

**METHODS, APPARATUS, INSTRUCTIONS AND LOGIC TO PROVIDE VECTOR
PACKED TUPLE CROSS-COMPARISON FUNCTIONALITY**

CROSS-REFERENCE TO RELATED APPLICATION

[0001] This Application is related to: Application Ser. No. 12/974,157, titled "Mechanism for Conflict Detection Using SIMD," filed Dec. 21, 2010; Application Ser. No. 12/976,616, titled "Vector Conflict Instructions," filed Dec. 22, 2010; and Application Ser. No. 13/731,006, titled "Method, Apparatus, Instructions and Logic to Provide Vector Address Conflict Detection Functionality," filed Dec. 29, 2012.

FIELD OF THE DISCLOSURE

[0002] 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. In particular, the disclosure relates to processor instructions and logic to provide single-instruction multiple-data (SIMD) vector packed tuple cross-comparison functionality.

BACKGROUND OF THE DISCLOSURE

[0003] Modern processors often include instructions to provide operations that are computationally intensive, but offer a high level of data parallelism that can be exploited through an efficient implementation using various data storage devices, such as for example, SIMD vector registers. In SIMD execution, a single instruction operates on multiple data elements concurrently or simultaneously. This is typically implemented by extending the width of various resources such as registers and arithmetic logic units (ALUs), allowing them to hold or operate on multiple data elements, respectively.

[0004] The central processing unit (CPU) may provide such parallel hardware to support the SIMD processing of vectors. A vector is a data structure that holds a number of consecutive data elements. A vector register of size L may contain N vector elements of size M, where $N=L/M$. For instance, a 64-byte vector register may be partitioned into (a) 64 vector elements, with each element holding a data item that occupies 1 byte, (b) 32 vector elements to hold data items that occupy 2 bytes (or one "word") each, (c) 16 vector elements

to hold data items that occupy 4 bytes (or one "doubleword") each, or (d) 8 vector elements to hold data items that occupy 8 bytes (or one "quadword") each.

[0005] A number of applications have large amounts of data-level parallelism and may be able to benefit from SIMD support. However, some applications spend a significant amount of time in operations on a set of sparse locations. Moreover, sometimes sequential operations are performed on the same sparse locations and so these applications may see only limited benefit from SIMD, operations. To maintain SIMD efficiency, some architectures allow not only SIMD arithmetic operations but also SIMD memory reads and writes (through gather and scatter operations). Scatter reductions are common operations in many applications. For example, a scatter-add operation can be used to enable multiple values of a first array to be reduced into (i.e., added to) selected elements of a second array according to a distribution of indices, which can often be random. But because of this, it may be difficult to efficiently process multiple elements concurrently (i.e., in SIMD mode). One concern is to ensure that scalar program order is preserved when necessary. Another concern is to ensure that when data is scattered into memory, the resulting vector of memory addresses includes only unique addresses (i.e., there are no duplicate addresses).

[0006] For example, histogram calculations are common operations in many image processing applications. A histogram may be used to track the distribution of color values of pixels in an image. However, updates to the histogram array may be random, depending on input data to an array. In particular, indices of neighboring elements may point to the same histogram bin. Accordingly, conflict detection and resolution is required to detect multiple dependent updates to the same locations and to ensure that scalar program order is preserved. This is precisely the kind of condition that can make it very difficult to process multiple data concurrently or simultaneously (i.e., using SIMD operations).

[0007] To date, potential solutions to such conflict concerns and related processing difficulties have not been adequately explored.

BRIEF DESCRIPTION OF THE DRAWINGS

[0008] The present invention is illustrated by way of example and not limitation in the figures of the accompanying drawings.

[0009] **Figure 1A** is a block diagram of one embodiment of a system that executes instructions to provide SIMD vector packed tuple cross-comparison functionality.

[0010] **Figure 1B** is a block diagram of another embodiment of a system that executes instructions to provide SIMD vector packed tuple cross-comparison functionality.

[0011] **Figure 1C** is a block diagram of another embodiment of a system that executes instructions to provide SIMD vector packed tuple cross-comparison functionality.

[0012] **Figure 2** is a block diagram of one embodiment of a processor that executes instructions to provide SIMD vector packed tuple cross-comparison functionality.

[0013] **Figure 3A** illustrates packed data types according to one embodiment.

[0014] **Figure 3B** illustrates packed data types according to one embodiment.

[0015] **Figure 3C** illustrates packed data types according to one embodiment.

[0016] **Figure 3D** illustrates an instruction encoding to provide SIMD vector packed tuple cross-comparison functionality according to one embodiment.

[0017] **Figure 3E** illustrates an instruction encoding to provide SIMD vector packed tuple cross-comparison functionality according to another embodiment.

[0018] **Figure 3F** illustrates an instruction encoding to provide SIMD vector packed tuple cross-comparison functionality according to another embodiment.

[0019] **Figure 3G** illustrates an instruction encoding to provide SIMD vector packed tuple cross-comparison functionality according to another embodiment.

[0020] **Figure 3H** illustrates an instruction encoding to provide SIMD vector packed tuple cross-comparison functionality according to another embodiment.

[0021] **Figure 4A** illustrates elements of one embodiment of a processor micro-architecture to execute instructions that provide SIMD vector packed tuple cross-comparison functionality.

[0022] **Figure 4B** illustrates elements of another embodiment of a processor micro-architecture to execute instructions that provide SIMD vector packed tuple cross-comparison functionality.

[0023] **Figure 5** is a block diagram of one embodiment of a processor to execute instructions that provide SIMD vector packed tuple cross-comparison functionality.

[0024] **Figure 6** is a block diagram of one embodiment of a computer system to execute instructions that provide SIMD vector packed tuple cross-comparison functionality.

[0025] **Figure 7** is a block diagram of another embodiment of a computer system to execute instructions that provide SIMD vector packed tuple cross-comparison functionality.

[0026] **Figure 8** is a block diagram of another embodiment of a computer system to execute instructions that provide SIMD vector packed tuple cross-comparison functionality.

[0027] **Figure 9** is a block diagram of one embodiment of a system-on-a-chip to execute instructions that provide SIMD vector packed tuple cross-comparison functionality.

[0028] **Figure 10** is a block diagram of an embodiment of a processor to execute instructions that provide SIMD vector packed tuple cross-comparison functionality.

[0029] **Figure 11** is a block diagram of one embodiment of an IP core development system that provides SIMD vector packed tuple cross-comparison functionality.

[0030] **Figure 12** illustrates one embodiment of an architecture emulation system that provides SIMD vector packed tuple cross-comparison functionality.

[0031] **Figure 13** illustrates one embodiment of a system to translate instructions that provide SIMD vector packed tuple cross-comparison functionality.

[0032] **Figure 14A** illustrates a flow diagram for one embodiment of an example of using a SIMD vector leading zero count instruction for generating permute controls for efficient vector address conflict resolution.

[0033] **Figure 14B** illustrates a flow diagram for one embodiment of an example of using SIMD vector address conflict masks for efficient vector address conflict resolution.

[0034] **Figure 15A** illustrates a flow diagram for one embodiment of a process in a processor to execute instructions to provide SIMD vector packed tuple cross-comparison functionality.

[0035] **Figure 15B** illustrates a flow diagram for one embodiment of a process to use SIMD vector address conflict masks for efficient vector address conflict resolution.

[0036] **Figure 16** illustrates a prior art embodiment of an apparatus for executing an instruction to provide SIMD vector leading zero count functionality useful for generating permute controls for efficient vector address conflict resolution.

[0037] **Figure 17A-17C** illustrate some alternative embodiments of instructions to provide SIMD vector packed tuple cross-comparison functionality.

[0038] **Figure 18** illustrates an embodiment of a method combining results of an instruction to provide SIMD vector packed tuple cross-comparison functionality to generate address conflict masks for efficient vector address conflict resolution.

[0039] **Figure 19A** illustrates an embodiment of an apparatus for executing an instruction to provide SIMD vector packed tuple cross-comparison functionality.

[0040] **Figure 19B** illustrates an alternative embodiment of an apparatus for executing an instruction to provide SIMD vector packed tuple cross-comparison functionality.

DETAILED DESCRIPTION

[0041] The following description discloses instructions and processing logic to provide SIMD vector packed tuple cross-comparison functionality within or in association with a processor, computer system, or other processing apparatus. Instructions and logic are disclosed herein to provide SIMD vector packed tuple cross-comparison functionality. Some embodiments include processors with a first and a second registers with a variable plurality of data fields, each of the data fields to store an element of a first data type. The processor executes a SIMD instruction for vector packed tuple cross-comparison in some embodiments, which for each data field of a portion of data fields in a tuple of the first register, compares its corresponding element with every element of a corresponding portion of data fields in a tuple of the second register and sets a mask bit corresponding to each element of the second register portion, in a bit-mask corresponding to each unmasked element of the corresponding first register portion, according to the corresponding comparison. The packed tuples in some embodiments may each comprise two, four or eight elements. In some embodiments bit-masks may also be shifted by corresponding elements in data fields of a third register before being stored in a destination register. The comparison type may be indicated by an immediate operand specified by the instruction. In some embodiments the third register may be the same register as the destination register.

[0042] When the elements being compared, e.g. indices, are the same size as the masks required to represent comparison results (e.g. sixteen 16-bit elements in 256-bit registers) then performing an instruction to cross-compare all elements may be appropriate. But when there are more elements to compare than bits available to represent the comparison results, one may need an alternative solution. It will be appreciated that SIMD vector packed tuple comparison instructions, as disclosed herein, may be used as an alternative solution for variable sized elements and memory offsets to provide SIMD vector address conflict detection functionality and the results combined to generate conflict masks for efficient SIMD address conflict resolution. It will also be appreciated that SIMD vector packed tuple comparison instructions may be of more general-purpose applicability (e.g. when any one of a variety of different comparison operations may be useful), and require less implementation cost in terms of area and power than an instruction to cross-compare all elements.

[0043] In the following description, numerous specific details such as processing logic, processor types, micro-architectural conditions, events, enablement mechanisms, and the like

are set forth in order to provide a more thorough understanding of embodiments of the present invention. It will be appreciated, however, by one skilled in the art that the invention may be practiced without such specific details. Additionally, some well known structures, circuits, and the like have not been shown in detail to avoid unnecessarily obscuring embodiments of the present invention.

[0044] Although the following embodiments are described with reference to a processor, other embodiments are applicable to other types of integrated circuits and logic devices. Similar techniques and teachings of embodiments of the present invention can be applied to other types of circuits or semiconductor devices that can benefit from higher pipeline throughput and improved performance. The teachings of embodiments of the present invention are applicable to any processor or machine that performs data manipulations. However, the present invention is not limited to processors or machines that perform 512 bit, 256 bit, 128 bit, 64 bit, 32 bit, or 16 bit data operations and can be applied to any processor and machine in which manipulation or management of data is performed. In addition, the following description provides examples, and the accompanying drawings show various examples for the purposes of illustration. However, these examples should not be construed in a limiting sense as they are merely intended to provide examples of embodiments of the present invention rather than to provide an exhaustive list of all possible implementations of embodiments of the present invention.

[0045] Although the below examples describe instruction handling and distribution in the context of execution units and logic circuits, other embodiments of the present invention can be accomplished by way of data and/or instructions stored on a machine-readable, tangible medium, which when performed by a machine cause the machine to perform functions consistent with at least one embodiment of the invention. In one embodiment, functions associated with embodiments of the present invention are embodied in machine-executable instructions. The instructions can be used to cause a general-purpose or special-purpose processor that is programmed with the instructions to perform the steps of the present invention. Embodiments of the present invention may be provided as a computer program product or software which may include a machine or computer-readable medium having stored thereon instructions which may be used to program a computer (or other electronic devices) to perform one or more operations according to embodiments of the present invention. Alternatively, steps of embodiments of the present invention might be performed

by specific hardware components that contain fixed-function logic for performing the steps, or by any combination of programmed computer components and fixed-function hardware components.

[0046] Instructions used to program logic to perform embodiments of the invention can be stored within a memory in the system, such as DRAM, cache, flash memory, or other storage. Furthermore, the instructions can be distributed via a network or by way of other computer readable media. Thus a machine-readable medium may include any mechanism for storing or transmitting information in a form readable by a machine (e.g., a computer), but is not limited to, floppy diskettes, optical disks, Compact Disc, Read-Only Memory (CD-ROMs), and magneto-optical disks, Read-Only Memory (ROMs), Random Access Memory (RAM), Erasable Programmable Read-Only Memory (EPROM), Electrically Erasable Programmable Read-Only Memory (EEPROM), magnetic or optical cards, flash memory, or a tangible, machine-readable storage used in the transmission of information over the Internet via electrical, optical, acoustical or other forms of propagated signals (e.g., carrier waves, infrared signals, digital signals, etc.). Accordingly, the computer-readable medium includes any type of tangible machine-readable medium suitable for storing or transmitting electronic instructions or information in a form readable by a machine (e.g., a computer).

[0047] A design may go through various stages, from creation to simulation to fabrication. Data representing a design may represent the design in a number of manners. First, as is useful in simulations, the hardware may be represented using a hardware description language or another functional description language. Additionally, a circuit level model with logic and/or transistor gates may be produced at some stages of the design process. Furthermore, most designs, at some stage, reach a level of data representing the physical placement of various devices in the hardware model. In the case where conventional semiconductor fabrication techniques are used, the data representing the hardware model may be the data specifying the presence or absence of various features on different mask layers for masks used to produce the integrated circuit. In any representation of the design, the data may be stored in any form of a machine readable medium. A memory or a magnetic or optical storage such as a disc may be the machine readable medium to store information transmitted via optical or electrical wave modulated or otherwise generated to transmit such information. When an electrical carrier wave indicating or carrying the code or design is transmitted, to the extent that copying, buffering, or re-transmission of the electrical signal is

performed, a new copy is made. Thus, a communication provider or a network provider may store on a tangible, machine-readable medium, at least temporarily, an article, such as information encoded into a carrier wave, embodying techniques of embodiments of the present invention.

