

INSTRUCTION AND LOGIC TO PERFORM A VECTOR SATURATED DOUBLEWORD/QUADWORD ADD

FIELD OF THE INVENTION

[0001] The present disclosure pertains to the field of processing logic, microprocessors, and associated instruction set architecture that, when executed by the processor or other processing logic, perform logical, mathematical, or other functional operations.

DESCRIPTION OF RELATED ART

[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 (quadword (Q) size data elements), eight separate 32-bit packed data elements (doubleword (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).

DESCRIPTION OF THE FIGURES

[0003] Embodiments are illustrated by way of example and not limitation in the Figures of the accompanying drawings, in which:

[0004] **FIG. 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;

[0005] **FIG. 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;

[0006] **FIG. 2A-B** are block diagrams of a more specific exemplary in-order core architecture

[0007] **FIG. 3** is a block diagram of a single core processor and a multicore processor with integrated memory controller and special purpose logic;

[0008] **FIG. 4** illustrates a block diagram of a system in accordance with an embodiment;

[0009] **FIG. 5** illustrates a block diagram of a second system in accordance with an embodiment;

[0010] **FIG. 6** illustrates a block diagram of a third system in accordance with an embodiment;

[0011] **FIG. 7** illustrates a block diagram of a system on a chip (SoC) in accordance with an embodiment;

[0012] **FIG. 8** 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;

[0013] **FIG. 9** is a block diagram illustrating a write masked vector add, according to an embodiment;

[0014] **FIG. 10** is a block diagram of exemplary processor logic to perform instructions according to embodiments described herein;

[0015] **FIGS. 11** is a block diagram of a processing system including instructions to perform a vector saturated add, according to an embodiment;

[0016] **FIGS. 12** is a flow diagram of logic to perform instructions according to embodiments described herein

[0017] **FIG. 13A-13B** are block diagrams illustrating a generic vector friendly instruction format and instruction templates thereof according to an embodiment;

[0018] **FIG. 14A-B** are block diagrams illustrating an exemplary specific vector friendly instruction format according to an embodiment; and

[0019] **FIG. 14C** is a block diagram illustrating the fields of the specific vector friendly instruction format that make up the register index field according to one embodiment;

[0020] **FIG. 14D** is a block diagram illustrating the fields of the specific vector friendly instruction format that make up the augmentation operation field according to one embodiment;

[0021] **FIG. 15** is a block diagram of a register architecture 1500 according to one embodiment.

DETAILED DESCRIPTION

[0022] 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, September 2014; and see Intel® Intel® Architecture Instruction Set Extensions Programming Reference, September 2014). Architectural extensions are described which extend the Intel Architecture (IA). However, the underlying principles are not limited to any particular ISA.

[0023] In one embodiment, a processing device implements a set of instructions to perform a saturated doubleword or quadword add operation. In one embodiment a vector saturated add instructions performs a parallel add on corresponding elements of two vector registers indicated by a first and second operand and writes the results to a third vector register indicated by a third operand. In one embodiment a scalar doubleword or quadword data element may be added to each element of a vector register. In one embodiment when an individual result is beyond the range of the target data element, a saturation value is written to the destination operand for the out of range data element.

[0024] Described below are processor core architectures followed by descriptions of exemplary processors and computer architectures according to embodiments described herein. Numerous specific details are set forth in order to provide a thorough understanding of the embodiments of the invention described below. It will be apparent, however, to one skilled in the art that the embodiments may be practiced without some of these specific details. In other instances, well-known structures and devices are shown in block diagram form to avoid obscuring the underlying principles of the various embodiments.

[0025] Processor cores may be implemented in different ways, for different purposes, and in different processors. For instance, implementations of such cores may include: 1) a general purpose in-order core intended for general-purpose computing; 2) a high performance general purpose out-of-order core intended for general-purpose computing; 3) a special purpose core

intended primarily for graphics and/or scientific (throughput) computing. Processors may be implemented using a single processor core or can include a multiple processor cores. The processor cores within the processor may be homogenous or heterogeneous in terms of architecture instruction set.

[0026] Implementations of different processors include: 1) a central processor including one or more general purpose in-order cores for general-purpose computing and/or one or more general purpose out-of-order cores intended for general-purpose computing; and 2) a coprocessor including one or more special purpose cores intended primarily for graphics and/or scientific (e.g., many integrated core processors). Such different processors lead to different computer system architectures including: 1) the coprocessor on a separate chip from the central system processor; 2) the coprocessor on a separate die, but in the same package as the central system processor; 3) the coprocessor on the same die as other processor cores (in which case, such a coprocessor is sometimes referred to as special purpose logic, such as integrated graphics and/or scientific (throughput) logic, or as special purpose cores); and 4) a system on a chip that may include on the same die the described processor (sometimes referred to as the application core(s) or application processor(s)), the above described coprocessor, and additional functionality.

Exemplary Core Architectures

In-order and out-of-order core block diagram

[0027] **Figure 1A** is a block diagram illustrating an exemplary in-order pipeline and an exemplary register renaming out-of-order issue/execution pipeline, according to an embodiment. **Figure 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 an embodiment. The solid lined boxes in Figures 1A-B illustrate the in-order pipeline and in-order core, while the optional addition of the dashed lined boxes illustrates the register renaming, out-of-order issue/execution pipeline and core. Given that the in-order aspect is a subset of the out-of-order aspect, the out-of-order aspect will be described.

[0028] 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.

[0029] 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.

[0030] 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.

[0031] 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.

[0032] 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.

[0033] 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.

[0034] 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 Cambridge, England), 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, etc.), allowing the operations used by many multimedia applications to be performed using packed data.

[0035] 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® Hyper-Threading Technology).

[0036] 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.

Specific Exemplary In-Order Core Architecture

[0037] **Figures 2A-B** are block diagrams 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.

