordance with an embodiment of the present invention. Like

elements in **Figures 4** and **5** bear like reference numerals, and certain aspects of **Figure 4** have been omitted from **Figure 5** in order to avoid obscuring other aspects of **Figure 5**.

[0050] **Figure 5** illustrates that the processors 470, 480 may include integrated memory and I/O control logic (“CL”) 472 and 482, respectively. Thus, the CL 472, 482 include integrated memory controller units and include I/O control logic. **Figure 5** illustrates that not only are the memories 432, 434 coupled to the CL 472, 482, but also that I/O devices 514 are also coupled to the control logic 472, 482. Legacy I/O devices 515 are coupled to the chipset 490.

[0051] Referring now to **Figure 6**, shown is a block diagram of a SoC 600 in accordance with an embodiment of the present invention. Similar elements in **Figure 2** bear like reference numerals. Also, dashed lined boxes are optional features on more advanced SoCs. In **Figure 6**, an interconnect unit(s) 602 is coupled to: an application processor 610 which includes a set of one or more cores 202A-N and shared cache unit(s) 206; a system agent unit 210; a bus controller unit(s) 216; an integrated memory controller unit(s) 214; a set or one or more coprocessors 620 which may include integrated graphics logic, an image processor, an audio processor, and a video processor; an static random access memory (SRAM) unit 630; a direct memory access (DMA) unit 632; and a display unit 640 for coupling to one or more external displays. In one embodiment, the coprocessor(s) 620 include a special-purpose processor, such as, for example, a network or communication processor, compression engine, GPGPU, a high-throughput MIC processor, embedded processor, or the like.

[0052] Embodiments of the mechanisms disclosed herein may be implemented in hardware, software, firmware, or a combination of such implementation approaches. Embodiments of the invention may be implemented as computer programs or program code executing on programmable systems comprising at least one processor, a storage system (including volatile and non-volatile memory and/or storage elements), at least one input device, and at least one output device.

[0053] Program code, such as code 430 illustrated in **Figure 4**, may be applied to input instructions to perform the functions described herein and generate output information. The output information may be applied to one or more output devices, in known fashion. For purposes of this application, a processing system includes any system that has a processor, such as, for example; a digital signal processor (DSP), a microcontroller, an application specific integrated circuit (ASIC), or a microprocessor.

[0054] The program code may be implemented in a high level procedural or object oriented programming language to communicate with a processing system. The program code may also

be implemented in assembly or machine language, if desired. In fact, the mechanisms described herein are not limited in scope to any particular programming language. In any case, the language may be a compiled or interpreted language.

[0055] One or more aspects of at least one embodiment may be implemented by representative instructions stored on a machine-readable medium which represents various logic within the processor, which when read by a machine causes the machine to fabricate logic to perform the techniques described herein. Such representations, known as “IP cores” may be stored on a tangible, machine readable medium and supplied to various customers or manufacturing facilities to load into the fabrication machines that actually make the logic or processor.

[0056] Such machine-readable storage media may include, without limitation, non-transitory, tangible arrangements of articles manufactured or formed by a machine or device, including storage media such as hard disks, any other type of disk including floppy disks, optical disks, compact disk read-only memories (CD-ROMs), compact disk rewritable’s (CD-RWs), and magneto-optical disks, semiconductor devices such as read-only memories (ROMs), random access memories (RAMs) such as dynamic random access memories (DRAMs), static random access memories (SRAMs), erasable programmable read-only memories (EPROMs), flash memories, electrically erasable programmable read-only memories (EEPROMs), phase change memory (PCM), magnetic or optical cards, or any other type of media suitable for storing electronic instructions.

[0057] Accordingly, embodiments of the invention also include non-transitory, tangible machine-readable media containing instructions or containing design data, such as Hardware Description Language (HDL), which defines structures, circuits, apparatuses, processors and/or system features described herein. Such embodiments may also be referred to as program products.

[0058] In some cases, an instruction converter may be used to convert an instruction from a source instruction set to a target instruction set. For example, the instruction converter may translate (e.g., using static binary translation, dynamic binary translation including dynamic compilation), morph, emulate, or otherwise convert an instruction to one or more other instructions to be processed by the core. The instruction converter may be implemented in software, hardware, firmware, or a combination thereof. The instruction converter may be on processor, off processor, or part on and part off processor.

[0059] **Figure 7** is a block diagram contrasting the use of a software instruction converter to convert binary instructions in a source instruction set to binary instructions in a target instruction set according to embodiments of the invention. In the illustrated embodiment, the instruction converter is a software instruction converter, although alternatively the instruction converter may be implemented in software, firmware, hardware, or various combinations thereof. **Figure 7** shows a program in a high level language 702 may be compiled using an x86 compiler 704 to generate x86 binary code 706 that may be natively executed by a processor with at least one x86 instruction set core 716. The processor with at least one x86 instruction set core 716 represents any processor that can perform substantially the same functions as an Intel processor with at least one x86 instruction set core by compatibly executing or otherwise processing (1) a substantial portion of the instruction set of the Intel x86 instruction set core or (2) object code versions of applications or other software targeted to run on an Intel processor with at least one x86 instruction set core, in order to achieve substantially the same result as an Intel processor with at least one x86 instruction set core. The x86 compiler 704 represents a compiler that is operable to generate x86 binary code 706 (e.g., object code) that can, with or without additional linkage processing, be executed on the processor with at least one x86 instruction set core 716.

[0060] Similarly, **Figure 7** shows the program in the high level language 702 may be compiled using an alternative instruction set compiler 708 to generate alternative instruction set binary code 710 that may be natively executed by a processor without at least one x86 instruction set core 714 (e.g., a processor with cores that execute the MIPS instruction set of MIPS Technologies of Sunnyvale, CA and/or that execute the ARM instruction set of ARM Holdings of Sunnyvale, CA). The instruction converter 712 is used to convert the x86 binary code 706 into code that may be natively executed by the processor without an x86 instruction set core 714. This converted code is not likely to be the same as the alternative instruction set binary code 710 because an instruction converter capable of this is difficult to make; however, the converted code will accomplish the general operation and be made up of instructions from the alternative instruction set. Thus, the instruction converter 712 represents software, firmware, hardware, or a combination thereof that, through emulation, simulation or any other process, allows a processor or other electronic device that does not have an x86 instruction set processor or core to execute the x86 binary code 706.

[0061] As mentioned above, multiplying a Boolean (bit) matrix and Boolean vector can be inefficient. Thus, a more efficient method of multiplying a Boolean matrix and vector is desirable. In particular, in some embodiments, an instruction is used to perform the matrix multiplication. The instruction performs a bitwise AND of a packed data operand that may represent a column of the bit matrix and a corresponding bit value of the bit vector that is broadcasted to a temporary vector. This bit vector may be represented by a second packed data operand. A bitwise exclusive OR operation is then performed with this resulting value and destination vector represented by a third packed data operand. When this instruction is repeated for all the columns of the bit matrix, the destination vector accumulates the results of all the operations and represents the result of the matrix multiplication. Note that modular arithmetic is used for matrix operations for bit vectors and matrices, and so a product operation becomes a bitwise AND, and a sum operation becomes a bitwise exclusive OR (i.e., XOR). Such an instruction enables a processor to efficiently process the multiplication of a bit matrix by a bit vector by eliminating overhead in term of extra logical instructions in a traditional instruction set repertoire.

[0062] **Figure 8** is a block diagram illustrating a system 800 that is operable to perform an embodiment of a vector broadcast and XORAND logical instruction. In some embodiments, system 800 may be part of a general-purpose processor (e.g., of the type commonly used in desktop, laptop, or other computers). Alternatively, system 800 may be a special-purpose processor. Examples of suitable special-purpose processors include, but are not limited to, cryptographic processors, network processors, communications processors, co-processors, graphics processors, embedded processors, digital signal processors (DSPs), and controllers (e.g., microcontrollers), to name just a few examples. The processor may be any of various complex instruction set computing (CISC) processors, various reduced instruction set computing (RISC) processors, various very long instruction word (VLIW) processors, various hybrids thereof, or other types of processors.

[0063] During operation, the system 800 may receive the embodiment of the vector broadcast and XORAND logical instruction 802 (hereafter referred to as instruction 802). For example, the instruction 802 may be received from an instruction fetch unit, an instruction queue, or the like. The instruction 802 may represent a macroinstruction, assembly language instruction, machine code instruction, or other instruction or control signal of an instruction set of the processor. In some embodiments, the instruction 802 may explicitly specify (e.g., through one

or more fields or a set of bits), or otherwise indicate (e.g., implicitly indicate), a first source packed data operand 810, and may explicitly specify or otherwise indicate a second source packed data operand 812. The instruction 802 may also explicitly specify or otherwise indicate a destination packed data operand 814, and may explicitly specify or otherwise indicate an immediate operand 808.

[0064] Referring again to Figure 8, the system 800 includes a decode unit or decoder 804. The decode unit may receive and decode instructions, including the instruction 802. The decode unit may output one or more microinstructions, micro-operations, micro-code entry points, decoded instructions or control signals, or other relatively lower-level instructions or control signals that reflect, represent, and/or are derived from the instruction 802. The one or more relatively lower-level instructions or control signals may implement the relatively higher-level instruction 802 through one or more relatively lower-level (e.g., circuit-level or hardware-level) operations. In some embodiments, the decode unit 804 may include one or more input structures (e.g., input port(s), input interconnect(s), an input interface, etc.) to receive the instruction 802, an instruction recognition logic coupled with the input structures to receive and recognize the instruction 802, a decode logic coupled with the recognition logic to receive and decode the instruction 802, and one or more output structures (e.g., output port(s), output interconnect(s), an output interface, etc.) coupled with the decode logic to output one or more corresponding lower level instructions or control signals. The recognition logic and the decode logic may be implemented using various different mechanisms including, but not limited to, microcode read only memories (ROMs), look-up tables, hardware implementations, programmable logic arrays (PLAs), and other mechanisms used to implement decode units known in the art. In some embodiments, decode unit 804 may be the same as decode unit 140 as illustrated in **Figure 1**.

