United States Patent Application Publication	20250258674
Kind Code	A1
Publication Date	August 14, 2025
Inventor(s)	ZBICIAK; Joseph

STREAM DATA UNIT WITH MULTIPLE HEAD REGISTERS

Abstract

A streaming engine employed in a digital signal processor specifies a fixed read only data stream. Once fetched the data stream is stored in two head registers for presentation to functional units in the fixed order. Data use by the functional unit is preferably controlled using the input operand fields of the corresponding instruction. A first read only operand coding supplies data from the first head register. A first read/advance operand coding supplies data from the first head register and also advances the stream to the next sequential data elements. Corresponding second read only operand coding and second read/advance operand coding operate similarly with the second head register. A third read only operand coding supplies double width data from both head registers.

Inventors:	**ZBICIAK; Joseph (San Jose, CA)**
Applicant:	**TEXAS INSTRUMENTS INCORPORATED** (Dallas, TX)
Family ID:	**60910379**
Appl. No.:	**19/191006**
Filed:	**April 28, 2025**

Related U.S. Application Data

parent US continuation 18607703 20240318 parent-grant-document US 12288069 child US 19191006
parent US continuation 17557712 20211221 parent-grant-document US 11934833 child US 18607703
parent US continuation 16458756 20190701 parent-grant-document US 11210097 child US 17557712
parent US continuation 15205246 20160708 ABANDONED child US 16458756

Publication Classification

Int. Cl.: **G06F9/30** (20180101); **G06F9/345** (20180101); **G06F9/38** (20180101); **G06F12/0875** (20160101)

U.S. Cl.:

CPC **G06F9/30149** (20130101); **G06F9/30036** (20130101); **G06F9/30038** (20230801); **G06F9/30047** (20130101); **G06F9/30065** (20130101); **G06F9/3016** (20130101); **G06F9/345** (20130101); **G06F9/3824** (20130101); **G06F9/383** (20130101); **G06F12/0875** (20130101); G06F2212/452 (20130101)

Background/Summary

CROSS-REFERENCE TO RELATED APPLICATIONS [0001] This application is a continuation of U.S. patent application Ser. No. 18/607,703, filed Mar. 18, 2024, which is a continuation of U.S. patent application Ser. No. 17/557,712, filed Dec. 21, 2021, now U.S. Pat. No. 11,934,833, which is a continuation of U.S. patent application Ser. No. 16/458,756, filed Jul. 1, 2019, now U.S. Pat. No. 11,210,097, which is a continuation of U.S. patent application Ser. No. 15/205,246, filed Jul. 8, 2016, now abandoned, each of which is incorporated herein by reference in its entirety. [0002] This application is an improvement over U.S. patent application Ser. No. 14/331,986 filed on Jul. 15, 2014, which is now U.S. Pat. No. 9,606,803, which claims priority to U.S. Provisional Patent Application No. 61/846,148 filed on Jul. 15, 2013, each of which is incorporated herein by reference.

TECHNICAL FIELD

[0003] The technical field of this invention is digital data processing and more specifically control of streaming engine used for operand fetching.

BACKGROUND

[0004] Modern digital signal processors (DSP) face multiple challenges. Workloads continue to increase, requiring increasing bandwidth. Systems on a chip (SOC) continue to grow in size and complexity. Memory system latency severely impacts certain classes of algorithms. As transistors get smaller, memories and registers become less reliable. As software stacks get larger, the number of potential interactions and errors becomes larger.

[0005] Memory bandwidth and scheduling are a problem for digital signal processors operating on real-time data. Digital signal processors operating on real-time data typically receive an input data stream, perform a filter function on the data stream (such as encoding or decoding) and output a transformed data stream. The system is called real-time because the application fails if the transformed data stream is not available for output when scheduled. Typical video encoding requires a predictable but non-sequential input data pattern. Often the corresponding memory accesses are difficult to achieve with available address generation and memory access resources. A typical application requires memory access to load data registers in a data register file and then supply to functional units which perform the data processing.

SUMMARY

[0006] This invention is a streaming engine employed in a digital signal processor. A fixed data stream sequence is specified by storing corresponding parameters in a control register. Once started the data stream is read only and cannot be written. This generally corresponds to the needs of a real-time filtering operation.

[0007] Once fetched the data stream is stored in a first-in-first-out buffer before supply to the functional unit. The data can only be presented to the functional unit in the fixed order. An exemplary embodiment supplies data elements of the specified data stream to sequentially pack a pair of head registers each having the data width of the functional unit.

[0008] The pair of head registers permits varied access to the data stream. A first stream read instruction reads from the first head register, which stores the latest data elements of the data stream. A second stream read instruction reads from the second head register, which stores the next following data elements after the first head register. This permits slight re-arrangement of access within the data stream sequence upon use of the stream data.

[0009] An exemplary embodiment uses the pair of head registers to supply data to double data width instructions. At least one functional unit is capable of operating on data twice the normal data width. This may be implemented using associated functional units, each operating on the normal data width. A third stream read instruction reads from both the first and second head registers supplying double width data.

[0010] In the preferred embodiment each of the first, second and third stream read instructions has an associated stream read/advance instruction. A stream read/advance instruction supplies data like its associated stream read instruction. Each stream read/advance instruction also advances the data stream by storing the next sequential data elements of said stream in the first and second stream head registers.

Description

BRIEF DESCRIPTION OF THE DRAWINGS

[0011] These and other aspects of this invention are illustrated in the drawings, in which:

[0012] FIG. **1** illustrates a dual scalar/vector datapath processor according to one embodiment of this invention;

[0013] FIG. **2** illustrates the registers and functional units in the dual scalar/vector datapath processor illustrated in FIG. **1**;

[0014] FIG. **3** illustrates a global scalar register file;

[0015] FIG. **4** illustrates a local scalar register file shared by arithmetic functional units;

[0016] FIG. **5** illustrates a local scalar register file shared by multiply functional units;

[0017] FIG. **6** illustrates a local scalar register file shared by load/store units;

[0018] FIG. **7** illustrates a global vector register file;

[0019] FIG. **8** illustrates a predicate register file;

[0020] FIG. **9** illustrates a local vector register file shared by arithmetic functional units;

[0021] FIG. **10** illustrates a local vector register file shared by multiply and correlation functional units;

[0022] FIG. **11** illustrates pipeline phases of a central processing unit according to a preferred embodiment of this invention;

[0023] FIG. **12** illustrates sixteen instructions of a single fetch packet;

[0024] FIG. **13** illustrates an example of the instruction coding of instructions used by this invention;

[0025] FIG. **14** illustrates the bit coding of a condition code extension slot 0;

[0026] FIG. **15** illustrates the bit coding of a condition code extension slot 1;

[0027] FIG. **16** illustrates the bit coding of a constant extension slot 0;

[0028] FIG. **17** is a partial block diagram illustrating constant extension;

[0029] FIG. **18** illustrates the carry control for SIMD operations according to this invention;

[0030] FIG. **19** illustrates a conceptual view of the streaming engines of this invention;

[0031] FIG. **20** illustrates a first example of lane allocation in a vector;

[0032] FIG. **21** illustrates a second example of lane allocation in a vector;

[0033] FIG. **22** illustrates a basic two dimensional stream;

[0034] FIG. **23** illustrates the order of elements within the example stream of FIG. **21**;

[0035] FIG. **24** illustrates extracting a smaller rectangle from a larger rectangle;

[0036] FIG. **25** illustrates how the streaming engine would fetch the stream of this example with a transposition granularity of 4 bytes;

[0037] FIG. **26** illustrates how the streaming engine would fetch the stream of this example with a transposition granularity of 8 bytes;

[0038] FIG. **27** illustrates the details of streaming engine of this invention;

[0039] FIG. **28** illustrates a stream template register of this invention;

[0040] FIG. **29** illustrates sub-field definitions of the flags field of the stream template register of this invention; and

[0041] FIG. **30** illustrates a partial schematic diagram showing the streaming engine supply of data of this invention;

[0042] FIG. **31** illustrates the details of streaming engine of an alternative embodiment this invention; and

[0043] FIG. **32** illustrates a partial schematic diagram showing the streaming engine supply of data in example of FIG. **31**.

DETAILED DESCRIPTION

[0044] FIG. **1** illustrates a dual scalar/vector datapath processor according to a preferred embodiment of this invention. Processor **100** includes separate level one instruction cache (L1I) **121** and level one data cache (L1D) **123**. Processor **100** includes a level two combined instruction/data cache (L2) **130** that holds both instructions and data. FIG. **1** illustrates connection between level one instruction cache **121** and level two combined instruction/data cache **130** (bus **142**). FIG. **1** illustrates connection between level one data cache **123** and level two combined instruction/data cache **130** (bus **145**). In the preferred embodiment of processor **100** level two combined instruction/data cache **130** stores both instructions to back up level one instruction cache **121** and data to back up level one data cache **123**. In the preferred embodiment level two combined instruction/data cache **130** is further connected to higher level cache and/or main memory in a manner known in the art and not illustrated in FIG. **1**. In the preferred embodiment central processing unit core **110**, level one instruction cache **121**, level one data cache **123** and level two combined instruction/data cache **130** are formed on a single integrated circuit. This signal integrated circuit optionally includes other circuits.

[0045] Central processing unit core **110** fetches instructions from level one instruction cache **121** as controlled by instruction fetch unit **111**. Instruction fetch unit **111** determines the next instructions to be executed and recalls a fetch packet sized set of such instructions. The nature and size of fetch packets are further detailed below. As known in the art, instructions are directly fetched from level one instruction cache **121** upon a cache hit (if these instructions are stored in level one instruction cache **121**). Upon a cache miss (the specified instruction fetch packet is not stored in level one instruction cache **121**), these instructions are sought in level two combined cache **130**. In the preferred embodiment the size of a cache line in level one instruction cache 121 equals the size of a fetch packet. The memory locations of these instructions are either a hit in level two combined cache **130** or a miss. A hit is serviced from level two combined cache **130**. A miss is serviced from another level of cache (not illustrated) or from main memory (not illustrated). As is known in the art, the requested instruction may be simultaneously supplied to both level one instruction cache **121** and central processing unit core **110** to speed use.

[0046] In the preferred embodiment of this invention, central processing unit core **110** includes plural functional units to perform instruction specified data processing tasks. Instruction dispatch unit **112** determines the target functional unit of each fetched instruction. In the preferred embodiment central processing unit **110** operates as a very long instruction word (VLIW) processor capable of operating on plural instructions in corresponding functional units simultaneously. Preferably a complier organizes instructions in execute packets that are executed together. Instruction dispatch unit **112** directs each instruction to its target functional unit. The functional unit assigned to an instruction is completely specified by the instruction produced by a compiler. The hardware of central processing unit core **110** has no part in this functional unit assignment. In the preferred embodiment instruction dispatch unit **112** may operate on plural instructions in parallel. The number of such parallel instructions is set by the size of the execute packet. This will be further detailed below.

[0047] One part of the dispatch task of instruction dispatch unit **112** is determining whether the instruction is to execute on a functional unit in scalar datapath side A **115** or vector datapath side B **116**. An instruction bit within each instruction called the s bit determines which datapath the instruction controls. This will be further detailed below.

[0048] Instruction decode unit **113** decodes each instruction in a current execute packet. Decoding includes identification of the functional unit performing the instruction, identification of registers used to supply data for the corresponding data processing operation from among possible register files and identification of the register destination of the results of the corresponding data processing operation. As further explained below, instructions may include a constant field in place of one register number operand field. The result of this decoding is signals for control of the target functional unit to perform the data processing operation specified by the corresponding instruction on the specified data.

[0049] Central processing unit core **110** includes control registers **114**. Control registers **114** store information for control of the functional units in scalar datapath side A **115** and vector datapath side B **116** in a manner not relevant to this invention. This information could be mode information or the like.

[0050] The decoded instructions from instruction decode **113** and information stored in control registers **114** are supplied to scalar datapath side A **115** and vector datapath side B **116**. As a result, functional units within scalar datapath side A **115** and vector datapath side B **116** perform instruction specified data processing operations upon instruction specified data and store the results in an instruction specified data register or registers. Each of scalar datapath side A **115** and vector datapath side B **116** includes plural functional units that preferably operate in parallel. These will be further detailed below in conjunction with FIG. **2**. There is a datapath **117** between scalar datapath side A **115** and vector datapath side B **116** permitting data exchange.

[0051] Central processing unit core **110** includes further non-instruction based modules. Emulation unit **118** permits determination of the machine state of central processing unit core **110** in response to instructions. This capability will typically be employed for algorithmic development. Interrupts/exceptions unit **119** enable central processing unit core **110** to be responsive to external, asynchronous events (interrupts) and to respond to attempts to perform improper operations (exceptions).

[0052] Central processing unit core **110** includes streaming engine **125**. Streaming engine **125** supplies two data streams from predetermined addresses typically cached in level two combined cache **130** to register files of vector datapath side B. This provides controlled data movement from memory (as cached in level two combined cache **130**) directly to functional unit operand inputs. This is further detailed below.

[0053] FIG. **1** illustrates exemplary data widths of busses between various parts. Level one instruction cache **121** supplies instructions to instruction fetch unit **111** via bus **141**. Bus **141** is preferably a 512-bit bus. Bus **141** is unidirectional from level one instruction cache **121** to central processing unit **110**. Level two combined cache **130** supplies instructions to level one instruction cache **121** via bus **142**. Bus **142** is preferably a 512-bit bus. Bus **142** is unidirectional from level two combined cache **130** to level one instruction cache **121**.

[0054] Level one data cache **123** exchanges data with register files in scalar datapath side A **115** via bus **143**. Bus **143** is preferably a 64-bit bus. Level one data cache **123** exchanges data with register files in vector datapath side B **116** via bus **144**. Bus **144** is preferably a 512-bit bus. Busses **143** and **144** are illustrated as bidirectional supporting both central processing unit **110** data reads and data writes. Level one data cache **123** exchanges data with level two combined cache **130** via bus **145**. Bus **145** is preferably a 512-bit bus. Bus **145** is illustrated as bidirectional supporting cache service for both central processing unit **110** data reads and data writes.

[0055] As known in the art, CPU data requests are directly fetched from level one data cache **123** upon a cache hit (if the requested data is stored in level one data cache **123**). Upon a cache miss (the specified data is not stored in level one data cache **123**), this data is sought in level two combined cache **130**. The memory locations of this requested data is either a hit in level two combined cache **130** or a miss. A hit is serviced from level two combined cache **130**. A miss is serviced from another level of cache (not illustrated) or from main memory (not illustrated). As is known in the art, the requested instruction may be simultaneously supplied to both level one data cache **123** and central processing unit core **110** to speed use.

[0056] Level two combined cache **130** supplies data of a first data stream to streaming engine **125** via bus **146**. Bus **146** is preferably a 512-bit bus. Streaming engine **125** supplies data of this first data stream to functional units of vector datapath side B **116** via bus **147**. Bus **147** is preferably a 512-bit bus. Level two combined cache **130** supplies data of a second data stream to streaming engine **125** via bus **148**. Bus **148** is preferably a 512-bit bus. Streaming engine **125** supplies data of this second data stream to functional units of vector datapath side B **116** via bus **149**. Bus **149** is preferably a 512-bit bus. Busses **146**, **147**, **148** and **149** are illustrated as unidirectional from level two combined cache **130** to streaming engine **125** and to vector datapath side B **116** in accordance with the preferred embodiment of this invention.