[0048] In modern processors, a number of different execution units are used to process and execute a variety of code and instructions. Not all instructions are created equal as some are quicker to complete while others can take a number of clock cycles to complete. The faster the throughput of instructions, the better the overall performance of the processor. Thus it would be advantageous to have as many instructions execute as fast as possible. However, there are certain instructions that have greater complexity and require more in terms of execution time and processor resources. For example, there are floating point instructions, load/store operations, data moves, etc.

[0049] As more computer systems are used in internet, text, and multimedia applications, additional processor support has been introduced over time. In one embodiment, an instruction set may be associated with one or more computer architectures, including data types, instructions, register architecture, addressing modes, memory architecture, interrupt and exception handling, and external input and output (I/O).

[0050] In one embodiment, the instruction set architecture (ISA) may be implemented by one or more micro-architectures, which includes processor logic and circuits used to implement one or more instruction sets. Accordingly, processors with different micro-architectures can share at least a portion of a common instruction set. For example, Intel® Pentium 4 processors, Intel® Core™ processors, and processors from Advanced Micro Devices, Inc. of Sunnyvale CA implement nearly identical versions of the x86 instruction set (with some extensions that have been added with newer versions), but have different internal designs. Similarly, processors designed by other processor development companies, such as ARM Holdings, Ltd., MIPS, or their licensees or adopters, may share at least a portion a common instruction set, but may include different processor designs. For example, the same register architecture of the ISA may be implemented in different ways in different micro-architectures using new or well-known techniques, including dedicated physical registers, one or more dynamically allocated physical registers using a register renaming mechanism (e.g., the use of a Register Alias Table (RAT), a Reorder Buffer (ROB) and a retirement register file. In one embodiment, registers may include one or more registers, register architectures,

register files, or other register sets that may or may not be addressable by a software programmer.

[0051] In one embodiment, an instruction may include one or more instruction formats. In one embodiment, an instruction format may indicate various fields (number of bits, location of bits, etc.) to specify, among other things, the operation to be performed and the operand(s) on which that operation is to be performed. Some instruction formats may be further broken defined by instruction templates (or sub formats). For example, the instruction templates of a given instruction format may be defined to have different subsets of the instruction format's fields and/or defined to have a given field interpreted differently. In one embodiment, an instruction is expressed using an instruction format (and, if defined, in a given one of the instruction templates of that instruction format) and specifies or indicates the operation and the operands upon which the operation will operate.

[0052] Scientific, financial, auto-vectorized general purpose, RMS (recognition, mining, and synthesis), and visual and multimedia applications (e.g., 2D/3D graphics, image processing, video compression/decompression, voice recognition algorithms and audio manipulation) may require the same operation to be performed on a large number of data items. In one embodiment, Single Instruction Multiple Data (SIMD) refers to a type of instruction that causes a processor to perform an operation on multiple data elements. SIMD technology may be used in processors that can logically divide the bits in a register into a number of fixed-sized or variable-sized data elements, each of which represents a separate value. For example, in one embodiment, the bits in a 64-bit register may be organized as a source operand containing four separate 16-bit data elements, each of which represents a separate 16-bit value. This type of data may be referred to as 'packed' data type or 'vector' data type, and operands of this data type are referred to as packed data operands or vector operands. In one embodiment, a packed data item or vector may be a sequence of packed data elements stored within a single register, and a packed data operand or a vector operand may a source or destination operand of a SIMD instruction (or 'packed data instruction' or a 'vector instruction'). In one embodiment, a SIMD instruction specifies a single vector operation to be performed on two source vector operands to generate a destination vector operand (also referred to as a result vector operand) of the same or different size, with the same or different number of data elements, and in the same or different data element order.

[0053] 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, ARM processors, such as the ARM Cortex® family of processors having an instruction set including the Vector Floating Point (VFP) and/or NEON instructions, and MIPS processors, such as the Loongson family of processors developed by the Institute of Computing Technology (ICT) of the Chinese Academy of Sciences, has enabled a significant improvement in application performance (Core™ and MMX™ are registered trademarks or trademarks of Intel Corporation of Santa Clara, Calif.).

[0054] In one embodiment, destination and source registers/data are generic terms to represent the source and destination of the corresponding data or operation. In some embodiments, they may be implemented by registers, memory, or other storage areas having other names or functions than those depicted. For example, in one embodiment, “DEST1” may be a temporary storage register or other storage area, whereas “SRC1” and “SRC2” may be a first and second source storage register or other storage area, and so forth. In other embodiments, two or more of the SRC and DEST storage areas may correspond to different data storage elements within the same storage area (e.g., a SIMD register). In one embodiment, one of the source registers may also act as a destination register by, for example, writing back the result of an operation performed on the first and second source data to one of the two source registers serving as a destination registers.

[0055] **Figure 1A** is a block diagram of an exemplary computer system formed with a processor that includes execution units to execute an instruction in accordance with one embodiment of the present invention. System 100 includes a component, such as a processor 102 to employ execution units including logic to perform algorithms for process data, in accordance with the present invention, such as in the embodiment described herein. System 100 is representative of processing systems based on the PENTIUM® III, PENTIUM® 4, Xeon™, Itanium®, XScale™ and/or StrongARM™ microprocessors available from Intel Corporation of Santa Clara, California, although other systems (including PCs having other microprocessors, engineering workstations, set-top boxes and the like) may also be used. In one embodiment, sample system 100 may execute a version of the WINDOWS™ operating system available from Microsoft Corporation of Redmond, Washington, although other operating systems (UNIX and Linux for example), embedded software, and/or graphical user

interfaces, may also be used. Thus, embodiments of the present invention are not limited to any specific combination of hardware circuitry and software.

[0056] Embodiments are not limited to computer systems. Alternative embodiments of the present invention can be used in other devices such as handheld devices and embedded applications. Some examples of handheld devices include cellular phones, Internet Protocol devices, digital cameras, personal digital assistants (PDAs), and handheld PCs. Embedded applications can include a micro controller, a digital signal processor (DSP), system on a chip, network computers (NetPC), set-top boxes, network hubs, wide area network (WAN) switches, or any other system that can perform one or more instructions in accordance with at least one embodiment.

[0057] **Figure 1A** is a block diagram of a computer system 100 formed with a processor 102 that includes one or more execution units 108 to perform an algorithm to perform at least one instruction in accordance with one embodiment of the present invention. One embodiment may be described in the context of a single processor desktop or server system, but alternative embodiments can be included in a multiprocessor system. System 100 is an example of a 'hub' system architecture. The computer system 100 includes a processor 102 to process data signals. The processor 102 can be a complex instruction set computer (CISC) microprocessor, a reduced instruction set computing (RISC) microprocessor, a very long instruction word (VLIW) microprocessor, a processor implementing a combination of instruction sets, or any other processor device, such as a digital signal processor, for example. The processor 102 is coupled to a processor bus 110 that can transmit data signals between the processor 102 and other components in the system 100. The elements of system 100 perform their conventional functions that are well known to those familiar with the art.

[0058] In one embodiment, the processor 102 includes a Level 1 (L1) internal cache memory 104. Depending on the architecture, the processor 102 can have a single internal cache or multiple levels of internal cache. Alternatively, in another embodiment, the cache memory can reside external to the processor 102. Other embodiments can also include a combination of both internal and external caches depending on the particular implementation and needs. Register file 106 can store different types of data in various registers including integer registers, floating point registers, status registers, and instruction pointer register.

[0059] Execution unit 108, including logic to perform integer and floating point operations, also resides in the processor 102. The processor 102 also includes a microcode

(ucode) ROM that stores microcode for certain macroinstructions. For one embodiment, execution unit 108 includes logic to handle a packed instruction set 109. By including the packed instruction set 109 in the instruction set of a general-purpose processor 102, along with associated circuitry to execute the instructions, the operations used by many multimedia applications may be performed using packed data in a general-purpose processor 102. Thus, many multimedia applications can be accelerated and executed more efficiently by using the full width of a processor's data bus for performing operations on packed data. This can eliminate the need to transfer smaller units of data across the processor's data bus to perform one or more operations one data element at a time.

[0060] Alternate embodiments of an execution unit 108 can also be used in micro controllers, embedded processors, graphics devices, DSPs, and other types of logic circuits. System 100 includes a memory 120. Memory 120 can be a dynamic random access memory (DRAM) device, a static random access memory (SRAM) device, flash memory device, or other memory device. Memory 120 can store instructions and/or data represented by data signals that can be executed by the processor 102.

[0061] A system logic chip 116 is coupled to the processor bus 110 and memory 120. The system logic chip 116 in the illustrated embodiment is a memory controller hub (MCH). The processor 102 can communicate to the MCH 116 via a processor bus 110. The MCH 116 provides a high bandwidth memory path 118 to memory 120 for instruction and data storage and for storage of graphics commands, data and textures. The MCH 116 is to direct data signals between the processor 102, memory 120, and other components in the system 100 and to bridge the data signals between processor bus 110, memory 120, and system I/O 122. In some embodiments, the system logic chip 116 can provide a graphics port for coupling to a graphics controller 112. The MCH 116 is coupled to memory 120 through a memory interface 118. The graphics card 112 is coupled to the MCH 116 through an Accelerated Graphics Port (AGP) interconnect 114.

[0062] System 100 uses a proprietary hub interface bus 122 to couple the MCH 116 to the I/O controller hub (ICH) 130. The ICH 130 provides direct connections to some I/O devices via a local I/O bus. The local I/O bus is a high-speed I/O bus for connecting peripherals to the memory 120, chipset, and processor 102. Some examples are the audio controller, firmware hub (flash BIOS) 128, wireless transceiver 126, data storage 124, legacy I/O controller containing user input and keyboard interfaces, a serial expansion port such as

Universal Serial Bus (USB), and a network controller 134. The data storage device 124 can comprise a hard disk drive, a floppy disk drive, a CD-ROM device, a flash memory device, or other mass storage device.

[0063] For another embodiment of a system, an instruction in accordance with one embodiment can be used with a system on a chip. One embodiment of a system on a chip comprises of a processor and a memory. The memory for one such system is a flash memory. The flash memory can be located on the same die as the processor and other system components. Additionally, other logic blocks such as a memory controller or graphics controller can also be located on a system on a chip.

[0064] **Figure 1B** illustrates a data processing system 140 which implements the principles of one embodiment of the present invention. It will be readily appreciated by one of skill in the art that the embodiments described herein can be used with alternative processing systems without departure from the scope of embodiments of the invention.

[0065] Computer system 140 comprises a processing core 159 capable of performing at least one instruction in accordance with one embodiment. For one embodiment, processing core 159 represents a processing unit of any type of architecture, including but not limited to a CISC, a RISC or a VLIW type architecture. Processing core 159 may also be suitable for manufacture in one or more process technologies and by being represented on a machine readable media in sufficient detail, may be suitable to facilitate said manufacture.

[0066] Processing core 159 comprises an execution unit 142, a set of register file(s) 145, and a decoder 144. Processing core 159 also includes additional circuitry (not shown) which is not necessary to the understanding of embodiments of the present invention. Execution unit 142 is used for executing instructions received by processing core 159. In addition to performing typical processor instructions, execution unit 142 can perform instructions in packed instruction set 143 for performing operations on packed data formats. Packed instruction set 143 includes instructions for performing embodiments of the invention and other packed instructions. Execution unit 142 is coupled to register file 145 by an internal bus. Register file 145 represents a storage area on processing core 159 for storing information, including data. As previously mentioned, it is understood that the storage area used for storing the packed data is not critical. Execution unit 142 is coupled to decoder 144. Decoder 144 is used for decoding instructions received by processing core 159 into control signals and/or microcode entry points. In response to these control signals and/or microcode

entry points, execution unit 142 performs the appropriate operations. In one embodiment, the decoder is used to interpret the opcode of the instruction, which will indicate what operation should be performed on the corresponding data indicated within the instruction.

[0067] Processing core 159 is coupled with bus 141 for communicating with various other system devices, which may include but are not limited to, for example, synchronous dynamic random access memory (SDRAM) control 146, static random access memory (SRAM) control 147, burst flash memory interface 148, personal computer memory card international association (PCMCIA)/compact flash (CF) card control 149, liquid crystal display (LCD) control 150, direct memory access (DMA) controller 151, and alternative bus master interface 152. In one embodiment, data processing system 140 may also comprise an I/O bridge 154 for communicating with various I/O devices via an I/O bus 153. Such I/O devices may include but are not limited to, for example, universal asynchronous receiver/transmitter (UART) 155, universal serial bus (USB) 156, Bluetooth wireless UART 157 and I/O expansion interface 158.

[0068] One embodiment of data processing system 140 provides for mobile, network and/or wireless communications and a processing core 159 capable of performing SIMD operations including a text string comparison operation. Processing core 159 may be programmed with various audio, video, imaging and communications algorithms including discrete transformations such as a Walsh-Hadamard transform, a fast Fourier transform (FFT), a discrete cosine transform (DCT), and their respective inverse transforms; compression/decompression techniques such as color space transformation, video encode motion estimation or video decode motion compensation; and modulation/demodulation (MODEM) functions such as pulse coded modulation (PCM).

[0069] **Figure 1C** illustrates another alternative embodiments of a data processing system capable of executing instructions to provide SIMD vector packed tuple cross-comparison functionality. In accordance with one alternative embodiment, data processing system 160 may include a main processor 166, a SIMD coprocessor 161, a cache memory 167, and an input/output system 168. The input/output system 168 may optionally be coupled to a wireless interface 169. SIMD coprocessor 161 is capable of performing operations including instructions in accordance with one embodiment. Processing core 170 may be suitable for manufacture in one or more process technologies and by being represented on a machine

readable media in sufficient detail, may be suitable to facilitate the manufacture of all or part of data processing system 160 including processing core 170.

[0070] For one embodiment, SIMD coprocessor 161 comprises an execution unit 162 and a set of register file(s) 164. One embodiment of main processor 166 comprises a decoder 165 to recognize instructions of instruction set 163 including instructions in accordance with one embodiment for execution by execution unit 162. For alternative embodiments, SIMD coprocessor 161 also comprises at least part of decoder 165B to decode instructions of instruction set 163. Processing core 170 also includes additional circuitry (not shown) which is not necessary to the understanding of embodiments of the present invention.

