(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
11 August 2016 (11.08.2016)

WIPO | PCT

(10) International Publication Number
WO 2016/126433 A1

(51) International Patent Classification:
G06F 9/30 (2006.01) *H03M 7/40* (2006.01)
G06F 9/38 (2006.01)

(21) International Application Number:
PCT/US2016/014288

(22) International Filing Date:
21 January 2016 (21.01.2016)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
62/110,840 2 February 2015 (02.02.2015) US
14/727,051 1 June 2015 (01.06.2015) US

(71) Applicant: **OPTIMUM SEMICONDUCTOR TECHNOLOGIES, INC.** [US/US]; 120 White Plains Road, 4th Floor, Tarrytown, NY 10591 (US).

(72) Inventors: **MOUDGILL, Mayan**; 25 Random Farms Drive, Chappaqua, NY 10514 (US). **HOANE, Arthur, Joseph**; 10 Oakland Avenue, Yonkers, NY 10710 (US). **HURTLEY, Paul**; 6 Craven Lane, White Plains, NY 10605 (US).

(74) Agents: **CAPPO, Raymond** et al.; Lowenstein Sandler LLP, 65 Livingston Avenue, Roseland, NJ 07068 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JP, KE, KG, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Published:

— *with international search report (Art. 21(3))*

(54) Title: VECTOR PROCESSOR CONFIGURED TO OPERATE ON VARIABLE LENGTH VECTORS USING INSTRUCTIONS THAT CHANGE ELEMENT WIDTHS

(57) Abstract: A computer processor is disclosed. The computer processor may comprises a vector unit comprising a vector register file comprising at least one register to hold a varying number of elements. The computer processor may further comprise processing logic configured to operate on the varying number of elements in the vector register file using one or more instructions that produce results with elements of widths different than that of the input elements. The computer processor may be implemented as a monolithic integrated circuit.

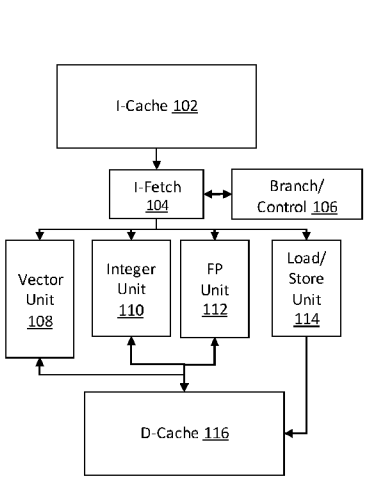


Figure 1

WO 2016/126433 A1

VECTOR PROCESSOR CONFIGURED TO OPERATE ON VARIABLE LENGTH VECTORS USING INSTRUCTIONS THAT CHANGE ELEMENT WIDTHS

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application claims the benefit of U.S. provisional patent application 62/110,840, filed February 2, 2015, and U. S. Utility Patent Application No. 14/727,051, filed June 1, 2015, the disclosures of which are incorporated herein by reference in their entirety.

TECHNICAL FIELD

[0002] Embodiments of the present disclosure relate to a vector processor, and in particular, to the architecture and implementation of a vector processor that includes one or more instructions that operate on variable length vectors using instructions that change element widths.

BACKGROUND

[0003] A vector instruction is an instruction that executes on a group of values using one instruction. For example, in the x86 architecture, the streaming SIMD extension (SSE) instruction ADDPS $xmm0,$xmm1 (add packed single-precision floating-point values), the two xmm registers each holds 4 single precision floating point values that are added together and stored in the first register. This behavior is equivalent to the pseudo-code sequence:

```
for(i = 0; i< 4; i++ )
    $xmm0[i] = $xmm0[i] + $xmm1[i]
```

[0004] The group of values can come from registers, memory, or a combination of both. Registers that hold groups of values, generally intended for use by vector instructions, are referred to as vector registers. The number of values in a group is called the vector length. In some examples,the vector length is also used to describe the number of operations performed by the vector instruction. Generally, the number of values in a vector register and the number of operations in a corresponding vector instruction calling for the vector register are the same, but they can be different in certain situations.

[0005] An instruction set architecture (ISA) including vector instructions is known as a vector ISA or *vector architecture*. A processor that implements a vector ISA is known as a *vector processor*.

[0006] A vector ISA where all vector instructions read their vector inputs from memory and write to memory without using any vector registers is known as a memory-to-memory vector or *memory-vector* architecture.

[0007] A vector ISA where all vector instructions, other than loads or stores, use only vector registers without accessing memory, is known as a *register-vector* architecture.

[0008] Vector instructions (such as the ADDPS above) can implicitly specify a fixed number of operations (four in the case of the ADDPS instructions). These are called *fixed-length* vector instructions. Another term for fixed-length register-vector instructions is *SIMD (Single Instruction Multiple Data)* instructions.

[0009] Previous generation vector processors were implemented on multiple boards using customized techniques to improve performance. The majority of them were targeted at high-performance computer applications, such as weather prediction, which often require supercomputers. However, technology development enabled single-chip microprocessors to out-perform these multi-board implementations, resulting in these vector processors being

phasing out. Instead, supercomputers became multi-processors that combined multiple of these high-performance microprocessors together.

[0010] A common characteristic of these processors is that they were not generally compatible with earlier models from the same company because the instruction set varies from model to model. This practice was motivated by the fact that they were targeted at problem domains where it was critical to extract as much performance as possible, and people were willing to rewrite the application to do so. But, this practice may result in implementation details of the machine being exposed in the instruction set, and instruction sets may change as the machine implementation details change from model to model. For example, the maximum vector length that could be specified was determined by the maximum number of elements that the vector register could hold in each implementation.