[0057] Streaming engine data requests are directly fetched from level two combined cache **130** upon a cache hit (if the requested data is stored in level two combined cache **130**). Upon a cache miss (the specified data is not stored in level two combined cache **130**), this data is sought from another level of cache (not illustrated) or from main memory (not illustrated). It is technically feasible in some embodiments for level one data cache **123** to cache data not stored in level two combined cache **130**. If such operation is supported, then upon a streaming engine data request that is a miss in level two combined cache **130**, level two combined cache **130** should snoop level one data cache **123** for the stream engine requested data. If level one data cache **123** stores this data its snoop response would include the data, which is then supplied to service the streaming engine request. If level one data cache **123** does not store this data its snoop response would indicate this and level two combined cache **130** must service this streaming engine request from another level of cache (not illustrated) or from main memory (not illustrated).

[0058] In one embodiment of this invention, both level one data cache **123** and level two combined cache **130** may be configured as selected amounts of cache or directly addressable memory in accordance with U.S. Pat. No. 6,606,686 entitled UNIFIED MEMORY SYSTEM ARCHITECTURE INCLUDING CACHE AND DIRECTLY ADDRESSABLE STATIC RANDOM ACCESS MEMORY.

[0059] FIG. **2** illustrates further details of functional units and register files within scalar datapath side A **115** and vector datapath side B **116**. Scalar datapath side A **115** includes global scalar register file **211**, L1/S1 local register file **212**, M1/N1 local register file **213** and D1/D2 local register file **214**. Scalar datapath side A **115** includes L1 unit **221**, S1 unit **222**, M1 unit **223**, N1 unit **224**, D1 unit **225** and D2 unit **226**. Vector datapath side B **116** includes global vector register file **231**, L2/S2 local register file **232**, M2/N2/C local register file **233** and predicate register file **234**. Vector datapath side B **116** includes L2 unit **241**, S2 unit **242**, M2 unit **243**, N2 unit **244**, C unit **245**, and P unit **246**. There are limitations upon which functional units may read from or write to which register files. These will be detailed below.

[0060] Scalar datapath side A **115** includes L1 unit **221**. L1 unit **221** generally accepts two 64-bit operands and produces one 64-bit result. The two operands are each recalled from an instruction specified register in either global scalar register file **211** or L1/S1 local register file **212.** L1 unit **221** preferably performs the following instruction selected operations: 64-bit add/subtract operations; 32-bit min/max operations; 8-bit Single Instruction Multiple Data (SIMD) instructions such as sum of absolute value, minimum and maximum determinations; circular min/max operations; and various move operations between register files. The result may be written into an instruction specified register of global scalar register file **211**, L1/S1 local register file **212**, M1/N1 local register file **213**, or D1/D2 local register file **214**.

[0061] Scalar datapath side A **115** includes S1 unit **222**. S1 unit **222** generally accepts two 64-bit operands and produces one 64-bit result. The two operands are each recalled from an instruction specified register in either global scalar register file **211** or L1/S1 local register file **212**. S1 unit **222** preferably performs the same type operations as L1 unit **221**. There optionally may be slight variations between the data processing operations supported by L1 unit **221** and S1 unit **222**. The result may be written into an instruction specified register of global scalar register file **211**, L1/S1 local register file **212**, M1/N1 local register file **213**, or D1/D2 local register file **214**.

[0062] Scalar datapath side A **115** includes M1 unit **223**. M1 unit **223** generally accepts two 64-bit operands and produces one 64-bit result. The two operands are each recalled from an instruction specified register in either global scalar register file **211** or M1/N1 local register file **213**. M1 unit **223** preferably performs the following instruction selected operations: 8-bit multiply operations; complex dot product operations; 32-bit bit count operations; complex conjugate multiply operations; and bit-wise Logical Operations, moves, adds and subtracts. The result may be written into an instruction specified register of global scalar register file **211**, L1/S1 local register file **212**, M1/N1 local register file **213**, or D1/D2 local register file **214**.

[0063] Scalar datapath side A **115** includes N1 unit **224**. N1 unit **224** generally accepts two 64-bit operands and produces one 64-bit result. The two operands are each recalled from an instruction specified register in either global scalar register file **211** or M1/N1 local register file **213**. N1 unit **224** preferably performs the same type operations as M1 unit **223**. There may be certain double operations (called dual issued instructions) that employ both the M1 unit **223** and the N1 unit **224** together. The result may be written into an instruction specified register of global scalar register file **211**, L1/S1 local register file **212**, M1/N1 local register file **213**, or D1/D2 local register file **214**.

[0064] Scalar datapath side A **115** includes D1 unit **225** and D2 unit **226**. D1 unit **225** and D2 unit **226** generally each accept two 64-bit operands and each produce one 64-bit result. D1 unit **225** and D2 unit **226** generally perform address calculations and corresponding load and store operations. D1 unit **225** is used for scalar loads and stores of 64 bits. D2 unit **226** is used for vector loads and stores of 512 bits. D1 unit **225** and D2 unit **226** preferably also perform: swapping, pack and unpack on the load and store data; 64-bit SIMD arithmetic operations; and 64-bit bit-wise logical operations. D1/D2 local register file **214** will generally store base and offset addresses used in address calculations for the corresponding loads and stores. The two operands are each recalled from an instruction specified register in either global scalar register file **211** or D1/D2 local register file **214**. The calculated result may be written into an instruction specified register of global scalar register file **211**, L1/S1 local register file **212**, M1/N1 local register file **213**, or D1/D2 local register file **214**.

[0065] Vector datapath side B **116** includes L2 unit **241**. L2 unit **241** generally accepts two 512-bit operands and produces one 512-bit result. The two operands are each recalled from an instruction specified register in either global vector register file **231**, L2/S2 local register file **232** or predicate register file **234**. L2 unit **241** preferably performs instruction similar to L1 unit **221** except on wider 512-bit data. The result may be written into an instruction specified register of global vector register file **231**, L2/S2 local register file **232**, M2/N2/C local register file **233**, or predicate register file **234**.

[0066] Vector datapath side B **116** includes S2 unit **242**. S2 unit **242** generally accepts two 512-bit operands and produces one 512-bit result. The two operands are each recalled from an instruction specified register in either global vector register file **231**, L2/S2 local register file **232** or predicate register file **234**. S2 unit **242** preferably performs instructions similar to S1 unit **222** except on wider 512-bit data. The result may be written into an instruction specified register of global vector register file **231**, L2/S2 local register file **232**, M2/N2/C local register file **233**, or predicate register file **234**.

[0067] Vector datapath side B **116** includes M2 unit **243**. M2 unit **243** generally accepts two 512-bit operands and produces one 512-bit result. The two operands are each recalled from an instruction specified register in either global vector register file **231** or M2/N2/C local register file **233**. M2 unit **243** preferably performs instructions similar to M1 unit **223** except on wider 512-bit data. The result may be written into an instruction specified register of global vector register file **231**, L2/S2 local register file **232**, or M2/N2/C local register file **233**.

[0068] Vector datapath side B **116** includes N2 unit **244**. N2 unit **244** generally accepts two 512-bit operands and produces one 512-bit result. The two operands are each recalled from an instruction specified register in either global vector register file **231** or M2/N2/C local register file **233**. N2 unit **244** preferably performs the same type operations as M2 unit **243**. There may be certain double operations (called dual issued instructions) that employ both M2 unit **243** and the N2 unit **244** together. The result may be written into an instruction specified register of global vector register file **231**, L2/S2 local register file **232**, or M2/N2/C local register file **233**.

[0069] Vector datapath side B **116** includes C unit **245**. C unit **245** generally accepts two 512-bit operands and produces one 512-bit result. The two operands are each recalled from an instruction specified register in either global vector register file **231** or M2/N2/C local register file **233**. C unit **245** preferably performs: “Rake” and “Search” instructions; up to 512 2-bit PN*8-bit multiplies I/Q complex multiplies per clock cycle; 8-bit and 16-bit Sum-of-Absolute-Difference (SAD) calculations, up to 512 SADs per clock cycle; horizontal add and horizontal min/max instructions; and vector permutes instructions. C unit **245** includes also contains 4 vector control registers (CUCR0 to CUCR3) used to control certain operations of C unit **245** instructions. Control registers CUCR0 to CUCR3 are used as operands in certain C unit **245** operations. Control registers CUCR0 to CUCR3 are preferably used: in control of a general permutation instruction (VPERM); and as masks for SIMD multiple DOT product operations (DOTPM) and SIMD multiple Sum-of-Absolute-Difference (SAD) operations. Control register CUCR0 is preferably used to store the polynomials for Galois Field Multiply operations (GFMPY). Control register CUCR1 is preferably used to store the Galois field polynomial generator function.

[0070] Vector datapath side B **116** includes P unit **246**. P unit **246** performs basic logic operations on registers of local predicate register file **234**. P unit **246** has direct access to read from and write to predication register file **234**. These operations include AND, ANDN, OR, XOR, NOR, BITR, NEG, SET, BITCNT, RMBD, BIT Decimate and Expand. A commonly expected use of P unit **246** includes manipulation of the SIMD vector comparison results for use in control of a further SIMD vector operation.

[0071] FIG. **3** illustrates global scalar register file **211**. There are 16 independent 64-bit wide scalar registers designated A0 to A15. Each register of global scalar register file **211** can be read from or written to as 64-bits of scalar data. All scalar datapath side A **115** functional units (L1 unit **221**, S1 unit **222**, M1 unit **223**, N1 unit **224**, D1 unit **225** and D2 unit **226**) can read or write to global scalar register file **211**. Global scalar register file **211** may be read as 32-bits or as 64-bits and may only be written to as 64-bits. The instruction executing determines the read data size. Vector datapath side B **116** functional units (L2 unit **241**, S2 unit **242**, M2 unit **243**, N2 unit **244**, C unit **245**, and P unit **246**) can read from global scalar register file **211** via crosspath **117** under restrictions that will be detailed below.

[0072] FIG. **4** illustrates D1/D2 local register file **214**. There are 16 independent 64-bit wide scalar registers designated D0 to D16. Each register of D1/D2 local register file **214** can be read from or written to as 64-bits of scalar data. All scalar datapath side A **115** functional units (L1 unit **221**, S1 unit **222**, M1 unit **223**, N1 unit **224**, D1 unit **225** and D2 unit **226**) can write to global scalar register file **211**. Only D1 unit **225** and D2 unit **226** can read from D1/D2 local scalar register file **214**. It is expected that data stored in D1/D2 local scalar register file **214** will include base addresses and offset addresses used in address calculation.

[0073] FIG. **5** illustrates L1/S1 local register file **212**. The embodiment illustrated in FIG. **5** has 8 independent 64-bit wide scalar registers designated AL0 to AL7. The preferred instruction coding (see FIG. **13**) permits L1/S1 local register file **212** to include up to 16 registers. The embodiment of FIG. **5** implements only 8 registers to reduce circuit size and complexity. Each register of L1/S1 local register file **212** can be read from or written to as 64-bits of scalar data. All scalar datapath side A **115** functional units (L1 unit **221**, S1 unit **222**, M1 unit **223**, N1 unit **224**, D1 unit **225**, and D2 unit **226**) can write to L1/S1 local scalar register file **212**. Only L1 unit **221** and S1 unit **222** can read from L1/S1 local scalar register file **212**.

[0074] FIG. **6** illustrates M1/N1 local register file **213**. The embodiment illustrated in FIG. **6** has 8 independent 64-bit wide scalar registers designated AM0 to AM7. The preferred instruction coding (see FIG. **13**) permits M1/N1 local register file **213** to include up to 16 registers. The embodiment of FIG. **6** implements only 8 registers to reduce circuit size and complexity. Each register of M1/N1 local register file **213** can be read from or written to as 64-bits of scalar data. All scalar datapath side A **115** functional units (L1 unit **221**, S1 unit **222**, M1 unit **223**, N1 unit **224**, D1 unit **225** and D2 unit **226**) can write to M1/N1 local scalar register file **213**. Only M1 unit **223** and N1 unit **224** can read from M1/N1 local scalar register file **213**.

[0075] FIG. **7** illustrates global vector register file **231**. There are 16 independent 512-bit wide vector registers. Each register of global vector register file **231** can be read from or written to as 64-bits of scalar data designated B0 to B15. Each register of global vector register file **231** can be read from or written to as 512-bits of vector data designated VB0 to VB15. The instruction type determines the data size. All vector datapath side B **116** functional units (L2 unit **241**, S2 unit **242**, M2 unit **243**, N2 unit **244**, C unit **245** and P unit **246**) can read or write to global scalar register file **231**. Scalar datapath side A **115** functional units (L1 unit **221**, S1 unit **222**, M1 unit **223**, N1 unit **224**, D1 unit **225**, and D2 unit **226**) can read from global vector register file **231** via crosspath **117** under restrictions that will be detailed below.

[0076] FIG. **8** illustrates P local register file **234**. There are 8 independent 64-bit wide registers designated P0 to P15. Each register of P local register file **234** can be read from or written to as 64-bits of scalar data. Vector datapath side B **116** functional units L2 unit **241**, S2 unit **242**, C unit **244** and P unit **246** can write to P local register file **234**. Only L2 unit **241**, S2 unit **242** and P unit **246** can read from P local scalar register file **234**. A commonly expected use of P local register file **234** includes: writing one bit SIMD vector comparison results from L2 unit **241**, S2 unit **242**, or C unit **245**; manipulation of the SIMD vector comparison results by P unit **246**; and use of the manipulated results in control of a further SIMD vector operation.

[0077] FIG. **9** illustrates L2/S2 local register file **232**. The embodiment illustrated in FIG. **9** has 8 independent 512-bit wide vector registers. The preferred instruction coding (see FIG. **13**) permits L2/S2 local register file **232** to include up to 16 registers. The embodiment of FIG. **9** implements only 8 registers to reduce circuit size and complexity. Each register of L2/S2 local vector register file **232** can be read from or written to as 64-bits of scalar data designated BL0 to BL7. Each register of L2/S2 local vector register file **232** can be read from or written to as 512-bits of vector data designated VBL0 to VBL7. The instruction type determines the data size. All vector datapath side B **116** functional units (L2 unit **241**, S2 unit **242**, M2 unit **243**, N2 unit **244**, C unit **245**, and P unit **246**) can write to L2/S2 local vector register file **232**. Only L2 unit **241** and S2 unit **242** can read from L2/S2 local vector register file **232**.

[0078] FIG. **10** illustrates M2/N2/C local register file **233**. The embodiment illustrated in FIG. **10** has 8 independent 512-bit wide vector registers. The preferred instruction coding (see FIG. **13**) permits M2/N2/C local register file **233** to include up to 16 registers. The embodiment of FIG. **10** implements only 8 registers to reduce circuit size and complexity. Each register of M2/N2/C local vector register file **233** can be read from or written to as 64-bits of scalar data designated BM0 to

BM7. Each register of M2/N2/C local vector register file **233** can be read from or written to as 512-bits of vector data designated VBM0 to VBM7. All vector datapath side B **116** functional units (L2 unit **241**, S2 unit **242**, M2 unit **243**, N2 unit **244**, C unit **245**, and P unit **246**) can write to M2/N2/C local vector register file **233**. Only M2 unit **243**, N2 unit **244**, and C unit **245** can read from M2/N2/C local vector register file **233**.

[0079] The provision of global register files accessible by all functional units of a side and local register files accessible by only some of the functional units of a side is a design choice. This invention could be practiced employing only one type of register file corresponding to the disclosed global register files.