[0065] The system 800 may also include a set of registers. In some embodiments, the registers may include general-purpose registers operable to hold data. The term general-purpose is often used to refer to an ability to store data or addresses in the registers, although this is not required. Each of the general-purpose registers may represent an on-die storage location that is operable to store data. The general-purpose registers may represent architecturally-visible registers (e.g., an architectural register file). The architecturally-visible or architectural registers are visible to software and/or a programmer and/or are the registers indicated by instructions to identify operands. These architectural registers are contrasted to other non-architectural or non-architecturally visible registers in a given microarchitecture (e.g.,

temporary registers, reorder buffers, retirement registers, etc.). The registers may be implemented in different ways in different microarchitectures using well-known techniques and are not limited to any particular type of circuit. Various different types of registers are suitable. Examples of suitable types of registers include, but are not limited to, dedicated physical registers, dynamically allocated physical registers using register renaming, and combinations thereof.

[0066] In some embodiments, the first source packed data operand 810 may be stored in a first general-purpose register, the second source packed data operand 812 may be stored in a second general-purpose register, the destination packed data operand 814 may be stored in a third general-purpose register. Alternatively, memory locations, or other storage locations, may be used for one or more of the source operands. For example, in some embodiments, memory operations may potentially be used for the second source packed data operand, although this is not required.

[0067] Execution unit 806 receives the control signals from decode unit 804 and executes instruction 802. Execution unit 806 is instructed to receive an immediate 8 bit value, a first source storage location, a second source storage location, and a destination storage location. These may be indicated by the immediate operand 808, the first source packed data operand 810, the second source packed data operand 812, and the destination source packed data operand 814, respectively. In some embodiments, the storage locations indicate registers, e.g., physical register file unit 158. In some embodiments, the storage locations indicate memory locations, such as a location in a memory unit, e.g., memory unit 170. The operations and functionality of the execution unit 806 may be described with further detail with reference to execution engine unit 150 in **Figure 1**.

[0068] Referring again to **Figure 8**, the execution unit 806 is coupled with the decode unit 804 and the registers. By way of example, the execution unit may include an arithmetic unit, an arithmetic logic unit, a digital circuit to perform arithmetic and logical operations, a digital circuit including a multiplier and adders, or the like. The execution unit may receive the one or more decoded or otherwise converted instructions or control signals that represent and/or are derived from the instruction 802. The execution unit may also receive the first source packed data operand 810, the second source packed data operand 812, the destination packed data operand 814, and the immediate operand 808. In some embodiments, the immediate operand has an 8-bit value. In some embodiments, the first source packed data operand 810, the second source packed data operand 812, and the destination packed data operand 814 indicate storage

locations with values that are multiples of 64 bits up to 512 bits. The execution unit is operable in response to and/or as a result of the instruction 802 (e.g., in response to one or more instructions or control signals decoded directly or indirectly (e.g., through emulation) from the instruction) to store a result.

[0069] In some embodiments, the packed data elements (bits) in the first source packed data operand 810, the second source packed data operand 812, and the destination packed data operand 814 are 64-bit packed data elements (quadwords). In such an embodiment, the operations performed on each 64-bit packed data element section are repeated, and the execution unit 806 may perform the operations on each 64-bit packed data element section in parallel or sequentially. In some embodiments, the length of the value indicated by the packed data operands may be many multiples of 64 in length, and may include multiple 64-bit sections.

[0070] The execution unit, as a result of the instruction 802, determines a bit in each 64-bit packed data element section indicated by the second source packed data operand 812 using as an index position the value indicated by the immediate operand. The immediate operand is an 8 bit value in one embodiment, and thus may represent 64 index positions, from 0 to 63, which may indicate a bit position in each of the 64-bit packed data element sections of the second source packed data operand.

[0071] Once the execution unit 806 determines a bit in one or more of the 64 bit packed data element sections, in some embodiments, this bit is then broadcasted to a temporary vector array of 64 bits for each of the 64-bit packed data element sections. In other words, this one bit value is repeated and placed in all 64-bit positions of this temporary vector array. In some embodiments, this temporary vector array may be a temporary or internal register that is not accessible through any programming language interface to the system.

[0072] For each of the 64-bit packed data element sections, the execution unit 806 then takes the corresponding temporary vector array with the broadcasted bit and performs a bitwise AND of this temporary vector array with the corresponding 64-bit packed data element sections indicated by the first source packed data operand. In other words, the execution unit 806 takes the determined bit and performs a bitwise AND between it and the corresponding bit values in the first source packed data operand.

[0073] The execution unit 806 further takes the resultant values (i.e., intermediate result) from this bitwise AND operand and performs a bitwise exclusive OR (XOR) operation of these resultant values and the corresponding packed data elements in the destination packed data operand. The execution unit 806 then takes the values determined from this bitwise XOR

operation and stores these values in the corresponding positions in the destination packed data operand.

[0074] These embodiments described above allow the system 800 to efficiently multiply a bit matrix with a bit vector using modular arithmetic. In some embodiments, some or all of the bit matrix is stored in the storage location indicated by the first source packed data operand, and the bit vector is stored in the storage location indicated by the second packed data operand. In some embodiments, the bit matrix is stored by row (i.e., position 0 in the storage location stores the first element of the first row of the bit matrix, position 1 stores the second element of the same first row of the bit matrix, etc.). In such a scenario, the bit matrix should be transposed such that it is stored in the storage location by column before performing the operations described above. When the operations described above are performed by the execution unit 806 repeatedly for all the values of the bit vector by setting different values in the immediate operand and all columns of the bit matrix, the values stored in the storage location indicated by the destination packed data operand represent the result of the matrix multiplication of the bit matrix and the bit vector. Note that the value indicated in the immediate operand for each operation should match the column number of the bit matrix indicated by the first source packed data operand such that the correct column in the bit matrix is multiplied with the correct value in the bit vector (i.e., value at position 0 in the bit vector should be multiplied with column at position 0 of the bit matrix). To achieve this, the storage location or storage location address indicated by the first source packed data operand may be changed for each iteration of the instruction 802.

[0075] In some embodiments, the bit matrix is a 64 x 64 bit matrix. As the storage location indicated by the first source packed data operand may not be able to store all 64 columns of the bit matrix (e.g., if the storage location were a 512-bit register), different storage locations may be indicated by the first source packed data operand for each iteration of the instruction 802 in order to complete the matrix multiplication operation.

[0076] Further details regarding the above embodiments will be describe below with reference to **Figures 9-11**.

[0077] The execution unit and/or the processor may include specific or particular logic (e.g., transistors, integrated circuitry, or other hardware potentially combined with firmware (e.g., instructions stored in non-volatile memory) and/or software) that is operable to perform the instruction 802 and/or store the result in response to and/or as a result of the instruction 802 (e.g., in response to one or more instructions or control signals decoded or otherwise derived

from instruction 802). In some embodiments, the execution unit may include one or more input structures (e.g., input port(s), input interconnect(s), an input interface, etc.) to receive source operands, circuitry or logic (e.g., a multiplier and at least one adder) coupled with the input structure(s) to receive and process the source operands and generate the result operand, and one or more output structures (e.g., output port(s), output interconnect(s), an output interface, etc.) coupled with the circuitry or logic to output the result operand.

[0078] To avoid obscuring the description, a relatively simple system 800 has been shown and described. In other embodiments, the system 800 may optionally include other well-known processor components. Possible examples of such components include, but are not limited to, an instruction fetch unit, instruction and data caches, second or higher level caches, out-of-order execution logic, an instruction scheduling unit, a register renaming unit, a retirement unit, a bus interface unit, instruction and data translation lookaside buffers, prefetch buffers, microinstruction queues, microinstruction sequencers, other components included in processors, and various combinations thereof. Numerous different combinations and configurations of such components are suitable. Embodiments are not limited to any known combination or configuration. Moreover, embodiments may be included in processors have multiple cores, logical processors, or execution engines at least one of which has a decode unit and an execution unit to perform an embodiment of instruction 802.

[0079] **Figure 9** illustrates logic 900 for performing a vector broadcast and XORAND logical instruction in accordance with one embodiment of the invention. In some embodiments, the execution unit 806 includes logic 900 to execute the instruction 802. In some embodiments, the instruction 802 specifies an immediate operand 808 (IMM8), a first source packed data operand 810 (SRC1), a second source packed data operand 812 (SRC2), and a destination packed data operand 814 (DEST). While the operands depicted in logic 900 indicate particular binary values, these values are included for illustrative purposes only and the operands may in other embodiments include different values.

[0080] Note that the storage locations indicated by SRC1 810, SRC2 812, and DEST 814 may each be able to store multiple packed 64-bit values. In such a scenario, logic 900 indicates that these 64-bit packed data element sections may each be processed in a similar fashion. In some embodiments, each section is processed in parallel to other sections. The length and demarcation of each of these sections is indicated by the numbers at 904. The total length of the operand values is indicated by the numbers at 906 and, in one embodiment, comprises 512 bits (i.e., 8 64-bit packed data elements stored in a 512 bit vector register).