[0038] Figure 2A is a block diagram of a single processor core, along with its connection to the on-die interconnect network 202 and with its local subset of the Level 2 (L2) cache 204, according to an embodiment. In one embodiment, an instruction decoder 200 supports the x86 instruction set with a packed data instruction set extension. An L1 cache 206 allows low-latency accesses to cache memory into the scalar and vector units. While in one embodiment (to simplify the design), a scalar unit 208 and a vector unit 210 use separate register sets (respectively, scalar registers 212 and vector registers 214) and data transferred between them is written to memory and then read back in from a level 1 (L1) cache 206, alternative embodiments 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).

[0039] The local subset of the L2 cache 204 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 204. Data read by a processor core is stored in its L2 cache subset 204 and can be accessed quickly and 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 204 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.

[0040] Figure 2B is an expanded view of part of the processor core in Figure 2A according to an embodiment. Figure 2B includes an L1 data cache 206A part of the L1 cache 204, as well as more detail regarding the vector unit 210 and the vector registers 214. Specifically, the vector unit 210 is a 16-wide vector-processing unit (VPU) (see the 16-wide ALU 228), 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 220, numeric conversion with numeric convert units 222A-B, and replication with replication unit 224 on the memory input. Write mask registers 226 allow predicating resulting vector writes.

Processor with integrated memory controller and special purpose logic

[0041] **Figure 3** is a block diagram of a processor 300 that may have more than one core, may have an integrated memory controller, and may have integrated graphics according to an embodiment. The solid lined boxes in Figure 3 illustrate a processor 300 with a single core 302A, a system agent 310, a set of one or more bus controller units 316, while the optional addition of the dashed lined boxes illustrates an alternative processor 300 with multiple cores 302A-N, a set of one or more integrated memory controller unit(s) 314 in the system agent unit 310, and special purpose logic 308.

[0042] Thus, different implementations of the processor 300 may include: 1) a CPU with the special purpose logic 308 being integrated graphics and/or scientific (throughput) logic (which may include one or more cores), and the cores 302A-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 302A-N being a large number of special purpose cores intended primarily for graphics and/or scientific (throughput); and 3) a coprocessor with the cores 302A-N being a large number of general purpose in-order cores. Thus, the processor 300 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 300 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.

[0043] The memory hierarchy includes one or more levels of cache within the cores, a set or one or more shared cache units 306, and external memory (not shown) coupled to the set of integrated memory controller units 314. The set of shared cache units 306 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 312 interconnects the integrated graphics logic 308, the set of shared cache units 306, and the system agent unit 310/integrated memory controller unit(s) 314, 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 306 and cores 302-A-N.

[0044] In some embodiments, one or more of the cores 302A-N are capable of multi-threading. The system agent 310 includes those components coordinating and operating cores 302A-N. The system agent unit 310 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 302A-N and the integrated graphics logic 308. The display unit is for driving one or more externally connected displays.

[0045] The cores 302A-N may be homogenous or heterogeneous in terms of architecture instruction set; that is, two or more of the cores 302A-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.

Exemplary Computer Architectures

[0046] **Figures 4-7** 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.

[0047] **Figure 4** shows a block diagram of a system 400 in accordance with an embodiment. The system 400 may include one or more processors 410, 415, which are coupled to a controller hub 420. In one embodiment the controller hub 420 includes a graphics memory controller hub (GMCH) 490 and an Input/Output Hub (IOH) 450 (which may be on separate chips); the GMCH 490 includes memory and graphics controllers to which are coupled memory 440 and a coprocessor 445; the IOH 450 is couples input/output (I/O) devices 460 to the GMCH 490. Alternatively, one or both of the memory and graphics controllers are integrated within the processor (as described herein), the memory 440 and the coprocessor 445 are coupled directly to the processor 410, and the controller hub 420 in a single chip with the IOH 450.

[0048] The optional nature of additional processors 415 is denoted in Figure 4 with broken lines. Each processor 410, 415 may include one or more of the processing cores described herein and may be some version of the processor 300.

[0049] The memory 440 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 420 communicates with the processor(s) 410, 415 via a multi-drop bus, such as a frontside bus (FSB), point-to-point interface such as QuickPath Interconnect (QPI), or similar connection 495.

[0050] In one embodiment, the coprocessor 445 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 420 may include an integrated graphics accelerator.

[0051] There can be a variety of differences between the physical resources 410, 415 in terms of a spectrum of metrics of merit including architectural, microarchitectural, thermal, power consumption characteristics, and the like.

[0052] In one embodiment, the processor 410 executes instructions that control data processing operations of a general type. Embedded within the instructions may be coprocessor instructions. The processor 410 recognizes these coprocessor instructions as being of a type that

should be executed by the attached coprocessor 445. Accordingly, the processor 410 issues these coprocessor instructions (or control signals representing coprocessor instructions) on a coprocessor bus or other interconnect, to coprocessor 445. Coprocessor(s) 445 accept and execute the received coprocessor instructions.

[0053] **Figure 5** shows a block diagram of a first more specific exemplary system 500 in accordance with an embodiment. As shown in Figure 5, multiprocessor system 500 is a point-to-point interconnect system, and includes a first processor 570 and a second processor 580 coupled via a point-to-point interconnect 550. Each of processors 570 and 580 may be some version of the processor 300. In one embodiment of the invention, processors 570 and 580 are respectively processors 410 and 415, while coprocessor 538 is coprocessor 445. In another embodiment, processors 570 and 580 are respectively processor 410 coprocessor 445.

[0054] Processors 570 and 580 are shown including integrated memory controller (IMC) units 572 and 582, respectively. Processor 570 also includes as part of its bus controller units point-to-point (P-P) interfaces 576 and 578; similarly, second processor 580 includes P-P interfaces 586 and 588. Processors 570, 580 may exchange information via a point-to-point (P-P) interface 550 using P-P interface circuits 578, 588. As shown in Figure 5, IMCs 572 and 582 couple the processors to respective memories, namely a memory 532 and a memory 534, which may be portions of main memory locally attached to the respective processors.