[0080] Crosspath **117** permits limited exchange of data between scalar datapath side A **115** and vector datapath side B **116**. During each operational cycle one 64-bit data word can be recalled from global scalar register file A **211** for use as an operand by one or more functional units of vector datapath side B **116** and one 64-bit data word can be recalled from global vector register file **231** for use as an operand by one or more functional units of scalar datapath side A **115**. Any scalar datapath side A **115** functional unit (L1 unit **221**, S1 unit **222**, M1 unit **223**, N1 unit **224**, D1 unit **225**, and D2 unit **226**) may read a 64-bit operand from global vector register file **231**. This 64-bit operand is the least significant bits of the 512-bit data in the accessed register of global vector register file **232**. Plural scalar datapath side A **115** functional units may employ the same 64-bit crosspath data as an operand during the same operational cycle. However, only one 64-bit operand is transferred from vector datapath side B **116** to scalar datapath side A **115** in any single operational cycle. Any vector datapath side B **116** functional unit (L2 unit **241**, S2 unit **242**, M2 unit **243**, N2 unit **244**, C unit **245** and P unit **246**) may read a 64-bit operand from global scalar register file **211**. If the corresponding instruction is a scalar instruction, the crosspath operand data is treated as any other 64-bit operand. If the corresponding instruction is a vector instruction, the upper 448 bits of the operand are zero filled. Plural vector datapath side B **116** functional units may employ the same 64-bit crosspath data as an operand during the same operational cycle. Only one 64-bit operand is transferred from scalar datapath side A **115** to vector datapath side B **116** in any single operational cycle.

[0081] Streaming engine **125** transfers data in certain restricted circumstances. Streaming engine **125** controls two data streams. A stream consists of a sequence of elements of a particular type. Programs that operate on streams read the data sequentially, operating on each element in turn. Every stream has the following basic properties. The stream data have a well-defined beginning and ending in time. The stream data have fixed element size and type throughout the stream. The stream data have fixed sequence of elements. Thus programs cannot seek randomly within the stream. The stream data is read-only while active. Programs cannot write to a stream while simultaneously reading from it. Once a stream is opened streaming engine **125**: calculates the address; fetches the defined data type from level two unified cache (which may require cache service from a higher level memory); performs data type manipulation such as zero extension, sign extension, data element sorting/swapping such as matrix transposition; and delivers the data directly to the programmed data register file within CPU **110**. Streaming engine **125** is thus useful for real-time digital filtering operations on well-behaved data. Streaming engine **125** frees these memory fetch tasks from the corresponding CPU enabling other processing functions.

[0082] Streaming engine **125** provides the following benefits. Streaming engine **125** permits multi-dimensional memory accesses. Streaming engine **125** increases the available bandwidth to the functional units. Streaming engine **125** reduces the number of cache miss stalls since the stream buffer bypasses level one data cache **123**. Streaming engine **125** reduces the number of scalar operations required to maintain a loop. Streaming engine **125** manages address pointers. Streaming engine **125** handles address generation automatically freeing up the address generation instruction slots and D1 unit **225** and D2 unit **226** for other computations.

[0083] CPU **110** operates on an instruction pipeline. Instructions are fetched in instruction packets of fixed length further described below. All instructions require the same number of pipeline phases for fetch and decode, but require a varying number of execute phases.

[0084] FIG. **11** illustrates the following pipeline phases: program fetch phase **1110**, dispatch and decode phases **1120** and execution phases **1130**. Program fetch phase **1110** includes three stages for all instructions. Dispatch and decode phases **1120** include three stages for all instructions. Execution phase **1130** includes one to four stages dependent on the instruction.

[0085] Fetch phase **1110** includes program address generation stage **1111** (PG), program access stage **1112** (PA) and program receive stage **1113** (PR). During program address generation stage **1111** (PG), the program address is generated in the CPU and the read request is sent to the memory controller for the level one instruction cache L1I. During the program access stage **1112** (PA) the level one instruction cache L1I processes the request, accesses the data in its memory and sends a fetch packet to the CPU boundary. During the program receive stage **1113** (PR) the CPU registers the fetch packet.

[0086] Instructions are always fetched sixteen 32-bit wide slots, constituting a fetch packet, at a time. FIG. **12** illustrates 16 instructions **1201** to **1216** of a single fetch packet. Fetch packets are aligned on 512-bit (16-word) boundaries. The preferred embodiment employs a fixed 32-bit instruction length. Fixed length instructions are advantageous for several reasons. Fixed length instructions enable easy decoder alignment. A properly aligned instruction fetch can load plural instructions into parallel instruction decoders. Such a properly aligned instruction fetch can be achieved by predetermined instruction alignment when stored in memory (fetch packets aligned on 512-bit boundaries) coupled with a fixed instruction packet fetch. An aligned instruction fetch permits operation of parallel decoders on instruction-sized fetched bits. Variable length instructions require an initial step of locating each instruction boundary before they can be decoded. A fixed length instruction set generally permits more regular layout of instruction fields. This simplifies the construction of each decoder which is an advantage for a wide issue VLIW central processor.

[0087] The execution of the individual instructions is partially controlled by a p bit in each instruction. This p bit is preferably bit 0 of the 32-bit wide slot. The p bit determines whether an instruction executes in parallel with a next instruction. Instructions are scanned from lower to higher address. If the p bit of an instruction is 1, then the next following instruction (higher memory address) is executed in parallel with (in the same cycle as) that instruction. If the p bit of an instruction is 0, then the next following instruction is executed in the cycle after the instruction.

[0088] CPU **110** and level one instruction cache L1I **121** pipelines are de-coupled from each other. Fetch packet returns from level one instruction cache L1I can take different number of clock cycles, depending on external circumstances such as whether there is a hit in level one instruction cache **121** or a hit in level two combined cache **130**. Therefore program access stage **1112** (PA) can take several clock cycles instead of 1 clock cycle as in the other stages.

[0089] The instructions executing in parallel constitute an execute packet. In the preferred embodiment an execute packet can contain up to sixteen instructions. No two instructions in an execute packet may use the same functional unit. A slot is one of five types: 1) a self-contained instruction executed on one of the functional units of CPU **110** (L1 unit **221**, S1 unit **222**, M1 unit **223**, N1 unit **224**, D1 unit **225**, D2 unit **226**, L2 unit **241**, S2 unit **242**, M2 unit **243**, N2 unit **244**, C unit **245** and P unit **246**); 2) a unitless instruction such as a NOP (no operation) instruction or multiple NOP instruction; 3) a branch instruction; 4) a constant field extension; and 5) a conditional code extension. Some of these slot types will be further explained below.

[0090] Dispatch and decode phases **1120** include instruction dispatch to appropriate execution unit stage **1121** (DS), instruction pre-decode stage **1122** (DC1); and instruction decode, operand reads stage **1123** (DC2). During instruction dispatch to appropriate execution unit stage **1121** (DS), the fetch packets are split into execute packets and assigned to the appropriate functional units. During the instruction pre-decode stage **1122** (DC1), the source registers, destination registers, and associated paths are decoded for the execution of the instructions in the functional units. During the instruction decode, operand reads stage **1123** (DC2), more detailed unit decodes are done, as well as reading operands from the register files.

[0091] Execution phases **1130** includes execution stages **1131** to **1135** (E1 to E5). Different types of instructions require different numbers of these stages to complete their execution. These stages of the pipeline play an important role in understanding the device state at CPU cycle boundaries.

[0092] During execute 1 stage **1131** (E1) the conditions for the instructions are evaluated and operands are operated on. As illustrated in FIG. **11**, execute 1 stage **1131** may receive operands from a stream buffer **1141** and one of the register files shown schematically as **1142**. For load and store instructions, address generation is performed and address modifications are written to a register file. For branch instructions, branch fetch packet in PG phase **1111** is affected. As illustrated in FIG. **11**, load and store instructions access memory here shown schematically as memory **1151**. For single-cycle instructions, results are written to a destination register file. This assumes that any conditions for the instructions are evaluated as true. If a condition is evaluated as false, the instruction does not write any results or have any pipeline operation after execute 1 stage **1131**.

[0093] During execute 2 stage **1132** (E2) load instructions send the address to memory. Store instructions send the address and data to memory. Single-cycle instructions that saturate results set the SAT bit in the control status register (CSR) if saturation occurs. For 2-cycle instructions, results are written to a destination register file.

[0094] During execute 3 stage **1133** (E3) data memory accesses are performed. Any multiply instructions that saturate results set the SAT bit in the control status register (CSR) if saturation occurs. For 3-cycle instructions, results are written to a destination register file.

[0095] During execute 4 stage **1134** (E4) load instructions bring data to the CPU boundary. For 4-cycle instructions, results are written to a destination register file.

[0096] During execute 5 stage **1135** (E5) load instructions write data into a register. This is illustrated schematically in FIG. **11** with input from memory **1151** to execute 5 stage **1135**.

[0097] FIG. **13** illustrates an example of the instruction coding **1300** of functional unit instructions used by this invention. Those skilled in the art would realize that other instruction codings are feasible and within the scope of this invention. Each instruction consists of 32 bits and controls the operation of one of the individually controllable functional units (L1 unit **221**, S1 unit **222**, M1 unit **223**, N1 unit **224**, D1 unit **225**, D2 unit **226**, L2 unit **241**, S2 unit **242**, M2 unit **243**, N2 unit **244**, C unit **245**, and P unit **246**). The bit fields are defined as follows.

[0098] The creg field **1301** (bits 29 to 31) and the z bit **1302** (bit 28) are optional fields used in conditional instructions. These bits are used for conditional instructions to identify the predicate register and the condition. The z bit **1302** (bit 28) indicates whether the predication is based upon zero or not zero in the predicate register. If z=1, the test is for equality with zero. If z=0, the test is for nonzero. The case of creg=0 and z=0 is treated as always true to allow unconditional instruction execution. The creg field **1301** and the z field **1302** are encoded in the instruction as shown in Table 1.

TABLE-US-00001 TABLE 1 Conditional creg z Register 31 30 29 28 Unconditional 0 0 0 0 Reserved 0 0 0 1 A0 0 0 1 z A1 0 1 0 z A2 0 1 1 z A3 1 0 0 z A4 1 0 1 z A5 1 1 0 z Reserved 1 1 x x

[0099] Execution of a conditional instruction is conditional upon the value stored in the specified data register. This data register is in the global scalar register file **211** for all functional units. Note that "z" in the z bit column refers to the zero/not zero comparison selection noted above and "x" is a don't care state. This coding can only specify a subset of the 16 global registers as predicate registers. This selection was made to preserve bits in the instruction coding. Note that unconditional instructions do not have these optional bits. For unconditional instructions these bits in fields **1301** and **1302** (**28** to **31**) are preferably used as additional opcode bits.

[0100] The dst field **1303** (bits 23 to 27) specifies a register in a corresponding register file as the destination of the instruction results.

[0101] The src2/cst field **1304** (bits 18 to 22) has several meanings depending on the instruction opcode field (bits 3 to 12 for all instructions and additionally bits 28 to 31 for unconditional instructions). The first meaning specifies a register of a corresponding register file as the second operand. The second meaning is an immediate constant. Depending on the instruction type, this is treated as an unsigned integer and zero extended to a specified data length or is treated as a signed integer and sign extended to the specified data length.

[0102] The src1 field **1305** (bits 13 to 17) specifies a register in a corresponding register file as the first source operand.

[0103] The opcode field **1306** (bits 3 to 12) for all instructions (and additionally bits 28 to 31 for unconditional instructions) specifies the type of instruction and designates appropriate instruction options. This includes unambiguous designation of the functional unit used and operation performed. A detailed explanation of the opcode is beyond the scope of this invention except for the instruction options detailed below.

[0104] The e bit **1307** (bit 2) is only used for immediate constant instructions where the constant may be extended. If e=1, then the immediate constant is extended in a manner detailed below. If e=0, then the immediate constant is not extended. In that case the immediate constant is specified by the src2/cst field **1304** (bits 18 to 22). Note that this e bit **1307** is used for only some instructions. Accordingly, with proper coding this e bit **1307** may be omitted from instructions which do not need it and this bit used as an additional opcode bit.

[0105] The s bit **1308** (bit 1) designates scalar datapath side A **115** or vector datapath side B **116**. If s=0, then scalar datapath side A **115** is selected. This limits the functional unit to L1 unit **221**, S1 unit **222**, M1 unit **223**, N1 unit **224**, D1 unit **225** and D2 unit **226** and the corresponding register files illustrated in FIG. **2**. Similarly, s=1 selects vector datapath side B **116** limiting the functional unit to L2 unit **241**, S2 unit **242**, M2 unit **243**, N2 unit **244**, P unit **246** and the corresponding register file illustrated in FIG. **2**.

[0106] The p bit **1309** (bit 0) marks the execute packets. The p-bit determines whether the instruction executes in parallel with the following instruction. The p-bits are scanned from lower to higher address. If p=1 for the current instruction, then the next instruction executes in parallel with the current instruction. If p=0 for the current instruction, then the next instruction executes in the cycle after the current instruction. All instructions executing in parallel constitute an execute packet. An execute packet can contain up to twelve instructions. Each instruction in an execute packet must use a different functional unit.

[0107] There are two different condition code extension slots. Each execute packet can contain one each of these unique 32-bit condition code extension slots which contains the 4-bit creg/z fields for the instructions in the same execute packet. FIG. **14** illustrates the coding for condition code extension slot 0 **1400** and FIG. **15** illustrates the coding for condition code extension slot 1 **1500**.

[0108] FIG. **14** illustrates the coding for condition code extension slot 0 **1400** having 32 bits. Field **1401** (bits 28 to 31) specify 4 creg/z bits assigned to the L1 unit **221** instruction in the same execute packet. Field **1402** (bits 27 to 24) specify 4 creg/z bits assigned to the L2 unit **241** instruction in the same execute packet. Field **1403** (bits 19 to 23) specify 4 creg/z bits assigned to the S1 unit **222** instruction in the same execute packet. Field **1404** (bits 16 to 19) specify 4 creg/z bits assigned to the S2 unit **242** instruction in the same execute packet. Field **1405** (bits 12 to 15) specify 4 creg/z bits assigned to the D1 unit **225** instruction in the same execute packet. Field **1406** (bits 8 to 11) specify 4 creg/z bits assigned to the D2 unit **226** instruction in the same execute packet. Field **1407** (bits 6 and 7) is unused/reserved. Field **1408** (bits 0 to 5) are coded a set of unique bits (CCEX0) to identify the condition code extension slot 0. Once this unique ID of condition code extension slot 0 is detected, the corresponding creg/z bits are employed to control conditional execution of any L1 unit **221**, L2 unit **241**, S1 unit **222**, S2 unit **242**, D1 unit **225**, and D2 unit **226** instruction in the same execution packet. These creg/z bits are interpreted as shown in Table 1. If the corresponding instruction is conditional (includes creg/z bits) the corresponding bits in the condition code extension slot 0 override the condition code bits in the instruction. Note that no execution packet can have more than one instruction directed to a particular execution unit. No execute packet of instructions can contain more than one condition code extension slot 0. Thus the mapping of creg/z bits to functional unit instruction is unambiguous. Setting the creg/z bits equal to “0000” makes the instruction unconditional. Thus a properly coded condition code extension slot 0 can make some corresponding instructions conditional and some unconditional.