[0081] In one embodiment, IMM8 808 is an 8-bit value which is able to specify a number from 0 to 63. In the exemplary binary values of **Figure 9**, this value is “4” (i.e., 100b in binary). An execution unit, following logic 900, determines a value of SRC2 812 at a bit position corresponding to the IMM8 808 value. In the exemplary values of **Figure 9**, this corresponds to the value “1” in the first 64-bit section of SRC2 812, and a “0” in the second 64-bit section of SRC2.

[0082] The execution unit then replicates or broadcasts each selected value of SRC2 812 64 times into a temporary vector B 902. Thus, for each 64-bit section of SRC2, a corresponding section in temporary vector B 902, with the same length, is populated with the selected value of SRC2 812 as selected using the index value of IMM8 808. For example, in **Figure 9**, the selected value for SRC2 812 in the first 64-bit section is a “1”. Thus, each of the 64 values in the corresponding 64-bit section of B 902 is set to “1”. In some embodiments, temporary vector B 902 may be a hidden register not accessible through any programming interface. In some embodiments, temporary vector B 902 may be stored in memory (e.g., RAM), or in cache, or other storage medium.

[0083] Although in **Figure 9** temporary vector B 902 includes multiple 64-bit sections corresponding to the 64-bit sections of SRC2 812, in some embodiments each 64-bit section of SRC2 is processed sequentially and thus temporary vector B 902 includes only a single 64-bit section. In other embodiments, each 64-bit sections of SRC2 812 is processed in parallel but separately, and thus multiple copies of temporary vector B 902 may exist at one time but are separate from each other.

[0084] In some embodiments, when the execution unit broadcasts the single selected value in SRC2 812 to the 64 values in temporary vector B 902, less than 64 processor cycles or operations are required, and instead the values are placed in a simultaneous fashion into the temporary vector B 902. In some embodiments, the broadcast is performed according to the broadcast method known by those skilled in the art.

[0085] The execution unit further executes logic 900 by performing a bitwise AND of the repeated values of temporary vector B 902 and the corresponding values of SRC1 810.

[0086] The execution unit further executes logic 900 to take the bitwise XOR between the results of the above bitwise AND operation (at 908) and the original values of DEST, shown by DEST 814a. The bitwise XOR results are then stored back into the storage location indicated by DEST, as shown by DEST 814b. For example, at bit position 0 (as indicated by 906), the bitwise AND between the value “1” of temporary vector B 902 and the value “1” at the

corresponding bit position of SRC1 810 is “1”, and the bitwise XOR between this resulting “1” value and the “0” value in the corresponding bit position of DEST 814a is “1”. This value of “1” is stored in the corresponding bit position of the storage location indicated by DEST, as shown at bit position 0 of DEST 814b.

[0087] In some cases, logic 900 may be used to efficiently determine the result of a matrix multiplication between a bit matrix of size 64x64 and a bit vector of size 64x1. In such an embodiment, the values of the bit vector are indicated by SRC2 812, and the values in one or more of the columns of the bit matrix are indicated by SRC1 810. If the bit matrix were originally represented in storage using a row by row format, the bit matrix would first be transposed by the execution unit to a column by column format. Matrix multiplication for bit matrices is represented using modular arithmetic. Thus, multiplication is represented by the bitwise AND, and addition is represented by the bitwise XOR. The first step in using logic 900 to multiply the bit matrix by the bit vector is to take the first column of the bit matrix and multiply it by the first value in the bit vector. To do this, IMM8 is set to the decimal value “0” to indicate the first bit position in the bit vector, which is indicated by SRC2 812. The first column of the bit matrix, which is 64-bits long, is also indicated by SRC1 810. Then, taking the bitwise AND of the first bit of the bit vector in temporary vector B 902 and the values of SRC1 810 represents the multiplication of the first column of the bit matrix with the first bit of the bit vector. This is shown in **Figure 9** at the bitwise AND shown at 908 and described above.

[0088] To properly complete the matrix multiplication, the above operations are repeated for the subsequent columns of the bit matrix and the corresponding bits in the bit vector, where each column and bit multiplication produces a resulting 64-bit value. These resulting 64-bit values should be summed together to arrive at a single 64-bit value. This single 64-bit value represents the outcome of the matrix multiplication of the bit matrix and the bit vector. Summation in bit matrix operations is represented by the bitwise XOR. Thus, the bitwise XOR operation shown at 910 in logic 900 adds the current bit matrix column and bit vector bit product to the running sum for the entire multiplication operation.

[0089] In some cases the 64x64 bit matrix is stored column by column at a memory location. Thus, for each execution of the instruction, the storage address indicated by SRC1 810 may be shifted 64 bits forward to the next column of the bit matrix. In other cases, SRC1 810 may be changed to a different storage location that stores the next column in the bit matrix. In some other cases each 64-bit section of SRC2 812, which represents the bit vector, may be shifted (or rotated) by 1 bit from the previous 64-bit section, and each 64-bit section of SRC1 810, which

represents the bit matrix, may include consecutive columns of the same bit matrix. This would allow consecutive columns of the bit matrix to be processed at once. In yet other cases each 64-bit section of SRC1 and SRC2 may represent different sets of bit matrix and bit vector combinations, allowing multiple bit matrix and bit vector multiplication computations to proceed simultaneously.

[0090] Although the matrix multiplication example above depicts the multiplication of a bit matrix and bit vector that are 64 bits long, in other cases the bit matrix and bit vector are less than 64 bits long.

[0091] **Figure 10** is a flow diagram of a method 1000 for a system operable to perform an embodiment of a vector broadcast and XORAND logical instruction. In various embodiments, the method may be performed by a processor, instruction processing apparatus, or other digital logic device. In some embodiments, the operations and/or method of **Figure 10** may be performed by and/or within the processor of **Figure 8**. The components, features, and specific optional details described herein for the processor of **Figure 8** also optionally apply to the operations and/or method of **Figure 10**. Alternatively, the operations and/or method of **Figure 10** may be performed by and/or within a similar or different processor or apparatus, such as those described with reference to **Figures 1-8**. Moreover, the processor of **Figure 8** may perform operations and/or methods the same as, similar to, or different than those of **Figure 10**.

[0092] The method 1000 includes, at block 1002, fetching an instruction from memory indicating a destination packed data operand, a first source packed data operand, a second source packed data operand, and an immediate operand. In various aspects, the instruction may be fetched and received at a processor, an instruction processing apparatus, or a portion thereof (e.g., an instruction fetch unit, a decode unit, a bus interface unit, etc.). In various aspects, the instruction may be received from an off-die source (e.g., from memory, interconnect, etc.), or from an on-die source (e.g., from an instruction cache, instruction queue, etc.).

[0093] At block 1004, the instruction is decoded. In some embodiments, the decoding of the instruction may be performed by a decode unit, such as decode unit 804 in **Figure 8**.

[0094] At block 1006, the method 1000 includes, determining a bit in the second source packed data operand based a position corresponding to the value of the immediate operand. In some embodiments, the determination of the data element is performed by an execution unit such as execution unit 806 in **Figure 8**.

[0095] At block 1008, the method 1000 includes storing a result in a storage location indicated by the destination packed data operand, wherein the result is determined from

performing a bitwise XOR between the destination packed data operand and an intermediate result, and wherein the intermediate result is determined from performing a bitwise AND between the first source packed data operand and the determined bit.

[0096] The illustrated method involves architectural operations (e.g., those visible from a software perspective). In other embodiments, the method may optionally include one or more microarchitectural operations. By way of example, the instruction may be fetched, decoded, scheduled out-of-order, source operands may be accessed, an execution unit may perform microarchitectural operations to implement the instruction, results may be rearranged back into program order, etc. In some embodiments, the microarchitectural operations to implement the instruction may optionally include any of the operations described in **Figures 1-7** and **12-15**.

[0097] **Figure 11** illustrates exemplary pseudocode for logic operable to perform an embodiment of a vector broadcast and XORAND logical instruction. In some embodiments, this logic is logic 900. The instruction 802, which is here represented by the opcode “vxorandbcstq” 1152, may specify various operands, as shown in 1154-1160. The operand zmm0 1152 specifies the destination packed data operand. In some embodiments, zmm0 1154 is DEST 814. In some embodiments, the instruction specifies a writemask 1162, in this case “k1”. The values of the writemask may indicate to an execution unit executing the logic represented by this pseudocode whether or not to write values to a specified portion of the register indicated by the destination packed data operand. The operand zmm1 1156 specifies the first source packed data operand. In some embodiments, this is SRC1 810. The operand zmm2 1158 specifies the second source packed data operand. In some embodiments this is SRC2 812. The operand imm8 1160 specifies an immediate operand. In some embodiments, imm8 1160 is IMM8 808.

[0098] Line 1102 indicates that the instruction is compatible in some embodiments with vector lengths of 128, 256, and 512. The K length indicates the number of sections of 64 packed data elements that the corresponding vector length of binary values may be separated into. As noted above, the instruction operates on sections of 64 packed data elements.

[0099] In some embodiments the operand of the instruction specifies an operand indicating a storage location that may store up to 512 bits, and in such a case only a portion of the register is used for the execution of the instruction. In some embodiments, one or more of the operands may indicate a memory storage location instead of a register location.

[00100] In **Figure 11**, a colon symbol with an equal sign symbol (colon-equals symbol) indicates that a value on the right side of the colon-equals sign is assigned to the variable on the

left side of the colon-equals sign.

[00101] At line 1104, a loop is set to iterate for a number of loops equal to the K length. For example, if the vector length were 128, the K length would be 2, and the loop would iterate two times. In some embodiments, the loop variable is “j”, as illustrated in **Figure 11**.

[00102] At line 1106, a variable i is set to j multiplied by 64. For example, when j is “2”, the variable i would be “128”.