[0055] Processors 570, 580 may each exchange information with a chipset 590 via individual P-P interfaces 552, 554 using point to point interface circuits 576, 594, 586, 598. Chipset 590 may optionally exchange information with the coprocessor 538 via a high-performance interface 539. In one embodiment, the coprocessor 538 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.

[0056] 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.

[0057] Chipset 590 may be coupled to a first bus 516 via an interface 596. In one

embodiment, first bus 516 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.

[0058] As shown in Figure 5, various I/O devices 514 may be coupled to first bus 516, along with a bus bridge 518 that couples first bus 516 to a second bus 520. In one embodiment, one or more additional processor(s) 515, 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 516. In one embodiment, second bus 520 may be a low pin count (LPC) bus. Various devices may be coupled to a second bus 520 including, for example, a keyboard and/or mouse 522, communication devices 527 and a storage unit 528 such as a disk drive or other mass storage device that may include instructions/code and data 530, in one embodiment. Further, an audio I/O 524 may be coupled to the second bus 520. Note that other architectures are possible. For example, instead of the point-to-point architecture of Figure 5, a system may implement a multi-drop bus or other such architecture.

[0059] **Figure 6** shows a block diagram of a second more specific exemplary system 600 in accordance with an embodiment. Like elements in Figures 5 and 6 bear like reference numerals, and certain aspects of Figure 5 have been omitted from Figure 6 in order to avoid obscuring other aspects of Figure 6.

[0060] Figure 6 illustrates that the processors 570, 580 may include integrated memory and I/O control logic ("CL") 572 and 582, respectively. Thus, the CL 572, 582 include integrated memory controller units and include I/O control logic. Figure 6 illustrates that not only are the memories 532, 534 coupled to the CL 572, 582, but also that I/O devices 614 are also coupled to the control logic 572, 582. Legacy I/O devices 615 are coupled to the chipset 590.

[0061] **Figure 7** shows a block diagram of a SoC 700 in accordance with an embodiment. Similar elements in Figure 3 bear like reference numerals. Also, dashed lined boxes are optional features on more advanced SoCs. In Figure 7, an interconnect unit(s) 702 is coupled to: an application processor 710 which includes a set of one or more cores 202A-N and shared cache unit(s) 306; a system agent unit 310; a bus controller unit(s) 316; an integrated memory

controller unit(s) 314; a set or one or more coprocessors 720 which may include integrated graphics logic, an image processor, an audio processor, and a video processor; an static random access memory (SRAM) unit 730; a direct memory access (DMA) unit 732; and a display unit 740 for coupling to one or more external displays. In one embodiment, the coprocessor(s) 720 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.

[0062] Embodiments of the mechanisms disclosed herein are implemented in hardware, software, firmware, or a combination of such implementation approaches. Embodiments are 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.

[0063] Program code, such as code 530 illustrated in Figure 5, 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.

[0064] 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.

[0065] One or more aspects of at least one embodiment may be implemented by representative data 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 (“tape”) and supplied to various customers or manufacturing facilities to load into the fabrication machines that actually make the logic or processor. For

example, IP cores, such as processors developed by ARM Holdings, Ltd. and the Institute of Computing Technology (ICT) of the Chinese Academy of Sciences may be licensed or sold to various customers or licensees and implemented in processors produced by these customers or licensees.

[0066] 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), rewritable compact disks (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.

[0067] Accordingly, embodiments 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.

Emulation (including binary translation, code morphing, etc.)

[0068] 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.

[0069] **Figure 8** 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 an embodiment. 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 8 shows a program in a high level language 802 may be compiled using an x86 compiler 804 to generate x86 binary code 806 that may be natively executed by a processor with at least one x86 instruction set core 816.

[0070] The processor with at least one x86 instruction set core 816 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 804 represents a compiler that is operable to generate x86 binary code 806 (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 816.

Similarly, Figure 8 shows the program in the high level language 802 may be compiled using an alternative instruction set compiler 808 to generate alternative instruction set binary code 810 that may be natively executed by a processor without at least one x86 instruction set core 814 (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 Cambridge, England).

[0071] The instruction converter 812 is used to convert the x86 binary code 806 into code that may be natively executed by the processor without an x86 instruction set core 814. This converted code is not likely to be the same as the alternative instruction set binary code 810 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 812 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 806.

Vector Saturated Doubleword/Quadword Add Instructions

[0072] Saturation arithmetic enhances efficiency of many data processing algorithms, particularly in digital signal processing applications. Saturated additions are common in many algorithms. However, an expensive instruction sequence is required to implement saturation arithmetic using existing instructions. In several embodiments, vector extensions to an instruction set architecture include instructions to perform saturated signed and unsigned integer additions. In one embodiment, a vector signed integer add with signed saturation is provided. In one embodiment, a vector unsigned integer add with unsigned saturation is provided. In one embodiment, packed doubleword and quadword integers are supported for both signed and unsigned instructions.

[0073] For example, a vector packed add signed doubleword (e.g., VPADDSD) instruction causes a processor to perform a SIMD add of the packed signed doubleword integers with saturation from the first source operand and second source operand. The processor then stores packed integer results in a destination operand. When an individual doubleword result is beyond the range of a signed doubleword integer (that is, greater than 0x7FFFFFFF or less than 0x80000000), a saturation value of 0x7FFFFFFF or 0x80000000, respectively, is written to the destination operand. The quadword signed instruction (e.g., VPADDSQ) and the unsigned versions (for doubleword and quadword, e.g., VPADDUSD, VPADDUSQ respectively) work in a similar way with an unsigned and/or quadword saturation value. In one embodiment, vector registers of 128-bits, 256-bits, and 512-bits are supported, with 4, 8, or 16 vector elements supported for a doubleword instruction and 2, 4, or 8 vector elements supported for a quadword instruction.