[0071] In operation, the main processor 166 executes a stream of data processing instructions that control data processing operations of a general type including interactions with the cache memory 167, and the input/output system 168. Embedded within the stream of data processing instructions are SIMD coprocessor instructions. The decoder 165 of main processor 166 recognizes these SIMD coprocessor instructions as being of a type that should be executed by an attached SIMD coprocessor 161. Accordingly, the main processor 166 issues these SIMD coprocessor instructions (or control signals representing SIMD coprocessor instructions) on the coprocessor bus 171 where from they are received by any attached SIMD coprocessors. In this case, the SIMD coprocessor 161 will accept and execute any received SIMD coprocessor instructions intended for it.

[0072] Data may be received via wireless interface 169 for processing by the SIMD coprocessor instructions. For one example, voice communication may be received in the form of a digital signal, which may be processed by the SIMD coprocessor instructions to regenerate digital audio samples representative of the voice communications. For another example, compressed audio and/or video may be received in the form of a digital bit stream, which may be processed by the SIMD coprocessor instructions to regenerate digital audio samples and/or motion video frames. For one embodiment of processing core 170, main processor 166, and a SIMD coprocessor 161 are integrated into a single processing core 170 comprising an execution unit 162, a set of register file(s) 164, and a decoder 165 to recognize instructions of instruction set 163 including instructions in accordance with one embodiment.

[0073] **Figure 2** is a block diagram of the micro-architecture for a processor 200 that includes logic circuits to perform instructions in accordance with one embodiment of the present invention. In some embodiments, an instruction in accordance with one embodiment

can be implemented to operate on data elements having sizes of byte, word, doubleword, quadword, etc., as well as datatypes, such as single and double precision integer and floating point datatypes. In one embodiment the in-order front end 201 is the part of the processor 200 that fetches instructions to be executed and prepares them to be used later in the processor pipeline. The front end 201 may include several units. In one embodiment, the instruction prefetcher 226 fetches instructions from memory and feeds them to an instruction decoder 228 which in turn decodes or interprets them. For example, in one embodiment, the decoder decodes a received instruction into one or more operations called “micro-instructions” or “micro-operations” (also called micro op or uops) that the machine can execute. In other embodiments, the decoder parses the instruction into an opcode and corresponding data and control fields that are used by the micro-architecture to perform operations in accordance with one embodiment. In one embodiment, the trace cache 230 takes decoded uops and assembles them into program ordered sequences or traces in the uop queue 234 for execution. When the trace cache 230 encounters a complex instruction, the microcode ROM 232 provides the uops needed to complete the operation.

[0074] Some instructions are converted into a single micro-op, whereas others need several micro-ops to complete the full operation. In one embodiment, if more than four micro-ops are needed to complete a instruction, the decoder 228 accesses the microcode ROM 232 to do the instruction. For one embodiment, an instruction can be decoded into a small number of micro ops for processing at the instruction decoder 228. In another embodiment, an instruction can be stored within the microcode ROM 232 should a number of micro-ops be needed to accomplish the operation. The trace cache 230 refers to a entry point programmable logic array (PLA) to determine a correct micro-instruction pointer for reading the micro-code sequences to complete one or more instructions in accordance with one embodiment from the micro-code ROM 232. After the microcode ROM 232 finishes sequencing micro-ops for an instruction, the front end 201 of the machine resumes fetching micro-ops from the trace cache 230.

[0075] The out-of-order execution engine 203 is where the instructions are prepared for execution. The out-of-order execution logic has a number of buffers to smooth out and re-order the flow of instructions to optimize performance as they go down the pipeline and get scheduled for execution. The allocator logic allocates the machine buffers and resources that each uop needs in order to execute. The register renaming logic renames logic registers onto

entries in a register file. The allocator also allocates an entry for each uop in one of the two uop queues, one for memory operations and one for non-memory operations, in front of the instruction schedulers: memory scheduler, fast scheduler 202, slow/general floating point scheduler 204, and simple floating point scheduler 206. The uop schedulers 202, 204, 206, determine when a uop is ready to execute based on the readiness of their dependent input register operand sources and the availability of the execution resources the uops need to complete their operation. The fast scheduler 202 of one embodiment can schedule on each half of the main clock cycle while the other schedulers can only schedule once per main processor clock cycle. The schedulers arbitrate for the dispatch ports to schedule uops for execution.

[0076] Register files 208, 210, sit between the schedulers 202, 204, 206, and the execution units 212, 214, 216, 218, 220, 222, 224 in the execution block 211. There is a separate register file 208, 210, for integer and floating point operations, respectively. Each register file 208, 210, of one embodiment also includes a bypass network that can bypass or forward just completed results that have not yet been written into the register file to new dependent uops. The integer register file 208 and the floating point register file 210 are also capable of communicating data with the other. For one embodiment, the integer register file 208 is split into two separate register files, one register file for the low order 32 bits of data and a second register file for the high order 32 bits of data. The floating point register file 210 of one embodiment has 128 bit wide entries because floating point instructions typically have operands from 64 to 128 bits in width.

[0077] The execution block 211 contains the execution units 212, 214, 216, 218, 220, 222, 224, where the instructions are actually executed. This section includes the register files 208, 210, that store the integer and floating point data operand values that the micro-instructions need to execute. The processor 200 of one embodiment is comprised of a number of execution units: address generation unit (AGU) 212, AGU 214, fast ALU 216, fast ALU 218, slow ALU 220, floating point ALU 222, floating point move unit 224. For one embodiment, the floating point execution blocks 222, 224, execute floating point, MMX, SIMD, and SSE, or other operations. The floating point ALU 222 of one embodiment includes a 64 bit by 64 bit floating point divider to execute divide, square root, and remainder micro-ops. For embodiments of the present invention, instructions involving a floating point value may be handled with the floating point hardware. In one embodiment, the ALU

operations go to the high-speed ALU execution units 216, 218. The fast ALUs 216, 218, of one embodiment can execute fast operations with an effective latency of half a clock cycle. For one embodiment, most complex integer operations go to the slow ALU 220 as the slow ALU 220 includes integer execution hardware for long latency type of operations, such as a multiplier, shifts, flag logic, and branch processing. Memory load/store operations are executed by the AGUs 212, 214. For one embodiment, the integer ALUs 216, 218, 220, are described in the context of performing integer operations on 64 bit data operands. In alternative embodiments, the ALUs 216, 218, 220, can be implemented to support a variety of data bits including 16, 32, 128, 256, etc. Similarly, the floating point units 222, 224, can be implemented to support a range of operands having bits of various widths. For one embodiment, the floating point units 222, 224, can operate on 128 bits wide packed data operands in conjunction with SIMD and multimedia instructions.

[0078] In one embodiment, the uops schedulers 202, 204, 206, dispatch dependent operations before the parent load has finished executing. As uops are speculatively scheduled and executed in processor 200, the processor 200 also includes logic to handle memory misses. If a data load misses in the data cache, there can be dependent operations in flight in the pipeline that have left the scheduler with temporarily incorrect data. A replay mechanism tracks and re-executes instructions that use incorrect data. Only the dependent operations need to be replayed and the independent ones are allowed to complete. The schedulers and replay mechanism of one embodiment of a processor are also designed to catch instructions that provide SIMD vector packed tuple cross-comparison functionality.

[0079] The term “registers” may refer to the on-board processor storage locations that are used as part of instructions to identify operands. In other words, registers may be those that are usable from the outside of the processor (from a programmer’s perspective). However, the registers of an embodiment should not be limited in meaning to a particular type of circuit. Rather, a register of an embodiment is capable of storing and providing data, and performing the functions described herein. The registers described herein can be implemented by circuitry within a processor using any number of different techniques, such as dedicated physical registers, dynamically allocated physical registers using register renaming, combinations of dedicated and dynamically allocated physical registers, etc. In one embodiment, integer registers store thirty-two bit integer data. A register file of one embodiment also contains eight multimedia SIMD registers for packed data. For the

discussions below, the registers are understood to be data registers designed to hold packed data, such as 64 bits wide MMX™ registers (also referred to as ‘mm’ registers in some instances) in microprocessors enabled with MMX technology from Intel Corporation of Santa Clara, California. These MMX registers, available in both integer and floating point forms, can operate with packed data elements that accompany SIMD and SSE instructions. Similarly, 128 bits wide XMM registers relating to SSE2, SSE3, SSE4, or beyond (referred to generically as “SSEx”) technology can also be used to hold such packed data operands. In one embodiment, in storing packed data and integer data, the registers do not need to differentiate between the two data types. In one embodiment, integer and floating point are either contained in the same register file or different register files. Furthermore, in one embodiment, floating point and integer data may be stored in different registers or the same registers.

[0080] In the examples of the following figures, a number of data operands are described. **Figure 3A** illustrates various packed data type representations in multimedia registers according to one embodiment of the present invention. **Fig. 3A** illustrates data types for a packed byte 310, a packed word 320, and a packed doubleword (dword) 330 for 128 bits wide operands. The packed byte format 310 of this example is 128 bits long and contains sixteen packed byte data elements. A byte is defined here as 8 bits of data. Information for each byte data element is stored in bit 7 through bit 0 for byte 0, bit 15 through bit 8 for byte 1, bit 23 through bit 16 for byte 2, and finally bit 120 through bit 127 for byte 15. Thus, all available bits are used in the register. This storage arrangement increases the storage efficiency of the processor. As well, with sixteen data elements accessed, one operation can now be performed on sixteen data elements in parallel.

[0081] Generally, a data element is an individual piece of data that is stored in a single register or memory location with other data elements of the same length. In packed data sequences relating to SSEx technology, the number of data elements stored in a XMM register is 128 bits divided by the length in bits of an individual data element. Similarly, in packed data sequences relating to MMX and SSE technology, the number of data elements stored in an MMX register is 64 bits divided by the length in bits of an individual data element. Although the data types illustrated in **Fig. 3A** are 128 bit long, embodiments of the present invention can also operate with 64 bit wide, 256 bit wide, 512 bit wide, or other sized operands. The packed word format 320 of this example is 128 bits long and contains eight

packed word data elements. Each packed word contains sixteen bits of information. The packed doubleword format 330 of **Fig. 3A** is 128 bits long and contains four packed doubleword data elements. Each packed doubleword data element contains thirty two bits of information. A packed quadword is 128 bits long and contains two packed quad-word data elements.

[0082] **Figure 3B** illustrates alternative in-register data storage formats. Each packed data can include more than one independent data element. Three packed data formats are illustrated; packed half 341, packed single 342, and packed double 343. One embodiment of packed half 341, packed single 342, and packed double 343 contain fixed-point data elements. For an alternative embodiment one or more of packed half 341, packed single 342, and packed double 343 may contain floating-point data elements. One alternative embodiment of packed half 341 is one hundred twenty-eight bits long containing eight 16-bit data elements. One embodiment of packed single 342 is one hundred twenty-eight bits long and contains four 32-bit data elements. One embodiment of packed double 343 is one hundred twenty-eight bits long and contains two 64-bit data elements. It will be appreciated that such packed data formats may be further extended to other register lengths, for example, to 96-bits, 160-bits, 192-bits, 224-bits, 256-bits, 512-bits or more.

[0083] **Figure 3C** illustrates various signed and unsigned packed data type representations in multimedia registers according to one embodiment of the present invention. Unsigned packed byte representation 344 illustrates the storage of an unsigned packed byte in a SIMD register. Information for each byte data element is stored in bit seven through bit zero for byte zero, bit fifteen through bit eight for byte one, bit twenty-three through bit sixteen for byte two, etc., and finally bit one hundred twenty through bit one hundred twenty-seven for byte fifteen. Thus, all available bits are used in the register. This storage arrangement can increase the storage efficiency of the processor. As well, with sixteen data elements accessed, one operation can now be performed on sixteen data elements in a parallel fashion. Signed packed byte representation 345 illustrates the storage of a signed packed byte. Note that the eighth bit of every byte data element is the sign indicator. Unsigned packed word representation 346 illustrates how word seven through word zero are stored in a SIMD register. Signed packed word representation 347 is similar to the unsigned packed word in-register representation 346. Note that the sixteenth bit of each word data element is the sign indicator. Unsigned packed doubleword representation 348 shows how doubleword

data elements are stored. Signed packed doubleword representation 349 is similar to unsigned packed doubleword in-register representation 348. Note that the necessary sign bit is the thirty-second bit of each doubleword data element.

[0084] **Figure 3D** is a depiction of one embodiment of an operation encoding (opcode) format 360, having thirty-two or more bits, and register/memory operand addressing modes corresponding with a type of opcode format described in the “Intel® 64 and IA-32 Intel Architecture Software Developer’s Manual Combined Volumes 2A and 2B: Instruction Set Reference A-Z,” which is available from Intel Corporation, Santa Clara, CA on the world-wide-web (www) at intel.com/products/processor/manuals/. In one embodiment, an instruction may be encoded by one or more of fields 361 and 362. Up to two operand locations per instruction may be identified, including up to two source operand identifiers 364 and 365. For one embodiment, destination operand identifier 366 is the same as source operand identifier 364, whereas in other embodiments they are different. For an alternative embodiment, destination operand identifier 366 is the same as source operand identifier 365, whereas in other embodiments they are different. In one embodiment, one of the source operands identified by source operand identifiers 364 and 365 is overwritten by the results of the instruction, whereas in other embodiments identifier 364 corresponds to a source register element and identifier 365 corresponds to a destination register element. For one embodiment, operand identifiers 364 and 365 may be used to identify 32-bit or 64-bit source and destination operands.