[00103] At line 1108, 64 bits of a temporary vector B, which may be stored in an internal register, is set to the value of SRC2[i+IMM8] replicated 64 times for the 64 bits of the temporary vector B. In some embodiments, temporary vector B is temporary vector B 902. SRC2[i+IMM8] represents the value in SRC2 at position “i+IMM8”.

[00104] In some embodiments, the value SRC2[i+IMM8] is set to a temporary value “b”, which may be an internal register, and this temporary value b is then replicated or broadcast to the values of temporary vector B.

[00105] At line 1112, the 64 bits of DEST for the 64-bit section currently being processed as indicated by the loop at line 1104 are XOR’ed with the result of the previous bitwise AND operation, and this result is assigned back to the same 64 bits of DEST.

[00106] In some embodiments, the operation at line 1112 is predicated on whether the instruction 802 specifies a writemask. If a writemask is specified, then as shown in line 1110, the bit in the writemask at position j should be set to the value “1” for the operations on line 1112 to be executed by the execution unit 806. Otherwise, the operations on lines 1114-1118 are executed instead.

[00107] Line 1114 executes if the conditional at line 1110 is determined to be “0” or false. In some embodiments, at line 1114, a conditional statement checks to see if merge masking is enabled. In some embodiments, merging masking is indicated by a flag. In some embodiments, this flag is “EVEX.z”. In some embodiments, this flag is indicated by an operand (e.g., “{z}”) in the instruction. Merge masking, or merging masking, indicates to the execution unit to preserve the original values of the destination operand rather than overwrite these values with “0”. If merging masking is on, then the set of 64 packed data elements in DEST that are currently being processed are left unchanged, as shown in line 1116. Otherwise, as shown in line 1118, these values are overridden with “0” (i.e., the value “0” is stored in the corresponding positions of the register indicated by the destination operand).

[00108] In some embodiments, at line 1120, the remaining values in DEST which were not processed as part of the instruction, i.e., those beyond the vector length specified, are zeroed out

(i.e., the value “0” is stored in the corresponding positions of the register indicated by the destination operand).

[00109] Although the embodiments above are described with reference to registers that are 512 bits wide, other embodiments of the invention do not require registers with such a length, and the invention may be implemented with registers of any length.

EXEMPLARY INSTRUCTION FORMATS

[00110] Embodiments of the instruction(s) described herein may be embodied in different formats. Additionally, exemplary systems, architectures, and pipelines are detailed below. Embodiments of the instruction(s) may be executed on such systems, architectures, and pipelines, but are not limited to those detailed.

[00111] A vector friendly instruction format is an instruction format that is suited for vector instructions (e.g., there are certain fields specific to vector operations). While embodiments are described in which both vector and scalar operations are supported through the vector friendly instruction format, alternative embodiments use only vector operations the vector friendly instruction format.

[00112] **Figures 12A-12B** are block diagrams illustrating a generic vector friendly instruction format and instruction templates thereof according to embodiments of the invention. **Figure 12A** is a block diagram illustrating a generic vector friendly instruction format and class A instruction templates thereof according to embodiments of the invention; while **Figure 12B** is a block diagram illustrating the generic vector friendly instruction format and class B instruction templates thereof according to embodiments of the invention. Specifically, a generic vector friendly instruction format 1200 for which are defined class A and class B instruction templates, both of which include no memory access 1205 instruction templates and memory access 1220 instruction templates. The term generic in the context of the vector friendly instruction format refers to the instruction format not being tied to any specific instruction set.

[00113] While embodiments of the invention will be described in which the vector friendly instruction format supports the following: a 64 byte vector operand length (or size) with 32 bit (4 byte) or 64 bit (8 byte) data element widths (or sizes) (and thus, a 64 byte vector consists of either 16 doubleword-size elements or alternatively, 8 quadword-size elements); a 64 byte vector operand length (or size) with 16 bit (2 byte) or 8 bit (1 byte) data element widths (or sizes); a 32 byte vector operand length (or size) with 32 bit (4 byte), 64 bit (8 byte), 16 bit (2 byte), or 8 bit (1 byte) data element widths (or sizes); and a 16 byte vector operand length (or

size) with 32 bit (4 byte), 64 bit (8 byte), 16 bit (2 byte), or 8 bit (1 byte) data element widths (or sizes); alternative embodiments may support more, less and/or different vector operand sizes (e.g., 256 byte vector operands) with more, less, or different data element widths (e.g., 128 bit (16 byte) data element widths).

[00114] The class A instruction templates in **Figure 12A** include: 1) within the no memory access 1205 instruction templates there is shown a no memory access, full round control type operation 1210 instruction template and a no memory access, data transform type operation 1215 instruction template; and 2) within the memory access 1220 instruction templates there is shown a memory access, temporal 1225 instruction template and a memory access, non-temporal 1230 instruction template. The class B instruction templates in **Figure 12B** include: 1) within the no memory access 1205 instruction templates there is shown a no memory access, write mask control, partial round control type operation 1212 instruction template and a no memory access, write mask control, vsize type operation 1217 instruction template; and 2) within the memory access 1220 instruction templates there is shown a memory access, write mask control 1227 instruction template.

[00115] The generic vector friendly instruction format 1200 includes the following fields listed below in the order illustrated in **Figures 12A-12B**.

[00116] Format field 1240 – a specific value (an instruction format identifier value) in this field uniquely identifies the vector friendly instruction format, and thus occurrences of instructions in the vector friendly instruction format in instruction streams. As such, this field is optional in the sense that it is not needed for an instruction set that has only the generic vector friendly instruction format.

[00117] Base operation field 1242 – its content distinguishes different base operations.

[00118] Register index field 1244 – its content, directly or through address generation, specifies the locations of the source and destination operands, be they in registers or in memory. These include a sufficient number of bits to select N registers from a PxQ (e.g. 32x512, 16x128, 32x1024, 64x1024) register file. While in one embodiment N may be up to three sources and one destination register, alternative embodiments may support more or less sources and destination registers (e.g., may support up to two sources where one of these sources also acts as the destination, may support up to three sources where one of these sources also acts as the destination, may support up to two sources and one destination).

[00119] Modifier field 1246 – its content distinguishes occurrences of instructions in the generic vector instruction format that specify memory access from those that do not; that is,

between no memory access 1205 instruction templates and memory access 1220 instruction templates. Memory access operations read and/or write to the memory hierarchy (in some cases specifying the source and/or destination addresses using values in registers), while non-memory access operations do not (e.g., the source and destinations are registers). While in one embodiment this field also selects between three different ways to perform memory address calculations, alternative embodiments may support more, less, or different ways to perform memory address calculations.

[00120] Augmentation operation field 1250 – its content distinguishes which one of a variety of different operations to be performed in addition to the base operation. This field is context specific. In one embodiment of the invention, this field is divided into a class field 1268, an alpha field 1252, and a beta field 1254. The augmentation operation field 1250 allows common groups of operations to be performed in a single instruction rather than 2, 3, or 4 instructions.

[00121] Scale field 1260 – its content allows for the scaling of the index field's content for memory address generation (e.g., for address generation that uses $2^{\text{scale}} * \text{index} + \text{base}$).

[00122] Displacement Field 1262A – its content is used as part of memory address generation (e.g., for address generation that uses $2^{\text{scale}} * \text{index} + \text{base} + \text{displacement}$).

[00123] Displacement Factor Field 1262B (note that the juxtaposition of displacement field 1262A directly over displacement factor field 1262B indicates one or the other is used) – its content is used as part of address generation; it specifies a displacement factor that is to be scaled by the size of a memory access (N) – where N is the number of bytes in the memory access (e.g., for address generation that uses $2^{\text{scale}} * \text{index} + \text{base} + \text{scaled displacement}$). Redundant low-order bits are ignored and hence, the displacement factor field's content is multiplied by the memory operands total size (N) in order to generate the final displacement to be used in calculating an effective address. The value of N is determined by the processor hardware at runtime based on the full opcode field 1274 (described herein) and the data manipulation field 1254C. The displacement field 1262A and the displacement factor field 1262B are optional in the sense that they are not used for the no memory access 1205 instruction templates and/or different embodiments may implement only one or none of the two.

[00124] Data element width field 1264 – its content distinguishes which one of a number of data element widths is to be used (in some embodiments for all instructions; in other embodiments for only some of the instructions). This field is optional in the sense that it is not needed if only one data element width is supported and/or data element widths are supported

using some aspect of the opcodes.

[00125] Write mask field 1270 – its content controls, on a per data element position basis, whether that data element position in the destination vector operand reflects the result of the base operation and augmentation operation. Class A instruction templates support merging-writemasking, while class B instruction templates support both merging- and zeroing-writemasking. When merging, vector masks allow any set of elements in the destination to be protected from updates during the execution of any operation (specified by the base operation and the augmentation operation); in other one embodiment, preserving the old value of each element of the destination where the corresponding mask bit has a 0. In contrast, when zeroing vector masks allow any set of elements in the destination to be zeroed during the execution of any operation (specified by the base operation and the augmentation operation); in one embodiment, an element of the destination is set to 0 when the corresponding mask bit has a 0 value. A subset of this functionality is the ability to control the vector length of the operation being performed (that is, the span of elements being modified, from the first to the last one); however, it is not necessary that the elements that are modified be consecutive. Thus, the write mask field 1270 allows for partial vector operations, including loads, stores, arithmetic, logical, etc. While embodiments of the invention are described in which the write mask field's 1270 content selects one of a number of write mask registers that contains the write mask to be used (and thus the write mask field's 1270 content indirectly identifies that masking to be performed), alternative embodiments instead or additional allow the mask write field's 1270 content to directly specify the masking to be performed.