[0109] FIG. **15** illustrates the coding for condition code extension slot 1 **1500** having 32 bits. Field **1501** (bits 28 to 31) specify 4 creg/z bits assigned to the M1 unit **223** instruction in the same execute packet. Field **1502** (bits 27 to 24) specify 4 creg/z bits assigned to the M2 unit **243** instruction in the same execute packet. Field **1503** (bits 19 to 23) specify 4 creg/z bits assigned to the C unit **245** instruction in the same execute packet. Field **1504** (bits 16 to 19) specify 4 creg/z bits assigned to the N1 unit **224** instruction in the same execute packet. Field **1505** (bits 12 to 15) specify 4 creg/z bits assigned to the N2 unit **244** instruction in the same execute packet. Field **1506** (bits 6 to 11) is unused/reserved. Field **1507** (bits 0 to 5) are coded a set of unique bits (CCEX1) to identify the condition code extension slot 1. Once this unique ID of condition code extension slot 1 is detected, the corresponding creg/z bits are employed to control conditional execution of any M1 unit **223**, M2 unit **243**, C unit **245**, N1 unit **224** and N2 unit **244** instruction in the same execution packet. These creg/z bits are interpreted as shown in Table 1. If the corresponding instruction is conditional (includes creg/z bits) the corresponding bits in the condition code extension slot 1 override the condition code bits in the instruction. Note that no execution packet can have more than one instruction directed to a particular execution unit. No execute packet of instructions can contain more than one condition code extension slot 1. Thus the mapping of creg/z bits to functional unit instruction is unambiguous. Setting the creg/z bits equal to “0000” makes the instruction unconditional. Thus a properly coded condition code extension slot 1 can make some instructions conditional and some unconditional.

[0110] It is feasible for both condition code extension slot 0 and condition code extension slot 1 to include a p bit to define an execute packet as described above in conjunction with FIG. **13**. In the preferred embodiment, as illustrated in FIGS. **14** and **15**, code extension slot 0 and condition code extension slot 1 preferably have bit 0 (p bit) always encoded as 1. Thus neither condition code extension slot 0 nor condition code extension slot 1 can be in the last instruction slot of an execute packet.

[0111] There are two different constant extension slots. Each execute packet can contain one each of these unique 32-bit constant extension slots which contains 27 bits to be concatenated as high order bits with the 5-bit constant field **1305** to form a 32-bit constant. As noted in the instruction coding description above only some instructions define the src2/cst field **1304** as a constant rather than a source register identifier. At least some of those instructions may employ a constant extension slot to extend this constant to 32 bits.

[0112] FIG. **16** illustrates the fields of constant extension slot 0. Each execute packet may include one instance of constant extension slot 0 and one instance of constant extension slot 1. FIG. **16** illustrates that constant extension slot 0 **1600** includes two fields. Field **1601** (bits 5 to 31) constitute the most significant 27 bits of an extended 32-bit constant including the target instruction scr2/cst field **1304** as the five least significant bits. Field **1602** (bits 0 to 4) are coded a set of unique bits (CSTX0) to identify the constant extension slot 0. In the preferred embodiment constant extension slot 0 **1600** can only be used to extend the constant of one of an L1 unit **221** instruction, data in a D1 unit **225** instruction, an S2 unit **242** instruction, an offset in a D2 unit **226** instruction, an M2 unit **243** instruction, an N2 unit **244** instruction, a branch instruction, or a C unit **245** instruction in the same execute packet. Constant extension slot 1 is similar to constant extension slot 0 except that bits 0 to 4 are coded a set of unique bits (CSTX1) to identify the constant extension slot 1. In the preferred embodiment constant extension slot 1 can only be used to extend the constant of one of an L2 unit **241** instruction, data in a D2 unit **226** instruction, an S1 unit **222** instruction, an offset in a D1 unit **225** instruction, an M1 unit **223** instruction or an N1 unit **224** instruction in the same execute packet.

[0113] Constant extension slot 0 and constant extension slot 1 are used as follows. The target instruction must be of the type permitting constant specification. As known in the art this is implemented by replacing one input operand register specification field with the least significant bits of the constant as described above with respect to scr2/cst field **1304**. Instruction decoder **113** determines this case, known as an immediate field, from the instruction opcode bits. The target instruction also includes one constant extension bit (e bit **1307**) dedicated to signaling whether the specified constant is not extended (preferably constant extension bit=0) or the constant is extended (preferably constant extension bit=1). If instruction decoder **113** detects a constant extension slot 0 or a constant extension slot 1, it further checks the other instructions within that execute packet for an instruction corresponding to the detected constant extension slot. A constant extension is made only if one corresponding instruction has a constant extension bit (e bit **1307**) equal to 1.

[0114] FIG. **17** is a partial block diagram **1700** illustrating constant extension. FIG. **17** assumes that instruction decoder **113** detects a constant extension slot and a corresponding instruction in the same execute packet. Instruction decoder **113** supplies the 27 extension bits from the constant extension slot (bit field **1601**) and the 5 constant bits (bit field **1305**) from the corresponding instruction to concatenator **1701**. Concatenator **1701** forms a single 32-bit word from these two parts. In the preferred embodiment the 27 extension bits from the constant extension slot (bit field **1601**) are the most significant bits and the 5 constant bits (bit field **1305**) are the least significant bits. This combined 32-bit word is supplied to one input of multiplexer **1702**. The 5 constant bits from the corresponding instruction field **1305** supply a second input to multiplexer **1702**. Selection of multiplexer **1702** is controlled by the status of the constant extension bit. If the constant extension bit (e bit **1307**) is 1 (extended), multiplexer **1702** selects the concatenated 32-bit input. If the constant extension bit is 0 (not extended), multiplexer **1702** selects the 5 constant bits from the corresponding instruction field **1305**. Multiplexer **1702** supplies this output to an input of sign extension unit **1703**.

[0115] Sign extension unit **1703** forms the final operand value from the input from multiplexer **1703**. Sign extension unit **1703** receives control inputs Scalar/Vector and Data Size. The Scalar/Vector input indicates whether the corresponding instruction is a scalar instruction or a vector instruction. The functional units of data path side A **115** (L1 unit **221**, S1 unit **222**, M1 unit **223**, N1 unit **224**, D1 unit **225**, and D2 unit **226**) can only perform scalar instructions. Any instruction directed to one of these functional units is a scalar instruction. Data path side B functional units L2 unit **241**, S2 unit **242**, M2 unit **243**, N2 unit **244** and C unit **245** may perform scalar instructions or vector instructions. Instruction decoder **113** determines whether the instruction is a scalar instruction or a vector instruction from the opcode bits. P unit **246** may only perform scalar instructions. The Data Size may be 8 bits (byte B), 16 bits (half-word H), 32 bits (word W) or 64 bits (double word D).

[0116] Table 2 lists the operation of sign extension unit **1703** for the various options.

TABLE-US-00002 TABLE 2

Instruction Type	Operand Size	Constant Length	Action
Scalar	B/H/W/D	5 bits	Sign extend to 64 bits
Scalar	B/H/W/D	32 bits	Sign extend to 64 bits
Vector	B/H/W/D	5 bits	Sign extend to operand size and replicate across whole vector
Vector	B/H/W	32 bits	Replicate 32-bit constant across each 32-bit (W) lane
Vector	D	32 bits	Sign extend to 64 bits and replicate across each 64-bit (D) lane

[0117] It is feasible for both constant extension slot 0 and constant extension slot 1 to include a p bit to define an execute packet as described above in conjunction with FIG. **13**. In the preferred embodiment, as in the case of the condition code extension slots, constant extension slot 0 and constant extension slot 1 preferably have bit 0 (p bit) always encoded as 1. Thus neither constant extension slot 0 nor constant extension slot 1 can be in the last instruction slot of an execute packet.

[0118] It is technically feasible for an execute packet to include a constant extension slot 0 or 1 and more than one corresponding instruction marked constant extended (e bit=1). For constant extension slot 0 this would mean more than one of an L1 unit **221** instruction, data in a D1 unit **225** instruction, an S2 unit **242** instruction, an offset in a D2 unit **226** instruction, an M2 unit **243** instruction or an N2 unit **244** instruction in an execute packet have an e bit of 1. For constant extension slot 1 this would mean more than one of an L2 unit **241** instruction, data in a D2 unit **226** instruction, an S1 unit **222** instruction, an offset in a D1 unit **225** instruction, an M1 unit **223** instruction or an N1 unit **224** instruction in an execute packet have an e bit of 1. Supplying the same constant extension to more than one instruction is not expected to be a useful function. Accordingly, in one embodiment instruction decoder **113** may determine this case an invalid operation and not supported. Alternately, this combination may be supported with extension bits of the constant extension slot applied to each corresponding functional unit instruction marked constant extended.

[0119] Special vector predicate instructions use registers in predicate register file **234** to control vector operations. In the current embodiment all these SIMD vector predicate instructions operate on selected data sizes. The data sizes may include byte (8 bit) data, half word (16 bit) data, word (32 bit) data, double word (64 bit) data, quad word (128 bit) data and half vector (256 bit) data. Each bit of the predicate register controls whether a SIMD operation is performed upon the corresponding byte of data. The operations of P unit **245** permit a variety of compound vector SIMD operations based upon more than one vector comparison. For example a range determination can be made using two comparisons. A candidate vector is compared with a first vector reference having the minimum of the range packed within a first data register. A second comparison of the candidate vector is made with a second reference vector having the maximum of the range packed within a second data register. Logical combinations of the two resulting predicate registers would permit a vector conditional operation to determine whether each data part of the candidate vector is within range or out of range.

[0120] L1 unit **221**, S1 unit **222**, L2 unit **241**, S2 unit **242** and C unit **245** often operate in a single instruction multiple data (SIMD) mode. In this SIMD mode the same instruction is applied to packed data from the two operands. Each operand holds plural data elements disposed in predetermined slots. SIMD operation is enabled by carry control at the data boundaries. Such carry control enables operations on varying data widths.

[0121] FIG. **18** illustrates the carry control. AND gate **1801** receives the carry output of bit N within the operand wide arithmetic logic unit (64 bits for scalar datapath side A **115** functional units and 512 bits for vector datapath side B **116** functional units). AND gate **1801** also receives a carry control signal which will be further explained below. The output of AND gate **1801** is supplied to the carry input of bit N+1 of the operand wide arithmetic logic unit. AND gates such as AND gate **1801** are disposed between every pair of bits at a possible data boundary. For example, for 8-bit data such an AND gate will be between bits 7 and 8, bits 15 and 16, bits 23 and 24, etc. Each such AND gate receives a corresponding carry control signal. If the data size is of the minimum, then each carry control signal is 0, effectively blocking carry transmission between the adjacent bits. The corresponding carry control signal is 1 if the selected data size requires both arithmetic logic unit sections. Table 3 below shows example carry control signals for the case of a 512 bit wide operand such as used by vector datapath side B **116** functional units which may be divided into sections of 8 bits, 16 bits, 32 bits, 64 bits, 128 bits or 256 bits. In Table 3 the upper 32 bits control the upper bits (bits 128 to 511) carries and the lower 32 bits control the lower bits (bits 0 to 127) carries. No control of the carry output of the most significant bit is needed, thus only 63 carry control signals are required.

TABLE-US-00003 TABLE 3

Data Size	Carry Control Signals
8 bits	−000 0000 0000 0000 0000 0000 0000 0000 (B) 0000 0000 0000 0000 0000 0000 0000 0000
16 bits	−101 0101 0101 0101 0101 0101 0101 0101 (H) 0101 0101 0101 0101 0101 0101 0101 0101
32 bits	−111 0111 0111 0111 0111 0111 0111 0111 (W) 0111 0111 0111 0111 0111 0111 0111 0111
64 bits	−111 1111 0111

1111 0111 1111 0111 1111 (D) 0111 1111 0111 1111 0111 1111 0111 1111 128 bits −111 1111 1111 1111 0111 1111 1111 1111 0111 1111 1111 1111 0111 1111 1111 1111 256 bits −111 1111 1111 1111 1111 1111 1111 1111 0111 1111 1111 1111 1111 1111 1111 1111

[0122] It is typical in the art to operate on data sizes that are integral powers of 2 (2.sup.N). However, this carry control technique is not limited to integral powers of 2. One skilled in the art would understand how to apply this technique to other data sizes and other operand widths.

[0123] FIG. **19** illustrates a conceptual view of the streaming engines of this invention. FIG. **19** illustrates the process of a single stream. Streaming engine **1900** includes stream address generator **1901**. Stream address generator **1901** sequentially generates addresses of the elements of the stream and supplies these element addresses to system memory **1910**. Memory **1910** recalls data stored at the element addresses (data elements) and supplies these data elements to data first-in-first-out (FIFO) memory **1902**. Data FIFO **1902** provides buffering between memory **1910** and CPU **1920**. Data formatter **1903** receives the data elements from data FIFO memory **1902** and provides data formatting according to the stream definition. This process will be described below. Streaming engine **1900** supplies the formatted data elements from data formatter **1903** to the CPU **1920**. The program on CPU **1920** consumes the data and generates an output.

[0124] Stream elements typically reside in normal memory. The memory itself imposes no particular structure upon the stream. Programs define streams and therefore impose structure, by specifying the following stream attributes: address of the first element of the stream; size and type of the elements in the stream; formatting for data in the stream; and the address sequence associated with the stream.

[0125] The streaming engine defines an address sequence for elements of the stream in terms of a pointer walking through memory. A multiple-level nested loop controls the path the pointer takes. An iteration count for a loop level indicates the number of times that level repeats. A dimension gives the distance between pointer positions of that loop level.

[0126] In a basic forward stream the innermost loop always consumes physically contiguous elements from memory. The implicit dimension of this innermost loop is 1 element. The pointer itself moves from element to element in consecutive, increasing order. In each level outside the inner loop, that loop moves the pointer to a new location based on the size of that loop level's dimension.

[0127] This form of addressing allows programs to specify regular paths through memory in a small number of parameters. Table 4 lists the addressing parameters of a basic stream.

TABLE-US-00004 TABLE 4 Parameter Definition ELEM_BYTES Size of each element in bytes ICNT0 Number of iterations for the innermost loop level 0 At loop level 0 all elements are physically contiguous DIM0 is ELEM_BYTES ICNT1 Number of iterations for loop level 1 DIM1 Number of elements between the starting points for consecutive iterations of loop level 1 ICNT2 Number of iterations for loop level 2 DIM2 Number of elements between the starting points for consecutive iterations of loop level 2 ICNT3 Number of iterations for loop level 3 DIM3 Number of elements between the starting points for consecutive iterations of loop level 3

[0128] The definition above maps consecutive elements of the stream to increasing addresses in memory. This works well for most algorithms but not all. Some algorithms are better served by reading elements in decreasing memory addresses, reverse stream addressing. For example, a discrete convolution computes vector dot-products, as per the formula:

$$[00001](f,g)[t] = \overset{\infty}{\underset{x=-\infty}{.Math.}} f[x]g[t-x]$$

[0129] In most DSP code, f[ ] and g[ ] represent arrays in memory. For each output, the algorithm reads f[ ] in the forward direction, but reads g[ ] in the reverse direction. Practical filters limit the range of indices for [x] and [t−x] to a finite number elements. To support this pattern, the streaming engine supports reading elements in decreasing address order.

[0130] Matrix multiplication presents a unique problem to the streaming engine. Each element in the matrix product is a vector dot product between a row from the first matrix and a column from the second. Programs typically store matrices all in row-major or column-major order. Row-major order stores all the elements of a single row contiguously in memory. Column-major order stores all elements of a single column contiguously in memory. Matrices typically get stored in the same order as the default array order for the language. As a result, only one of the two matrices in a matrix multiplication map on to the streaming engine's 2-dimensional stream definition. In a typical example a first index steps through columns on array first array but rows on second array. This problem is not unique to the streaming engine. Matrix multiplication's access pattern fits poorly with most general-purpose memory hierarchies. Some software libraries transposed one of the two matrices, so that both get accessed row-wise (or column-wise) during multiplication. The streaming engine supports implicit matrix transposition with transposed streams. Transposed streams avoid the cost of explicitly transforming the data in memory. Instead of accessing data in strictly consecutive-element order, the streaming engine effectively interchanges the inner two loop dimensions in its traversal order, fetching elements along the second dimension into contiguous vector lanes.