[0074] **Figure 9** is a block diagram illustrating a write masked vector add, according to an embodiment. In one embodiment, a write mask register K₁ 910 which, on a per data element position basis, controls whether that data element position in the destination vector operand reflects the result of the instruction operation. Based on the write masking configuration, each data element position in a destination operand (e.g., DEST operand 907) contains either the output of the sum of corresponding data elements of a vector register identified by a first source operand (e.g., SRC1 operand 901) and a second source operand (e.g., SRC2 operand 902). For

example, destination element zero 910a has an associated write mask value of one, and receives the result of the sum of element zero of the SRC1 operand 901 (e.g., 0x9) and element zero of the SRC2 operand 902 (e.g., 0x8). Destination element one 910b has an associated write mask value of zero and, based on the write mask configuration, is zero masked as illustrated, or the original value of the element is unchanged. While both SRC1 operand 901 and SRC2 operand 902 are illustrated as vectors, in one embodiment the SRC2 of the instruction may be a memory address storing a scalar integer value that is to be added to each element of the vector register specified by the SRC1 operand 901.

[0075] **Figure 10** is a block diagram of exemplary processor logic to perform instructions according to embodiments described herein. According to embodiments, vector addition logic 1006 includes a first source register (e.g., SRC1 register 1001), a second source register (e.g., SRC2 register 1002) and a destination register (e.g., DEST register 1007). In one embodiment the SRC1 register 1002 contains exemplary source vector A, while the SRC2 register 1002 contains exemplary source vector B. The sum of corresponding vector elements are computed and at least some of those elements may be used to produce exemplary vector C, which, in one embodiment is output to the DEST register 1007. In one embodiment the first source register contains source vector A, while the second source register contains a scalar value B which was fetched from a specified memory location (e.g., an address specified by SRC2 of the instruction). The scalar value may be stored in a general-purpose register or broadcast to multiple elements of a vector register, according to embodiments. Saturation logic 1008 is included in the vector addition logic 1006 to substitute out of range results with the appropriate saturation value (e.g., minimum or maximum values, either signed or unsigned).

[0076] In the specific example shown in Figure 10, the SRC1 register 1001, SRC2 register 1002, and the DEST register 1007 are each 128-bits. However, the underlying principles of the embodiments described herein are not limited as such, and additional register sizes including 256 and 512 bits may be used in varying embodiments. In one embodiment, a mask bit may also be specified within a mask data structure 1010 for each of the destination register data elements. If the mask bit associated with a particular data element in the destination register is set to true (e.g., one), then the vector addition logic 1006 outputs a sum of associated data elements. If the

mask bit is set to false (e.g., zero), then in one embodiment, the vector addition logic 1006 writes zeroes to the associated destination register entry. The foregoing technique of writing zeroes to a destination data element in response to a mask value is referred to herein as “zeroing masking.” Alternatively, one embodiment uses “merging masking” in which the previous data element values stored in the destination register are maintained. Thus, if merging masking were used, the bits of destination vector C would maintain their prior values. It is understood by having ordinary skill in the art that the masking bits described above may be reversed while still complying with the underlying principles of the embodiments (e.g., true = masking; false = no masking).

[0077] In operation, if any resulting element exceeds the maximum or minimum data element value, saturation logic 1008 (using either signed or unsigned saturation) substitutes the maximum or minimum value for the element. As illustrated, in one embodiment the conversion logic 1006 accesses the registers 1001, 1002, and 1007 to perform the above operations by controlling multiplexers 1010, 1011, and 1012. The logic required for implementing a multiplexer is well understood by those of ordinary skill in the art and is not described in detail herein.

[0078] **Figure 11** is a block diagram of a processing system including instructions to perform a vector saturated add, according to an embodiment. The exemplary processing system includes a processor 1155 coupled to main memory 1100. The processor 1155 includes a decode unit 1130 with decode logic 1131 for decoding the vector saturated add instructions. Additionally, a processor execution engine unit 1140 includes additional execution logic 1141 for executing the vector saturated add instructions. Registers 1105 provide register storage for operands, control data and other types of data as the execution unit 1140 executes the instruction stream. In one embodiment the registers 1105 also include the physical registers used in implementing the vector saturated add instructions described herein.

[0079] The details of a single processor core (“Core 0”) are illustrated in **Figure 11** for simplicity. It will be understood, however, that each core shown in **Figure 11** may have the same set of logic as Core 0. As illustrated, each core may also include a dedicated Level 1 (L1) cache 1112 and Level 2 (L2) cache 1111 for caching instructions and data according to a specified cache

management policy. The L1 cache 1111 includes a separate instruction cache 1320 for storing instructions and a separate data cache 1121 for storing data. The instructions and data stored within the various processor caches are managed at the granularity of cache lines, which may be a fixed size (e.g., 64, 128, 512 Bytes in length). Each core of this exemplary embodiment has an instruction fetch unit 1110 for fetching instructions from main memory 1100 and/or a shared Level 3 (L3) cache 1116; a decode unit 1130 for decoding the instructions; an execution unit 1140 for executing the instructions; and a write back/retire unit 1150 for retiring the instructions and writing back the results to the registers 1105.

[0080] The instruction fetch unit 1110 includes various well known components including a next instruction pointer 1103 for storing the address of the next instruction to be fetched from memory 1100 (or one of the caches); an instruction translation look-aside buffer (ITLB) 1104 for storing a map of recently used virtual-to-physical instruction addresses to improve the speed of address translation; a branch prediction unit 1102 for speculatively predicting instruction branch addresses; and branch target buffers (BTBs) 1101 for storing branch addresses and target addresses. Once fetched, instructions are then streamed to the remaining stages of the instruction pipeline including the decode unit 1130, the execution unit 1140, and the write back/retire unit 1150.