[00126] Immediate field 1272 – its content allows for the specification of an immediate. This field is optional in the sense that it is not present in an implementation of the generic vector friendly format that does not support immediate and it is not present in instructions that do not use an immediate.

[00127] Class field 1268 – its content distinguishes between different classes of instructions. With reference to **Figures 12A-B**, the contents of this field select between class A and class B instructions. In **Figures 12A-B**, rounded corner squares are used to indicate a specific value is present in a field (e.g., class A 1268A and class B 1268B for the class field 1268 respectively in Figures 12A-B).

Instruction Templates of Class A

[00128] In the case of the non-memory access 1205 instruction templates of class A, the alpha

field 1252 is interpreted as an RS field 1252A, whose content distinguishes which one of the different augmentation operation types are to be performed (e.g., round 1252A.1 and data transform 1252A.2 are respectively specified for the no memory access, round type operation 1210 and the no memory access, data transform type operation 1215 instruction templates), while the beta field 1254 distinguishes which of the operations of the specified type is to be performed. In the no memory access 1205 instruction templates, the scale field 1260, the displacement field 1262A, and the displacement scale field 1262B are not present.

[00129] No-Memory Access Instruction Templates – Full Round Control Type Operation

[00130] In the no memory access full round control type operation 1210 instruction template, the beta field 1254 is interpreted as a round control field 1254A, whose content(s) provide static rounding. While in the described embodiments of the invention the round control field 1254A includes a suppress all floating point exceptions (SAE) field 1256 and a round operation control field 1258, alternative embodiments may support may encode both these concepts into the same field or only have one or the other of these concepts/fields (e.g., may have only the round operation control field 1258).

[00131] SAE field 1256 – its content distinguishes whether or not to disable the exception event reporting; when the SAE field's 1256 content indicates suppression is enabled, a given instruction does not report any kind of floating-point exception flag and does not raise any floating point exception handler.

[00132] Round operation control field 1258 – its content distinguishes which one of a group of rounding operations to perform (e.g., Round-up, Round-down, Round-towards-zero and Round-to-nearest). Thus, the round operation control field 1258 allows for the changing of the rounding mode on a per instruction basis. In one embodiment of the invention where a processor includes a control register for specifying rounding modes, the round operation control field's 1250 content overrides that register value.

No Memory Access Instruction Templates – Data Transform Type Operation

[00133] In the no memory access data transform type operation 1215 instruction template, the beta field 1254 is interpreted as a data transform field 1254B, whose content distinguishes which one of a number of data transforms is to be performed (e.g., no data transform, swizzle, broadcast).

[00134] In the case of a memory access 1220 instruction template of class A, the alpha field 1252 is interpreted as an eviction hint field 1252B, whose content distinguishes which one of

the eviction hints is to be used (in Figure 12A, temporal 1252B.1 and non-temporal 1252B.2 are respectively specified for the memory access, temporal 1225 instruction template and the memory access, non-temporal 1230 instruction template), while the beta field 1254 is interpreted as a data manipulation field 1254C, whose content distinguishes which one of a number of data manipulation operations (also known as primitives) is to be performed (e.g., no manipulation; broadcast; up conversion of a source; and down conversion of a destination). The memory access 1220 instruction templates include the scale field 1260, and optionally the displacement field 1262A or the displacement scale field 1262B.

[00135] Vector memory instructions perform vector loads from and vector stores to memory, with conversion support. As with regular vector instructions, vector memory instructions transfer data from/to memory in a data element-wise fashion, with the elements that are actually transferred is dictated by the contents of the vector mask that is selected as the write mask.

[00136] Memory Access Instruction Templates – Temporal

[00137] Temporal data is data likely to be reused soon enough to benefit from caching. This is, however, a hint, and different processors may implement it in different ways, including ignoring the hint entirely.

Memory Access Instruction Templates – Non-Temporal

[00138] Non-temporal data is data unlikely to be reused soon enough to benefit from caching in the 1st-level cache and should be given priority for eviction. This is, however, a hint, and different processors may implement it in different ways, including ignoring the hint entirely.

Instruction Templates of Class B

[00139] In the case of the instruction templates of class B, the alpha field 1252 is interpreted as a write mask control (Z) field 1252C, whose content distinguishes whether the write masking controlled by the write mask field 1270 should be a merging or a zeroing.

[00140] In the case of the non-memory access 1205 instruction templates of class B, part of the beta field 1254 is interpreted as an RL field 1257A, whose content distinguishes which one of the different augmentation operation types are to be performed (e.g., round 1257A.1 and vector length (VSIZE) 1257A.2 are respectively specified for the no memory access, write mask control, partial round control type operation 1212 instruction template and the no memory access, write mask control, VSIZE type operation 1217 instruction template), while the rest of the beta field 1254 distinguishes which of the operations of the specified type is to be

performed. In the no memory access 1205 instruction templates, the scale field 1260, the displacement field 1262A, and the displacement scale field 1262B are not present.

[00141] In the no memory access, write mask control, partial round control type operation 1210 instruction template, the rest of the beta field 1254 is interpreted as a round operation field 1259A and exception event reporting is disabled (a given instruction does not report any kind of floating-point exception flag and does not raise any floating point exception handler).

[00142] Round operation control field 1259A – just as round operation control field 1258, its content distinguishes which one of a group of rounding operations to perform (e.g., Round-up, Round-down, Round-towards-zero and Round-to-nearest). Thus, the round operation control field 1259A allows for the changing of the rounding mode on a per instruction basis. In one embodiment of the invention where a processor includes a control register for specifying rounding modes, the round operation control field's 1250 content overrides that register value.

[00143] In the no memory access, write mask control, VSIZE type operation 1217 instruction template, the rest of the beta field 1254 is interpreted as a vector length field 1259B, whose content distinguishes which one of a number of data vector lengths is to be performed on (e.g., 128, 256, or 512 byte).

[00144] In the case of a memory access 1220 instruction template of class B, part of the beta field 1254 is interpreted as a broadcast field 1257B, whose content distinguishes whether or not the broadcast type data manipulation operation is to be performed, while the rest of the beta field 1254 is interpreted the vector length field 1259B. The memory access 1220 instruction templates include the scale field 1260, and optionally the displacement field 1262A or the displacement scale field 1262B.

[00145] With regard to the generic vector friendly instruction format 1200, a full opcode field 1274 is shown including the format field 1240, the base operation field 1242, and the data element width field 1264. While one embodiment is shown where the full opcode field 1274 includes all of these fields, the full opcode field 1274 includes less than all of these fields in embodiments that do not support all of them. The full opcode field 1274 provides the operation code (opcode).

[00146] The augmentation operation field 1250, the data element width field 1264, and the write mask field 1270 allow these features to be specified on a per instruction basis in the generic vector friendly instruction format.

[00147] The combination of write mask field and data element width field create typed instructions in that they allow the mask to be applied based on different data element widths.

[00148] The various instruction templates found within class A and class B are beneficial in different situations. In some embodiments of the invention, different processors or different cores within a processor may support only class A, only class B, or both classes. For instance, a high performance general purpose out-of-order core intended for general-purpose computing may support only class B, a core intended primarily for graphics and/or scientific (throughput) computing may support only class A, and a core intended for both may support both (of course, a core that has some mix of templates and instructions from both classes but not all templates and instructions from both classes is within the purview of the invention). Also, a single processor may include multiple cores, all of which support the same class or in which different cores support different class. For instance, in a processor with separate graphics and general purpose cores, one of the graphics cores intended primarily for graphics and/or scientific computing may support only class A, while one or more of the general purpose cores may be high performance general purpose cores with out of order execution and register renaming intended for general-purpose computing that support only class B. Another processor that does not have a separate graphics core, may include one more general purpose in-order or out-of-order cores that support both class A and class B. Of course, features from one class may also be implement in the other class in different embodiments of the invention. Programs written in a high level language would be put (e.g., just in time compiled or statically compiled) into a variety of different executable forms, including: 1) a form having only instructions of the class(es) supported by the target processor for execution; or 2) a form having alternative routines written using different combinations of the instructions of all classes and having control flow code that selects the routines to execute based on the instructions supported by the processor which is currently executing the code.

[00149] **Figures 13A-D** are block diagrams illustrating an exemplary specific vector friendly instruction format according to embodiments of the invention. **Figures 13A-D** show a specific vector friendly instruction format 1300 that is specific in the sense that it specifies the location, size, interpretation, and order of the fields, as well as values for some of those fields. The specific vector friendly instruction format 1300 may be used to extend the x86 instruction set, and thus some of the fields are similar or the same as those used in the existing x86 instruction set and extension thereof (e.g., AVX). This format remains consistent with the prefix encoding field, real opcode byte field, MOD R/M field, SIB field, displacement field, and immediate fields of the existing x86 instruction set with extensions. The fields from **Figure 12** into which the fields from **Figure 13** map are illustrated.

[00150] It should be understood that, although embodiments of the invention are described with reference to the specific vector friendly instruction format 1300 in the context of the generic vector friendly instruction format 1200 for illustrative purposes, the invention is not limited to the specific vector friendly instruction format 1300 except where claimed. For example, the generic vector friendly instruction format 1200 contemplates a variety of possible sizes for the various fields, while the specific vector friendly instruction format 1300 is shown as having fields of specific sizes. By way of specific example, while the data element width field 1264 is illustrated as a one bit field in the specific vector friendly instruction format 1300, the invention is not so limited (that is, the generic vector friendly instruction format 1200 contemplates other sizes of the data element width field 1264).

[00151] The generic vector friendly instruction format 1200 includes the following fields listed below in the order illustrated in **Figure 13A**.