[0011] The second stream of vector processors emerged as the density of transistors kept going up. By the late 1990s, general-purpose microprocessors had reached a point of diminishing returns by increading the number of scalar functional units they could support even though there was still chip area that could be used to support more scalar function units. At the same time, there was a desire to support video encode and decode directly on these microprocessors. The confluence of these two trends led to the introduction of various fixed length vector extensions to existing general purpose architectures – MMX for the Intel x86, Altivec/VMX for the IBM PowerPC and MVI for the DEC Alpha, for instance.

[0012] These SIMD style architectures used registers with fixed byte lengths (8Bytes in the case of MMX, 16Bytes for Altivec). The registers were typically designed to hold multiple smaller length elements that can be operated on simultaneously. Thus, the MMX architecture could hold 2 4-Byte integers or 4 2-Byte integers or 8 1-Byte integers. The instructions PADDD/PADDW/PADDB would add the contents of two registers together, treating them as holding either 2 4-Byte/4 2-Byte/8 1-Byte values respectively.

[0013] As technology advances, it became possible to have the vector registers hold additional values. The MMX extension for the x86 architecture was followed by the 16-Byte SSE 32-Byte AVX2 and the 64-Byte AVX3. At each point, additional instructions were introduced to perform substantially the same operations.

[0014] In the case of implementations of general purpose architectures, for business reasons, different models are able to run the code written for older models. Thus, a newer implementation of the x86 architecture can support multiple different vector register widths, and instructions that operate on all of these instruction register widths.

SUMMARY

[0015] The above-described problems are addressed and a technical solution is achieved in the art by providing a computer processor comprising a vector unit comprising a vector register file comprising at least one vector register that may hold a varying number of elements. The computer processor may further comprise processing logic configured to operate on the varying number of elements in the vector register file using one or more instructions that produce results with elements of widths different than that of the input elements. The computer processor may be implemented as a monolithic integrated circuit.

[0016] In an example, the computer processor may further comprise a vector length register file comprising at least one register, where theat least one register of the vector length register file is used to specify the number of elements on which the processing logic operates.

[0017] In an example, the processing logic may be further configured to: read two vector registers from the vector register file, treat the contents of the two vector registers as two input vectors of input integer elements of a bit-size, element-wise multiply the elements of the first vector register with the corresponding elements of the second register to produce an output vector of output integer elements, the size of each output integer element having a

size that is twice the number of bits of a corresponding input integer element, and store the output vector of integer elements to a vector register in the vector register file. In an example, the size of each of the input integer elements may beone byte, and the size of each of the output integer elements may be two bytes. In an example, the size of each of the input integer elements may be two bytes, and the size of each of the output integer elements may be four bytes. In an example, the size of each of the input integer elements may be four bytes, and the size of each of the output integer elements may be eight bytes.

[0018] In an example, the processing logic may be further configured to: read a vector register from the vector register file, treat the contents of the vector register as an input vector of input integer elements of a smaller bit-size, convert the input vector of input integer elements into an output vector of output integer elements of a larger bit-size, and store the output vector of output integer elements to a vector register in the vector register file. In an example, the computer processor converting from a smaller bit-size to a larger bit-size may comprise the computer processor appending zeroes to upper bits of each of the input integer elements. In an example,the computer processor converting from a smaller bit-size to a larger bit-size may comprise the computer processor appending zeroes to the lower bits of each of the input integer elements. In an example, the computer processor converting from a smaller bit-size to a larger bit-size may comprise the computer processor appending copies of the highest-bit of the input element to the upper bits of the integer element for each of the input integer elements.

[0019] In an example, the size of each of the input integer elements may be one byte, and the size of each of the output integer elementsmay be two bytes. In an example, the size of each of the input integer elements may be two bytes, and the size of each of the output integer elements may be four bytes. In an example, the size of each of the input integer

elements may be four bytes, and the size of each of the output integer elementsmay be eight bytes.

[0020] In an example, the processing logic may be further configured to: read a vector register from the vector register file, treat the contents of the vector register as an input vector of input integer elements of larger bit-size,convert the contents of the vector register into output integer elements of a smaller bit-size, and store the resulting output vector of output integer elements of a smaller bit-size to a vector register in the vector register file. In an example, the computer processor converting from input integer elements of a larger bit-size to corresponding output integer elements of a smaller bit-size may comprise the computer processor discarding the high bits of each of the input integer elements. In an example, the computer processor converting from input integer elements of a larger bit-size to corresponding output integer elements of a smaller bit-size may comprise the computer processor discarding the low bits of each of the input integer elements.

[0021] In an example, the computer processor converting from input integer elements of a larger bit-size to corresponding output integer elements of a smaller bit-size may comprises the computer processor representing an integer value in the smaller bit-size, converting the integer value to the largest number representable at the smaller bit-size if the original integer value is larger than the largest number representable, or converting the integer value to the smallest number representable if the original integer value is smaller than the smallest number representable.

[0022] In an example, the bit-size of each of the input integer elements may be two bytes and the bit-size of each of the output integer elements may be one byte. In an example, the bit-size of each of the input integer elements may be four bytes and the bit-size of each of the output integer elements may betwo bytes. In an example, the bit-size of each of the

input integer element may be eight bytes and the bit-size of each of the output integer elements may befour bytes.

[0023] In an example, the processing logic may be further configured to: read a vector register from the vector register file, treat the contents of the vector register as an input vector of input floating-point elements of a smaller bit-size, convert the input floating-point elements of the vector register into an output vector of output floating-point numbers of a larger bit-size, and store the resulting output vector of output floating-point elements of a larger bit-size to a vector register in the vector register file. In an example, the size of each of the input floating-point elements may be two bytes, and the size of each of the output floating-point elements may be four bytes. In an example, the size of each of the input floating-point elements may be four bytes, and the size of the each of the output floating-point elements may be eight bytes.