[0085] **Figure 3E** is a depiction of another alternative operation encoding (opcode) format 370, having forty or more bits. Opcode format 370 corresponds with opcode format 360 and comprises an optional prefix byte 378. An instruction according to one embodiment may be encoded by one or more of fields 378, 371, and 372. Up to two operand locations per instruction may be identified by source operand identifiers 374 and 375 and by prefix byte 378. For one embodiment, prefix byte 378 may be used to identify 32-bit or 64-bit source and destination operands. For one embodiment, destination operand identifier 376 is the same as source operand identifier 374, whereas in other embodiments they are different. For an alternative embodiment, destination operand identifier 376 is the same as source operand identifier 375, whereas in other embodiments they are different. In one embodiment, an instruction operates on one or more of the operands identified by operand identifiers 374 and 375 and one or more operands identified by the operand identifiers 374 and 375 is overwritten

by the results of the instruction, whereas in other embodiments, operands identified by identifiers 374 and 375 are written to another data element in another register. Opcode formats 360 and 370 allow register to register, memory to register, register by memory, register by register, register by immediate, register to memory addressing specified in part by MOD fields 363 and 373 and by optional scale-index-base and displacement bytes.

[0086] Turning next to **Figure 3F**, in some alternative embodiments, 64-bit (or 128-bit, or 256-bit, or 512-bit or more) single instruction multiple data (SIMD) arithmetic operations may be performed through a coprocessor data processing (CDP) instruction. Operation encoding (opcode) format 380 depicts one such CDP instruction having CDP opcode fields 382 and 389. The type of CDP instruction, for alternative embodiments, operations may be encoded by one or more of fields 383, 384, 387, and 388. Up to three operand locations per instruction may be identified, including up to two source operand identifiers 385 and 390 and one destination operand identifier 386. One embodiment of the coprocessor can operate on 8, 16, 32, and 64 bit values. For one embodiment, an instruction is performed on integer data elements. In some embodiments, an instruction may be executed conditionally, using condition field 381. For some embodiments, source data sizes may be encoded by field 383. In some embodiments, Zero (Z), negative (N), carry (C), and overflow (V) detection can be done on SIMD fields. For some instructions, the type of saturation may be encoded by field 384.

[0087] Turning next to **Figure 3G** is a depiction of another alternative operation encoding (opcode) format 397, to provide SIMD vector packed tuple cross-comparison functionality according to another embodiment, corresponding with a type of opcode format described in the “Intel® Advanced Vector Extensions Programming Reference,” which is available from Intel Corp., Santa Clara, CA on the world-wide-web (www) at intel.com/products/processor/manuals/.

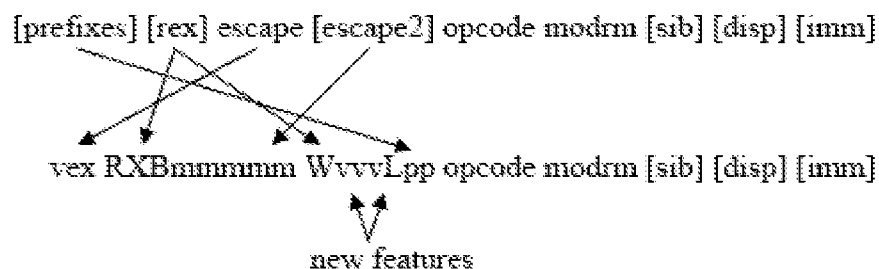
[0088] The original x86 instruction set provided for a 1-byte opcode with various formats of address syllable and immediate operand contained in additional bytes whose presence was known from the first "opcode" byte. Additionally, there were certain byte values that were reserved as modifiers to the opcode (called prefixes, as they had to be placed before the instruction). When the original palette of 256 opcode bytes (including these special prefix values) was exhausted, a single byte was dedicated as an escape to a new set of 256 opcodes. As vector instructions (e.g., SIMD) were added, a need for more opcodes was generated, and

the "two byte" opcode map also was insufficient, even when expanded through the use of prefixes. To this end, new instructions were added in additional maps which use 2 bytes plus an optional prefix as an identifier.

[0089] Additionally, in order to facilitate additional registers in 64-bit mode, an additional prefix may be used (called "REX") in between the prefixes and the opcode (and any escape bytes necessary to determine the opcode). In one embodiment, the REX may have 4 "payload" bits to indicate use of additional registers in 64-bit mode. In other embodiments it may have fewer or more than 4 bits. The general format of at least one instruction set (which corresponds generally with format 360 and/or format 370) is illustrated generically by the following:

[prefixes] [rex] escape [escape2] opcode modrm (etc.)

[0090] Opcode format 397 corresponds with opcode format 370 and comprises optional VEX prefix bytes 391 (beginning with C4 hex in one embodiment) to replace most other commonly used legacy instruction prefix bytes and escape codes. For example, the following illustrates an embodiment using two fields to encode an instruction, which may be used when a second escape code is present in the original instruction, or when extra bits (e.g. the XB and W fields) in the REX field need to be used. In the embodiment illustrated below, legacy escape is represented by a new escape value, legacy prefixes are fully compressed as part of the "payload" bytes, legacy prefixes are reclaimed and available for future expansion, the second escape code is compressed in a "map" field, with future map or feature space available, and new features are added (e.g., increased vector length and an additional source register specifier).



[0091] An instruction according to one embodiment may be encoded by one or more of fields 391 and 392. Up to four operand locations per instruction may be identified by field 391 in combination with source operand identifiers 374 and 375 and in combination with an optional scale-index-base (SIB) identifier 393, an optional displacement identifier 394, and an optional immediate byte 395. For one embodiment, VEX prefix bytes 391 may be used to

identify 32-bit or 64-bit source and destination operands and/or 128-bit or 256-bit SIMD register or memory operands. For one embodiment, the functionality provided by opcode format 397 may be redundant with opcode format 370, whereas in other embodiments they are different. Opcode formats 370 and 397 allow register to register, memory to register, register by memory, register by register, register by immediate, register to memory addressing specified in part by MOD field 373 and by optional (SIB) identifier 393, an optional displacement identifier 394, and an optional immediate byte 395.

[0092] Turning next to **Figure 3H** is a depiction of another alternative operation encoding (opcode) format 398, to provide SIMD vector packed tuple cross-comparison functionality according to another embodiment. Opcode format 398 corresponds with opcode formats 370 and 397 and comprises optional EVEX prefix bytes 396 (beginning with 62 hex in one embodiment) to replace most other commonly used legacy instruction prefix bytes and escape codes and provide additional functionality. An instruction according to one embodiment may be encoded by one or more of fields 396 and 392. Up to four operand locations per instruction and a mask may be identified by field 396 in combination with source operand identifiers 374 and 375 and in combination with an optional scale-index-base (SIB) identifier 393, an optional displacement identifier 394, and an optional immediate byte 395. For one embodiment, EVEX prefix bytes 396 may be used to identify 32-bit or 64-bit source and destination operands and/or 128-bit, 256-bit or 512-bit SIMD register or memory operands. For one embodiment, the functionality provided by opcode format 398 may be redundant with opcode formats 370 or 397, whereas in other embodiments they are different. Opcode format 398 allows register to register, memory to register, register by memory, register by register, register by immediate, register to memory addressing, with masks, specified in part by MOD field 373 and by optional (SIB) identifier 393, an optional displacement identifier 394, and an optional immediate byte 395. The general format of at least one instruction set (which corresponds generally with format 360 and/or format 370) is illustrated generically by the following:

evex1 RxBmmmmm WvvvLpp evex4 opcode modrm [sib] [disp] [imm]

[0093] For one embodiment an instruction encoded according to the EVEX format 398 may have additional "payload" bits that may be used to provide SIMD vector packed tuple cross-comparison functionality with additional new features such as, for example, a user

configurable mask register, or an additional operand, or selections from among 128-bit, 256-bit or 512-bit vector registers, or more registers from which to select, etc.

[0094] For example, where VEX format 397 may be used to provide SIMD vector packed tuple cross-comparison functionality without a mask, the EVEX format 398 may be used to provide SIMD vector packed tuple cross-comparison functionality with an explicit user configurable mask. Additionally, where VEX format 397 may be used to provide SIMD vector packed tuple cross-comparison functionality on 128-bit or 256-bit vector registers, EVEX format 398 may be used to provide SIMD vector packed tuple cross-comparison functionality on 128-bit, 256-bit, 512-bit or larger (or smaller) vector registers.

[0095] Example instructions to provide SIMD vector packed tuple cross-comparison functionality for efficient vector address conflict resolution are illustrated by the following examples:

<i>Instruction</i>	<i>dest/ source1</i>	<i>source2</i>	<i>source3</i>	<i>source4</i>	<i>source5</i>	<i>description</i>
VPDCMPUW Vector packed double compare unsigned words	Vmm1	Vmm2	Vmm3/ Mem1	Mask1	Imm8	Compare each 16-bit unsigned element of every double (2-tuple) in register Vmm2, if a corresponding mask bit in Mask1 is 1, with every element in the corresponding double in register Vmm3 or at memory location, Mem1, according to the comparison type in Imm8 and store the comparison masks as corresponding elements in destination register Vmm1.
VPQCMPD Vector packed quadruple compare [signed] doublewords	Vmm1	Vmm2/ Mem1	Mask1	Imm8		Compare each 32-bit signed element of every quadruple (4-tuple) in source/destination register Vmm1, if a corresponding mask bit in Mask1 is 1, with every element in the corresponding quadruple in register Vmm2 or at memory location, Mem1, according to the comparison type in Imm8 and store the comparison masks as corresponding elements in register Vmm1.
VPOSHCMPB Vector packed octuple shift compare [signed] bytes	Vmm1	Vmm2	Vmm3/ Mem1	Mask1	Imm8	Compare each 8-bit signed element of every octuple (8-tuple) in register Vmm2, if a corresponding mask bit in Mask1 is 1, with every element in the corresponding octuple in register Vmm3 or at memory location, Mem1, according to the comparison type in Imm8, left shift resulting comparison masks respectively by shift counts in source/destination register Vmm1. Store the left shifted comparison masks as elements in register Vmm1.
VPDCMPQ Vector packed double compare [signed] quadwords	Vmm1	Vmm2	Vmm3/ Mem1	Mask1	Imm8	Compare each 64-bit signed element of every double (2-tuple) in register Vmm2, if a corresponding mask bit in Mask1 is 1, with every element in the corresponding double in register Vmm3 or at memory location, Mem1, according to the comparison type in Imm8 and store the comparison masks as corresponding elements in destination register Vmm1.

[0096] When the elements being compared, e.g. indices, are the same size as the masks required to represent comparison results (e.g. sixteen 16-bit elements in 256-bit registers) then performing an instruction to cross-compare all elements may be appropriate. But when

there are more elements to compare than bits available to represent the comparison results, one may need an alternative solution. It will be appreciated that SIMD vector packed tuple comparison instructions, as in the examples above, may be used as an alternative solution for variable sized elements and memory offsets to provide SIMD vector address conflict detection functionality and the results combined to generate conflict masks for efficient SIMD address conflict resolution.

[0097] Examples include processors with a first and a second registers with a variable plurality of data fields, each of the data fields to store an element of a first data type. The processor in some embodiments executes a SIMD instruction for vector packed tuple (e.g. double, quadruple or octuple) cross-comparison, which for each data field of a portion of data fields constituting a tuple of the first register, compares its corresponding element with every element of a corresponding portion of data fields of a tuple in the second register (or vector in memory), and sets a mask bit corresponding to each element of the second vector portion, in a bit-mask corresponding to each unmasked element of the corresponding first register portion, according to the corresponding comparison. The packed tuples may each comprise two, four or eight elements, wherein the elements may be signed or unsigned packed bytes (8 bits), words (16 bits), doublewords (32 bits) or quadwords (64 bits). In some embodiments any comparison bit-masks may also be left shifted by corresponding elements in data fields of another source/destination register before being stored into the source/destination register. The comparison type may typically be indicated (e.g. as one of equal, less-than, less-than or equal, always false, not equal, not less-than, not less-than or equal, and always true) by an immediate operand specified by the instruction. It will also be appreciated that SIMD vector leading zero count instructions may be used with variable sized elements and conflict masks to provide SIMD permute controls, which will be described in greater detail below (e.g. with regard to Figure 14A), thereby permitting efficient SIMD address conflict resolution in registers without performing dependent computations through memory. Thus the instructions disclosed herein provide for efficient SIMD address conflict resolution especially in gather-modify-scatter applications.

[0098] **Figure 4A** is a block diagram illustrating an in-order pipeline and a register renaming stage, out-of-order issue/execution pipeline according to at least one embodiment of the invention. **Figure 4B** is a block diagram illustrating an in-order architecture core and a register renaming logic, out-of-order issue/execution logic to be included in a processor

according to at least one embodiment of the invention. The solid lined boxes in **Figure 4A** illustrate the in-order pipeline, while the dashed lined boxes illustrates the register renaming, out-of-order issue/execution pipeline. Similarly, the solid lined boxes in **Figure 4B** illustrate the in-order architecture logic, while the dashed lined boxes illustrates the register renaming logic and out-of-order issue/execution logic.

[0099] In **Figure 4A**, a processor pipeline 400 includes a fetch stage 402, a length decode stage 404, a decode stage 406, an allocation stage 408, a renaming stage 410, a scheduling (also known as a dispatch or issue) stage 412, a register read/memory read stage 414, an execute stage 416, a write back/memory write stage 418, an exception handling stage 422, and a commit stage 424.

[00100] In **Figure 4B**, arrows denote a coupling between two or more units and the direction of the arrow indicates a direction of data flow between those units. Figure 4B shows processor core 490 including a front end unit 430 coupled to an execution engine unit 450, and both are coupled to a memory unit 470.

[00101] The core 490 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 490 may be a special-purpose core, such as, for example, a network or communication core, compression engine, graphics core, or the like.

[00102] The front end unit 430 includes a branch prediction unit 432 coupled to an instruction cache unit 434, which is coupled to an instruction translation lookaside buffer (TLB) 436, which is coupled to an instruction fetch unit 438, which is coupled to a decode unit 440. The decode unit 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 decoder 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. The instruction cache unit 434 is further coupled to a level 2 (L2) cache unit 476 in the memory unit 470. The decode unit 440 is coupled to a rename/allocator unit 452 in the execution engine unit 450.