[0081] **Figure 12** is a flow diagram of logic to perform instructions according to embodiments described herein. In one embodiment a processor includes logic to perform instruction operations including fetching an instruction to perform a vector saturated add instruction, as shown at 1202. As shown at 1204, decode logic is configured to decode the fetched instruction into a decoded instruction. As shown at 1206, processor execution logic executes the decoded instruction to perform a vector add operation. At 1208, saturation logic replaces any out of range results within any computed data elements with the appropriate saturation values (e.g., signed or unsigned, doubleword or quadword). At 1210, the execution logic writes one or more results of the executed instruction to the processor register file based on the processor write mask configuration and the write mask value for each data element. In one embodiment, writing the results of the executed instruction includes committing the results of the saturated add operation to a location, such as an architecture register, that is indicated by a

destination operand of the vector saturated add operation. The results can include one or more vector data elements including the sum of associated data elements stored in the source vectors, and one or more data elements storing a zero value based on the write mask associated with the data element and the write mask configuration. In one embodiment the results include one or more vector data elements that are unmodified and contain a previous value or a result of a previous operation.

[0082] Pseudocode describing an implementation of one embodiment is set forth in **Table 1** below.

Table 1 – Exemplary VPADDSD instruction logic

```

00  (KL, VL) = (4, 128), (8, 256), (16, 512)
01  FOR j ← 0 TO KL-1
02      i ← j * 32
03      IF k1[j] OR *no writemask* THEN
04          IF (EVEX.b == 1) AND (SRC2 *is memory*) THEN
05              DEST[i+31:i] ← SaturateToSignedDWord (SRC1[i+31:i] + SRC2[31:0])
06          ELSE
07              DEST[i+31:i] ← SaturateToSignedDWord (SRC1[i+31:i] + SRC2[i+31:i])
08          FI;
09      ELSE
10          IF *merging-masking*; merging-masking THEN
11              *DEST[i+31:i] remains unchanged*
12          ELSE *zeroing-masking*; zeroing-masking
13              DEST[i+31:i] = 0
14          FI
15      FI;
16  ENDFOR;
17  DEST[MAX_VL-1:VL] ← 0

```

[0083] The exemplary pseudocode shown in **Table 1** provides for a vector processor add saturated signed doubleword instruction. In the exemplary pseudocode, vector lengths (VL) of 128, 256, and 512 bits are supported with 4, 8, or 16 doubleword vector elements respectively. It will be understood, however, that the underlying principles of the embodiments are not limited to the specific implementation described in the pseudocode of **Table 1**, as embodiments provide additional instructions including signed quadword and unsigned doubleword and quadword instructions. Additionally, while vector add operations are to be performed, in one embodiment the SRC2 operand may be a memory address storing a doubleword or quadword data element, which is added to each element of the SRC1 vector. In such embodiment, an implicit load operation is performed from the memory address specified. In one embodiment, the load operation broadcasts the data element from memory to all elements of a SRC2 vector register before the processor execution unit performs the add operation.

[0084] In one embodiment a no-writemask operation can be performed, or a writemask operation can be performed. If no writemask is used, a sum of associated source data elements is written to a destination data element or a saturation value is written for results that are out of range of the data type for the destination data element (e.g., doubleword or quadword). If a writemask is used, each destination element will receive a result, a saturation value, a zero value, or will remain unmodified based on the write mask value associated with a data element and the write mask configuration for the instruction.

Exemplary Instruction Formats

[0085] Embodiments of the instruction(s) described herein may be embodied in different formats. 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.

[0086] **Figures 13A-13B** are block diagrams illustrating a generic vector friendly instruction format and instruction templates thereof according to an embodiment. Figure 13A is a block diagram illustrating a generic vector friendly instruction format and class A instruction templates

thereof according to an embodiment; while Figure 13B is a block diagram illustrating the generic vector friendly instruction format and class B instruction templates thereof according to an embodiment. Specifically, a generic vector friendly instruction format 1300 for which are defined class A and class B instruction templates, both of which include no memory access 1305 instruction templates and memory access 1320 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.

[0087] Embodiments 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). However, alternate embodiments 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).

[0088] The class A instruction templates in Figure 13A include: 1) within the no memory access 1305 instruction templates there is shown a no memory access, full round control type operation 1310 instruction template and a no memory access, data transform type operation 1315 instruction template; and 2) within the memory access 1320 instruction templates there is shown a memory access, temporal 1325 instruction template and a memory access, non-temporal 1330 instruction template. The class B instruction templates in Figure 13B include: 1) within the no memory access 1305 instruction templates there is shown a no memory access, write mask control, partial round control type operation 1312 instruction template and a no memory access, write mask control, vsize type operation 1317 instruction template; and 2) within the memory access 1320 instruction templates there is shown a memory access, write mask control 1327 instruction template.

[0089] The generic vector friendly instruction format 1300 includes the following fields listed below in the order illustrated in Figures 13A-13B.

[0090] Format field 1340 – 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.

[0091] Base operation field 1342 – its content distinguishes different base operations.

[0092] Register index field 1344 – 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).

[0093] Modifier field 1346 – 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 1305 instruction templates and memory access 1320 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.

[0094] Augmentation operation field 1350 – 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, this field is divided into a class field 1368, an alpha field 1352, and a beta field 1354. The augmentation operation field 1350 allows common groups of

operations to be performed in a single instruction rather than 2, 3, or 4 instructions.

[0095] Scale field 1360 – its content allows for the scaling of the index field's content for memory address generation (e.g., for address generation that uses $2^{\text{scale}} * \text{index} + \text{base}$).

[0096] Displacement Field 1362A – its content is used as part of memory address generation (e.g., for address generation that uses $2^{\text{scale}} * \text{index} + \text{base} + \text{displacement}$).

[0097] Displacement Factor Field 1362B (note that the juxtaposition of displacement field 1362A directly over displacement factor field 1362B 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 1374 (described later herein) and the data manipulation field 1354C. The displacement field 1362A and the displacement factor field 1362B are optional in the sense that they are not used for the no memory access 1305 instruction templates and/or different embodiments may implement only one or none of the two.

[0098] Data element width field 1364 – 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.

[0099] Write mask field 1370 – 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 1370 allows for partial vector operations, including loads, stores, arithmetic, logical, etc. While embodiments are described in which the write mask field's 1370 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 1370 content indirectly identifies that masking to be performed), alternative embodiments instead or additionally allow the mask write field's 1370 content to directly specify the masking to be performed.