[0131] This algorithm works, but is impractical to implement for small element sizes. Some algorithms work on matrix tiles which are multiple columns and rows together. Therefore, the streaming engine defines a separate transposition granularity. The hardware imposes a minimum granularity. The transpose granularity must also be at least as large as the element size. Transposition granularity causes the streaming engine to fetch one or more consecutive elements from dimension 0 before moving along dimension 1. When the granularity equals the element size, this results in fetching a single column from a row-major array. Otherwise, the granularity specifies fetching 2, 4 or more columns at a time from a row-major array. This is also applicable for column-major layout by exchanging row and column in the description. A parameter GRANULE indicates the transposition granularity in bytes.

[0132] Another common matrix multiplication technique exchanges the innermost two loops of the matrix multiply. The resulting inner loop no longer reads down the column of one matrix while reading across the row of another. For example the algorithm may hoist one term outside the inner loop, replacing it with the scalar value. On a vector machine, the innermost loop can be implements very efficiently with a single scalar-by-vector multiply followed by a vector add. The DSP CPU of this invention lacks a scalar-by-vector multiply. Programs must instead duplicate the scalar value across the length of the vector and use a vector-by-vector multiply. The streaming engine of this invention directly supports this and related use models with an element duplication mode. In this mode, the streaming engine reads a granule smaller than the full vector size and replicates that granule to fill the next vector output.

[0133] The streaming engine treats each complex number as a single element with two sub-elements that give the real and imaginary (rectangular) or magnitude and angle (polar) portions of the complex number. Not all programs or peripherals agree what order these sub-elements should appear in memory. Therefore, the streaming engine offers the ability to swap the two sub-elements of a complex number with no cost. This feature swaps the halves of an element without interpreting the contents of the element and can be used to swap pairs of sub-elements of any type, not just complex numbers.

[0134] Algorithms generally prefer to work at high precision, but high precision values require more storage and bandwidth than lower precision values. Commonly, programs will store data in memory at low precision, promote those values to a higher precision for calculation and then demote the values to lower precision for storage. The streaming engine supports this directly by allowing algorithms to specify one level of type promotion. In the preferred embodiment of this invention every sub-element may be promoted to the next larger type size with either sign or zero extension for integer types. It is also feasible that the streaming engine may support floating point promotion, promoting 16-bit and 32-bit floating point values to 32-bit and 64-bit formats, respectively.

[0135] The streaming engine defines a stream as a discrete sequence of elements, the DSP CPU consumes elements packed contiguously in vectors. Vectors resemble streams in as much as they contain multiple homogeneous elements with some implicit sequence. Because the streaming engine reads streams, but the DSP CPU consumes vectors, the streaming engine must map streams onto vectors in a consistent way.

[0136] Vectors consist of equal-sized lanes, each lane containing a sub-element. The DSP CPU designates the rightmost lane of the vector as lane 0, regardless of device's current endian mode. Lane numbers increase right-to-left. The actual number of lanes within a vector varies depending on the length of the vector and the data size of the sub-element.

[0137] FIG. **20** illustrates a first example of lane allocation in a vector. Vector **2000** is divided into 8 64-bit lanes (8×64 bits=512 bits the vector length). Lane 0 includes bits 0 to 63; line 1 includes bits 64 to 127; lane 2 includes bits 128 to 191; lane 3 includes bits 192 to 255, lane 4 includes bits 256 to 319, lane 5 includes bits 320 to 383, lane 6 includes bits 384 to 447 and lane 7 includes bits 448 to 511.

[0138] FIG. **21** illustrates a second example of lane allocation in a vector. Vector **2100** is divided into 16 32-bit lanes (16×32 bits=512 bits the vector length). Lane 0 includes bits 0 to 31; line 1 includes bits 32 to 63; lane 2 includes bits 64 to 95; lane 3 includes bits 96 to 127; lane 4 includes bits 128 to 159; lane 5 includes bits 160 to 191; lane 6 includes bits 192 to 223; lane 7 includes bits 224 to 255; lane 8 includes bits 256 to 287; line 9 occupied bits 288 to 319; lane 10 includes bits 320 to 351; lane 11 includes bits 352 to 383; lane 12 includes bits 384 to 415; lane 13 includes bits 416 to 447; lane 14 includes bits 448 to 479; and lane 15 includes bits 480 to 511.

[0139] The streaming engine maps the innermost stream dimension directly to vector lanes. It maps earlier elements within that dimension to lower lane numbers and later elements to higher lane numbers. This is true regardless of whether this particular stream advances in increasing or decreasing address order. Whatever order the stream defines, the streaming engine deposits elements in vectors in increasing-lane order. For non-complex data, it places the first element in lane 0 of the first vector the CPU fetches, the second in lane 1, and so on. For complex data, the streaming engine places the first element in lanes 0 and 1, second in lanes 2 and 3, and so on. Sub-elements within an element retain the same relative ordering regardless of the stream direction. For non-swapped complex elements, this places the sub-elements with the lower address of each pair in the even numbered lanes, and the sub-elements with the higher address of each pair in the odd numbered lanes. Swapped complex elements reverse this mapping.

[0140] The streaming engine fills each vector the CPU fetches with as many elements as it can from the innermost stream dimension. If the innermost dimension is not a multiple of the vector length, the streaming engine pads that dimension out to a multiple of the vector length with zeros. Thus for higher-dimension streams, the first element from each iteration of an outer dimension arrives in lane 0 of a vector. The streaming engine always maps the innermost dimension to consecutive lanes in a vector. For transposed streams, the innermost dimension consists of groups of sub-elements along dimension 1, not dimension 0, as transposition exchanges these two dimensions.

[0141] Two dimensional streams exhibit great variety as compared to one dimensional streams. A basic two dimensional stream extracts a smaller rectangle from a larger rectangle. A transposed 2-D stream reads a rectangle column-wise instead of row-wise. A looping stream, where the second dimension overlaps first executes a finite impulse response (FIR) filter taps which loops repeatedly or FIR filter samples which provide a sliding window of input samples.

[0142] FIG. **22** illustrates a basic two dimensional stream. The inner two dimensions, represented by ELEM_BYTES, ICNT0, DIM1 and ICNT1 give sufficient flexibility to describe extracting a smaller rectangle **2220** having dimensions **2221** and **2222** from a larger rectangle **2210** having dimensions **2211** and **2212**. In this example rectangle **2220** is a 9 by 13 rectangle of 64-bit values and rectangle **2210** is a larger 11 by 19 rectangle. The following stream parameters define this stream: [0143] ICNT0=9 [0144] ELEM_BYTES=8 [0145] ICNT1=13 [0146] DIM1=88 (11 times 8)

Thus the iteration count in the 0 dimension **2221** is 9. The iteration count in the 1 direction **2222** is 13. Note that the ELEM_BYTES only scales the innermost dimension. The first dimension has ICNT0 elements of size ELEM_BYTES. The stream address generator does not scale the outer dimensions. Therefore, DIM1=88, which is 11 elements scaled by 8 bytes per element.

[0147] FIG. **23** illustrates the order of elements within this example stream. The streaming engine fetches elements for the stream in the order illustrated in order **2300**. The first 9 elements come from the first row of rectangle **2220**, left-to-right in hops 1 to 8. The 10th through 24th elements comes from the second row, and so on. When the stream moves from the 9th element to the 10th element (hop 9 in FIG. **23**), the streaming engine computes the new location based on the pointer's position at the start of the inner loop, not where the pointer ended up at the end of the first dimension. This makes DIM1 independent of ELEM_BYTES and ICNT0. DIM1 always represents the distance between the first bytes of each consecutive row.

[0148] Transposed streams access along dimension 1 before dimension 0. The following examples illustrate a couple transposed streams, varying the transposition granularity. FIG. **24** illustrates extracting a smaller rectangle **2420** (12×8) having dimensions **2421** and **2422** from a larger rectangle **2410** (14×13) having dimensions **2411** and **2412**. In FIG. **24** ELEM_BYTES equals 2.

[0149] FIG. **25** illustrates how the streaming engine would fetch the stream of this example with a transposition granularity of 4 bytes. Fetch pattern **2500** fetches pairs of elements from each row (because the granularity of 4 is twice the ELEM_BYTES of 2), but otherwise moves down the columns. Once it reaches the bottom of a pair of columns, it repeats this pattern with the next pair of columns.

[0150] FIG. **26** illustrates how the streaming engine would fetch the stream of this example with a transposition granularity of 8 bytes. The overall structure remains the same. The streaming engine fetches 4 elements from each row (because the granularity of 8 is four times the ELEM_BYTES of 2) before moving to the next row in the column as shown in fetch pattern **2600**.

[0151] The streams examined so far read each element from memory exactly once. A stream can read a given element from memory multiple times, in effect looping over a piece of memory. FIR filters exhibit two common looping patterns. FIRs re-read the same filter taps for each output. FIRs also read input samples from a sliding window. Two consecutive outputs will need inputs from two overlapping windows.

[0152] FIG. **27** illustrates the details of streaming engine **2700**. Streaming engine **2700** contains three major sections: Stream 0 **2710**; Stream 1 **2720**; and Shared L2 Interfaces **2730**. Stream 0 **2710** and Stream 1 **2720** both contain identical hardware that operates in parallel. Stream 0 **2710** and Stream 1 **2720** both share L2 interfaces **2730**. Each stream **2710** and **2720** provides the CPU with up to 512 bits/cycle, every cycle. The streaming engine architecture enables this through its dedicated stream paths and shared dual L2 interfaces.

[0153] Each streaming engine **2700** includes a dedicated 4-dimensional stream address generator **2711**/**2721** that can each generate one new non-aligned request per cycle. Address generators **2711**/**2721** output 512-bit aligned addresses that overlap the elements in the sequence defined by the stream parameters. This will be further described below.

[0154] Each address generator **2711**/**2711** connects to a dedicated micro table look-aside buffer (pTLB) **2712**/**2722**. The pTLB **2712**/**2722** converts a single 48-bit virtual address to a 44-bit physical address each cycle. Each pTLB **2712**/**2722** has 8 entries, covering a minimum of 32 kB with 4 kB pages or a maximum of 16 MB with 2 MB pages. Each address generator **2711**/**2721** generates 2 addresses per cycle. The pTLB **2712**/**2722** only translates 1 address per cycle. To maintain throughput, streaming engine **2700** takes advantage of the fact that most stream references will be within the same 4 kB page. Thus the address translation does not modify bits 0 to 11 of the address. If aout0 and aout1 line in the same 4 kB page (aout0[47:12] are the same aout1[47:12]), then the pTLB **2712**/**2722** only translates aout0 and reuses the translation for the upper bits of both addresses.

[0155] Translated addresses are queued in command queue **2713**/**2723**. These addresses are aligned with information from the corresponding Storage Allocation and Tracking block **2714**/**2724**. Streaming engine **2700** does not explicitly manage pTLB **2712**/**2722**. The system memory management unit (MMU) invalidates μTLBs as necessary during context switches.

[0156] Storage Allocation and Tracking **2714**/**2724** manages the stream's internal storage, discovering data reuse and tracking the lifetime of each piece of data. Storage Allocation and Tracking **2714**/**2724** accepts 2 virtual addresses per cycle and binds those addresses to slots in the stream's data storage. Streaming engine **2700** organizes its data store as an array of slots. Streaming engine **2700** maintains the following metadata listed in Table 5 to track the contents and lifetime of the data in each slot.

TABLE-US-00005 TABLE 5 Address 48-bit virtual address associated with the slot Valid Single bit indicating whether the tag address is valid Ready Single bit indicating the data has arrived for this address Active Single bit indicating whether there are any references outstanding to this data Last Value indicating the most recent reference Reference to this slot in the reference queue

[0157] Table 6 details the interaction of the valid, ready and active bits.

TABLE-US-00006 TABLE 6 Available for Valid Ready Active Interpretation Allocation 0 — — Address invalid Yes 1 0 0 Invalid, cannot have — data pending without reference in flight 1 0 1 Request sent for slot, No data pending 1 1 0 No active references in Yes flight 1 1 1 Reference in flight, No data available

[0158] Using this metadata, the storage allocation and tracking **2714**/**2724** can identify data reuse opportunities in the stream. Storage allocation and tracking **2714**/**2724** performs the following steps for each address. It compares the address against the relevant tags in its tag array. On a hit, it cancels the command associated with this address. On a miss, it allocates a free slot, setting Valid=1, Ready=0 and updates the outgoing command to direct the data it is fetching to this slot. In either case, a slot number is associated with the address. Storage allocation and tracking **2714**/**2724** inserts the reference in the reference queue. Storage allocation and tracking **2714**/**2724** sets Active=1 and updates Last Reference to the position of the reference in the reference queue. This is the value of the reference queue's insertion pointer at the time of insertion. This process converts the generated addresses into the slot numbers that represent the data. From this point forward, the streaming engine need not track addresses directly.

[0159] To maximize reuse and minimize stalls, streaming engine **2700** allocates slots in the following order: the slot one after the most recent allocation if available in FIFO order; the lowest number available slot, if any; and if no slot available, stall and iterate these two steps until allocation succeeds. This will tend to allocate slots in FIFO order, but avoids stalling if a particular reuse pattern works against that order.

[0160] Reference queue **2715**/**2725** stores the sequence of references generated by the corresponding address generator **2711**/**2721**. This information drives the data formatting network so that it can present data to the CPU in the correct order. Each entry in reference queue **2715**/**2725** contains the information necessary to read data out of the data store and align it for the CPU. Reference queue **2715**/**2725** maintains the following information listed in Table 7 in each slot:

TABLE-US-00007 TABLE 7 Data Slot Low Slot number for the lower half of data associated with aout0 Data Slot High Slot number for the upper half of data associated with aout1 Rotation Number of bytes to rotate data to align next element with lane 0 Length Number of valid bytes in this reference

[0161] Storage allocation and tracking **2714**/**2724** inserts references in reference queue **2715**/**2725** as address generator **2711**/**2721** generates new addresses. Storage allocation and tracking **2714**/**2724** removes references from reference queue **2715**/**2725** when the data becomes available and there is room in the stream holding registers. As storage allocation and tracking **2714**/**2724** removes slot references from reference queue **2715**/**2725** and formats data, it checks whether the references represent the last reference to the corresponding slots. Storage allocation and tracking **2714**/**2724** compares reference queue **2715**/**2725** removal pointer against the slot's recorded Last Reference. If they match, then storage allocation and tracking **2714**/**2724** marks the slot inactive once it's done with the data.

[0162] Streaming engine **2700** has data storage **2716**/**2726** for an arbitrary number of elements. Deep buffering allows the streaming engine to fetch far ahead in the stream, hiding memory system latency. The right amount of buffering might vary from product generation to generation. In the current preferred embodiment streaming engine **2700** dedicates 32 slots to each stream. Each slot holds 64 bytes of data.

[0163] Butterfly network **2717**/**2727** consists of a 7 stage butterfly network. Butterfly network **2717**/**2727** receives 128 bytes of input and generates 64 bytes of output. The first stage of the butterfly is actually a half-stage. It collects bytes from both slots that match a non-aligned fetch and merges them into a single, rotated 64-byte array. The remaining 6 stages form a standard butterfly network. Butterfly network **2717**/**2727** performs the following operations: rotates the next element down to byte lane 0; promotes data types by one power of 2, if requested; swaps real and imaginary components of complex numbers, if requested; converts big endian to little endian if the CPU is presently in big endian mode. The user specifies element size, type promotion and real/imaginary swap as part of the stream's parameters.