[00103] The execution engine unit 450 includes the rename/allocator unit 452 coupled to a retirement unit 454 and a set of one or more scheduler unit(s) 456. The scheduler unit(s) 456 represents any number of different schedulers, including reservations stations, central instruction window, etc. The scheduler unit(s) 456 is coupled to the physical register file(s) unit(s) 458. Each of the physical register file(s) units 458 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, etc., status (e.g., an instruction pointer that is the address of the next instruction to be executed), etc. The physical register file(s) unit(s) 458 is overlapped by the retirement unit 454 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.). Generally, the architectural registers are visible from the outside of the processor or from a programmer's perspective. The registers are not limited to any known particular type of circuit. Various different types of registers are suitable as long as they are capable of storing and providing data as described herein. Examples of suitable registers include, but are not limited to, dedicated physical registers, dynamically allocated physical registers using register renaming, combinations of dedicated and dynamically allocated physical registers, etc. The retirement unit 454 and the physical register file(s) unit(s) 458 are coupled to the execution cluster(s) 460. The execution cluster(s) 460 includes a set of one or more execution units 462 and a set of one or more memory access units 464. The execution units 462 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) 456, physical register file(s) unit(s) 458, and execution cluster(s) 460 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) 464). 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.

[00104] The set of memory access units 464 is coupled to the memory unit 470, which includes a data TLB unit 472 coupled to a data cache unit 474 coupled to a level 2 (L2) cache unit 476. In one exemplary embodiment, the memory access units 464 may include a load unit, a store address unit, and a store data unit, each of which is coupled to the data TLB unit 472 in the memory unit 470. The L2 cache unit 476 is coupled to one or more other levels of cache and eventually to a main memory.

[00105] By way of example, the exemplary register renaming, out-of-order issue/execution core architecture may implement the pipeline 400 as follows: 1) the instruction fetch 438 performs the fetch and length decoding stages 402 and 404; 2) the decode unit 440 performs the decode stage 406; 3) the rename/allocator unit 452 performs the allocation stage 408 and renaming stage 410; 4) the scheduler unit(s) 456 performs the schedule stage 412; 5) the physical register file(s) unit(s) 458 and the memory unit 470 perform the register read/memory read stage 414; the execution cluster 460 perform the execute stage 416; 6) the memory unit 470 and the physical register file(s) unit(s) 458 perform the write back/memory write stage 418; 7) various units may be involved in the exception handling stage 422; and 8) the retirement unit 454 and the physical register file(s) unit(s) 458 perform the commit stage 424.

[00106] The core 490 may support one or more instructions sets (e.g., the x86 instruction set (with some extensions that have been added with newer versions); the MIPS instruction set of MIPS Technologies of Sunnyvale, CA; the ARM instruction set (with optional additional extensions such as NEON) of ARM Holdings of Sunnyvale, CA).

[00107] It should be understood that the core may support multithreading (executing two or more parallel sets of operations or threads), and may do so in a variety of ways including time sliced multithreading, simultaneous multithreading (where a single physical core provides a logical core for each of the threads that physical core is simultaneously multithreading), or a combination thereof (e.g., time sliced fetching and decoding and simultaneous multithreading thereafter such as in the Intel® Hyperthreading technology).

[00108] 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 a separate instruction and data cache units 434/474 and a shared L2 cache unit 476, 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.

[00109] **Figure 5** is a block diagram of a single core processor and a multicore processor 500 with integrated memory controller and graphics according to embodiments of the invention. The solid lined boxes in **Figure 5** illustrate a processor 500 with a single core 502A, a system agent 510, a set of one or more bus controller units 516, while the optional addition of the dashed lined boxes illustrates an alternative processor 500 with multiple cores 502A-N, a set of one or more integrated memory controller unit(s) 514 in the system agent unit 510, and an integrated graphics logic 508.

[00110] The memory hierarchy includes one or more levels of cache within the cores, a set or one or more shared cache units 506, and external memory (not shown) coupled to the set of integrated memory controller units 514. The set of shared cache units 506 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 512 interconnects the integrated graphics logic 508, the set of shared cache units 506, and the system agent unit 510, alternative embodiments may use any number of well-known techniques for interconnecting such units.

[00111] In some embodiments, one or more of the cores 502A-N are capable of multi-threading. The system agent 510 includes those components coordinating and operating cores 502A-N. The system agent unit 510 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 502A-N and the integrated graphics logic 508. The display unit is for driving one or more externally connected displays.

[00112] The cores 502A-N may be homogenous or heterogeneous in terms of architecture and/or instruction set. For example, some of the cores 502A-N may be in order while others are out-of-order. As another example, two or more of the cores 502A-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.

[00113] The processor may be a general-purpose processor, such as a Core™ i3, i5, i7, 2 Duo and Quad, Xeon™, Itanium™, XScale™ or StrongARM™ processor, which are available from Intel Corporation, of Santa Clara, Calif. Alternatively, the processor may be from another company, such as ARM Holdings, Ltd, MIPS, etc.. The processor may be a special-purpose processor, such as, for example, a network or communication processor, compression engine, graphics processor, co-processor, embedded processor, or the like. The processor may be implemented on one or more chips. The processor 500 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.

[00114] Figures 6-8 are exemplary systems suitable for including the processor 500, while Figure 9 is an exemplary system on a chip (SoC) that may include one or more of the cores 502. 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.

[00115] Referring now to **Figure 6**, shown is a block diagram of a system 600 in accordance with one embodiment of the present invention. The system 600 may include one or more processors 610, 615, which are coupled to graphics memory controller hub (GMCH) 620. The optional nature of additional processors 615 is denoted in **Figure 6** with broken lines.

[00116] Each processor 610,615 may be some version of the processor 500. However, it should be noted that it is unlikely that integrated graphics logic and integrated memory control units would exist in the processors 610,615. Figure 6 illustrates that the GMCH 620 may be coupled to a memory 640 that may be, for example, a dynamic random access memory (DRAM). The DRAM may, for at least one embodiment, be associated with a non-volatile cache.

[00117] The GMCH 620 may be a chipset, or a portion of a chipset. The GMCH 620 may communicate with the processor(s) 610, 615 and control interaction between the processor(s) 610, 615 and memory 640. The GMCH 620 may also act as an accelerated bus interface between the processor(s) 610, 615 and other elements of the system 600. For at least one embodiment, the GMCH 620 communicates with the processor(s) 610, 615 via a multi-drop bus, such as a frontside bus (FSB) 695.

[00118] Furthermore, GMCH 620 is coupled to a display 645 (such as a flat panel display). GMCH 620 may include an integrated graphics accelerator. GMCH 620 is further coupled to an input/output (I/O) controller hub (ICH) 650, which may be used to couple various peripheral devices to system 600. Shown for example in the embodiment of Figure 6 is an external graphics device 660, which may be a discrete graphics device coupled to ICH 650, along with another peripheral device 670.

[00119] Alternatively, additional or different processors may also be present in the system 600. For example, additional processor(s) 615 may include additional processors(s) that are the same as processor 610, additional processor(s) that are heterogeneous or asymmetric to processor 610, accelerators (such as, e.g., graphics accelerators or digital signal processing (DSP) units), field programmable gate arrays, or any other processor. There can be a variety of differences between the physical resources 610, 615 in terms of a spectrum of metrics of merit including architectural, micro-architectural, thermal, power consumption characteristics, and the like. These differences may effectively manifest themselves as asymmetry and heterogeneity amongst the processors 610, 615. For at least one embodiment, the various processors 610, 615 may reside in the same die package.

[00120] Referring now to **Figure 7**, shown is a block diagram of a second system 700 in accordance with an embodiment of the present invention. As shown in **Figure 7**, multiprocessor system 700 is a point-to-point interconnect system, and includes a first processor 770 and a second processor 780 coupled via a point-to-point interconnect 750. Each of processors 770 and 780 may be some version of the processor 500 as one or more of the processors 610, 615.

[00121] While shown with only two processors 770, 780, it is to be understood that the scope of the present invention is not so limited. In other embodiments, one or more additional processors may be present in a given processor.

[00122] Processors 770 and 780 are shown including integrated memory controller units 772 and 782, respectively. Processor 770 also includes as part of its bus controller units point-to-point (P-P) interfaces 776 and 778; similarly, second processor 780 includes P-P interfaces 786 and 788. Processors 770, 780 may exchange information via a point-to-point (P-P) interface 750 using P-P interface circuits 778, 788. As shown in Figure 7, IMCs 772 and 782 couple the processors to respective memories, namely a memory 732 and a memory 734, which may be portions of main memory locally attached to the respective processors.

[00123] Processors 770, 780 may each exchange information with a chipset 790 via individual P-P interfaces 752, 754 using point to point interface circuits 776, 794, 786, 798. Chipset 790 may also exchange information with a high-performance graphics circuit 738 via a high-performance graphics interface 739.

[00124] 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.

[00125] Chipset 790 may be coupled to a first bus 716 via an interface 796. In one embodiment, first bus 716 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.

[00126] As shown in **Figure 7**, various I/O devices 714 may be coupled to first bus 716, along with a bus bridge 718 which couples first bus 716 to a second bus 720. In one embodiment, second bus 720 may be a low pin count (LPC) bus. Various devices may be coupled to second bus 720 including, for example, a keyboard and/or mouse 722, communication devices 727 and a storage unit 728 such as a disk drive or other mass storage device which may include instructions/code and data 730, in one embodiment. Further, an audio I/O 724 may be coupled to second bus 720. Note that other architectures are possible. For example, instead of the point-to-point architecture of **Figure 7**, a system may implement a multi-drop bus or other such architecture.

[00127] Referring now to **Figure 8**, shown is a block diagram of a third system 800 in accordance with an embodiment of the present invention. Like elements in **Figure 7** and **Figure 8** bear like reference numerals, and certain aspects of **Figure 7** have been omitted from **Figure 8** in order to avoid obscuring other aspects of **Figure 8**.

[00128] **Figure 8** illustrates that the processors 870, 880 may include integrated memory and I/O control logic (“CL”) 872 and 882, respectively. For at least one embodiment, the CL 872, 882 may include integrated memory controller units such as that described above in connection with **Figures 5** and **7**. In addition, CL 872, 882 may also include I/O control logic. **Figure 8** illustrates that not only are the memories 832, 834 coupled to the CL 872, 882, but also that I/O devices 814 are also coupled to the control logic 872, 882. Legacy I/O devices 815 are coupled to the chipset 890.

[00129] Referring now to **Figure 9**, shown is a block diagram of a SoC 900 in accordance with an embodiment of the present invention. Similar elements in Figure 5 bear like reference numerals. Also, dashed lined boxes are optional features on more advanced SoCs. In **Figure 9**, an interconnect unit(s) 902 is coupled to: an application processor 910 which includes a set of one or more cores 502A-N and shared cache unit(s) 506; a system agent unit 510; a bus controller unit(s) 516; an integrated memory controller unit(s) 514; a set of one or more media processors 920 which may include integrated graphics logic 508, an image processor 924 for providing still and/or video camera functionality, an audio processor 926 for providing hardware audio acceleration, and a video processor 928 for providing video encode/decode acceleration; an static random access memory (SRAM) unit 930; a direct memory access (DMA) unit 932; and a display unit 940 for coupling to one or more external displays.

[00130] **Figure 10** illustrates a processor containing a central processing unit (CPU) and a graphics processing unit (GPU), which may perform at least one instruction according to one embodiment. In one embodiment, an instruction to perform operations according to at least one embodiment could be performed by the CPU. In another embodiment, the instruction could be performed by the GPU. In still another embodiment, the instruction may be performed through a combination of operations performed by the GPU and the CPU. For example, in one embodiment, an instruction in accordance with one embodiment may be received and decoded for execution on the GPU. However, one or more operations within the decoded instruction may be performed by a CPU and the result returned to the GPU for final retirement of the instruction. Conversely, in some embodiments, the CPU may act as the primary processor and the GPU as the co-processor.

[00131] In some embodiments, instructions that benefit from highly parallel, throughput processors may be performed by the GPU, while instructions that benefit from the

performance of processors that benefit from deeply pipelined architectures may be performed by the CPU. For example, graphics, scientific applications, financial applications and other parallel workloads may benefit from the performance of the GPU and be executed accordingly, whereas more sequential applications, such as operating system kernel or application code may be better suited for the CPU.

[00132] In Figure 10, processor 1000 includes a CPU 1005, GPU 1010, image processor 1015, video processor 1020, USB controller 1025, UART controller 1030, SPI/SDIO controller 1035, display device 1040, High-Definition Multimedia Interface (HDMI) controller 1045, MIPI controller 1050, flash memory controller 1055, dual data rate (DDR) controller 1060, security engine 1065, and I²S/I²C (Integrated Interchip Sound/Inter-Integrated Circuit) interface 1070. Other logic and circuits may be included in the processor of Figure 10, including more CPUs or GPUs and other peripheral interface controllers.

[00133] 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 the Cortex™ family of processors developed by ARM Holdings, Ltd. and Loongson IP cores developed the Institute of Computing Technology (ICT) of the Chinese Academy of Sciences may be licensed or sold to various customers or licensees, such as Texas Instruments, Qualcomm, Apple, or Samsung and implemented in processors produced by these customers or licensees.

[00134] Figure 11 shows a block diagram illustrating the development of IP cores according to one embodiment. Storage 1130 includes simulation software 1120 and/or hardware or software model 1110. In one embodiment, the data representing the IP core design can be provided to the storage 1130 via memory 1140 (e.g., hard disk), wired connection (e.g., internet) 1150 or wireless connection 1160. The IP core information generated by the simulation tool and model can then be transmitted to a fabrication facility where it can be fabricated by a third party to perform at least one instruction in accordance with at least one embodiment.

[00135] In some embodiments, one or more instructions may correspond to a first type or architecture (e.g., x86) and be translated or emulated on a processor of a different type or architecture (e.g., ARM). An instruction, according to one embodiment, may therefore be performed on any processor or processor type, including ARM, x86, MIPS, a GPU, or other processor type or architecture.