[0024] In an example, the processing logic may be further configured to: read a vector register from the vector register file, treat the contents of the vector register as an input vector of input floating-point elements of a larger bit-size, convert the contents of the vector register into floating-point elements of a smaller bit-size, and store the resulting output vector of output floating-point elements of a smaller bit-size to a vector register in the vector register file. In an example, the bit-size of each of the input floating-point elements may be four bytes and the bit-size of each of the output floating-point elements may be two bytes. In an example, the bit-size of each of the input floating-point elements may be eight bytes and the bit-size of each of the output floating-point elements may be four bytes.

[0025] In an example, the computer processor may be further configured to control rounding and saturation modes. In an example, the value of a smaller floating-point number may be obtained from a larger floating-point number by rounding and saturation defined by the

rounding and saturation modes. In an example, the rounding and saturation modes may be defined as per an IEEE standard.

[0026] The above-described problems are addressed and a technical solution is achieved in the art by providing a method. The method may comprise holding, by a vector unit of a computer processor comprising a vector register file comprising at least one register, a varying number of elements. The method may further comprise operating, by processing logic of the computer processor, on the varying number of elements in the vector register file using one or more instructions that produce results with elements of widths different than that of the input elements. The computer processor may be implemented as a monolithic integrated circuit. In an example, the computer processor may further comprise a vector length register file comprising at least one register, where theat least one register of the vector length register file is used to specify the number of elements on which the processing logic operates.

[0027] In an example, the processing logic may be further configured to: read two vector registers from the vector register file, treat the contents of the two vector registers as two input vectors of input integer elements of a bit-size, element-wise multiply the elements of the first vector register with the corresponding elements of the second register to produce an output vector of output integer elements, the size of each output integer element having a size that is twice the number of bits of a corresponding input integer element, and store the output vector of integer elements to a vector register in the vector register file. In an example, the size of each of the input integer elements may beone byte, and the size of each of the output integer elements may be two bytes. In an example, the size of each of the input integer elements may be two bytes, and the size of each of the output integer elements may be four bytes. In an example, the size of each of the input integer elements may be four bytes, and the size of each of the output integer elements may be eight bytes.

[0028] In an example, the processing logic may be further configured to: read a vector register from the vector register file, treat the contents of the vector register as an input vector of input integer elements of a smaller bit-size, convert the input vector of input integer elements into an output vector of output integer elements of a larger bit-size, and store the output vector of output integer elements to a vector register in the vector register file. In an example, the computer processor converting from a smaller bit-size to a larger bit-size may comprise the computer processor appending zeroes to upper bits of each of the input integer elements. In an example,the computer processor converting from a smaller bit-size to a larger bit-size may comprise the computer processor appending zeroes to the lower bits of each of the input integer elements. In an example, the computer processor converting from a smaller bit-size to a larger bit-size may comprise the computer processor appending copies of the highest-bit of the input element to the upper bits of the integer element for each of the input integer elements.

[0029] In an example, the size of each of the input integer elements may be one byte, and the size of each of the output integer elementsmay be two bytes. In an example, the size of each of the input integer elements may be two bytes, and the size of each of the output integer elements may be four bytes. In an example, the size of each of the input integer elements may be four bytes, and the size of each of the output integer elementsmay be eight bytes.

[0030] In an example, the processing logic may be further configured to: read a vector register from the vector register file, treat the contents of the vector register as an input vector of input integer elements of larger bit-size,convert the contents of the vector register into output integer elements of a smaller bit-size;, and store the resulting output vector of output integer elements of a smaller bit-size to a vector register in the vector register file. In an example, the computer processor converting from input integer elements of a larger bit-

size to corresponding output integer elements of a smaller bit-size may comprise the computer processor discarding the high bits of each of the input integer elements. In an example, the computer processor converting from input integer elements of a larger bit-size to corresponding output integer elements of a smaller bit-size may comprise the computer processor discarding the low bits of each of the input integer elements.

[0031] In an example, the computer processor converting from input integer elements of a larger bit-size to corresponding output integer elements of a smaller bit-size may comprises the computer processor representing an integer value in the smaller bit-size, converting the integer value to the largest number representable at the smaller bit-size if the original integer value is larger than the largest number representable, or converting the integer value to the smallest number representable if the original integer value is smaller than the smallest number representable.

[0032] In an example, the bit-size of each of the input integer elements may be two bytes and the bit-size of each of the output integer elements may be one byte. In an example, the bit-size of each of the input integer elements may be four bytes and the bit-size of each of the output integer elements may betwo bytes. In an example, the bit-size of each of the input integer element may be eight bytes and the bit-size of each of the output integer elements may befour bytes.

[0033] In an example, the processing logic may be further configured to: read a vector register from the vector register file, treat the contents of the vector register as an input vector of input floating-point elements of a smaller bit-size, convert the input floating-point elements of the vector register into an output vector of output floating-point numbers of a larger bit-size, and store the resulting output vector of output floating-point elements of a larger bit-size to a vector register in the vector register file. In an example, the size of each of the input floating-point elements may be two bytes, and the size of each of the output

floating-point elements may be four bytes. In an example, the size of each of the input floating-point elements may be four bytes, and the size of the each of the output floating-point elements may be eight bytes.