[0100] Immediate field 1372 – 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.

[0101] Class field 1368 – its content distinguishes between different classes of instructions. With reference to Figures 13A-B, the contents of this field select between class A and class B instructions. In Figures 13A-B, rounded corner squares are used to indicate a specific value is present in a field (e.g., class A 1368A and class B 1368B for the class field 1368 respectively in Figures 13A-B).

Instruction Templates of Class A

[0102] In the case of the non-memory access 1305 instruction templates of class A, the alpha field 1352 is interpreted as an RS field 1352A, whose content distinguishes which one of the different augmentation operation types are to be performed (e.g., round 1352A.1 and data transform 1352A.2 are respectively specified for the no memory access, round type operation 1310 and the no memory access, data transform type operation 1315 instruction templates), while the beta field 1354 distinguishes which of the operations of the specified type is to be

performed. In the no memory access 1305 instruction templates, the scale field 1360, the displacement field 1362A, and the displacement scale field 1362B are not present.

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

[0103] In the no memory access full round control type operation 1310 instruction template, the beta field 1354 is interpreted as a round control field 1354A, whose content(s) provide static rounding. While in the described embodiments the round control field 1354A includes a suppress all floating point exceptions (SAE) field 1356 and a round operation control field 1358, 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 1358).

[0104] SAE field 1356 – its content distinguishes whether or not to disable the exception event reporting; when the SAE field's 1356 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.

[0105] Round operation control field 1358 – 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 1358 allows for the changing of the rounding mode on a per instruction basis. In one embodiment a processor includes a control register for specifying rounding modes and the round operation control field's 1350 content overrides that register value.

No Memory Access Instruction Templates – Data Transform Type Operation

[0106] In the no memory access data transform type operation 1315 instruction template, the beta field 1354 is interpreted as a data transform field 1354B, whose content distinguishes which one of a number of data transforms is to be performed (e.g., no data transform, swizzle, broadcast).

[0107] In the case of a memory access 1320 instruction template of class A, the alpha field 1352 is interpreted as an eviction hint field 1352B, whose content distinguishes which one of the eviction hints is to be used (in Figure 13A, temporal 1352B.1 and non-temporal 1352B.2 are respectively specified for the memory access, temporal 1325 instruction template and the

memory access, non-temporal 1330 instruction template), while the beta field 1354 is interpreted as a data manipulation field 1354C, 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 1320 instruction templates include the scale field 1360, and optionally the displacement field 1362A or the displacement scale field 1362B.

[0108] 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.

Memory Access Instruction Templates – Temporal

[0109] 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

[0110] 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

[0111] In the case of the instruction templates of class B, the alpha field 1352 is interpreted as a write mask control (Z) field 1352C, whose content distinguishes whether the write masking controlled by the write mask field 1370 should be a merging or a zeroing.

[0112] In the case of the non-memory access 1305 instruction templates of class B, part of the beta field 1354 is interpreted as an RL field 1357A, whose content distinguishes which one of the different augmentation operation types are to be performed (e.g., round 1357A.1 and vector length (VSIZE) 1357A.2 are respectively specified for the no memory access, write mask control, partial round control type operation 1312 instruction template and the no memory access, write mask control, VSIZE type operation 1317 instruction template), while the rest of the beta field 1354 distinguishes which of the operations of the specified type is to be performed.

In the no memory access 1305 instruction templates, the scale field 1360, the displacement field 1362A, and the displacement scale field 1362B are not present.

[0113] In the no memory access, write mask control, partial round control type operation 1310 instruction template, the rest of the beta field 1354 is interpreted as a round operation field 1359A 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).

[0114] Round operation control field 1359A – just as round operation control field 1358, 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 1359A allows for the changing of the rounding mode on a per instruction basis. In one embodiment a processor includes a control register for specifying rounding modes and the round operation control field's 1350 content overrides that register value.

[0115] In the no memory access, write mask control, VSIZE type operation 1317 instruction template, the rest of the beta field 1354 is interpreted as a vector length field 1359B, whose content distinguishes which one of a number of data vector lengths is to be performed on (e.g., 128, 256, or 512 byte).

[0116] In the case of a memory access 1320 instruction template of class B, part of the beta field 1354 is interpreted as a broadcast field 1357B, whose content distinguishes whether or not the broadcast type data manipulation operation is to be performed, while the rest of the beta field 1354 is interpreted the vector length field 1359B. The memory access 1320 instruction templates include the scale field 1360, and optionally the displacement field 1362A or the displacement scale field 1362B.

[0117] With regard to the generic vector friendly instruction format 1300, a full opcode field 1374 is shown including the format field 1340, the base operation field 1342, and the data element width field 1364. While one embodiment is shown where the full opcode field 1374 includes all of these fields, the full opcode field 1374 includes less than all of these fields in embodiments that do not support all of them. The full opcode field 1374 provides the operation code (opcode).

[0118] The augmentation operation field 1350, the data element width field 1364, and the

write mask field 1370 allow these features to be specified on a per instruction basis in the generic vector friendly instruction format.

[0119] 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.

[0120] The various instruction templates found within class A and class B are beneficial in different situations. In some embodiments, 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. Programs written in a high level language would be put (e.g., just in time compiled or statically compiled) into an 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.

Exemplary Specific Vector Friendly Instruction Format

[0121] **Figure 14** is a block diagram illustrating an exemplary specific vector friendly instruction format according to an embodiment. Figure 14 shows a specific vector friendly

instruction format 1400 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 1400 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 13 into which the fields from Figure 14 map are illustrated.

[0122] It should be understood that, although embodiments are described with reference to the specific vector friendly instruction format 1400 in the context of the generic vector friendly instruction format 1300 for illustrative purposes, the invention is not limited to the specific vector friendly instruction format 1400 except where claimed. For example, the generic vector friendly instruction format 1300 contemplates a variety of possible sizes for the various fields, while the specific vector friendly instruction format 1400 is shown as having fields of specific sizes. By way of specific example, while the data element width field 1364 is illustrated as a one bit field in the specific vector friendly instruction format 1400, the invention is not so limited (that is, the generic vector friendly instruction format 1300 contemplates other sizes of the data element width field 1364).