[00136] **Figure 12** illustrates how an instruction of a first type is emulated by a processor of a different type, according to one embodiment. In **Figure 12**, program 1205 contains some instructions that may perform the same or substantially the same function as an instruction according to one embodiment. However the instructions of program 1205 may be of a type and/or format that is different or incompatible with processor 1215, meaning the instructions of the type in program 1205 may not be able to be executed natively by the processor 1215. However, with the help of emulation logic, 1210, the instructions of program 1205 are translated into instructions that are natively capable of being executed by the processor 1215. In one embodiment, the emulation logic is embodied in hardware. In another embodiment, the emulation logic is embodied in a tangible, machine-readable medium containing software to translate instructions of the type in the program 1205 into the type natively executable by the processor 1215. In other embodiments, emulation logic is a combination of fixed-function or programmable hardware and a program stored on a tangible, machine-readable medium. In one embodiment, the processor contains the emulation logic, whereas in other embodiments, the emulation logic exists outside of the processor and is provided by a third party. In one embodiment, the processor is capable of loading the emulation logic embodied in a tangible, machine-readable medium containing software by executing microcode or firmware contained in or associated with the processor.

[00137] **Figure 13** is a block diagram contrasting the use of a software instruction converter to convert binary instructions in a source instruction set to binary instructions in a target instruction set according to embodiments of the invention. In the illustrated embodiment, the instruction converter is a software instruction converter, although alternatively the instruction converter may be implemented in software, firmware, hardware, or various combinations thereof. **Figure 13** shows a program in a high level language 1302 may be compiled using an x86 compiler 1304 to generate x86 binary code 1306 that may be natively executed by a processor with at least one x86 instruction set core 1316. The processor with at least one x86 instruction set core 1316 represents any processor that can

perform substantially the same functions as a 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 1304 represents a compiler that is operable to generate x86 binary code 1306 (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 1316. Similarly, Figure 13 shows the program in the high level language 1302 may be compiled using an alternative instruction set compiler 1308 to generate alternative instruction set binary code 1310 that may be natively executed by a processor without at least one x86 instruction set core 1314 (e.g., a processor with cores that execute the MIPS instruction set of MIPS Technologies of Sunnyvale, CA and/or that execute the ARM instruction set of ARM Holdings of Sunnyvale, CA). The instruction converter 1312 is used to convert the x86 binary code 1306 into code that may be natively executed by the processor without an x86 instruction set core 1314. This converted code is not likely to be the same as the alternative instruction set binary code 1310 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 1312 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 1306.

[00138] Figure 14A illustrates a flow diagram for one embodiment of an example of a process 1401 using a SIMD vector leading zero count instruction for generating permute controls for efficient vector address conflict resolution. Process 1401 and other processes herein disclosed are performed by processing blocks that may comprise dedicated hardware or software or firmware operation codes executable by general purpose machines or by special purpose machines or by a combination of both.

[00139] In processing block 1410 of process 1401 the set of indices is initialized, e.g. to the values 7, 2, 7, 1 and 7. It will be appreciated that the number of offset indices and the values of the offset indices is intended to be illustrative and not restrictive of embodiments of the invention. In particular, the number of offset indices in a register may be determined by

the particular application and/or the size of data elements corresponding to the indices. The example values of offset indices are simply intended to illustrate resolution of dependency conflicts. In processing block 1415 the elements left mask is set to all (e.g. five) ones. In processing block 1420 the dependencies masks may be set according to a technique of process 1405 (which is described in greater detail below with regard to Figure 18) to reflect all matching offset indices. Then in processing block 1425 a set of order masks is initialized. In processing block 1430 the conflict masks are computed by performing a bitwise AND between the dependencies masks and the order masks.

[00140] Proceeding next to processing block 1439, the leading zero counts are computed using a vector packed leading zero count instruction, VPLZCNT, on the conflict masks by counting the number of most significant contiguous bits that are set to zero for each data field of the conflict masks and storing each count as a value in a corresponding data field of the leading zero counts. In processing block 1440 the maximum counts are initialized to all fours, which is a value one less than the number of bits used to represent a conflict mask in the example illustrated. In processing block 1445 a set of permute indices is computed by performing a SIMD packed subtraction, PSUB, subtracting the value of each leading zero count from a corresponding maximum count value, one less than the number of bits used to represent a conflict mask, to generate a corresponding difference of the maximum counts minus the leading zero counts. It will be appreciated that those locations corresponding to offset indices that are not dependent on a prior computation have permute indices of negative one, which may also be used advantageously.

[00141] In processing block 1450 data is gathered from memory according to the offset indices and an elements left mask which is (optionally) all ones. Next in processing block 1455 the elements left mask is recomputed using a packed compare not equal instruction, PCMPNEQ, comparing the permute indices advantageously with all negative ones. In processing block 1460 a determination is made whether the elements left mask is zero (0). If so in processing block 1465 the data is scattered to memory using the offset indices, and the processing of this set of SIMD data ends in processing block 1499.

[00142] Otherwise in processing block 1470 the data is permuted by performing a SIMD vector packed permutation, VPERM, employing the permute indices and the elements left mask as a completion mask to advantageously update only the elements left to process. In processing block 1475 the required computations are performed on the data elements left to

process using the elements left mask as a completion mask. Then in a prior technique process 1406 a set of conflicted out elements is initialized by broadcasting the elements left mask to all elements in processing block 1480. Then the set of conflict masks is recomputed using a vector packed AND instruction on the previous conflict masks and the new conflicted out elements. The new elements left mask is then computed using a packed compare not equal instruction, PCMPNEQ, comparing each conflict mask with all zeroes. Processing then reiterates beginning in processing block 1460 with the testing of a new elements left mask. It will also be appreciated that SIMD vector leading zero count instructions, as described herein, may be used advantageously with variable sized elements and conflict masks to provide SIMD permute controls, thereby permitting efficient SIMD address conflict resolution in registers without performing dependent computations through memory.

[00143] **Figure 14B** illustrates a flow diagram for one embodiment of an example of using SIMD vector address conflict masks for efficient vector address conflict resolution. In processing block 1410 of process 1402 the set of offset indices is initialized, e.g. to the values 7, 2, 7, 1 and 7. It will again be appreciated that the number of offset indices and the values of the offset indices is intended to be illustrative and not restrictive of embodiments of the invention. In particular, the number of offset indices in a register may be determined by the particular application and/or the size of data elements corresponding to the indices. The example values of offset indices are simply intended to illustrate resolution of dependency conflicts. In processing block 1415 the elements left mask is set to all (e.g. five) ones. In processing block 1420 the dependencies masks may be set according to a technique of process 1405 (which is described in greater detail below with regard to Figure 18) to include matching offset indices. Then in processing block 1425 a set of order masks is initialized. In processing block 1430 the conflict masks are computed by performing a bitwise AND between the dependencies masks and the order masks.

[00144] In processing block 1450 data is gathered from memory according to the offset indices and an elements left mask which is (optionally) all ones. Next in the process 1406 a set of conflicted out elements is initialized by broadcasting the elements left mask to all elements in processing block 1480. In processing block 1485 the set of conflict masks is recomputed using a vector packed AND instruction on the previous conflict masks and the new conflicted out elements. Then the new elements left mask is computed in processing block 1490 using a packed compare not equal instruction, PCMPNEQ, comparing each

conflict mask with all zeroes. In processing block 1462 the elements left mask are negated to form a did these mask, and the data elements are scattered in processing block 1466 to memory according to the offset indices. In processing block 1460 a determination is made whether the elements left mask is zero (0). If so in processing of this set of SIMD data ends in processing block 1499. Otherwise processing reiterates beginning in processing block 1450 of process 1402.

[00145] **Figure 15A** illustrates a flow diagram for one embodiment of a process 1501 in a processor to execute instructions to provide SIMD vector packed tuple cross-comparison functionality. The packed tuples may each comprise two, four or eight elements. In some embodiments the comparison type may be indicated (e.g. as one of equal, less-than, less-than or equal, false, not equal, not less-than, not less-than or equal, and true) by an immediate operand specified by the instruction. In processing block 1510 elements of a first data type are stored in each of a plurality of n data fields of a vector register. In processing block 1520 elements of the first data type are stored in each of n data fields of a second vector. In processing block 1530 a SIMD instruction for vector packed tuple cross-comparison is decoded in a processor. Responsive to the SIMD instruction for vector packed tuple cross-comparison, processing proceeds in process 1503 with processing block 1540.

[00146] In processing block 1540 a next tuple in the vector register is selected for processing. In processing block 1550 the next element in the tuple is compared with every element of a corresponding tuple of the plurality of n data fields of the second vector. In processing block 1560 a mask bit corresponding to each tuple element of the second vector is set, in the next bit-mask corresponding to the element concurrently being compared from the vector register tuple selected for processing, according to the corresponding comparison. In some embodiments, all bit-masks corresponding to the elements of the vector register may be stored in a destination register aligned with the least significant bit of their corresponding elements in the vector register. In some alternative embodiments, bit-masks corresponding to the elements of the vector register may instead be stored in a source/destination register realigned according to shift counts stored in corresponding elements of the source/destination register. In processing block 1570 a determination is made as to whether or not all comparisons for the tuple currently selected for processing have finished. If not, processing reiterates beginning with processing block 1550 until each data field of the tuple currently selected for processing have finished. In some embodiments, data fields of the vector register

may be implicitly unmasked, requiring each of them to be cross-compared. In some alternative embodiments, data fields of the vector register may be explicitly masked or unmasked (e.g. in a mask register specified by the instruction), requiring only elements in unmasked data fields to be cross-compared. At such time that all required comparisons have finished for the currently selected tuple, processing proceeds to processing block 1580 where a determination is made as to whether or not processing of all n data fields of the vector register has finished. If not, processing reiterates beginning with processing block 1540 where another tuple in the vector register is selected for processing. Otherwise, processing ends in processing block 1590.

[00147] It will be appreciated that SIMD vector packed tuple cross-comparison instructions, as in the embodiments described herein, may be used for variable sized elements and memory offsets to provide SIMD vector address conflict detection functionality and to generate conflict masks for efficient SIMD address conflict resolution. It will also be appreciated that SIMD vector packed tuple cross-comparison instructions may be used in combination with SIMD vector leading zero count instructions, as in the embodiments described herein, to provide SIMD permute controls, for efficient SIMD address conflict resolution especially in certain gather-modify-scatter applications permitting address conflict resolution in registers without performing dependent computations through memory.

[00148] **Figure 15B** illustrates a flow diagram for one embodiment of a process 1502 to use SIMD vector address conflict masks for efficient vector address conflict resolution. In processing block 1505 an offset in each element of a first register or memory vector source is compared with an offset in every less significant element of the vector source to determine if they hold matching offsets. In processing block 1515 any mask bits corresponding to less significant elements in the first vector source holding offsets matching the offset in the element corresponding to a mask in a first vector destination are set to one (e.g. as shown in process 1801). Any other bits in the destination register are set to a second value (e.g. zero) in processing block 1525 (e.g. as shown in process 1405).

[00149] In processing block 1535 the values of each of the masks stored in the first destination register are compared with a corresponding copy of a specific value (e.g. of zero) in another vector, in order to generate a mask of the elements that depend upon computations of less significant elements. In processing block 1545 the mask of elements that depend upon incomplete computations on less significant elements is used to detect elements ready to

process and generate a completion mask. In processing block 1555 the SIMD computation is performed on the data using the completion mask. Then in processing block 1565 the mask of elements that depend upon incomplete computations on less significant elements is updated. A determination is made in processing block 1575 whether or not processing of all the elements is finished, and if so processing of these elements ends in processing block 1595. Otherwise, processing reiterates beginning in processing block 1545. It will also be appreciated that while process 1502, as illustrated, does not presume that the processed elements need to be stored to memory, any such storing, or scattering to memory, together with additional subsequent iterations of process 1502 may be performed in accordance with alternative embodiments of the invention.

[00150] **Figure 16** illustrates a prior art embodiment of an apparatus 1601 for executing an instruction to provide SIMD vector leading zero count functionality useful for generating permute controls for efficient vector address conflict resolution. Embodiments of apparatus 1601 may be part of a pipeline 400 (e.g. execution stage 416) or part of a core 490 (e.g. execution unit(s) 462) for execution of an instruction to provide SIMD vector leading zero count functionality. Embodiments of apparatus 1601 may be coupled with a decode stage (e.g. decode 406) or a decoder (e.g. decode unit 440) to decode an instruction for a SIMD vector leading zero count, which may permit efficient vector address conflict resolution. One or more execution units (e.g. execution apparatus 1601) responsive to the decoded instruction, read the plurality of bits of each of the data fields in memory vector operand or vector register 1610 or 1620, and for each data field of the memory vector operand or vector register 1610 or 1620, count the number of most significant contiguous bits set to zero, and store the count as a value in a corresponding data field of a SIMD destination register 1650 or 1660.

[00151] For example, embodiments of apparatus 1601 may be coupled with vector registers (e.g. physical register files unit(s) 458) comprising a variable plurality of n variable sized data fields to store values of a variable plurality of n variable sized data elements. Embodiments of the instruction to provide SIMD vector leading zero count functionality specify a vector leading zero count operation and a data field size for performing the SIMD leading zero count for each data field of the memory vector operand or vector register, e.g. 1610 or 1620, and storing the counts as values in corresponding data fields of the specified size in the SIMD destination register, e.g. 1650 or 1660.