[0034] In an example, the processing logic may be further configured to: read a vector register from the vector register file, treat the contents of the vector register as an input vector of input floating-point elements of a larger bit-size, convert the contents of the vector register into floating-point elements of a smaller bit-size, and store the resulting output vector of output floating-point elements of a smaller bit-size to a vector register in the vector register file. In an example, the bit-size of each of the input floating-point elements may be four bytes and the bit-size of each of the output floating-point elements may be two bytes. In an example, the bit-size of each of the input floating-point elements may be eight bytes and the bit-size of each of the output floating-point elements may be four bytes.

[0035] In an example, the computer processor may be further configured to control rounding and saturation modes. In an example, the value of a smaller floating-point number may be obtained from a larger floating-point number by rounding and saturation defined by the rounding and saturation modes. In an example, the rounding and saturation modes may be defined as per an IEEE standard.

BRIEF DESCRIPTION OF THE DRAWINGS

[0036] The present invention may be more readily understood from the detailed description of an exemplary embodiment presented below considered in conjunction with the following drawings:

[0037] **Figure 1** is a processor according to an example of the present disclosure.

[0038] **Figure 2** is a vector unit according to an example of the present disclosure.

[0039] **Figure 3** is an execution pipeline according to an example of the present disclosure.

[0040] Figure 4 is another execution pipeline according to an example of the present disclosure.

[0041] Figure 5 is a vector execution pipeline according to an example of the present disclosure.

[0042] Figure 6 is a non-overlapping sub-vector issue according to an example of the present disclosure.

[0043] Figure 7 is an overlapping sub-vector issue according to an example of the present disclosure.

[0044] Figure 8 is an instruction issue structure according to an example of the present disclosure.

[0045] Figure 9 shows renaming while executing instruction sequence according to an example of the present disclosure.

[0046] Figure 10 shows register file organization according to an example of the present disclosure.

[0047] Figure 11 is a memory bank according to an example of the present disclosure.

[0048] Figure 12 is vector register file according to an example of the present disclosure.

DETAILED DESCRIPTION

[0049] Examples of the present disclosure include an architecture that uses register-to-register variable-length vector instructions. This architecture is designed to allow for varying implementations that can execute the same instructions but at differing rates, allowing for both accommodation for future architecture designs and different price/performance trade-offs. Examples of the present disclosure include features to support both digital signal processing and graphics processing, as well as high-performance computing.

[0050] Examples of the present disclosure are designed for efficient implementations in the context of a modern out-of-order superscalar processor, relating to how register renaming and out-of-order execution are applied to the implementation of the variable-length vector register architectures.

[0051] Examples of the present disclosure include a variable-length register vector architecture suitable for implementation in the context of a modern general-purpose microprocessor. In one examples, the architecture may:

- allow an instruction to specify a vector lengths with values that are larger than those that can currently be implemented in hardware, and
- have the vector length specify the number of operations that are to be performed, independent of the number of elements that can be packed into the registers of specific implementation.

[0052] Examples of the present disclosure allow for backward compatible – namely, examples allow for an implementation that has larger registers to execute instructions with vector lengths that are optimized for a processor with shorter registers, while supporting the same set of instructions.

[0053] In some examples, the architecture supports precise exceptions for vector instructions , and causes an exception for a vector instruction that cannot be executed directly by the hardware in a specific implementation. These features make it possible to allow forward compatibility. In this case, if an instruction is optimized for vectors of length larger than supported on a particular implementation, embodiments of the present disclosure may trap the instructionand emulate the instruction in software then when the hardware attempts to execute the instruction.

[0054] The separation of vector length from the actual implementation provides flexibility with various implementations that support the execution of operations at differing rates. In

general, the old-style variable length vector processors would start processing one element of the vector every cycle, while the SIMD fixed-length vector implementation typically start processing all elements simultaneously. In contrast, embodiments of the present disclosure allow for different units to process operations at different rates. For example, one or more embodiments may choose to start upto 16 ADD operations at a time, so that an addition of vector length 38 would be started over three clock cycle, with 16 ADD operations in the first two clock cycles and the remaining six ADD operationsin the last clock cycle. The same embodiment may choose to only implement one divide unit, so that a divide of vector length 38 would be started over 38 clock cycles, with one divide in each cycle.

[0055] Old-style vector processors were targeted at high-performance floating-point processing, while modern SIMD style architectures are more general purpose. Embodiments of the present disclosure includeinstructions that are specifically targeted to digital signal processing and graphics applications in a manner that takes advantage of the variable length nature of the instructions and of the larger register sizes available.

[0056] In one embodiment, the vector architecture may be implemented as an in-order processor. In another embodiment, the vector architecture may be implemented using techniques such as out-of-order issue and register renaming to achieve better performance. This disclosure shows how a variable-length vector processor can take fit into an architecture utilizing features such as out-of-order issue and register renaming, and take advantage of these features to improve performance.

[0057] One of the inputs to a vector instruction can be a register that specifies the number of operations. For example, a vector ISA may be implemented where the vector ADD instruction is VADD $n0,$v1,$v2,$v3, with the behavior of:

```
for(i = 0; i< $n0; i++ )
```

$v1[i] = $v2[i] + $v3[i]

[0058] There is no specific term for this style of vector instruction. A prototypical vector processor, the Cray-1, uses vector count registers to specify the length of vector instructions, and by default, Cray-1 style vector instructions are associated with a vector length register. However, in the following discussion, to explicitly distinguish from the fixed-length vector instructions, the vector instructions that get their length from a register are referred to as variable-length vector instructions.

[0059] In one embodiment, the same instruction set architecture can have different realizations. In one possible realization, it is assumed that the vector unit is one of several units in a modern out-of-order processor, as shown in **Figure 1** that illustrates a processor according to an embodiment of the present disclosure.