[0123] The generic vector friendly instruction format 1300 includes the following fields listed below in the order illustrated in Figure 14A.

[0124] EVEX Prefix (Bytes 0-3) 1402 - is encoded in a four-byte form.

[0125] Format Field 1340 (EVEX Byte 0, bits [7:0]) - the first byte (EVEX Byte 0) is the format field 1340 and it contains 0x62 (the unique value used for distinguishing the vector friendly instruction format in one embodiment of the invention).

[0126] The second-fourth bytes (EVEX Bytes 1-3) include a number of bit fields providing specific capability.

[0127] REX field 1405 (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 1357BEX 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 1111B, 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.

[0128] REX' field 1310 – this is the first part of the REX' field 1310 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, 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 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.

[0129] Opcode map field 1415 (EVEX byte 1, bits [3:0] – mmmm) – its content encodes an implied leading opcode byte (0F, 0F 38, or 0F 3).

[0130] Data element width field 1364 (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).

[0131] EVEX.vvvv 1420 (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 1111b. Thus, EVEX.vvvv field 1420 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.

[0132] EVEX.U 1368 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.

[0133] Prefix encoding field 1425 (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.

[0134] Alpha field 1352 (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.

[0135] Beta field 1354 (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.

[0136] REX' field 1310 – 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.

[0137] Write mask field 1370 (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, 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).

[0138] Real Opcode Field 1430 (Byte 4) is also known as the opcode byte. Part of the

opcode is specified in this field.

[0139] MOD R/M Field 1440 (Byte 5) includes MOD field 1442, Reg field 1444, and R/M field 1446. As previously described, the MOD field's 1442 content distinguishes between memory access and non-memory access operations. The role of Reg field 1444 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 1446 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.

[0140] Scale, Index, Base (SIB) Byte (Byte 6) - As previously described, the scale field's 1350 content is used for memory address generation. SIB.xxx 1454 and SIB.bbb 1456 – the contents of these fields have been previously referred to with regard to the register indexes Xxxx and Bbbb.

[0141] Displacement field 1362A (Bytes 7-10) – when MOD field 1442 contains 10, bytes 7-10 are the displacement field 1362A, and it works the same as the legacy 32-bit displacement (disp32) and works at byte granularity.

[0142] Displacement factor field 1362B (Byte 7) – when MOD field 1442 contains 01, byte 7 is the displacement factor field 1362B. 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 1362B is a reinterpretation of disp8; when using displacement factor field 1362B, 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 $\text{disp8} * N$. 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 1362B substitutes the legacy x86 instruction set 8-bit displacement. Thus, the displacement factor field 1362B 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).

[0143] Immediate field 1372 operates as previously described.

Full Opcode Field

[0144] Figure 14B is a block diagram illustrating the fields of the specific vector friendly instruction format 1400 that make up the full opcode field 1374 according to one embodiment. Specifically, the full opcode field 1374 includes the format field 1340, the base operation field 1342, and the data element width (W) field 1364. The base operation field 1342 includes the prefix encoding field 1425, the opcode map field 1415, and the real opcode field 1430.

Register Index Field

[0145] Figure 14C is a block diagram illustrating the fields of the specific vector friendly instruction format 1400 that make up the register index field 1344 according to one embodiment. Specifically, the register index field 1344 includes the REX field 1405, the REX' field 1410, the MODR/M.reg field 1444, the MODR/M.r/m field 1446, the VVVV field 1420, xxx field 1454, and the bbb field 1456.

Augmentation Operation Field

[0146] Figure 14D is a block diagram illustrating the fields of the specific vector friendly instruction format 1400 that make up the augmentation operation field 1350 according to one embodiment. When the class (U) field 1368 contains 0, it signifies EVEX.U0 (class A 1368A); when it contains 1, it signifies EVEX.U1 (class B 1368B). When U=0 and the MOD field 1442 contains 11 (signifying a no memory access operation), the alpha field 1352 (EVEX byte 3, bit [7] – EH) is interpreted as the rs field 1352A. When the rs field 1352A contains a 1 (round 1352A.1), the beta field 1354 (EVEX byte 3, bits [6:4]– SSS) is interpreted as the round control field 1354A. The round control field 1354A includes a one bit SAE field 1356 and a two bit

round operation field 1358. When the rs field 1352A contains a 0 (data transform 1352A.2), the beta field 1354 (EVEX byte 3, bits [6:4]- SSS) is interpreted as a three bit data transform field 1354B. When U=0 and the MOD field 1442 contains 00, 01, or 10 (signifying a memory access operation), the alpha field 1352 (EVEX byte 3, bit [7] – EH) is interpreted as the eviction hint (EH) field 1352B and the beta field 1354 (EVEX byte 3, bits [6:4]- SSS) is interpreted as a three bit data manipulation field 1354C.

[0147] When U=1, the alpha field 1352 (EVEX byte 3, bit [7] – EH) is interpreted as the write mask control (Z) field 1352C. When U=1 and the MOD field 1442 contains 11 (signifying a no memory access operation), part of the beta field 1354 (EVEX byte 3, bit [4]- S₀) is interpreted as the RL field 1357A; when it contains a 1 (round 1357A.1) the rest of the beta field 1354 (EVEX byte 3, bit [6-5]- S₂₋₁) is interpreted as the round operation field 1359A, while when the RL field 1357A contains a 0 (VSIZE 1357.A2) the rest of the beta field 1354 (EVEX byte 3, bit [6-5]- S₂₋₁) is interpreted as the vector length field 1359B (EVEX byte 3, bit [6-5]- L₁₋₀). When U=1 and the MOD field 1442 contains 00, 01, or 10 (signifying a memory access operation), the beta field 1354 (EVEX byte 3, bits [6:4]- SSS) is interpreted as the vector length field 1359B (EVEX byte 3, bit [6-5]- L₁₋₀) and the broadcast field 1357B (EVEX byte 3, bit [4]- B).