[0164] Streaming engine **2700** attempts to fetch and format data ahead of the CPU's demand for it, so that it can maintain full throughput. Holding registers **2718**/**2728** provide a small amount of buffering so that the process remains fully pipelined. Holding registers **2718**/**2728** are not directly architecturally visible, except for the fact that streaming engine **2700** provides full throughput.

[0165] The two streams **2710**/**2720** share a pair of independent L2 interfaces **2730**: L2 Interface A (IFA) **2733** and L2 Interface B (IFB) **2734**. Each L2 interface provides 512 bits/cycle throughput direct to the L2 controller for an aggregate bandwidth of 1024 bits/cycle. The L2 interfaces use the credit-based multicore bus architecture (MBA) protocol. The L2 controller assigns each interface its own pool of command credits. The pool should have sufficient credits so that each interface can send sufficient commands to achieve full read-return bandwidth when reading L2 RAM, L2 cache and MSMC RAM.

[0166] To maximize performance, both streams can use both L2 interfaces, allowing a single stream to send a peak command rate of 2 commands/cycle. Each interface prefers one stream over the other, but this preference changes dynamically from request to request. IFA **2733** and IFB **2734** always prefer opposite streams, when IFA **2733** prefers Stream 0, IFB **2734** prefers Stream 1 and vice versa.

[0167] Arbiter **2731**/**2732** ahead of each interface **2733**/**2734** applies the following basic protocol on every cycle it has credits available. Arbiter **2731**/**2732** checks if the preferred stream has a command ready to send. If so, arbiter **2731**/**2732** chooses that command. Arbiter **2731**/**2732** next checks if an alternate stream has at least two commands ready to send, or one command and no credits. If so, arbiter **2731**/**2732** pulls a command from the alternate stream. If either interface issues a command, the notion of preferred and alternate streams swap for the next request. Using this simple algorithm, the two interfaces dispatch requests as quickly as possible while retaining fairness between the two streams. The first rule ensures that each stream can send a request on every cycle that has available credits. The second rule provides a mechanism for one stream to borrow the other's interface when the second interface is idle. The third rule spreads the bandwidth demand for each stream across both interfaces, ensuring neither interface becomes a bottleneck by itself.

[0168] Coarse Grain Rotator **2735**/**2736** enables streaming engine **2700** to support a transposed matrix addressing mode. In this mode, streaming engine **2700** interchanges the two innermost dimensions of its multidimensional loop. This accesses an array column-wise rather than row-wise. Rotator **2735**/**2736** is not architecturally visible, except as enabling this transposed access mode.

[0169] The stream definition template provides the full structure of a stream that contains data. The iteration counts and dimensions provide most of the structure, while the various flags provide the rest of the details. For all data-containing streams, the streaming engine defines a single stream template. All stream types it supports fit this template. The numbers above each field indicate byte numbers within a 256-bit vector. The streaming engine defines a four-level loop nest for addressing elements within the stream. Most of the fields in the stream template map directly to the parameters in that algorithm. FIG. **28** illustrates stream template register **2800**. The numbers above the fields are byte numbers within a 256-bit vector. Table 8 shows the stream field definitions of a stream template, which includes ICNT0 field (**2810**), ICNT1 field (**2811**), ICNT2 field (**2812**), ICNT3 field (**2813**), FLAGS field (**2831**), DIM1 field (**2821**), DIM2 field (**2822**), and DIM3 field (**2823**).

TABLE-US-00008 TABLE 8 Field Size Name Description Bits ICNT0 Iteration count for loop 0 (innermost) 32 ICNT1 Iteration count for loop 1 32 ICNT2 Iteration count for loop 2 32 ICNT3 Iteration count for loop 3 (outermost) 8 DIM1 Signed dimension for loop 1 32 DIM2 Signed dimension for loop 2 32 DIM3 Signed dimension for loop 3 32 FLAGS Stream modifier flags 24

[0170] In the current example DIM0 is always equal to is ELEM_BYTES defining physically contiguous data. The stream template includes mostly 32-bit fields. The stream template limits ICNT3 to 8 bits and the FLAGS field to 24 bits. Streaming engine **2700** interprets all iteration counts as unsigned integers and all dimensions as unscaled signed integers. The template above fully specifies the type of elements, length and dimensions of the stream. The stream instructions separately specify a start address. This would typically be by specification of a scalar register in scalar register file **211** which stores this start address. This allows a program to open multiple streams using the same template.

[0171] FIG. **29** illustrates sub-field definitions of the flags field **2900**, which may correspond to field **2831** of FIG. **28**. As shown in FIG. **28** the flags field **2900** is 3 bytes or 24 bits. FIG. **29** shows bit numbers of the fields. Table 9 shows the definition of these fields, which includes ELTYPE field (**2901**), DIR field (**2902**), TRANSPOSE field (**2903**), PROMOTE field (**2904**), and THROTTLE field (**2905**).

TABLE-US-00009 TABLE 9 Size Field Name Description Bits ELTYPE Type of data element 4 DIR Stream direction 1 0 forward direction 1 reverse direction TRANSPOSE Two dimensional transpose mode 3 PROMOTE Promotion mode 2 THROTTLE Fetch ahead throttle mode 2

[0172] The Element Type (ELTYPE) field defines the data type of the elements in the stream. The coding of the four bits of this field is defined as shown in Table 10.

TABLE-US-00010 TABLE 10 Total Sub-element Element Real- ELTYPE Size Bits Size Bits Complex Bytes/Element 0000 8 8 real 1 0001 16 16 real 2 0010 32 32 real 4 0011 64 64 real 8 0100 reserved 0101 reserved 0110 reserved 0111 reserved 1000 8 16 complex 2 no swap 1001 16 32 complex 4 no swap 1010 32 64 complex 8 no swap 1011 64 128 complex 16 no swap 1100 8 16 complex 2 swapped 1101 16 32 complex 4 swapped 1110 32 64 complex 8 swapped 1111 64 128 complex 16 swapped

[0173] Sub-Element Size determines the type for purposes of type promotion and vector lane width. For example, 16-bit sub-elements get promoted to 32-bit sub-elements when a stream requests type promotion. The vector lane width matters when the DSP CPU operates in big endian mode, as it always lays out vectors in little endian order.

[0174] Total Element Size determines the minimal granularity of the stream. In the stream addressing model, it determines the number of bytes the stream fetches for each iteration of the innermost loop. Streams always read whole elements, either in increasing or decreasing order. Therefore, the innermost dimension of a stream spans ICNT0×total-element-size bytes.

[0175] Real-Complex Type determines whether the streaming engine treats each element as a real number or two parts (real/imaginary or magnitude/angle) of a complex number. This field also specifies whether to swap the two parts of complex numbers. Complex types have a total element size that is twice their sub-element size. Otherwise, the sub-element size equals total element size.

[0176] The TRANSPOSE field determines whether the streaming engine accesses the stream in a transposed order. The transposed order exchanges the inner two addressing levels. The TRANSPOSE field also indicated the granularity it transposes the stream. The coding of the four bits of this field is defined as shown in Table 11.

TABLE-US-00011 TABLE 11 Transpose Granule Stream Advance TRANSPOSE Duplication Bytes Rate 0000 both disabled 64 bytes 0001 reserved 0010 reserved 0011 transpose 4 16 rows 0100 transpose 8   8 rows 0101 transpose 16   4 rows 0110 transpose 32   2 rows 0111 reserved 1000 duplicate 1   1 byte 1001 duplicate 2   2 bytes 1010 duplicate 4   4 bytes 1011 duplicate 8   8 bytes 1100 duplicate 16 16 bytes 1101 duplicate 32 32 bytes 1110 reserved 1111 reserved

[0177] Streaming engine **2700** actually transposes at a different granularity than the element size. This allows programs to fetch multiple columns of elements from each row. The transpose granularity must be no smaller than the element size.

[0178] The PROMOTE field controls whether the streaming engine promotes sub-elements in the stream and the type of promotion. When enabled, streaming engine **2700** promotes types by a single power-of-2 size. The coding of the two bits of this field is defined as shown in Table 12.

TABLE-US-00012 TABLE 12 PROMOTE Description 00 no promotion 01 unsigned integer promotion, zero extend 10 signed integer promotion, sign extend 11 floating point promotion

[0179] When the stream specifies No promotion, each sub-element occupies a vector lane equal in width to the size specified by ELTYPE. Otherwise, each sub-element occupies a vector lane twice as large. When PROMOTE is 00, the streaming engine fetches half as much data from memory to satisfy the same number of stream fetches.

[0180] Promotion modes 01b and 10b treat the incoming sub-elements as unsigned and signed integers, respectively. For unsigned integers, the streaming engine promotes by filling the new bits with zeros. For signed integers the streaming engine promotes by filling the new bits with copies of the sign bit. Positive signed integers have a most significant bit equal to 0. On promotion of positive signed integers, the new bits are zero filled. Negative signed integers have a most significant bit equal to 1. On promotion of negative signed integers, the new bits are 1 filled.

[0181] Promotion mode 11b treats the incoming sub-elements as floating point numbers. Floating point promotion treats each sub-element as a floating point type. The streaming engine supports two floating point promotions: short float (16-bit) to single precision float (32-bit); single precision float (32-bit) to double precision float (64-bit).

[0182] The THROTTLE field controls how aggressively the streaming engine fetches ahead of the CPU. The coding of the two bits of this field is defined as shown in Table 13.

TABLE-US-00013 TABLE 13 THROTTLE Description 00 Minimum throttling, maximum fetch ahead 01 Less throttling, more fetch ahead 10 More throttling, less fetch ahead 11 Maximum throttling, minimum fetch ahead

THROTTLE does not change the meaning of the stream, and serves only as a hint. The streaming engine may ignore this field. Programs should not rely on the specific throttle behavior for program correctness, because the architecture does not specify the precise throttle behavior. THROTTLE allows programmers to provide hints to the hardware about the program's own behavior. By default, the streaming engine attempts to get as far ahead of the CPU as it can to hide as much latency as possible, while providing full stream throughput to the CPU. While several key applications need this level of throughput, it can lead to bad system level behavior for others. For example, the streaming engine discards all fetched data across context switches. Therefore, aggressive fetch-ahead can lead to wasted bandwidth in a system with large numbers of context switches. Aggressive fetch-ahead only makes sense in those systems if the CPU consumes data very quickly.

[0183] The DSP CPU exposes the streaming engine to programs through a small number of instructions and specialized registers. A STROPEN instruction opens a stream. The STROPEN command specifies a stream number indicating opening stream 0 or stream 1. The STROPEN specifies a stream template register which stores the stream template as described above. The arguments of the STROPEN instruction are listed in Table 14.

TABLE-US-00014 TABLE 14 Argument Description Stream Start Scaler register storing stream start Address Register address Steam Number Stream 0 or Stream 1 Stream Template Vector register storing stream Register template data

[0184] The stream start address register is preferably a scalar register in general scalar register file **211**. The STROPEN instruction specifies stream 0 or stream 1 by its opcode. The stream template register is preferably a vector register in general vector register file **221**. If the specified stream is active the STROPEN instruction closes the prior stream and replaces the stream with the specified stream.

[0185] A STRCLOSE instruction closes a stream. The STRCLOSE command specifies the stream number of the stream to be closed.

[0186] A STRSAVE instruction captures sufficient state information of a specified stream to restart that stream in the future. A STRRSTR instruction restores a previously saved stream. A STRSAVE instruction does not save any of the data of the stream. A STRSAVE instruction saves only metadata. The stream re-fetches data in response to a STRRSTR instruction.

[0187] Streaming engine is in one of three states: Inactive; Active; or Frozen. When inactive the streaming engine does nothing. Any attempt to fetch data from an inactive streaming engine is an error. Until the program opens a stream, the streaming engine is inactive. After the program consumes all the elements in the stream or the program closes the stream, the streaming engine also becomes inactive. Programs which use streams explicitly activate and inactivate the streaming engine. The operating environment manages streams across context-switch boundaries via the streaming engine's implicit freeze behavior, coupled with its own explicit save and restore actions.

[0188] Active streaming engines have a stream associated with them. Programs can fetch new stream elements from active streaming engines. Streaming engines remain active until one of the following. When the stream fetches the last element from the stream, it becomes inactive. When program explicitly closes the stream, it becomes inactive. When CPU responds to an interrupt or exception, the streaming engine freezes. Frozen streaming engines capture all the state necessary to resume the stream where it was when the streaming engine froze. The streaming engines freeze in response to interrupts and exceptions. This combines with special instructions to save and restore the frozen stream context, so that operating environments can cleanly switch contexts. Frozen streams reactivate when the CPU returns to the interrupted context.

[0189] Programs access stream data via holding register **2718** for stream 0 and holding register **2728** for stream 1. These registers are outside the other register files. These registers represent the head of stream for respective streams 0 and 1. Dedicated bit codings of the src1 field **1305** and the src2/cst field **1304** in the corresponding program instruction read stream data and the control advance of the stream. Table 15 shows an exemplary coding of the source operand fields **1305** and **1304** in accordance with the preferred embodiment of this invention.

TABLE-US-00015 TABLE 15 Bit Coding src1 field 1305 or Designated Register src2/cst field 1304 Source 00000 Global Register 0 First 00001 Global Register 1 subset 00010 Global Register 2 00011 Global Register 3 00100 Global Register 4 00101 Global Register 5 00110 Global Register 6 00111 Global Register 7 01000 Global Register 8 01001 Global Register 9 01010 Global Register 10 01011 Global Register 11 01100 Global Register 12 01101 Global Register 13 01110 Global Register 14 01111 Global Register 15 10000 Local Register 0 Second 10001 Local Register 1 Subset 10010 Local Register 2 10011 Local Register 3 10100 Local Register 4 10101 Local Register 5 10110 Local Register 6 10111 Local Register 7 11000 Reserved Reserved 11001 Reserved 11010 Reserved 11011 Reserved 11100 Stream 0 read Stream 0 11101 Stream 0 read/advance 11110 Stream 1 read Stream 1 11111 Stream 1 read/advance

[0190] Bit codings 00000 to 01111 (first subset) specify a corresponding register in global vector register file **231**. Note that only vector data path side B includes streaming engines. For an instruction having a src1 field **1305** or src2/cst field **1304** bit coding in this first subset, instruction decoder **113** supplies an input operand for the corresponding functional unit from the specified register number in global vector register file **231**. Bit codings 10000 to 10111 (second subset) specify a corresponding register in the corresponding vector local register file. For instructions directed to L2 unit **241** or S2 unit **242** that local register file is L2/S2 local register file **232**. For instructions directed to M2 unit **243**, N2 unit **244** or C unit **245** that local register file is M2/N2/C local register file **233**. For an instruction having a src1 field **1305** or src2/cst field **1304** bit coding in this second subset, instruction decoder **113** supplies an input operand for the corresponding functional unit from the specified register number in the corresponding local register file, in this embodiment L2/S2 local register file **232** or M2/N2/C local register file **233**.

[0191] Bit codings 11000 to 11011 are reserved in this embodiment. These coding are not used. Instruction decoder **113** may ignore these bit codings or may generate an error. The compiler will not generate these codings.