[0060] The processor as shown in **Figure 1** may include:

- An instruction cache 102 that holds instructions for execution;
- An instruction fetch unit 104 that fetches instructions from the instruction cache 102;
- A control/branch execution unit 106 that controls the instructions that are fetched from the instruction cache 102 based on both the instructions that are fetched and various prediction methods;
- A variety of units including integer units 110 for executing integer instructions and floating point units 112 for floating point instructions;
- A load/store unit 114 that is responsible for coordinating the movement of data from the memory to the registers associated with the various units;
- A data-cache 116 that holds data elements;
- The vector unit 108.

[0061] The vector unit 108may include various circuit blocks, as shown in **Figure 2**. **Figure 2** illustrates a vector unit 200 according to an embodiment of the present disclosure. The vector unit as shown in **Figure 2** may include:

- A register rename block 202 that renames the architected vector unit registers to physical registers;
- An out-of-order issue block 204 that holds vector instructions that are not yet complete, and is responsible for sending these instructions for execution. Note that a vector instruction may be repeatedly issued based on the vector length and the number of functional units available for that instruction;
- A variety of physical vector register files 214 that include:
    - A vector register file 206 that holds vectors of elements;
    - A vector length register file 208 that is used to specify the number of operations performed by a vector instruction;
    - A vector accumulator register file 210 that holds scalar values resulting from operations such as summing up vectors;
    - A vector mask register file 212 that holds single bit values, and is used to control the execution of vector operations;
- A variety of functional blocks (as shown in the figure) include:
    - 2 multipliers 216;
    - 4 ALUs 218;
    - A single divide and/or square root 220;
    - A search block 222 for searching for minimum/maximum values;
    - A reduction block 224 for summing vector elements into a single value.

[0062] This combination of blocks as shown in **Figure 2** is selected for illustrative purposes, and is not exhaustive.

Unity vector architecture

[0063] A Unity ISA defines scalar instructions that perform the usual control, integer, floating-point and memory access instructions. It also defines a subset of instructions that manipulate vectors of instructions. In the taxonomy described in the previous section, Unity is a variable-length vector-register architecture.

[0064] This section focuses on the vector subset of the Unity ISA which is decoupled from the scalar subset. Embodiments of the present disclosure include the instructions that can be adapted to other scalar instruction sets by changing the encoding of the instructions and the number of registers.

[0065] In one embodiment, the Unity ISA registers that form the vector subset include:

- vector registers that hold collections of elements operated on by vector instructions;
- vector count registers that are used to specify the length of vector instructions, and other places where element counts are used;
- vector mask registers that hold a vector of single bits, used to control behavior of vector instructions over single elements;
- vector accumulator registers that are used when scalar values are needed as input or output for vector instructions.

[0066] In one embodiment, the vector count register of the Unity ISA includes 16-bits and is capable of specifying vector lengths of up to 65535. A vector of 65535 double precision floating numbers would require almost 512K bytes of memory per register. This size is impractical to implement in current technology. Embodiments of the present disclosure allow for a variety of implementations that implement different sizes of vector registers, suitable for the price/performance of the target application and the prevailing technology. If

the vector count cannot fit in the implemented vector size, the processor may take an exception and emulate the instruction.

Instructions

[0067] One Unity vector instruction, such as one that adds two vectors, is specified as vadd $n0,$v1,$v2,$v3. This instruction adds $n0 elements from the two source vector registers $v2,$v3 and stores the results in the target vector register $v1. The semantics of this instruction are equivalent to the following pseudo-code:

```
for(i = 0; i< $n0; i++ )
    $v1[i] = $v2[i] + $v3[i]
```

Typing

[0068] In one embodiment, each element of a vector of the Unity vector ISA may be implicitly typed as one of the following types:

- byte: 1-byte integer
- short: 2-byte integer
- int: 4-byte integer
- long: 8-byte integer
- half: 16 bit IEEE floating point
- single: 32 bit IEEE floating point
- double: 64 bit IEEE floating point

[0069] All instructions that operate on vector registers specify the type of elements they expect as inputs and the type of the elements that they generate as output. They include vector register loads and stores, and operations that rearrange data among vector registers

without actually operating on them. They also include instructions that convert vectors of one type to vectors of other types. These instructions specify both the input and output types.

[0070] In the assembler mnemonics, and in the description of this disclosure, vector instructions that operate on floating-point elements are suffixed with an 'f'. So, vadd adds vectors of integers, and vaddf operates on vectors of floating-point elements. The size of the integer/floating-point elements is indicated by a further suffix of _b/_h/_i/_l for integers and _h/_s/_d for floating-point. Thus, vadd_b adds vectors of 1 byte integers, and vaddf_s adds vectors of single-precision floating-point values. Since the element size generally does not change the semantics of the instruction (other than modifying the size and type of the inputs and outputs), the size suffix in this disclosure is generally omitted.

[0071] If a vector register element is written as a particular type by a first instruction and is subsequently read by a second instruction that expects vector registers of some other type, the behavior of the read by the second instruction is undefined.

[0072] The Unity ISA specifies that for each implementation, there exists an implementation specific mechanism that may save and restore all vector registers.

[0073] The constraints on typing of vector elements mean that implementations of the Unity ISA are free to pick the appropriate internal representations of the various vector element types. For example, 32-bit floating point numbers can be stored in an expanded format, with a 25 bit signed mantissa, an 8 bit exponent, and bits to indicate NaN and Infinity – a total of 35 bits.