[00152] For example, one embodiment of apparatus 1601 for executing an instruction to provide SIMD vector leading zero count functionality reads the plurality of bits of each of the data fields of a first size (e.g. 16-bits or 32-bits) in memory vector operand or vector register 1620, and counts the number of most significant contiguous bits set to zero in leading zero counters 1630-1637 of a leading zero count circuit 1603, then stores the counts as values in corresponding data fields of the same specified size in the SIMD destination register 1660. Another embodiment of apparatus 1601 for executing an instruction to provide a SIMD vector leading zero count reads the plurality of bits of each of the data fields of a second size (e.g. 32-bits or 64-bits) in a memory vector operand or vector register 1610, and selects the counts of the most significant contiguous bits set to zero from odd leading zero counters 1631 to 1637, or adds a value m to the counts from even leading zero counters 1630 to 1636, respectively, in selective adder circuits 1640 to 1643 of a leading zero count combination circuit 1604, then stores the combined counts as values in corresponding data fields of the specified size in the SIMD destination register 1650. It will be appreciated that alternative embodiments of apparatus 1601 for executing an instruction to provide SIMD vector leading zero counts may be capable of performing both SIMD vector leading zero counts of a first data field size and SIMD vector leading zero counts of a second data field size as shown in Figure 16, or may be modified to perform SIMD vector leading zero counts for a variety of specified data field sizes. SIMD vector leading zero count instructions, as in the embodiments described herein, may be used with variable sized elements and conflict masks to provide SIMD permute controls, thereby permitting more efficient SIMD address conflict resolution in registers without performing dependent computations through memory. Thus the SIMD vector leading zero count instructions and logic disclosed herein provide for efficient SIMD address conflict resolution especially in certain gather-modify-scatter applications, for example as discussed with regard to Figure 14A.

[00153] When the elements being compared, e.g. indices, are the same size as the masks required to represent comparison results (e.g. sixteen 16-bit elements in 256-bit registers) then performing an instruction to cross-compare all elements may be appropriate. But when there are more elements to compare than bits available to represent the comparison results, one may need an alternative solution. It will be appreciated that SIMD vector packed tuple comparison instructions, as disclosed herein, may be used as an alternative solution for variable sized elements and memory offsets to provide SIMD vector address conflict

detection functionality and the results combined to generate conflict masks for efficient SIMD address conflict resolution. It will also be appreciated that SIMD vector packed tuple comparison instructions may be of more general-purpose applicability (e.g. when any one of a variety of different comparison operations may be useful), and may require less implementation cost in terms of area and power than an instruction to cross-compare all elements.

[00154] Figure 17A illustrates one embodiment of an example instruction 1701 to provide SIMD vector packed double cross-comparison functionality. In some embodiments an encoding of instruction 1701 may be recorded on some machine-readable medium to record functional descriptive material, for execution by a special purpose machine or by a general purpose processing machine. In some embodiments an encoding of instruction 1701 may specify a first vector register 1710 comprising a first plurality of data fields (e.g. n data fields), the first plurality divided into first portions, each first portion having a second plurality of data fields (e.g. two data fields) each data field either to be masked or to be unmasked. In some embodiments, data fields of the first vector register 1710 may be implicitly unmasked, requiring each of them to be compared. In some alternative embodiments, data fields of the first vector register 1710 may be explicitly masked or unmasked (e.g. in a mask register specified by the instruction 1701), requiring only elements in unmasked data fields to be compared. Instruction 1701 may also specify a second vector register 1720 or memory storage set representing a third plurality of data fields corresponding to the first plurality of data fields, the third plurality divided into second portions, each second portion also having said second plurality of data fields (e.g. two data fields). In some embodiments the packed elements may be signed or unsigned packed bytes (8 bits), words (16 bits), doublewords (32 bits) or quadwords (64 bits). Instruction 1701 may also specify an immediate byte operand 1731 encoding a particular type of comparison (e.g. greater-than or equal, $\geq$) and a destination vector register 1771 comprising a corresponding first plurality of data fields (e.g. n data fields) divided into corresponding first portions, each corresponding first portion having the second plurality of data fields (e.g. two data fields). In some embodiments the comparison type may be indicated as one of: equal, less-than, less-than or equal, always false, not equal, not less-than, not less-than or equal, and always true by immediate operand 1731.

[00155] For each element stored in a data field of the first portion (e.g. 2-tuple) of the first plurality of n data fields in first vector register 1710, instruction 1701 compares that element with every element of the corresponding portion (e.g. 2-tuple) representing said third plurality of n data fields of the second vector register 1720 or memory storage set. In destination vector register 1771, instruction 1701 sets mask bits corresponding to each element of the second vector 1720 portion, in a bit-mask corresponding to each unmasked element of the corresponding first register 1710 portion, according to the corresponding comparisons, and sets any other (e.g. more significant in the bit-mask) bits in the destination register to a zero value. In some embodiments, all bit-masks corresponding to the elements of the first vector register 1710 may be stored in destination vector register 1771 aligned with the least significant bit of their corresponding elements in the first vector register 1710. In some alternative embodiments, bit-masks corresponding to the elements of the first vector register 1710 may instead be stored in a source/destination vector register 1771 realigned according to shift counts stored in corresponding elements of the source/destination vector register 1771.

[00156] **Figure 17B** illustrates an alternative embodiment of an instruction 1702 to provide SIMD vector packed quadruple cross-comparison functionality. In some embodiments an encoding of instruction 1702 may also be recorded on some machine-readable medium to record functional descriptive material, for execution by a special purpose machine or by a general purpose processing machine. In some embodiments an encoding of instruction 1702 may specify a first vector register 1710 comprising a first plurality of data fields (e.g. n data fields), the first plurality divided into first portions, each first portion having a second plurality of data fields (e.g. four data fields) each data field either to be masked or to be unmasked. In some embodiments, data fields of the first vector register 1710 may be implicitly unmasked, requiring each of them to be compared. In some alternative embodiments, data fields of the first vector register 1710 may be explicitly masked or unmasked (e.g. in a mask register specified by the instruction 1702), requiring only elements in unmasked data fields to be compared. Instruction 1702 may also specify a second vector register 1720 or memory storage set representing a third plurality of data fields corresponding to the first plurality of data fields, the third plurality divided into second portions, each second portion also having said second plurality of data fields (e.g. four data fields). In some embodiments the packed elements may be signed or unsigned packed bytes (8 bits), words (16 bits), doublewords (32 bits) or quadwords (64 bits). Instruction 1702 may also specify an

immediate byte operand 1732 encoding a particular type of comparison (e.g. less-than, <) and a destination vector register 1772 comprising a corresponding first plurality of data fields (e.g. n data fields) divided into corresponding first portions, each corresponding first portion having the second plurality of data fields (e.g. four data fields). In some embodiments the comparison type may be indicated as one of: equal, less-than, less-than or equal, always false, not equal, not less-than, not less-than or equal, and always true by immediate operand 1732.

[00157] For each element stored in a data field of the first portion (e.g. 4-tuple) of the first plurality of n data fields in first vector register 1710, instruction 1702 compares that element with every element of the corresponding portion (e.g. 4-tuple) representing said third plurality of n data fields of the second vector register 1720 or memory storage set. In destination vector register 1772, instruction 1702 sets mask bits corresponding to each element of the second vector 1720 portion, in a bit-mask corresponding to each unmasked element of the corresponding first register 1710 portion, according to the corresponding comparisons, and sets any other (e.g. more significant in the bit-mask) bits in the destination register to a zero value. In some embodiments, all bit-masks corresponding to the elements of the first vector register 1710 may be stored in destination vector register 1772 aligned with the least significant bit of their corresponding elements in the first vector register 1710. In some alternative embodiments, bit-masks corresponding to the elements of the first vector register 1710 may instead be stored in a source/destination vector register 1772 realigned according to shift counts stored in corresponding elements of the source/destination vector register 1772.

[00158] **Figure 17C** illustrates another alternative embodiment of an instruction 1703 to provide SIMD vector packed octuple cross-comparison functionality. In some embodiments an encoding of instruction 1703 may also be recorded on some machine-readable medium to record functional descriptive material, for execution by a special purpose machine or by a general purpose processing machine. In some embodiments an encoding of instruction 1703 may specify a first vector register 1710 comprising a first plurality of data fields (e.g. n data fields), the first plurality divided into first portions, each first portion having a second plurality of data fields (e.g. eight data fields) each data field either to be masked or to be unmasked. In some embodiments, data fields of the first vector register 1710 may be implicitly unmasked, requiring each of them to be compared. In some alternative embodiments, data fields of the first vector register 1710 may be explicitly masked or unmasked (e.g. in a mask register specified by the instruction 1703), requiring only elements

in unmasked data fields to be compared. Instruction 1703 may also specify a second vector register 1720 or memory storage set representing a third plurality of data fields corresponding to the first plurality of data fields, the third plurality divided into second portions, each second portion also having said second plurality of data fields (e.g. eight data fields). In some embodiments the packed elements may be signed or unsigned packed bytes (8 bits), words (16 bits), doublewords (32 bits) or quadwords (64 bits). Instruction 1703 may also specify an immediate byte operand 1733 encoding a particular type of comparison (e.g. equal, =) and a destination vector register 1773 comprising a corresponding first plurality of data fields (e.g. n data fields) divided into corresponding first portions, each corresponding first portion having the second plurality of data fields (e.g. eight data fields). In some embodiments the comparison type may be indicated as one of: equal, less-than, less-than or equal, always false, not equal, not less-than, not less-than or equal, and always true by immediate operand 1733.

[00159] For each element stored in a data field of the first portion (e.g. 8-tuple) of the first plurality of n data fields in first vector register 1710, instruction 1703 compares that element with every element of the corresponding portion (e.g. 8-tuple) representing said third plurality of n data fields of the second vector register 1720 or memory storage set. In destination vector register 1773, instruction 1703 sets mask bits corresponding to each element of the second vector 1720 portion, in a bit-mask corresponding to each unmasked element of the corresponding first register 1710 portion, according to the corresponding comparisons, and sets any other (e.g. more significant in the bit-mask) bits in the destination register to a zero value. In some embodiments, all bit-masks corresponding to the elements of the first vector register 1710 may be stored in destination vector register 1773 aligned with the least significant bit of their corresponding elements in the first vector register 1710. In some alternative embodiments, bit-masks corresponding to the elements of the first vector register 1710 may instead be stored in a source/destination vector register 1773 realigned according to shift counts stored in corresponding elements of the source/destination vector register 1773.

[00160] **Figure 18** illustrates an embodiment of an example method 1801 combining results of an instruction to provide SIMD vector packed tuple cross-comparison functionality to generate address conflict masks (e.g. such as 1420) for efficient vector address conflict resolution. For a first vector packed double cross-comparison 1802, an element of a first data type is stored in each of n data fields of vector register 1810. An element of the first data type is also stored in each of n data fields of vector 1820, which may be stored in a vector register

or in a memory storage set representing vector 1820. In a processor, a SIMD instruction for vector packed double cross-comparison 1802 is decoded.

[00161] In response to the SIMD instruction for vector packed double cross-comparison 1802, for each element stored in a data field of each portion (e.g. 2-tuple) of the n data fields in vector register 1810, instruction 1802 compares that element with every element of the corresponding portion (e.g. 2-tuple) of the vector register or memory storage set representing the n data fields of vector 1820. In destination vector register 1830, instruction 1802 sets mask bits corresponding to each element of the vector 1820 portion, in a bit-mask corresponding to each unmasked element of the corresponding register 1810 portion, according to the corresponding comparisons (e.g. for equality) performed, and sets any other (e.g. more significant in the bit-mask) bits in the destination register to a zero value. In some embodiments, all bit-masks corresponding to the elements of the vector register 1810 may be first stored in destination vector register 1830 aligned with the least significant bit of their corresponding elements in the first vector register 1810, and then a packed shift 1812 may be preformed according to a separate SIMD instruction. In some alternative embodiments, bit-masks corresponding to the elements of the first vector register 1810 may instead be stored in a source/destination vector register 1830 realigned according to shift counts stored in corresponding elements of the source/destination vector register 1830.

[00162] For a second vector packed double cross-comparison 1803, an element of the first data type is stored in each of n data fields of vector register 1815. An element of the first data type is also stored in each of n data fields of vector 1825, which may be stored in a vector register or in a memory storage set representing vector 1825. A second SIMD instruction for vector packed double cross-comparison 1803 is decoded. In response to the second SIMD instruction for vector packed double cross-comparison 1803, for each element stored in a data field of each portion (e.g. 2-tuple) of the n data fields in vector register 1815, instruction 1803 compares that element with every element of the corresponding portion (e.g. 2-tuple) of the vector register or memory storage set representing the n data fields of vector 1825. In destination vector register 1835, instruction 1803 sets mask bits corresponding to each element of the vector 1825 portion, in a bit-mask corresponding to each unmasked element of the corresponding register 1815 portion, according to the corresponding comparisons (e.g. for equality) performed, and sets any other (e.g. more significant in the bit-mask) bits in the destination register to a zero value. In some embodiments, all bit-masks corresponding to the

elements of the vector register 1815 may be first stored in destination vector register 1835 aligned with the least significant bit of their corresponding elements in the first vector register 1815, and then a packed shift 1813 may be preformed according to a separate SIMD instruction. In some alternative embodiments, bit-masks corresponding to the elements of the first vector register 1815 may instead be stored in a source/destination vector register 1835 realigned according to shift counts also stored in corresponding elements of the source/destination vector register 1835. A vector permutation may then be performed upon the bit-mask elements of vector register 1835 according to a permutation vector 1840 and a separate SIMD vector permute instruction to generate a temporary results vector register 1845.

[00163] For a third vector packed double cross-comparison 1804, an element of the first data type is stored in each of n data fields of vector 1850, which may be stored in a vector register or in a memory storage set representing vector 1850. A third SIMD instruction for vector packed double cross-comparison 1804 is decoded. In response to the third SIMD instruction for vector packed double cross-comparison 1804, for each element stored in a data field of each portion (e.g. 2-tuple) of the n data fields in vector register 1815, instruction 1804 compares that element with every element of the corresponding portion (e.g. 2-tuple) of the vector register or memory storage set representing the n data fields of vector 1850. In destination vector register 1860, instruction 1804 sets mask bits corresponding to each element of the vector 1850 portion, in a bit-mask corresponding to each unmasked element of the corresponding register 1815 portion, according to the corresponding comparisons (e.g. for equality) performed, and sets any other (e.g. more significant in the bit-mask) bits in the destination register to a zero value. In some embodiments, all bit-masks corresponding to the elements of the vector register 1815 may be first stored in destination vector register 1860 aligned with the least significant bit of their corresponding elements in the vector register 1815, and then a packed shift 1814 may be preformed according to a separate SIMD instruction. In some alternative embodiments, bit-masks corresponding to the elements of the first vector register 1815 may instead be stored in a source/destination vector register 1860 realigned according to shift counts also stored in corresponding elements of the source/destination vector register 1860.