[0192] Bit codings 11100 and 11101 are directed to stream 0. Bit codings 11110 and 11111 are directed to stream 1. Bit coding 11100 is a read of stream 0. Upon detection of an instruction having a src1 field **1305** or the src2/cst field **1304** with bit coding of 11100, instruction decoder **113** supplies the corresponding operand input of the functional unit of that instruction with the data stored in holding register **2718**. Holding register **2718** holds elements in the specified data stream 0 as disclosed above in conjunction with FIGS. **20** and **21**. This supply of data is analogous to supply of data from a data register. Similarly, bit coding 11110 is a read of stream 1. Upon detection of an instruction having a src1 field **1305** or the src2/cst field **1304** with bit coding of 11110, instruction decoder **113** supplies the corresponding operand input of the functional unit of that instruction with the data stored in holding register **2728**.

[0193] The stream reading bit codings 11100 and 11110 are treated similarly to register specifying bit codings. Thus more than one functional unit may receive input from the same stream holding register **2718** or **2728**. A single instruction may specify the same stream holding register **2718** of **2728** for both input operands. An instruction may specify one input operand from holding register **2718** and the other input operand from holding register **2728** and in either order.

[0194] Bit codings 11101 and 11111 trigger read/advance stream operations. Bit coding 11101 is a read/advance of stream 0. Upon detection of an instruction having a src1 field **1305** or the src2/cst field **1304** with bit coding of 11101, instruction decoder **113** supplies the corresponding operand input of the functional unit of that instruction with the data stored in holding register **2718**. Streaming engine **2700** then advances stream 0 to the next set of elements in the specified data stream 0 as disclosed above in conjunction with FIGS. **20** and **21**. Thus, holding register **2718** will store the next elements in stream 0. Similarly, bit coding 11111 is a read/advance of stream 1. Upon detection of an instruction having a src1 field **1305** or the src2/cst field **1304** with bit coding of 11111, instruction decoder **113** supplies the corresponding operand input of the functional unit of that instruction with the data stored in holding register **2728** and then triggers streaming engine **2700** to advance stream 1 to store the next data elements in holding register **2728**. The data input operation takes place in the same manner as the read operations of bit codings 11100 and 11110. The read/advance operand bit codings add the advance to the defined next stream data elements as described above.

[0195] As previously described the same stream holding register data may be supplied to more than one input of a functional unit and to more than one functional unit. Thus it is possible to code instructions in the same execute packet where one of these inputs is a read coding and another input for the same stream is a read/advance coding. In that case the corresponding stream is advanced. Thus if any stream 0 or stream 1 operand bit coding is a read/advance bit coding the stream advances, whether or not any other stream 0 operand bit codings are read or read/advance.

[0196] In accordance with the nature of streams, stream data is read only. Thus the bit coding of Table 15 cannot be used for dst field **1303**. Instruction decoder **113** may ignore these bit codings in dst field **1303** or may generate an error. The compiler will not generate these codings.

[0197] FIG. **30** is a partial schematic diagram **3000** illustrating the stream input operand coding described above. FIG. **30** illustrates decoding src1 field **1305** of one instruction of a corresponding src1 input of functional unit **3020**. These same circuits are duplicated for src2/cst field **1304** and the src2 input of functional unit **3020**. In addition, these circuits are duplicated for each instruction within an execute packet that can be dispatched simultaneously.

[0198] Instruction decoder **113** receives bits 13 to 17 comprising src1 field **1305** of an instruction. The opcode field opcode field (bits 4 to 12 for all instructions and additionally bits 28 to 31 for unconditional instructions) unambiguously specifies a corresponding functional unit **3020**. In this embodiment functional unit **3020** could be L2 unit **241**, S2 unit **242**, M2 unit **243**, N2 unit **244**, or C unit **245**. The relevant part of instruction decoder **113** illustrated in FIG. **30** decodes src1 bit field **1305**. Sub-decoder **3011** determines whether src1 bit field **1305** is in the range from 00000 to 01111. If this is the case, sub-decoder **3011** supplies a corresponding register number to global vector register file **231**. In this example this register field is the four least significant bits of src1 bit field **1305**. Global vector register file **231** recalls data stored in the register corresponding to this register number and supplies this data to the src1 input of functional unit **3020**. This decoding is generally known in the art.

[0199] Sub-decoder **3012** determines whether src1 bit field **1305** is in the range from 10000 to 10111. If this is the case, sub-decoder **3012** supplies a corresponding register number to the corresponding local vector register file. If the instruction is directed to L2 unit **241** or S2 unit **242**, the corresponding local vector register file is local vector register field **232**. If the instruction is directed to M2 unit **243**, N2 unit **244** or C unit **245**, the corresponding local vector register file is local vector register field **233**. In this example this register field is the three least significant bits of src1 bit field **1305**. Local vector register file **231** recalls data stored in the register corresponding to this register number and supplies this data to the src1 input of functional unit **3020**. The corresponding local vector register file **232/233** recalls data stored in the register corresponding to this register number and supplies this data to the src1 input of functional unit **3020**. This decoding is generally known in the art.

[0200] Sub-decoder **3013** determines whether src1 bit field **1305** is 11100. If this is the case, sub-decoder **3013** supplies a stream 0 read signal to streaming engine **2700**. Streaming engine **2700** then supplies stream 0 data stored in holding register **2718** to the src1 input of functional unit **3020**.

[0201] Sub-decoder **3014** determines whether src1 bit field **1305** is 11101. If this is the case, sub-decoder **3014** supplies a stream 0 read signal to streaming engine **2700**. Streaming engine **2700** then supplies stream 0 data stored in holding register **2718** to the src1 input of functional unit **3020**. Sub-decoder **3014** also supplies an advance signal to stream 0. As previously described, streaming engine **2700** advances to store the next sequential data elements of stream 0 in holding register **2718**.

[0202] Sub-decoder **3015** determines whether src1 bit field **1305** is 11110. If this is the case, sub-decoder **3015** supplies a stream 1 read signal to streaming engine **2700**. Streaming engine **2700** then supplies stream 1 data stored in holding register **2728** to the src1 input of functional unit **3020**.

[0203] Sub-decoder **3016** determines whether src1 bit field **1305** is 11111. If this is the case, sub-decoder **3016** supplies a stream 1 read signal to streaming engine **2700**. Streaming engine **2700** then supplies stream 1 data stored in holding register **2728** to the src1 input of functional unit **3020**. Sub-decoder **3014** also supplies an advance signal to stream 1. As previously described, streaming engine **2700** advances to store the next sequential data elements of stream 2 in holding register **2728**.

[0204] Similar circuits are used to select data supplied to scr2 input of functional unit **3002** in response to the bit coding of src2/cst field **1304**. The src2 input of functional unit **3020** may be supplied with a constant input in a manner described above.

[0205] The exact number of instruction bits devoted to operand specification and the number of data registers and streams are design choices. Those skilled in the art would realize that other number selections that described in the application are feasible. In particular, the specification of a single global vector register file and omission of local vector register files is feasible. This invention employs a bit coding of an input operand selection field to designate a stream read and another bit coding to designate a stream read and advancing the stream.

[0206] FIG. **31** illustrates alternate streaming engine **3100**. Streaming engine **3100** is similar to streaming engine **2700** illustrated in FIG. **27**. Streaming engine **3100** includes address generators **2211**/**2221**, μTLBs **2212**/**2222**, command queues **2213**/**2223**, storage allocation and tracking blocks **2214**/**2224**, reference queues **2215**/**2225**, arbitrators **2231**/**2232**, L2 interfaces **2233**/**2234**, coarse rotators **2235**/**2236**, data storage **2216**/**2226** and butterfly networks **2217** and **2227** as previously described in conjunction with FIG. **27**. Holding register **2718** is replaced with two holding registers **3118** and **3119**, designated SE0L and SE0H. Each of holding registers **3118** and **3119** are the vector width of 512 bits. Holding register **3119** stores the next vector of stream 0 data following holding register **3118**. Similarly, holding register **2728** is replaced with like sized holding registers **3128** and **3129**, designated SE1L and SE1H. Holding register **3129** stores the next vector of stream 1 data following holding register **3128**.

[0207] Streaming engine **3100** permits differing stream access than streaming engine **2700**. Streaming engine **3100** permits direct access to SE0H and SE1 H data in a manner described below. Streaming engine **3100** permits access to a stream 0 double vector of data corresponding to combined SE0L and SE0H data (SE0). Streaming engine **3100** permits access to a stream 1 double vector of data corresponding to combined SE1L and SE1H data (SE1). This structure of holding registers **3118**/**3119** and **3128**/**3129** supports these operations a follows. Table 16 lists these stream access operations.

TABLE-US-00016 TABLE 16 Bit Coding src1 field 1305 or Designated src2/cst field Register 1304 Source Action First Subset Global Register Recall Global Register Data Number at Register Number Second Subset Local Register Recall Local Register Data at Number Register Number Stream 0 SE0 Recall Stream 0 Data from Registers 3118/3119 SE0++ Recall Stream 0 Data from Registers 3118/3119, Advance Stream 0 by Double Vector SE0L Recall Stream 0 Data from Register 3118 SE0L++ Recall Stream 0 Data from Register 3118, Advance Stream 0 by Double Vector SE0H Recall Stream 0 Data from Register 3119 SE0H++ Recall Stream 0 Data from Register 3119, Advance Stream 0 by Double Vector Stream 1 SE1 Recall Stream 1 Data from Registers 3128/2629 SE1++ Recall Stream 1 Data from Registers 3128/2629, Advance Stream 1 by Double Vector SE1L Recall Stream 1 Data from Register 3128 SE1L++ Recall Stream 1 Data from Register 3128, Advance Stream 1 by Double Vector SE1H Recall Stream 1 Data from Register 3129 SE1H++ Recall Stream 1 Data from Register 3129, Advance Stream 1 by Double Vector

[0208] Table 16 lists first subset access to global vector register file **231** in the same manner as listed in Table 15. Table 16 lists second subject access to the corresponding local vector register file **242**/**423** in the same manner as listed in FIG. **15**. Table 16 lists an access designated SE0. Upon decoding this source register designation, streaming engine **3100** supplies data from holding register **3118** to the corresponding functional unit and data from holding register **3119** to a paired functional unit thus supporting double vector operation. Table 16 lists an access designated SE0++. Upon decoding this source register designation, streaming engine **3100** supplies double vector data from holding registers **3118** and **3119** to paired functional units supporting double vector operation and advances stream 0 by two vectors (1024 bits). Table 16 lists similar accesses designated SE1 and SE1++ having similar double vector data supply from holding registers **3128** and **3129**.

[0209] Table 16 lists four different single vector accesses. The accesses designated SE0L and SE0L++ correspond to the stream 0 read and read/advance operations listed in Table 15 which supply data from holding register **3118**. The accesses designated SE1 L and SE1 L++ correspond to the stream 1 read and read/advance operations listed in Table 15 which supply data from holding register **3128**. The access designated SE0H supplies data from holding register **3119**. As noted above, holding register **3119** stores the next vector of stream 0 data following holding register **3118**. The access designated SE0H++ supplies data from holding register **3119** and advances stream 0 by two vectors (1024 bits). The access designated SE1 H supplies data from holding register **3129**. As noted above, holding register **3129** stores the next vector of stream 1 data following holding register **3128**. The access designated S1 H++ supplies data from holding register **3129** via output OutL and advances stream 1 by two vectors (1024 bits).

[0210] It can be readily seen that a 5-bit src1 field **1305** cannot specify one operation from a set including: fetch from one of 16 global vector registers; fetch from one of 8 local vector registers; and above described 12 stream codings. This number of selections can be accommodated by increasing the number of bits in the src1 field **1305** to 6 bits. The number of selections can be reduced by restricting the number and identity of registers accessible in the global vector register file, the local vector register file or both. An alternative embodiment decodes src1 field **1305** differently for double vector operations than for vector operations. Table 17 lists the decoding of src1 field **1305** for vector operations in this alternative embodiment.

TABLE-US-00017 TABLE 17 Bit Coding src1 field 1305 or Designated Register src2/cst field 1304 Source 00000 Global Register 0 First 00001 Global Register 1 subset 00010 Global Register 2 00011 Global Register 3 00100 Global Register 4 00101 Global Register 5 00110 Global Register 6 00111 Global Register 7 01000 Global Register 8 01001 Global Register 9 01010 Global Register 10 01011 Global Register 11 01100 Global Register 12 01101 Global Register 13 01110 Global Register 14 01111 Global Register 15 10000 Local Register 0 Second 10001 Local Register 1 Subset 10010 Local Register 2 10011 Local Register 3 10100 Local Register 4 10101 Local Register 5 10110 Local Register 6 10111 Local Register 7 11000 SE0L read Stream 0 11001 SE0L++ read/advance 11010 SE0H read 11011 SE0H++ read/advance 11100 SE1L read Stream 1 11101 SE1L++ read/advance 11110 SE1H read 11111 SE1H++ read/advance

[0211] Table 18 lists the decoding of src1 field **1305** for double vector operations.

TABLE-US-00018 TABLE 18 Bit Coding src1 field 1305 or Designated Register src2/cst field 1304 Source 00000 Global Register 0 First 00001 Global Register 1 subset 00010 Global Register 2 00011 Global Register 3 00100 Global Register 4 00101 Global Register 5 00110 Global Register 6 00111 Global Register 7 01000 Global Register 8 01001 Global Register 9 01010 Global Register 10 01011 Global Register 11 01100 Global Register 12 01101 Global Register 13 01110 Global Register 14 01111 Global Register 15 10000 Local Register 0 Second 10001 Local Register 1 Subset 10010 Local Register 2 10011 Local Register 3 10100 Local Register 4 10101 Local Register 5 10110 Local Register 6 10111 Local Register 7 11000 Reserved Reserved 11001 Reserved 11010 Reserved 11011 Reserved 11100 SE0 double vector Stream 0 read 11101 SE0++ double vector read/ advance 11110 SE1 double vector Stream 1 read 11111 SE1++ double vector read/advance

[0212] For both codings of Tables 17 and 18 in this alternative embodiment, the decoding of the first subset and the second subset are identical to Table 15. This alternative does not require any change in the instruction coding of instructions **1300**. The opcode field **1306** (bits 4 to 12) for all instructions (and additionally bits 28 to 31 for unconditional instructions) must specify the type of instruction and unambiguously distinguish all double vector instructions from all vector instructions. Thus using this alternative embodiment merely requires making instruction decoder **113** operand field decoding conditional on the vector/double vector coding for streaming data accesses.

[0213] FIG. **32** is a partial schematic diagram **3200** similar to FIG. **25** illustrating the stream input operand coding of this alternative embodiment. FIG. **32** illustrates decoding src1 field **1305** of one instruction of a corresponding src1 input of functional unit **3230** and paired functional unit **3240**. These same circuits are duplicated for src2/cst field **1304** and the src2 input of functional units

3230 and **3240**. In addition, these circuits are duplicated for each instruction within an execute packet that can be dispatched simultaneously.

[0214] Instruction decoder **113** receives bits 13 to 17 comprising src1 field **1305** of an instruction. The opcode field opcode field (bits 4 to 12 for all instructions and additionally bits 28 to 31 for unconditional instructions) unambiguously specifies a corresponding functional unit **3230**. If the instruction is a double vector operation this field also unambiguously specifies the paired functional unit **3240**. In this embodiment functional units **3230** and **3240** could be L2 unit **241**, S2 unit **242**, M2 unit **243**, N2 unit **244** or C unit **245**. The relevant part of instruction decoder **113** illustrated in FIG. **32** decodes src1 bit field **1305**. As illustrated in FIG. **32**, instruction decoder **113** generates a vector/double vector (D/DV) signal indicating whether the instruction being decoded controls a normal vector operation or a special double vector operation. This vector/double vector signal is supplied to sub-decoders **3214**, **3215**, **3216**, **3217**, **3218**, **3219**, **3220**, **3221**, **3222**, **3223** and **3224** for control in the manner described below.