[0074] In fact, these restrictions allow for an implementation to use distinct storage for the same vector register depending on the type of vector elements. For example, every vector register could be mapped to one of two different register arrays – one for use by vector instructions that use floating-point types and one for fixed-point types. This would increase the total size of the vector file, but might increase the performance of the implementation by

allowing increased overlap of integer and floating-point vector operations instructionsand/or by allowing for closer coupling between registers and functional units, resulting in decreased cycle time and/or number of pipeline stage.

[0075] The requirement for the existence of an implementation-dependent way to save and/or restore all vector registers arises from the fact that during context switching, the vector registers of the executing process that is being swapped out is saved and the vector registers of the process that is going to be executed is swapped in. However, the context switch code does not know the types of elements that were last written to each register. Consequently, it is impossible to use the standard instructions without violating the constraint that they expect elements to be of an appropriate type.

[0076] There are several possible mechanisms available depending on the implementation of the vector register. If all vector elements are stored in the registers in the same format as in memory, it is a matter of using a byte move instruction, suppressing type checking (if any). If the vector elements are stored differently based on type, the implementation may keep track of the last type written to each vector register in an implementation specific (and possibly non-architected) register. The context switch code can use that information to save and then later restore the values using the store and load instructions of the appropriate type.

Unwritten vector elements

[0077] If a vector instruction writes N elements to a vector register, and a subsequent vector instruction reads an element from that vector register that was not written to, the behavior is undefined. In this situation, an implementation can choose to take an exception, or to return a predefined value for the unwritten elements.

[0078] In all cases, the behaviors, when executing an instruction that reads unwritten values,need to be reproducible. This means that if the same sequence of instructions is

repeatedly executed, it returns the same results. This is true even if there is an intervening interrupt or context switch.

[0079] The motivation for this requirement is to make it possible to isolate faults in the code being executed on the processor. If a program executes an instruction that, because of some fault, reads a vector element that was not written to by the prior write to the vector, and the value being read is not consistent, then the results of the execution of the program can vary from run to run, making it difficult to pin down the exact cause.

Floating point rounding-mode/exceptions

[0080] IEEE floating point rounding-modes and exceptions defined for scalar operations is extended to vectors of floating point operations, by treating a vector as a series of floating-point operations, each of which needs to be compliant to IEEE floating point rounding modes and handles the five IEEE floating-point exception types appropriately.

[0081] For each of the five loating-point exception types (invalid, division by zero, overflow, underflow, inexact), there is a control bit and a status bit.The control bit is used to suppress or take interrupts when an exception is encountered, and the status bit is used to record the occurrence of an unsuppressed floating point-exception.

[0082] When a floating point vector instruction executes a sequence of floating point operations, and one of the sequence of operations encounters input operands that would cause one of the five exceptions, the control bit is examined. If the control bit indicates that this exception does not cause an interrupt to be taken, the status bit corresponding to the exception type is set to 1. Note that if multiple exception types are encountered while executing the sequence of operations, and the control bits for all these exception types are such that interrupts are not taken, multiple status bits can be set during the execution of a single vector floating point instruction.

[0083] If an exception is encountered, and the control bit is set such that the exception is to cause an interrupt, the interrupt is taken as though the instruction did not execute. Enough information is recorded so that the interrupt handler can identify the exception type and the position in the operation sequence of the operation that caused the interrupt to be taken.

[0084] For complete IEEE floating point support, the floating point vector instructions support all 4 IEEE rounding modes –0, -∞,+∞, nearest (ties to even). The rounding mode is specified by a control register, which is used to specify the floating point rounding mode to be applied. When a vector floating point instruction is executed, the results of each of the sequence of operations is rounded based on the control register arithmetic unary instructions

[0085] The arithmetic vector instructions with 1 vector register input, V*OP* $n0,$v1,$v, executes $n0 unary operations on elements writing to $v1from $v2. In pseudo-code:

```
for(i = 0; i< $n0; i++ )
    $v1[i] = OP $v2[i]
```

[0086] The basic integer unary instructions are:

- vabs: compute the absolute value of each element
- vneg: negate each element

[0087] The basic floating-point arithmetic unary instructions include:

- vabsf: compute the absolute value of each element
- vnegf: negate each element
- vrecf: compute the reciprocal of each element
- vrecef: compute an initial estimate of the reciprocal of each element that can be refined to an exact result by software
- vsqrtf: compute the square root of each element

- vsqrtef: compute an initial estimate of the square root of each element that can be refined to an exact result by software

[0088] Note that the vabs/vabsf and vneg/vnegf instructions are equivalent to the binary vdiff/vdiffs and vsub/vsubf binary instructions (described below) with the first input containing all zero elements.

Arithmetic binary instructions

[0089] The arithmetic vector instructions with 2 vector register inputs, VOP $n0,$v1,$v2,$v3, executes $n0 binary operations, taking one input from $v2 and the other from $v3 and writing to $v1. In pseudo-code:

```
for(i = 0; i< $n0; i++ )
    $v1[i] = $v2[i] OP $v3[i]
```

[0090] The basic floating-pointarithmetic binary instructions include:

- vaddf: add elements
- vsubf: subtract elements
- vdifff: find the absolute difference of the elements
- vmulf: multiply the elements
- vminf: minimum of two elements
- vmaxf: maximum of two elements
- vdivf: floating-point divide

[0091] The basic integer arithmetic binary instructions include:

- vadd: add elements
- vsub: subtract elements
- vdiff: absolute difference of the elements
- vmul: multiply elements, keeping lower part of result

- vmin: minimum of the two elements, treated as signed integers
- vminu: minimum of the two elements, treated as unsigned integers
- vmax: maximum of the two elements, treated as signed integers
- vmaxu: maximum of the two elements, treated as unsigned integers
- vand: logical and of the bits of the elements
- vor: logical or of the bits of the elements
- vnand: logical nand of the bits of the elements
- vxor: logical xor of the bits of the elements
- vshl: left shift the value of the first element by the relevant 3/4/5/6 lower bits of the second element, depending on whether it is a byte/short/int/long integer type.
- vshr: right shift the value of the first element by the relevant lower bits of the second element, shifting in 0s (logical right shift)
- vshra: right shift the value of the first element by the relevant lower bits of the second element, shifting in the sign bit (arithmetic right shift)

[0092] In one embodiment, the Unity ISA provides for the byte variants of the shift instructions. In these instructions, the first argument is one of the integer types, while the second argument is a vector of byte elements. These instructions include vshbl, vshbr and vshbra, corresponding to vshl, vshr, vshra.