[00164] The bit-masks stored in source/destination vector register 1860 corresponding to elements of the vector register 1815 are combined, respectively with the bit-masks stored in

the temporary results vector register 1845 (e.g. according to a separate SIMD vector packed add instruction) to generate a second temporary results vector register 1870. The bit-masks stored in source/destination vector register 1835 corresponding to elements of the vector register 1815 are combined, respectively with the bit-masks stored in the second temporary results vector register 1870 (e.g. according to another separate SIMD vector packed add instruction) to generate a third temporary results vector register 1880. The bit-masks stored in source/destination vector register 1830 corresponding to elements of the vector register 1810 are combined, respectively with the bit-masks stored in the third temporary results vector register 1880 (e.g. according to another separate SIMD vector packed add instruction) to generate a results vector register 1890 having address conflict masks (e.g. such as 1420) corresponding to elements of the vector register 1810.

[00165] It will be appreciated that SIMD vector packed tuple comparison instructions, as disclosed herein, may be used as an alternative solution for variable sized elements and memory offsets to provide SIMD vector address conflict detection functionality and the results combined to generate conflict masks for efficient SIMD address conflict resolution. It will also be appreciated that SIMD vector packed tuple comparison instructions may be of more general-purpose applicability (e.g. when any one of a variety of different comparison operations may be useful), and require less implementation cost in terms of area and power than an instruction to cross-compare all elements.

[00166] **Figure 19A** illustrates an embodiment of an apparatus 1901 for executing an instruction to provide SIMD vector packed tuple cross-comparison functionality.

Embodiments of apparatus 1901 may be part of a pipeline 400 (e.g. execution stage 416) or part of a core 490 (e.g. execution unit(s) 462) for execution of an instruction to provide SIMD vector packed tuple cross-comparison functionality. Some embodiments of apparatus 1901 comprise packed tuple comparison apparatus 1903, which includes control logic 1940 and comparator array 1942, and optionally shifting apparatus 1905, which includes shifters 1950, 1951, 1952 and 1953, etc. Embodiments of apparatus 1901 may be coupled with a decode stage (e.g. decode 406) or a decoder (e.g. decode unit 440) to decode an instruction for a SIMD vector packed tuple cross-comparison, which may permit efficient vector address conflict resolution. One or more execution units (e.g. execution apparatus 1901) responsive to the decoded instruction, for each element stored in a data field of each portion (e.g. 2-tuple) of the n data fields in vector register 1910, compares that element with every element of the

corresponding portion (e.g. 2-tuple) of the vector register or memory storage set representing the n data fields of vector 1920. Embodiments of apparatus 1901 may be coupled with vector registers (e.g. physical register files unit(s) 458) comprising a variable plurality of n variable sized data fields to store values of a variable plurality of n variable sized data elements. In destination vector register 1970, apparatus 1901 sets mask bits corresponding to each element of the vector 1920 portion, in a bit-mask corresponding to each unmasked element of the corresponding register 1910 portion, according to the corresponding comparisons (e.g. specified by immediate operand 1930) performed, and sets any other (e.g. more significant in the bit-mask) bits in the destination register to a zero value. In some embodiments, all bit-masks corresponding to the elements of the vector register 1910 may be stored in destination vector register 1970 aligned with the least significant bit of their corresponding elements in the vector register 1910. In some alternative embodiments, bit-masks corresponding to the elements of the first vector register 1910 may optionally be stored in a source/destination vector register 1970 realigned (e.g. by shifting apparatus 1905) according to shift counts stored in corresponding elements of the source/destination vector register 1970.

[00167] **Figure 19B** illustrates an alternative embodiment of an apparatus 1902 for executing an instruction to provide SIMD vector packed tuple cross-comparison functionality. Embodiments of apparatus 1902 may be part of a pipeline 400 (e.g. execution stage 416) or part of a core 490 (e.g. execution unit(s) 462) for execution of an instruction to provide SIMD vector packed tuple cross-comparison functionality. Some embodiments of apparatus 1902 comprise packed tuple comparison apparatus 1904, which includes control logic 1940 and comparator array 1944, and optionally shifting apparatus 1906, which includes shifters 1960, 1961, 1962, 1963, etc. Embodiments of apparatus 1902 may be coupled with a decode stage (e.g. decode 406) or a decoder (e.g. decode unit 440) to decode an instruction for a SIMD vector packed tuple cross-comparison, which may permit efficient vector address conflict resolution. One or more execution units (e.g. execution apparatus 1902) responsive to the decoded instruction, for each element stored in a data field of each portion (e.g. 4-tuple) of the n data fields in vector register 1910, compares that element with every element of the corresponding portion (e.g. 4-tuple) of the vector register or memory storage set representing the n data fields of vector 1920. Embodiments of apparatus 1902 may be coupled with vector registers (e.g. physical register files unit(s) 458) comprising a variable plurality of n variable sized data fields to store values of a variable plurality of n variable sized data elements. In

destination vector register 1970, apparatus 1902 sets mask bits corresponding to each element of the vector 1920 portion, in a bit-mask corresponding to each unmasked element of the corresponding register 1910 portion, according to the corresponding comparisons (e.g. specified by immediate operand 1930) performed, and sets any other (e.g. more significant in the bit-mask) bits in the destination register to a zero value. In some embodiments, all bit-masks corresponding to the elements of the vector register 1910 may be stored in destination vector register 1970 aligned with the least significant bit of their corresponding elements in the vector register 1910. In some alternative embodiments, bit-masks corresponding to the elements of the first vector register 1910 may optionally be stored in a source/destination vector register 1970 realigned (e.g. by shifting apparatus 1906) according to shift counts stored in corresponding elements of the source/destination vector register 1970.

[00168] Use of SIMD vector packed tuple comparison instructions permits a variable plurality of n variable sized data elements including signed or unsigned packed bytes (8 bits), words (16 bits), doublewords (32 bits) or quadwords (64 bits). It will be appreciated that SIMD vector packed tuple comparison instructions may be of more general-purpose applicability than a vector conflict instruction (e.g. when any one of a variety of different comparison operations may be useful), and require less implementation cost in terms of area and power than an instruction to cross-compare all elements.

[00169] Embodiments of the mechanisms disclosed herein may be implemented in hardware, software, firmware, or a combination of such implementation approaches. Embodiments of the invention may be implemented as computer programs or program code executing on programmable systems comprising at least one processor, a storage system (including volatile and non-volatile memory and/or storage elements), at least one input device, and at least one output device.

[00170] Program code 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.

[00171] 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.

[00172] One or more aspects of at least one embodiment may be implemented by representative instructions stored on a machine-readable medium which represents various logic within the processor, which when read by a machine causes the machine to fabricate logic to perform the techniques described herein. Such representations, known as “IP cores” may be stored on a tangible, machine readable medium and supplied to various customers or manufacturing facilities to load into the fabrication machines that actually make the logic or processor.

[00173] Such machine-readable storage media may include, without limitation, non-transitory, tangible arrangements of articles manufactured or formed by a machine or device, including storage media such as hard disks, any other type of disk including floppy disks, optical disks, compact disk read-only memories (CD-ROMs), compact disk rewritable's (CD-RWs), and magneto-optical disks, semiconductor devices such as read-only memories (ROMs), random access memories (RAMs) such as dynamic random access memories (DRAMs), static random access memories (SRAMs), erasable programmable read-only memories (EPROMs), flash memories, electrically erasable programmable read-only memories (EEPROMs), magnetic or optical cards, or any other type of media suitable for storing electronic instructions.

[00174] Accordingly, embodiments of the invention also include non-transitory, tangible machine-readable media containing instructions or containing design data, such as Hardware Description Language (HDL), which defines structures, circuits, apparatuses, processors and/or system features described herein. Such embodiments may also be referred to as program products.

[00175] 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.

[00176] Thus, techniques for performing one or more instructions according to at least one embodiment are disclosed. While certain exemplary embodiments have been described and shown in the accompanying drawings, it is to be understood that such embodiments are merely illustrative of and not restrictive on the broad invention, and that this invention not be limited to the specific constructions and arrangements shown and described, since various other modifications may occur to those ordinarily skilled in the art upon studying this disclosure. In an area of technology such as this, where growth is fast and further advancements are not easily foreseen, the disclosed embodiments may be readily modifiable in arrangement and detail as facilitated by enabling technological advancements without departing from the principles of the present disclosure or the scope of the accompanying claims.

THE CLAIMS

What is claimed is:

1. A processor comprising:

a first vector register to store, in each of a plurality of n data fields, an element of a first data type;

a second vector register or memory storage set to store, in each of a plurality of n data fields, a corresponding element of the first data type;

a decode stage to decode a first instruction specifying a vector packed tuple cross-comparison operation and a tuple size; and

one or more execution units, responsive to the decoded first instruction, to:

for each data field of a tuple-sized portion of said plurality of n data fields in the first vector register, compare its corresponding element with every element of a corresponding tuple-sized portion of the plurality of n data fields of said second vector register or memory storage set, and

set a mask bit corresponding to each element of said second vector register or memory storage set tuple-sized portion, in a bit-mask corresponding to each unmasked element of the corresponding first vector register tuple-sized portion, according to the corresponding comparison.

2. The processor of claim 1, wherein a type of comparison for the cross-comparison operation is indicated by an immediate operand specified by said first instruction.
3. The processor of claim 2, wherein a type of comparison for the cross-comparison operation is selected from the group consisting of: equal, less-than, less-than or equal, false, not equal, not less-than, not less-than or equal, and true.
4. The processor of claim 1, wherein the first instruction is a vector packed double compare instruction to cross-compare pairs of data fields of the first vector register with pairs of data fields of the second vector register or memory storage set.

5. The processor of claim 1, wherein the first instruction is a vector packed quadruple compare instruction to cross-compare four-tuples of data fields of the first vector register with four-tuples of data fields of the second vector register or memory storage set.
6. The processor of claim 1, wherein the first instruction is a vector packed octuple compare instruction to cross-compare eight-tuples of data fields of the first vector register with eight-tuples of data fields of the second vector register or memory storage set.
7. The processor of claim 1, wherein the plurality of n data fields is 64 data fields.
8. The processor of claim 1, wherein the plurality of n data fields is 32 data fields.
9. The processor of claim 1, wherein the plurality of n data fields is 16 data fields.
10. The processor of claim 1, wherein the plurality of n data fields is 8 data fields.
11. The processor of claim 1, wherein each of the plurality of n data fields of data fields is 8 bits.
12. The processor of claim 1, wherein each of the plurality of n data fields of data fields is 16 bits.
13. The processor of claim 1, wherein a number of mask bits set corresponding to each element of said second vector register or memory storage set tuple-sized portion comprises 2 bits.
14. The processor of claim 1, wherein a number of mask bits set corresponding to each element of said second vector register or memory storage set tuple-sized portion comprises 4 bits.

15. The processor of claim 1, wherein a number of mask bits set corresponding to each element of said second vector register or memory storage set tuple-sized portion comprises 8 bits.
16. The processor of claim 1, wherein the first instruction specifies a mask register operand having a bit to indicate whether each data field, respectively, of the plurality of n data fields in the first vector register is masked or unmasked.
17. A method comprising:
 - storing in each of a plurality of n data fields of a first vector register, an element of a first data type;
 - storing in each of a plurality of n data fields of a second vector register, a corresponding element of the first data type;
 - executing, in a processor, a SIMD instruction for vector packed tuple cross-comparison; and
 - for each data field of a portion of said plurality of n data fields in the first vector register, comparing its corresponding element with every element of a corresponding portion of the plurality of n data fields of the second vector register, and
 - setting a mask bit corresponding to each element of the second register portion, in a bit-mask corresponding to each unmasked element of the corresponding first register portion, according to the corresponding comparison.
18. The method of claim 17, wherein each element of the first register is implicitly unmasked.
19. The method of claim 17, wherein n bits of a mask register are set to one or zero to explicitly indicate that a corresponding element of the first register is unmasked or masked respectively.
20. The method of claim 17, wherein the portion of said plurality of n data fields comprises two data fields.

21. The method of claim 17, wherein the portion of said plurality of n data fields comprises four data fields.
22. The method of claim 17, wherein the portion of said plurality of n data fields comprises eight data fields.
23. A processing system comprising:
a memory; and
a plurality of processors each processor comprising:
a first vector register to store, in each of a plurality of n data fields, an element of a first data type;
a second vector register or memory storage set to store, in each of a plurality of n data fields, a corresponding element of the first data type;
a decode stage to decode a first instruction specifying a vector packed tuple cross-comparison operation and a tuple size; and
one or more execution units, responsive to the decoded first instruction, to:
for each data field of a tuple-sized portion of said plurality of n data fields in the first vector register, compare its corresponding element with every element of a corresponding tuple-sized portion of the plurality of n data fields of said second vector register or memory storage set, and
set a mask bit corresponding to each element of said second vector register or memory storage set tuple-sized portion, in a bit-mask corresponding to each unmasked element of the corresponding first vector register tuple-sized portion, according to the corresponding comparison.
24. The processing system of claim 23, wherein a type of comparison for the cross-comparison operation is indicated by an immediate operand specified by said first instruction.
25. The processing system of claim 23, wherein a type of comparison for the cross-comparison operation is selected from the group consisting of: equal, less-than, less-than or equal, false, not equal, not less-than, not less-than or equal, and true.

26. The processing system of claim 23, wherein the first instruction is a vector packed double compare instruction to cross-compare pairs of data fields of the first vector register with pairs of data fields of the second vector register or memory storage set.
27. The processing system of claim 23, wherein the first instruction is a vector packed quadruple compare instruction to cross-compare four-tuples of data fields of the first vector register with four-tuples of data fields of the second vector register or memory storage set.
28. The processing system of claim 23, wherein the first instruction is a vector packed octuple compare instruction to cross-compare eight-tuples of data fields of the first vector register with eight-tuples of data fields of the second vector register or memory storage set.