EVEX Prefix (Bytes 0-3) 1302 - is encoded in a four-byte form.

[00152] Format Field 1240 (EVEX Byte 0, bits [7:0]) - the first byte (EVEX Byte 0) is the format field 1240 and it contains 0x62 (the unique value used for distinguishing the vector friendly instruction format in one embodiment of the invention).

[00153] The second-fourth bytes (EVEX Bytes 1-3) include a number of bit fields providing specific capability.

[00154] REX field 1305 (EVEX Byte 1, bits [7-5]) – consists of a EVEX.R bit field (EVEX Byte 1, bit [7] – R), EVEX.X bit field (EVEX byte 1, bit [6] – X), and 1257BEX byte 1, bit[5] – B). The EVEX.R, EVEX.X, and EVEX.B bit fields provide the same functionality as the corresponding VEX bit fields, and are encoded using 1s complement form, i.e. ZMM0 is encoded as 1211B, ZMM15 is encoded as 0000B. Other fields of the instructions encode the lower three bits of the register indexes as is known in the art (rrr, xxx, and bbb), so that Rrrr, Xxxx, and Bbbb may be formed by adding EVEX.R, EVEX.X, and EVEX.B.

[00155] REX' field 1210 – this is the first part of the REX' field 1210 and is the EVEX.R' bit field (EVEX Byte 1, bit [4] - R') that is used to encode either the upper 16 or lower 16 of the extended 32 register set. In one embodiment of the invention, this bit, along with others as indicated below, is stored in bit inverted format to distinguish (in the well-known x86 32-bit mode) from the BOUND instruction, whose real opcode byte is 62, but does not accept in the MOD R/M field (described below) the value of 11 in the MOD field; alternative embodiments of the invention do not store this and the other indicated bits below in the inverted format. A

value of 1 is used to encode the lower 16 registers. In other words, R'Rrrr is formed by combining EVEX.R', EVEX.R, and the other RRR from other fields.

[00156] Opcode map field 1315 (EVEX byte 1, bits [3:0] – mmmm) – its content encodes an implied leading opcode byte (0F, 0F 38, or 0F 3).

[00157] Data element width field 1264 (EVEX byte 2, bit [7] – W) - is represented by the notation EVEX.W. EVEX.W is used to define the granularity (size) of the datatype (either 32-bit data elements or 64-bit data elements).

[00158] EVEX.vvvv 1320 (EVEX Byte 2, bits [6:3]-vvvv)- the role of EVEX.vvvv may include the following: 1) EVEX.vvvv encodes the first source register operand, specified in inverted (1s complement) form and is valid for instructions with 2 or more source operands; 2) EVEX.vvvv encodes the destination register operand, specified in 1s complement form for certain vector shifts; or 3) EVEX.vvvv does not encode any operand, the field is reserved and should contain 1211b. Thus, EVEX.vvvv field 1320 encodes the 4 low-order bits of the first source register specifier stored in inverted (1s complement) form. Depending on the instruction, an extra different EVEX bit field is used to extend the specifier size to 32 registers.

[00159] EVEX.U 1268 Class field (EVEX byte 2, bit [2]-U) – If EVEX.U = 0, it indicates class A or EVEX.U0; if EVEX.U = 1, it indicates class B or EVEX.U1.

[00160] Prefix encoding field 1325 (EVEX byte 2, bits [1:0]-pp) – provides additional bits for the base operation field. In addition to providing support for the legacy SSE instructions in the EVEX prefix format, this also has the benefit of compacting the SIMD prefix (rather than requiring a byte to express the SIMD prefix, the EVEX prefix requires only 2 bits). In one embodiment, to support legacy SSE instructions that use a SIMD prefix (66H, F2H, F3H) in both the legacy format and in the EVEX prefix format, these legacy SIMD prefixes are encoded into the SIMD prefix encoding field; and at runtime are expanded into the legacy SIMD prefix prior to being provided to the decoder's PLA (so the PLA can execute both the legacy and EVEX format of these legacy instructions without modification). Although newer instructions could use the EVEX prefix encoding field's content directly as an opcode extension, certain embodiments expand in a similar fashion for consistency but allow for different meanings to be specified by these legacy SIMD prefixes. An alternative embodiment may redesign the PLA to support the 2 bit SIMD prefix encodings, and thus not require the expansion.

[00161] Alpha field 1252 (EVEX byte 3, bit [7] – EH; also known as EVEX.EH, EVEX.rs, EVEX.RL, EVEX.write mask control, and EVEX.N; also illustrated with α) – as previously described, this field is context specific.

[00162] Beta field 1254 (EVEX byte 3, bits [6:4]-SSS, also known as EVEX.s₂₋₀, EVEX.r₂₋₀, EVEX.rr1, EVEX.LL0, EVEX.LLB; also illustrated with $\beta\beta\beta$) – as previously described, this field is context specific.

[00163] REX' field 1210 – this is the remainder of the REX' field and is the EVEX.V' bit field (EVEX Byte 3, bit [3] - V') that may be used to encode either the upper 16 or lower 16 of the extended 32 register set. This bit is stored in bit inverted format. A value of 1 is used to encode the lower 16 registers. In other words, V'VVVV is formed by combining EVEX.V', EVEX.vvvv.

[00164] Write mask field 1270 (EVEX byte 3, bits [2:0]-kkk) – its content specifies the index of a register in the write mask registers as previously described. In one embodiment of the invention, the specific value EVEX.kkk=000 has a special behavior implying no write mask is used for the particular instruction (this may be implemented in a variety of ways including the use of a write mask hardwired to all ones or hardware that bypasses the masking hardware).

[00165] Real Opcode Field 1330 (Byte 4) is also known as the opcode byte. Part of the opcode is specified in this field.

[00166] MOD R/M Field 1340 (Byte 5) includes MOD field 1342, Reg field 1344, and R/M field 1346. As previously described, the MOD field's 1342 content distinguishes between memory access and non-memory access operations. The role of Reg field 1344 can be summarized to two situations: encoding either the destination register operand or a source register operand, or be treated as an opcode extension and not used to encode any instruction operand. The role of R/M field 1346 may include the following: encoding the instruction operand that references a memory address, or encoding either the destination register operand or a source register operand.

[00167] Scale, Index, Base (SIB) Byte (Byte 6) - As previously described, the scale field's 1250 content is used for memory address generation. SIB.xxx 1354 and SIB.bbb 1356 – the contents of these fields have been previously referred to with regard to the register indexes Xxxx and Bbbb.

[00168] Displacement field 1262A (Bytes 7-10) – when MOD field 1342 contains 10, bytes 7-10 are the displacement field 1262A, and it works the same as the legacy 32-bit displacement (disp32) and works at byte granularity.

[00169] Displacement factor field 1262B (Byte 7) – when MOD field 1342 contains 01, byte 7 is the displacement factor field 1262B. The location of this field is that same as that of the legacy x86 instruction set 8-bit displacement (disp8), which works at byte granularity. Since

disp8 is sign extended, it can only address between -128 and 127 bytes offsets; in terms of 64 byte cache lines, disp8 uses 8 bits that can be set to only four really useful values -128, -64, 0, and 64; since a greater range is often needed, disp32 is used; however, disp32 requires 4 bytes. In contrast to disp8 and disp32, the displacement factor field 1262B is a reinterpretation of disp8; when using displacement factor field 1262B, the actual displacement is determined by the content of the displacement factor field multiplied by the size of the memory operand access (N). This type of displacement is referred to as disp8*N. This reduces the average instruction length (a single byte of used for the displacement but with a much greater range). Such compressed displacement is based on the assumption that the effective displacement is multiple of the granularity of the memory access, and hence, the redundant low-order bits of the address offset do not need to be encoded. In other words, the displacement factor field 1262B substitutes the legacy x86 instruction set 8-bit displacement. Thus, the displacement factor field 1262B is encoded the same way as an x86 instruction set 8-bit displacement (so no changes in the ModRM/SIB encoding rules) with the only exception that disp8 is overloaded to disp8*N. In other words, there are no changes in the encoding rules or encoding lengths but only in the interpretation of the displacement value by hardware (which needs to scale the displacement by the size of the memory operand to obtain a byte-wise address offset).

[00170] Immediate field 1272 operates as previously described.

Full Opcode Field

[00171] **Figure 13B** is a block diagram illustrating the fields of the specific vector friendly instruction format 1300 that make up the full opcode field 1274 according to one embodiment of the invention. Specifically, the full opcode field 1274 includes the format field 1240, the base operation field 1242, and the data element width (W) field 1264. The base operation field 1242 includes the prefix encoding field 1325, the opcode map field 1315, and the real opcode field 1330.

Register Index Field

[00172] **Figure 13C** is a block diagram illustrating the fields of the specific vector friendly instruction format 1300 that make up the register index field 1244 according to one embodiment of the invention. Specifically, the register index field 1244 includes the REX field 1305, the REX' field 1310, the MODR/M.reg field 1344, the MODR/M.r/m field 1346, the VVVV field 1320, xxx field 1354, and the bbb field 1356.