[0093] The result of an integer multiply of N bytes may, in general, require 2N bytes to represent. Recognizing this, embodiments include several different flavors of vector multiply. The basic instruction, vmul, multiplies two N byte elements together and stores the lower N bytes of the result. In one embodiment, the Unity ISA may include vector instructions that promote the multipy results to the next larger integer type and store the whole result. The vector instructions may include:

- v2mul: multiply the two elements, treating both arguments assigned integers

- v2mulus: multiply the two elements, treating the first argument as a signed integer and the second as an unsigned integer
- v2muluu: multiply the two elements, treating both arguments as unsigned integers.

[0094] In some implementations, these vector instructions as discussed above are not defined for the long integer type.

[0095] Being able to process vectors of M N-byte elements and produce a single vector of M 2N-byte elements is something that fixed-vector architectures cannot do. Since the size of the vector registers is fixed, if they read M*N bytes and produce 2*M*N bytes, they need to either:

- use two separate instructions to process the vector, each of which reads M/2 inputs, or
- use an instruction that writes two separate vector registers, each with M/2 outputs.

Multiply and Add

[0096] The multiply-and-add vector instructions may include three-input instructions, such as, the instruction of VOP $n0,$v1,$v2,$v3,$v4 that executes $n0 multiply-and-add operations, multiplying the inputs from $v2 and $v3 and combining it with $v4, then writing to $v1. In pseudo-code, the VOP instrunction stands for:

```
for(i = 0; i< $n0; i++ )
    $v1[i] = ±( $v2[i] * $v3[i] ) ± $v4[i]
```

[0097] There are four floating point variants for the multiply-and-add instruction, corresponding to the ways the result of the multiply can be combined with the third argument.

- vmuladdf: add result of multiply to third argument
- vmulsubf: add result of multiply to negative of third argument
- vnmuladdf: add negative of result of multiply to third argument

- vnmulsubf: add negative of result of multiply to negative of third argument

[0098] There are two classes of integer multiply-and-add, based on whether the type of the result is the same as that of the inputs, or whether it is of twice the size.

[0099] In the four instructions of the first class, all three inputs are of the same length. The lower N bytes of the multiply are combined with the third argument, and the N bytes of the result are saved. The first class of instrunctions includes:

- vmuladd: add result of multiply to third argument
- vmulsub: add result of multiply to negative of third argument
- vnmuladd: add negative of result of multiply to third argument
- vnmulsub: add negative of result of multiply to negative of third argument

[00100] In the second class, the 2 inputs to the multiply are of size N, and the 3rd input are of size 2N. The first two arguments are multiplied together as signed integers, and the resulting 2N byte value is added to the 2N byte 3rd argument, with 2N byte result being stored. The second class of instructions includes:

- v2muladd: add result of multiply to third argument
- v2mulsub: add result of multiply to negative of third argument
- v2nmuladd: add negative of result of multiply to third argument
- v2nmulsub: add negative of result of multiply to negative of third argument

Reduction

[00101] Reduction operations combine vector registers into scalar results that are stored in a reduction register. Optionally, a reduction register can also be added to the result.

[00102] The first class of reduction instructions includes the vector sum reduce instructions. The floating point vector sum reduce instruction, vsumred0f $n0,$c1,$v2, adds

together $n0 elements of $v2 and stores them to the accumulator register $c1. The variant form, vsumredf $n0,$c1,$v2,$c3, also adds $c3 to the result. In pseudo-code:

```
sum = $c3 // or 0
for(i = 0; i< $n0; i++ )
    sum += $v2[i]
$c1 = sum
```

[00103] The floating point sum is computed at the same precision as the input types.

[00104] By contrast, the integer forms of the instructions, vsumred and vsumred0, sum up the results as 64 bit numbers independent of the type of the elements of vector $v2.

[00105] The second class of reduction instructions includes vector multiply sum instructions. The floating point multiply reduce instructions are vmulred0f $n0,$c1,$v2,$v3 and vmulred0f $n0,$c1,$v2,$v3,$c4. The behavior of these instructions is similar to that of the sum reduce, except that the elements of the two vector registers are multiplied together prior to being summed. In pseudo-code:

```
sum = $c4 // or 0
for(i = 0; i< $n0; i++ )
    sum += $v2[i] * $v3[i]
$c1 = sum
```

[00106] As in the case of the vector sum reduce, the floating point sum is computed in the same precision as the vector elements.

[00107] In the case of the integer form of the multiply reduce, vmulred and vmulred0, the elements of the two vectors are multiplied together as signed integers, and the double width products are summed together in 64 bits.

Partial Reduction

[00108] In one embodiment, instructions of the Unity ISA can perform partial reductions. These instructions may combine subsets of vectors, or of results of computations, into vectors with fewer elements.