[0215] Sub-decoder **3211** determines whether src1 bit field **1305** is in first subset specifying global vector register file **231**. If this is the case, sub-decoder **3211** supplies a corresponding register number to global vector register file **231**. Global vector register file **231** recalls data stored in the register corresponding to this register number and supplies this data to the src1 input of functional unit **3230**. This decoding is generally known in the art. Note that the decoding this first subset is not dependent upon the vector/double vector signal.

[0216] Sub-decoder **3212** determines whether src1 bit field **1305** is in the second subset specifying a corresponding local vector register file **232** or **233**. If this is the case, sub-decoder **2512** supplies a corresponding register number to the corresponding local vector register file. If the instruction is directed to L2 unit **241** or S2 unit **242**, the corresponding local vector register file is local vector register field **232**. If the instruction is directed to M2 unit **243**, N2 unit **244** or C unit **245**, the corresponding local vector register file is local vector register field **233**. Local vector register file **232**/**233** recalls data stored in the register corresponding to this register number and supplies this data to the src1 input of functional unit **3230**. The corresponding local vector register file **232**/**233** recalls data stored in the register corresponding to this register number and supplies this data to the src1 input of functional unit **2520**. This decoding is generally known in the art. Note that the decoding this second subset is not dependent upon the vector/double vector signal.

[0217] Sub-decoder **3213** is active if the vector/double vector signal indicates a double vector operation. Sub-decoder **3213** determines whether src1 bit field **1305** specifies SE0. As listed in Table 18, this is a bit coding of “11100.” If this is the case, sub-decoder **3213** supplies a SE0 read signal to streaming engine **3100**. Streaming engine **3100** then supplies stream 0 data stored in holding register **3118** via output OutL to the src1 input of functional unit **3230** and stream 0 data stored in holding register **3119** via output OutH to src1 input of paired functional unit **3240**. This is a double vector operation employing paired functional units.

[0218] Sub-decoder **3214** is active if the vector/double vector signal indicates a double vector operation. Sub-decoder **3214** determines whether src1 bit field **1305** specified SE0++. As listed in Table 18, this is a bit coding of “11101.” If this is the case, sub-decoder **3213** supplies a SE0 read signal to streaming engine **3100**. Streaming engine **3100** then supplies stream 0 data stored in holding register **3118** via output OutL to the src1 input of functional unit **3230** and stream 0 data stored in holding register **3119** via output OutH to src1 input of paired functional unit **3240**. This is a double vector operation employing paired functional units. Sub-decoder **3214** also supplies an advance signal to stream 0. Streaming engine **2200** advances a double vector amount (512 bits) and stores the next stream 0 data in holding registers **3118** and **3119**.

[0219] Sub-decoder **3215** is active if the vector/double vector signal indicates a vector operation. Sub-decoder **3215** determines whether src1 bit field **1305** specifies SE0L. As listed in Table 17, this is a bit coding of “11000.” If this is the case, sub-decoder **3215** supplies a SE0L read signal to streaming engine **3100**. Streaming engine **3100** then supplies stream 0 data stored in holding register **3128** via output OutL to the src1 input of functional unit **3230**.

[0220] Sub-decoder **3216** is active if the vector/double vector signal indicates a vector operation. Sub-decoder **3216** determines whether src1 bit field **1305** specifies SE0L++. As listed in Table 17, this is a bit coding of “11001.” If this is the case, sub-decoder **3215** supplies a SE0L read signal to streaming engine **3100**. Streaming engine **3100** then supplies stream 0 data stored in holding register **3128** via output OutL to the src1 input of functional unit **3230**. Sub-decoder **3215** also supplies an advance signal to stream 0. Streaming engine **2200** advances a double vector amount (512 bits) and stores the next stream 0 data in holding registers **3118** and **3119**. In the preferred embodiment all stream advances are in increments of a double vector (512 bits) even for single vector reads.

[0221] Sub-decoder **3217** is active if the vector/double vector signal indicates a vector operation. Sub-decoder **3217** determines whether src1 bit field **1305** specifies SE0H. As listed in Table 17, this is a bit coding of “11010.” If this is the case, sub-decoder **3217** supplies a SE0H read signal to streaming engine **3100**. Streaming engine **3100** then supplies stream 0 data stored in holding register **3129** via output OutL to the src1 input of functional unit **3230**.

[0222] Sub-decoder **3218** is active if the vector/double vector signal indicates a vector operation. Sub-decoder **3218** determines whether src1 bit field **1305** specifies SE0H++. As listed in Table 17, this is a bit coding of “11011.” If this is the case, sub-decoder **3218** supplies a SE0H read signal to streaming engine **3100**. Streaming engine **3100** then supplies stream 0 data stored in holding register **3129** via output OutL to the src1 input of functional unit **3230**. Sub-decoder **3218** also supplies an advance signal to stream 0. Streaming engine **2200** advances a double vector amount (512 bits) and stores the next stream 0 data in holding registers **3118** and **3119**. In the preferred embodiment all stream advances are in increments of a double vector (512 bits) even for single vector reads.

[0223] Sub-decoder **3219** is active if the vector/double vector signal indicates a double vector operation. Sub-decoder **3219** determines whether src1 bit field **1305** specifies SE1. As listed in Table 18, this is a bit coding of “11110.” If this is the case, sub-decoder **3219** supplies a SE1 read signal to streaming engine **3100**. Streaming engine **3100** then supplies stream 1 data stored in holding register **3128** via output OutL to the src1 input of functional unit **3230** and stream 1 data stored in holding register **3129** via output OutH to src1 input of paired functional unit **3240**. This is a double vector operation employing paired functional units.

[0224] Sub-decoder **3220** is active if the vector/double vector signal indicates a double vector operation. As listed in Table 18, this is a bit coding of “11111.” Sub-decoder **3220** determines whether src1 bit field **1305** specified SE1++. If this is the case, sub-decoder **3220** supplies a SE1 read signal to streaming engine **3100**. Streaming engine **3100** then supplies stream 1 data stored in holding register **3128** via output OutL to the src1 input of functional unit **3230** and stream 1 data stored in holding register **3129** via output OutH to src1 input of paired functional unit **3240**. This is a double vector operation employing paired functional units. Sub-decoder **3220** also supplies an advance signal to stream 1. Streaming engine **2200** advances a double vector amount (512 bits) and stores the next stream 1 data in holding registers **3128** and **3129**. In the preferred embodiment all stream advances are in increments of a double vector (512 bits) even for single vector reads.

[0225] Sub-decoder **3221** is active if the vector/double vector signal indicates a vector operation. Sub-decoder **3221** determines whether src1 bit field **1305** specifies SE1 L. As listed in Table 17, this is a bit coding of “11100.” If this is the case, sub-decoder **3221** supplies a SE1L read signal to streaming engine **3100**. Streaming engine **3100** then supplies stream 1 data stored in holding register **3128** via output OutL to the src1 input of functional unit **3230**.

[0226] Sub-decoder **3222** is active if the vector/double vector signal indicates a vector operation. Sub-decoder **3222** determines whether src1 bit field **1305** specifies SE1L++. As listed in Table 17, this is a bit coding of “11101.” If this is the case, sub-decoder **3222** supplies a SE1L read signal to streaming engine **3100**. Streaming engine **3100** then supplies stream 1 data stored in holding register **3128** via output OutL to the src1 input of functional unit **3230**. Sub-decoder **3222** also supplies an advance signal to stream 1. Streaming engine **2200** advances a double vector amount (512 bits) and stores the next stream 1 data in holding registers **3128** and **3129**. In the preferred embodiment all stream advances are in increments of a double vector (512 bits) even for single vector reads.

[0227] Sub-decoder **3223** is active if the vector/double vector signal indicates a vector operation. Sub-decoder **3223** determines whether src1 bit field **1305** specifies SE1 H. As listed in Table 17, this is a bit coding of "11110." If this is the case, sub-decoder **3223** supplies a SE1H read signal to streaming engine **3100**. Streaming engine **3100** then supplies stream 1 data stored in holding register **3129** via output OutL to the src1 input of functional unit **3230**.

[0228] Sub-decoder **3224** is active if the vector/double vector signal indicates a vector operation. Sub-decoder **3224** determines whether src1 bit field **1305** specifies SE1 H++. As listed in Table 17, this is a bit coding of "11111." If this is the case, sub-decoder **3224** supplies a SE1H read signal to streaming engine **3100**. Streaming engine **3100** then supplies stream 1 data stored in holding register **3129** via output OutL to the src1 input of functional unit **3230**. Sub-decoder **3224** also supplies an advance signal to stream 1. Streaming engine **2200** advances a double vector amount (512 bits) and stores the next stream 1 data in holding registers **3128** and **3129**. In the preferred embodiment all stream advances are in increments of a double vector (512 bits) even for single vector reads.

[0229] FIG. **32** further illustrates supply of non-stream operand for double vector operations. Upon decoding a double vector operation, instruction decode **113** supplies a register number corresponding to the src1 field **1305** to the appropriate register file, either global vector register file **231** or the corresponding local vector register file **232** or **233**. The register file supplies data stored at this register number to the src1 input of primary functional unit **3230**. Instruction decoder **113** supplies a corresponding register number to the appropriate register file to recall the other vector of the double vector operation. It is known in the art to restrict valid codings of src1 field **1305** in double vector operations to even numbered registers. Data in the even numbered register is supplied to primary functional unit **3230**. Data in the next higher register number (the next odd register number following the specified even number register number) is supplied to the src1 input of paired functional unit **3240**. This is known in the art and is only illustrated for completeness.

[0230] Similar circuits are used to select data supplied to scr2 input of functional units **3230** and **3240** in response to the bit coding of src2/cst field **1304**. The src2 input of functional units **3230** and **3240** may be supplied with a constant input in a manner described above.

[0231] Thus sub-decoders **3215**, **3216**, **3217**, **3218**, **3221**, **3222**, **3223** and **3243** (programmed as listed in Table 17) are enabled for vector instructions. Sub-decoders **3213**, **3214**, **3219** and **3220** (programmed as listed in Table 18) are enabled for double vector instructions. As listed in Tables 17 and 18 the coding for global vector register file **231** and the corresponding local vector register file **232** or **233** are the same for double vector instructions as for vector instructions.

[0232] The exact number of instruction bits devoted to operand specification and the number of data registers and streams are design choices. Those skilled in the art would realize that other number selections that described in the application are feasible. In particular, the specification of a single global vector register file and omission of local vector register files is feasible. This invention employs a bit coding of an input operand selection field to designate low and high stream reads, a double wide read, and other bit codings to designate stream reads and advancing the stream.

[0233] While this specification contains many specifics, these should not be construed as limitations on the scope of what may be claimed, but rather as descriptions of features that may be specific to particular embodiments. Certain features that are described in this specification in the context of separate embodiments can also be implemented in combination in a single embodiment. Conversely, various features that are described in the context of a single embodiment can also be implemented in multiple embodiments separately or in any suitable sub-combination. Moreover, although features may be described above as acting in certain combinations and even initially claimed as such, one or more features from a claimed combination can in some cases be excised from the combination, and the claimed combination may be directed to a sub-combination or variation of a sub-combination.

[0234] Similarly, while operations are depicted in the drawings in a particular order, this should not be understood as requiring that such operations be performed in the particular order shown or in sequential order, or that all illustrated operations be performed, to achieve desirable results unless such order is recited in one or more claims. In certain circumstances, multitasking and parallel processing may be advantageous. Moreover, the separation of various system components in the embodiments described above should not be understood as requiring such separation in all embodiments.

Claims

1. A device comprising: a cache control circuit that includes: a cache interface capable of coupling to a memory; a first register coupled to the cache interface; and a second register coupled to the cache interface; wherein the cache control circuit is capable of: in response to a first request: retrieving a first data element from the memory via the cache interface; storing the first data element in the first register; retrieving a second data element from the memory via the cache interface; and storing the second data element in the second register; and in response to a second request, selecting between: providing the first data element from the first register without providing the second data element; providing the second data element from the second register without providing the first data element; and providing the first data element from the first register and the second data element from the second register.

2. The device of claim 1, wherein the first data element and the second data element are portions of a double vector value.

3. The device of claim 1, wherein the cache control circuit is capable of: providing the first data element from the first register to a first functional unit of a processor; and providing the second data element from the second register to a second functional unit of the processor.

4. The device of claim 3, wherein: each of the first functional unit and the second functional unit has a vector width; and a combined width of the first register and the second register is greater than the vector width of the first functional unit and the second functional unit.

5. The device of claim 1, wherein the cache control circuit is capable of determining, based on the second request, whether to replace the first data element in the first register with a third data element and replace the second data element in the second register with a fourth data element.

6. The device of claim 1, wherein the second request includes a field that specifies whether to: provide the first data element from the first register without providing the second data element; provide the second data element from the second register without providing the first data element; or provide the first data element from the first register and the second data element from the second register.

7. The device of claim 6, wherein the field of the second request further specifies whether to: provide at least one of the first data element or the second data element to a functional unit of a processor; or provide a third data element from a register file to the functional unit of the processor.

8. The device of claim 1, wherein: the cache control circuit includes an address generator coupled to the cache interface; and the address generator is capable of, in response to the first request, generate a first address associated with the first data element and generate a second address associated with the second data element.

9. The device of claim 1, wherein: the memory is a level-two (L2) cache; and the cache control circuit is capable of providing at least one of the first data element or the second data element to a functional unit of a processor via a data path that does not includes a level-one (L1) cache.

10. A device comprising: a first functional unit having a vector width; a second functional unit having the vector width; and a cache control circuit coupled to the first functional unit and the second functional unit; wherein: the cache control circuit includes: a first register coupled to first functional unit; and a second register coupled to the second functional unit; each of the first register and the second register has a respective width that is the vector width; and the cache control circuit is capable of, in response to a request, select between: providing a first data element from the first register without providing a second data element; providing the second data element from the second register without providing the first data element; and providing the first data element from the first register and the second data element from the second register.

11. The device of claim 10, wherein the first data element and the second data element are portions of a double vector width value.

12. A method comprising: receiving a first request; based on the first request, storing a first data element in a first register and a second data element in a second register; receiving a second request that specifies whether to: provide the first data element from the first register without providing the second data element; provide the second data element from the second register without providing the first data element; and provide the first data element from the first register and the second data element from the second register; and based on the second request, providing at least one of the first data element or the second data element.

13. The method of claim 12, wherein the first data element and the second data element are portions of a double vector value.

14. The method of claim 12, wherein: the providing at least one of the first data element or the second data element includes at least one of: providing the first data element to a first functional unit of a processor; or providing the second data element to a second functional unit of the processor.

15. The method of claim 14, wherein: a combined width of the first register and the second register is greater than a respective vector width of each of the first functional unit and the second functional unit.

16. The method of claim 12 further comprising determining, based on the second request, whether to replace the first data element in the first register with a third data element and replace the second data element in the second register with a fourth data element.

17. The method of claim 12, wherein the second request includes a field that specifies whether to: provide the first data element from the first register without providing the second data element; provide the second data element from the second register without providing the first data element; or provide the first data element from the first register and the second data element from the second register.

18. The method of claim 17, wherein the field of the second request further specifies whether to: provide at least one of the first data element or the second data element to a functional unit of a processor; or provide a third data element from a register file to the functional unit of the processor.

19. The method of claim 12, wherein the storing of the first data element in the first register and the second data element in the second register includes: generating a first address associated with the first data element; generating a second address associated with the second data element; and retrieving the first data element and the second data element from a cache using the first address and the second address.

20. The method of claim 19, wherein the cache is a level-two (L2) cache.