Augmentation Operation Field

[00173] **Figure 13D** is a block diagram illustrating the fields of the specific vector friendly instruction format 1300 that make up the augmentation operation field 1250 according to one embodiment of the invention. When the class (U) field 1268 contains 0, it signifies EVEX.U0 (class A 1268A); when it contains 1, it signifies EVEX.U1 (class B 1268B). When U=0 and the MOD field 1342 contains 11 (signifying a no memory access operation), the alpha field 1252 (EVEX byte 3, bit [7] – EH) is interpreted as the rs field 1252A. When the rs field 1252A contains a 1 (round 1252A.1), the beta field 1254 (EVEX byte 3, bits [6:4]- SSS) is interpreted as the round control field 1254A. The round control field 1254A includes a one bit SAE field 1256 and a two bit round operation field 1258. When the rs field 1252A contains a 0 (data transform 1252A.2), the beta field 1254 (EVEX byte 3, bits [6:4]- SSS) is interpreted as a three bit data transform field 1254B. When U=0 and the MOD field 1342 contains 00, 01, or 10 (signifying a memory access operation), the alpha field 1252 (EVEX byte 3, bit [7] – EH) is interpreted as the eviction hint (EH) field 1252B and the beta field 1254 (EVEX byte 3, bits [6:4]- SSS) is interpreted as a three bit data manipulation field 1254C.

[00174] When U=1, the alpha field 1252 (EVEX byte 3, bit [7] – EH) is interpreted as the write mask control (Z) field 1252C. When U=1 and the MOD field 1342 contains 11 (signifying a no memory access operation), part of the beta field 1254 (EVEX byte 3, bit [4]- S₀) is interpreted as the RL field 1257A; when it contains a 1 (round 1257A.1) the rest of the beta field 1254 (EVEX byte 3, bit [6-5]- S₂₋₁) is interpreted as the round operation field 1259A, while when the RL field 1257A contains a 0 (VSIZE 1257A.2) the rest of the beta field 1254 (EVEX byte 3, bit [6-5]- S₂₋₁) is interpreted as the vector length field 1259B (EVEX byte 3, bit [6-5]- L₁₋₀). When U=1 and the MOD field 1342 contains 00, 01, or 10 (signifying a memory access operation), the beta field 1254 (EVEX byte 3, bits [6:4]- SSS) is interpreted as the vector length field 1259B (EVEX byte 3, bit [6-5]- L₁₋₀) and the broadcast field 1257B (EVEX byte 3, bit [4]- B).

[00175] **Figure 14** is a block diagram of a register architecture 1400 according to one embodiment of the invention. In the embodiment illustrated, there are 32 vector registers 1410 that are 512 bits wide; these registers are referenced as zmm0 through zmm31. The lower order 256 bits of the lower 16 zmm registers are overlaid on registers ymm0-16. The lower order 128 bits of the lower 16 zmm registers (the lower order 128 bits of the ymm registers) are overlaid on registers xmm0-15. The specific vector friendly instruction format 1300 operates on these overlaid register file as illustrated in the below tables.

Adjustable Vector Length	Class	Operations	Registers
Instruction Templates that do not include the vector length field 1259B	A (Figure 12A; U=0)	1210, 1215, 1225, 1230	zmm registers (the vector length is 64 byte)
	B (Figure 12B; U=1)	1212	zmm registers (the vector length is 64 byte)
Instruction Templates that do include the vector length field 1259B	B (Figure 12B; U=1)	1217, 1227	zmm, ymm, or xmm registers (the vector length is 64 byte, 32 byte, or 16 byte) depending on the vector length field 1259B

[00176] In other words, the vector length field 1259B selects between a maximum length and one or more other shorter lengths, where each such shorter length is half the length of the preceding length; and instructions templates without the vector length field 1259B operate on the maximum vector length. Further, in one embodiment, the class B instruction templates of the specific vector friendly instruction format 1300 operate on packed or scalar single/double-precision floating point data and packed or scalar integer data. Scalar operations are operations performed on the lowest order data element position in an zmm/ymm/xmm register; the higher order data element positions are either left the same as they were prior to the instruction or zeroed depending on the embodiment.

[00177] Write mask registers 1415 - in the embodiment illustrated, there are 8 write mask registers (k0 through k7), each 64 bits in size. In an alternate embodiment, the write mask registers 1415 are 16 bits in size. As previously described, in one embodiment of the invention, the vector mask register k0 cannot be used as a write mask; when the encoding that would normally indicate k0 is used for a write mask, it selects a hardwired write mask of 0xFFFF, effectively disabling write masking for that instruction.

[00178] General-purpose registers 1425 - in the embodiment illustrated, there are sixteen 64-bit general-purpose registers that are used along with the existing x86 addressing modes to address memory operands. These registers are referenced by the names RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, and R8 through R15.

[00179] Scalar floating point stack register file (x87 stack) 1445, on which is aliased the MMX packed integer flat register file 1450 - in the embodiment illustrated, the x87 stack is an eight-element stack used to perform scalar floating-point operations on 32/64/80-bit floating point data using the x87 instruction set extension; while the MMX registers are used to perform operations on 64-bit packed integer data, as well as to hold operands for some operations performed between the MMX and XMM registers.

[00180] Alternative embodiments of the invention may use wider or narrower registers. Additionally, alternative embodiments of the invention may use more, less, or different register files and registers.

[00181] **Figures 15A-B** illustrate a block diagram of a more specific exemplary in-order core architecture, which core would be one of several logic blocks (including other cores of the same type and/or different types) in a chip. The logic blocks communicate through a high-bandwidth interconnect network (e.g., a ring network) with some fixed function logic, memory I/O interfaces, and other necessary I/O logic, depending on the application.

[00182] **Figure 15A** is a block diagram of a single processor core, along with its connection to the on-die interconnect network 1502 and with its local subset of the Level 2 (L2) cache 1504, according to embodiments of the invention. In one embodiment, an instruction decoder 1500 supports the x86 instruction set with a packed data instruction set extension. An L1 cache 1506 allows low-latency accesses to cache memory into the scalar and vector units. While in one embodiment (to simplify the design), a scalar unit 1508 and a vector unit 1510 use separate register sets (respectively, scalar registers 1512 and vector registers 1514) and data transferred between them is written to memory and then read back in from a level 1 (L1) cache 1506, alternative embodiments of the invention may use a different approach (e.g., use a single register set or include a communication path that allow data to be transferred between the two register files without being written and read back).

[00183] The local subset of the L2 cache 1504 is part of a global L2 cache that is divided into separate local subsets, one per processor core. Each processor core has a direct access path to its own local subset of the L2 cache 1504. Data read by a processor core is stored in its L2 cache subset 1504 and can be accessed quickly, in parallel with other processor cores accessing

their own local L2 cache subsets. Data written by a processor core is stored in its own L2 cache subset 1504 and is flushed from other subsets, if necessary. The ring network ensures coherency for shared data. The ring network is bi-directional to allow agents such as processor cores, L2 caches and other logic blocks to communicate with each other within the chip. Each ring data-path is 1012-bits wide per direction.

[00184] **Figure 15B** is an expanded view of part of the processor core in **Figure 15A** according to embodiments of the invention. **Figure 15B** includes an L1 data cache 1506A part of the L1 cache 1504, as well as more detail regarding the vector unit 1510 and the vector registers 1514. Specifically, the vector unit 1510 is a 16-wide vector processing unit (VPU) (see the 16-wide ALU 1528), which executes one or more of integer, single-precision float, and double-precision float instructions. The VPU supports swizzling the register inputs with swizzle unit 1520, numeric conversion with numeric convert units 1522A-B, and replication with replication unit 1524 on the memory input. Write mask registers 1526 allow predicating resulting vector writes.

[00185] Embodiments of the invention may include various steps, which have been described above. The steps may be embodied in machine-executable instructions which may be used to cause a general-purpose or special-purpose processor to perform the steps. Alternatively, these steps may be performed by specific hardware components that contain hardwired logic for performing the steps, or by any combination of programmed computer components and custom hardware components.

[00186] As described herein, instructions may refer to specific configurations of hardware such as application specific integrated circuits (ASICs) configured to perform certain operations or having a predetermined functionality or software instructions stored in memory embodied in a non-transitory computer readable medium. Thus, the techniques shown in the figures can be implemented using code and data stored and executed on one or more electronic devices (e.g., an end station, a network element, etc.). Such electronic devices store and communicate (internally and/or with other electronic devices over a network) code and data using computer machine-readable media, such as non-transitory computer machine-readable storage media (e.g., magnetic disks; optical disks; random access memory; read only memory; flash memory devices; phase-change memory) and transitory computer machine-readable communication media (e.g., electrical, optical, acoustical or other form of propagated signals – such as carrier waves, infrared signals, digital signals, etc.). In addition, such electronic devices typically include a set of one or more processors coupled to one or more other components, such as one

or more storage devices (non-transitory machine-readable storage media), user input/output devices (e.g., a keyboard, a touchscreen, and/or a display), and network connections. The coupling of the set of processors and other components is typically through one or more busses and bridges (also termed as bus controllers). The storage device and signals carrying the network traffic respectively represent one or more machine-readable storage media and machine-readable communication media. Thus, the storage device of a given electronic device typically stores code and/or data for execution on the set of one or more processors of that electronic device. Of course, one or more parts of an embodiment of the invention may be implemented using different combinations of software, firmware, and/or hardware. Throughout this detailed description, for the purposes of explanation, numerous specific details were set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the invention may be practiced without some of these specific details. In certain instances, well known structures and functions were not described in elaborate detail in order to avoid obscuring the subject matter of the present invention. Accordingly, the scope and spirit of the invention should be judged in terms of the claims which follow.

[00187] An embodiment of the invention includes a processor comprising fetch logic to fetch an instruction from memory indicating a destination packed data operand, a first source packed data operand, a second source packed data operand, and an immediate value; and execution logic to determine a bit in the second source packed data operand based a position corresponding to the immediate value; perform a bitwise AND between the first source packed data operand and the determined bit to generate an intermediate result; perform a bitwise XOR between the destination packed data operand and the intermediate result to generate a final result; and store the final result in a storage location indicated by the destination packed data operand.