Exemplary Register Architecture

[0148] **Figure 15** is a block diagram of a register architecture 1500 according to one embodiment. In the embodiment illustrated, there are 32 vector registers 1510 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 1400 operates on these overlaid registers as illustrated in **Table 2** below.

Table 2 – Register File

Adjustable Vector Length	Class	Operations	Registers

Instruction Templates that do not include the vector length field 1359B	A (Figure 13A; U=0)	1310, 1315, 1325, 1330	zmm registers (the vector length is 64 byte)
	B (Figure 13B; U=1)	1312	zmm registers (the vector length is 64 byte)
Instruction Templates that do include the vector length field 1359B	B (Figure 13B; U=1)	1317, 1327	zmm, ymm, or xmm registers (the vector length is 64 byte, 32 byte, or 16 byte) depending on the vector length field 1359B

[0149] In other words, the vector length field 1359B 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 1359B operate on the maximum vector length. Further, in one embodiment, the class B instruction templates of the specific vector friendly instruction format 1400 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.

[0150] Write mask registers 1515 - 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 1515 are 16 bits in size. As previously described, in one embodiment 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.

[0151] General-purpose registers 1525 - 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.

[0152] Scalar floating point stack register file (x87 stack) 1545, on which is aliased the

MMX packed integer flat register file 1550 - 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.

[0153] Alternative embodiments may use wider or narrower registers. Additionally, alternative embodiments may use more, less, or different register files and registers.

[0154] In the foregoing specification, the invention has been described with reference to specific exemplary embodiments thereof. It will, however, be evident that various modifications and changes may be made thereto without departing from the broader spirit and scope of the invention as set forth in the appended claims. The specification and drawings are, accordingly, to be regarded in an illustrative rather than a restrictive sense.

[0155] The instructions described herein refer to specific configurations of hardware, such as application specific integrated circuits (ASICs), configured to perform certain operations or having a predetermined functionality. 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.

[0156] 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 that follow.

CLAIMS

What is claimed is:

1. A processing apparatus comprising:
 - decode logic to decode a first instruction into a decoded first instruction including a first operand and a second operand;
 - an execution unit to execute the first decoded instruction to perform a vector saturated add operation on the first and second operand; and
 - a register file unit to commit a result of the vector saturated add operation to a location indicated by a destination operand.
2. The processing apparatus as in claim 1 further comprising an instruction fetch unit to fetch the first instruction, wherein the instruction is a single machine-level instruction.
3. The processing apparatus as in claim 1 wherein the register file unit further to store a set of registers comprising:
 - a first register to store a first source operand value;
 - a second register to store a second source operand value; and
 - a third register to conditionally store a first data element of the result of the saturated add operation based upon a mask value associated with the at first data element.
4. The processing apparatus as in claim 3 wherein the register file unit further to not commit a second data element of the result of the saturated add operation based at least upon the mask value associated with the second element.
5. The processing apparatus as in claim 3 wherein the first or second register is a vector register.

6. The processing apparatus as in claim 5 wherein the second register is a vector register, the second operand indicates a memory address storing a scalar data element, and the scalar data element is broadcast to each element of the second register.
7. The processing apparatus as in claim 5 wherein the vector register is a 128-bit, 256-bit, or 512-bit register.
8. The processing apparatus as in claim 5 wherein the vector register stores packed doubleword or quadword data element.
9. The processing apparatus as in claim 5 wherein a result of the saturated add operation for a set of data elements is out of range of a data type of the data elements and a saturation value is written to a destination data element.
10. The processing apparatus as in claim 9 wherein the saturation value is an unsigned value.
11. The processing apparatus as in claim 9 wherein the saturation value is a signed value.
12. A method implemented by an integrated circuit, the method comprising:
 - fetching a single instruction to perform a vector saturated add operation, the instruction having two source operands and a destination operand;
 - decoding the single instruction into a decoded instruction;
 - fetching source operand values associated with the two source operands, the source operand values including multiple packed data elements;
 - executing the decoded instruction to compute a sum of associated data elements of the source operand values, wherein the sum of associated data elements is out of range of a data type of the associated data elements and a saturation value is written to a first destination data element as a result.
13. The method as in claim 12 further comprising writing a zero to a second data element based on a write mask value associated with the second data element.

14. The method as in claim 13 further comprising loading a data element from a memory address specified by a source operand and broadcasting the data element to each element of a source vector register.
15. A system for performing a vector saturated add operation, the system comprising:
 - means for fetching a single instruction to perform a vector saturated add operation, the instruction having two source operands and a destination operand;
 - means for decoding the single instruction into a decoded instruction;
 - means for fetching source operand values associated with the two source operands, the source operand values including multiple packed data elements; and
 - means for executing the decoded instruction to compute a sum of associated data elements of the source operand values.
16. The system as in claim 15 further comprising means for writing a sum computed from associated data elements of the source operand values to a first data element of a vector register file, the writing based on a write mask value associated with the first data element.
17. The system as in claim 15 further comprising means for writing a zero to a second data element based on a write mask value associated with the second data element.
18. The system as in claim 15 further comprising means for loading a data element from a memory address specified by a source operand.
19. The system as in claim 18 further comprising means for broadcasting the data element to each element of a source vector register.
20. The system as in claim 19 wherein the source vector register is a 128-bit register.
21. The system as in claim 19 wherein the source vector register is a 256-bit register.
22. The system as in claim 19 wherein the source vector register is 512-bit register.
23. The system as in claim 19 wherein the data element is a doubleword data element.

24. The system as in claim 19 wherein the data element is a quadword data element.
25. The system as in claim 24 wherein a sum of associated data elements is out of range of a data type of the associated data elements and further comprising means for writing a saturation value to a second destination data element as a result.