[00188] An additional embodiment includes, wherein to perform the bitwise AND between the first source packed data operand and the determined bit, the execution logic is further configured to perform the bitwise AND between the first source packed data operand and a temporary vector, wherein the value of the determined bit is to be broadcasted one or more times to the temporary vector.

[00189] An additional embodiment includes, wherein the storage locations indicated by the destination packed data operand, the first source packed data operand, and the second source packed data operand are to be processed in separate 64 bit sections, wherein the processor is to

execute the same logic for each of the 64 bit sections.

[00190] An additional embodiment includes, wherein the instruction further includes a writemask operand, and wherein the execution logic is to further set the values for the one of the 64-bit sections in the storage location indicated by the destination packed data operand to zero responsive to determining that the writemask operand indicates that a writemask is set for one of the 64 bit sections in the destination packed data operand.

[00191] An additional embodiment includes, wherein the storage locations indicated by the destination packed data operand, the first source packed data operand, and the second source packed data operand are at least one of a register and a memory location.

[00192] An additional embodiment includes, wherein the storage locations indicated by the destination packed data operand, the first source packed data operand, and the second source packed data operand are registers that are 512 bits long.

[00193] An additional embodiment includes, wherein the immediate value is 8 bits long.

[00194] An additional embodiment includes, wherein the instruction is used to perform a bit matrix multiplication operation between a bit matrix and a bit vector, wherein one or more columns of the bit matrix are stored in the storage location indicated by the first source packed data operand, and wherein values of the bit vector are stored in the storage location indicated by the second source packed data operand.

[00195] An embodiment of the invention includes, wherein the bit matrix is transposed such that the one or more columns of the bit matrix are stored column by column in the storage location indicated by the first source packed data operand.

[00196] An additional embodiment includes, wherein the storage location indicated by the destination packed data operand includes the result of the bit matrix multiplication operation between the bit matrix and the bit vector when the instruction is executed for each of the columns of the bit matrix, wherein for each execution of the instruction, the immediate value specifies a value that indicates a position in the bit vector corresponding to the column number of the bit matrix that is processed.

[00197] An embodiment of the invention includes a method in a computer processor comprising fetching an instruction from memory indicating a destination packed data operand, a first source packed data operand, a second source packed data operand, and an immediate value; determining a bit in the second source packed data operand based a position corresponding to the immediate value; performing a bitwise AND between the first source packed data operand and the determined bit to generate an intermediate result; performing a

bitwise XOR between the destination packed data operand and the intermediate result to generate a final result; and storing the final result in a storage location indicated by the destination packed data operand.

[00198] An additional embodiment includes, wherein the performing the bitwise AND between the first source packed data operand and the determined bit further includes performing the bitwise AND between the first source packed data operand and a temporary vector, wherein the value of the determined bit is to be broadcasted one or more times to the temporary vector.

[00199] An additional embodiment includes, wherein the storage locations indicated by the destination packed data operand, the first source packed data operand, and the second source packed data operand are to be processed in separate 64 bit sections, wherein the processor is to execute the same logic for each of the 64 bit sections.

[00200] An additional embodiment includes, wherein the instruction further includes a writemask operand, and wherein the method further comprises setting the values for the one of the 64-bit sections in the storage location indicated by the destination packed data operand to zero responsive to determining that the writemask operand indicates that a writemask is set for one of the 64 bit sections in the destination packed data operand.

[00201] An additional embodiment includes, wherein the storage locations indicated by the destination packed data operand, the first source packed data operand, and the second source packed data operand are at least one of a register and a memory location.

[00202] An additional embodiment includes, wherein the storage locations indicated by the destination packed data operand, the first source packed data operand, and the second source packed data operand are registers that are 512 bits long.

[00203] An additional embodiment includes, wherein the immediate value is 8 bits long.

[00204] An additional embodiment includes, wherein the instruction is used to perform a bit matrix multiplication operation between a bit matrix and a bit vector, wherein one or more columns of the bit matrix are stored in the storage location indicated by the first source packed data operand, and wherein values of the bit vector are stored in the storage location indicated by the second source packed data operand.

[00205] An embodiment of the invention includes, wherein the bit matrix is transposed such that the one or more columns of the bit matrix are stored column by column in the storage location indicated by the first source packed data operand.

[00206] An additional embodiment includes, wherein the storage location indicated by the

destination packed data operand includes the result of the bit matrix multiplication operation between the bit matrix and the bit vector when the instruction is executed for each of the columns of the bit matrix, wherein for each execution of the instruction, the immediate value specifies a value that indicates a position in the bit vector corresponding to the column number of the bit matrix that is processed.

[00207] While the invention has been described in terms of several embodiments, those skilled in the art will recognize that the invention is not limited to the embodiments described, can be practiced with modification and alteration within the spirit and scope of the appended claims. The description is thus to be regarded as illustrative instead of limiting.

CLAIMS

We claim:

1. A processor comprising:
fetch logic to fetch an instruction from memory indicating a destination packed data operand, a first source packed data operand, a second source packed data operand, and an immediate value; and
execution logic to:
determine a bit in the second source packed data operand based a position corresponding to the immediate value;
perform a bitwise AND between the first source packed data operand and the determined bit to generate an intermediate result;
perform a bitwise XOR between the destination packed data operand and the intermediate result to generate a final result; and
store the final result in a storage location indicated by the destination packed data operand.
2. The processor of claim 1, wherein to perform the bitwise AND between the first source packed data operand and the determined bit, the execution logic is further configured to perform the bitwise AND between the first source packed data operand and a temporary vector, wherein the value of the determined bit is to be broadcasted one or more times to the temporary vector.
3. The processor of claim 1, wherein the storage locations indicated by the destination packed data operand, the first source packed data operand, and the second source packed data operand are to be processed in separate 64 bit sections, wherein the processor is to execute the same logic for each of the 64 bit sections.
4. The processor of claim 3, wherein the instruction further includes a writemask operand, and wherein the execution logic is to further set the values for the one of the 64-bit sections in the storage location indicated by the destination packed data operand to zero responsive to determining that the writemask operand indicates that a writemask is set for one of the 64 bit sections in the destination packed data operand.

5. The processor of claim 1, wherein the storage locations indicated by the destination packed data operand, the first source packed data operand, and the second source packed data operand are at least one of a register and a memory location.
6. The processor of claim 5, wherein the storage locations indicated by the destination packed data operand, the first source packed data operand, and the second source packed data operand are registers that are 512 bits long.
7. The processor of claim 5, wherein the immediate value is 8 bits long.
8. The processor of claim 1, wherein the instruction is used to perform a bit matrix multiplication operation between a bit matrix and a bit vector, wherein one or more columns of the bit matrix are stored in the storage location indicated by the first source packed data operand, and wherein values of the bit vector are stored in the storage location indicated by the second source packed data operand.
9. The processor of claim 8, wherein the bit matrix is transposed such that the one or more columns of the bit matrix are stored column by column in the storage location indicated by the first source packed data operand.
10. The processor of claim 9, wherein the storage location indicated by the destination packed data operand includes the result of the bit matrix multiplication operation between the bit matrix and the bit vector when the instruction is executed for each of the columns of the bit matrix, wherein for each execution of the instruction, the immediate value specifies a value that indicates a position in the bit vector corresponding to the column number of the bit matrix that is processed.
11. A method in a computer processor, comprising:
 - fetching an instruction from memory indicating a destination packed data operand, a first source packed data operand, a second source packed data operand, and an immediate operand;
 - determining a bit in the second source packed data operand based a position corresponding to the immediate value;

performing a bitwise AND between the first source packed data operand and the determined bit to generate an intermediate result;
performing a bitwise XOR between the destination packed data operand and the intermediate result to generate a final result; and
storing the final result in a storage location indicated by the destination packed data operand.

12. The method of claim 11, wherein the performing the bitwise AND between the first source packed data operand and the determined bit further includes performing the bitwise AND between the first source packed data operand and a temporary vector, wherein the value of the determined bit is to be broadcasted one or more times to the temporary vector.

13. The method of claim 11, wherein the storage locations indicated by the destination packed data operand, the first source packed data operand, and the second source packed data operand are to be processed in separate 64 bit sections, wherein the processor is to execute the same logic for each of the 64 bit sections.

14. The method of claim 13, wherein the instruction further includes a writemask operand, and wherein the method further comprises setting the values for the one of the 64-bit sections in the storage location indicated by the destination packed data operand to zero responsive to determining that the writemask operand indicates that a writemask is set for one of the 64 bit sections in the destination packed data operand.

15. The method of claim 11, wherein the storage locations indicated by the destination packed data operand, the first source packed data operand, and the second source packed data operand are at least one of a register and a memory location.

16. The method of claim 15, wherein the storage locations indicated by the destination packed data operand, the first source packed data operand, and the second source packed data operand are registers that are 512 bits long.

17. The method of claim 15, wherein the immediate value is 8 bits long.

18. The method of claim 11, wherein the instruction is used to perform a bit matrix multiplication operation between a bit matrix and a bit vector, wherein one or more columns of

the bit matrix are stored in the storage location indicated by the first source packed data operand, and wherein values of the bit vector are stored in the storage location indicated by the second source packed data operand.

19. The method of claim 18, wherein the bit matrix is transposed such that the one or more columns of the bit matrix are stored column by column in the storage location indicated by the first source packed data operand.

20. The method of claim 19, wherein the storage location indicated by the destination packed data operand includes the result of the bit matrix multiplication operation between the bit matrix and the bit vector when the instruction is executed for each of the columns of the bit matrix, wherein for each execution of the instruction, the immediate value specifies a value that indicates a position in the bit vector corresponding to the column number of the bit matrix that is processed.