[00109] The basic floating-point combining instruction is vsumnf_xN $n0,$v1,$v2, where N is a whole number. This instruction sums up groups of N elements of $v2, and puts the $n0/N results in $v2. The pseudo-code for this instruction is:

```
for(i = 0; i< $n0; i += N )
        sum = 0
        for( j = 0; j < N; j++ )
                sum += $v2[i+j]
        $v1[i/N] = sum
```

[00110] In one embodiment, there is an integer equivalent (vsumn) and an equivalent that sums up integer valued elements using results that are twice the size as the inputs (v2sumn).

[00111] In another embodiment, the Unity ISA may include another partial reduction vector instruction that multiplies elements of two vectors and then sums up groups of products. The version that operates on floating point elements is vdotf_xN $n0,$v1,$v2,$v3, wherethe “dot” in the instruction is short for “dot-product,”because the operation is similar to that for a dot product. The pseudo-code for this instruction is:

```
for(i = 0; i< $n0; i += N )
        sum = 0
        for( j = 0; j < N; j++ )
                sum += $v2[i+j]*$v3[i+j]
        $v3[i/N] = sum
```

[00112] The integer and double-width integer equivalents are vdot and v2dot.

Complex instructions

[00113] In one embodiment, instructions of the Unity ISAmay include those that perform complex multiplies on vector elements, where the vector are treated as sequences of alternating real and imaginary values. For example, the vector complex floating-point multiply instruction, vxmulf $n0,$v1,$v2,$v3,includes the behavior described by the following pseudo-code:

```
for(i = 0; i< $n0; i += 2 )
    re2 = $v2[i+0]
    im2 = $v2[i+1]
    re3 = $v3[i+0]
    im3 = $v3[i+1]
    $v1[i+0] = re2*re3 – im2*im3;
    $v1[i+1] = re2*im3 + re3*im2;
```

[00114] Note that $n0 specifies twice the number of complex multiplies. The reason is that when performing other operations on complex multiplies, the normal vector operations such as vector add are used. These count the real and imaginary ADDs as separate ADDs. So, to add a vector of N/2 complex values, a count of N needs to be specified. The vector complex multiply instruction uses N so that the same vector count can be used to control both kinds of operations. The behavior of a vector complex multiply instruction is undefined if the count is odd.

[00115] All forms of vector instructions that use a floating-point multiply or a signed-signed integer multiply have their complex multiply equivalent. Apart from the vector floating point multiply instructions, these include vector integer multiplies (vxmul) and vector double width integer multiplies (v2xmul), floating point multiply add (vxmuladdf, vxmulsubf, vxnmuladdf, vxnmulsubf), integer multiply add

(vxmuladd,vxmulsub,vxnmuladd,vxmulsub) and integer double width multiply add (v2xmuladd, v2xmulsub, v2xnmuladd, v2xnmulsub).

[00116] There are also vector complex sum reduction and vector complex multiply reduction instruction. These instructions target two accumulators, one for the real and one for the imaginary sum. For example, vxmulred0f $n0,$c1,$c2,$v3,$v4 complex multiplies pairs of elements of the vector registers $v3 and $v4, treating them as complex numbers, and then sums the real and imaginary values separately. The real sum is written to $c0 and the imaginary sum to $c1. The non-zero variant vxmulredf $n0,$c1,$c2,$v3,$v4 add the original contents of $c0 and $c1 to the sums prior to writing back the results. This can be expressed by the following pseudo-code:

```
resum = $c1 // or 0
imsum = $c2 // or 0
for(i = 0; i< $n0; i += 2 )
    re3 = $v3[i+0]
    im3 = $v3[i+1]
    re4 = $v4[i+0]
    im4 = $v4[i+1]
    resum = re3*re4 – im3*im4;
    imsum = re3*im4 + re4*im3;
$c1 = resum
$c2 = imsum
```

[00117] The vector complex sum instructions such as vxsumred0f $n0,$c1,$c2,$v3 sum up alternate elements of the vector register and store them in the two specified accumulators. There are both floating-point (vxsumredf, vxsumred0f, vxmulredf, vxmulred0f) and integer equivalents (vxsumred, vxsumred0, vxmulred, xvmulred0) of the reduction instructions.

[00118] The partial reduction instructions have complex equivalents as well - vxsumnf, vxsumn, v2xsumn, vxdotf, vxdot, v2xdot.

Saturation

[00119] In one embodiment, instructions in the Unity ISA may include the fixed point saturating vector instructions. These instructions operate on integer typed elements. Each of these fixed point saturating instructions may include an equivalent non-saturating vector instruction that operates on integer typed data. In general, a fixed point saturating instruction differs from its non-saturating equivalent in the treatment of infinite-precision results that exceed the range of values representable by the type of the result element. In the case of the non-saturating operations, the result is truncated. In the case of saturating instructions, the result is saturated to the most positive/negative value representable, depending on the sign of the result.

[00120] The instructions that fit the basic pattern are:

- vadds: add saturating
- vsubs: subtract saturating
- vdiffs: absolute difference saturating; note that this can only saturate to largest positive value
- vshls: shift left saturating
- vshbls: shift by byte left saturating

[00121] Fixed point multiplies treat both inputs as signed. Fixed point multiplies include the additional property that the results of the multiply of two elements is further multiplied by 2. So, the result of an integer multiply and a fixed point multiply may differ by a factor of 2, even when no saturation takes place. If the result of a fixed-point multiply of size N is to

be stored back into size N, the upper N bytes of the multiply are saved, unlike an integer multiply where the lower N bytes are saved.

[00122] The multiply based fixed point instructions include:

- vmuls: fixed point multiply saturating
- vmuladds: fixed-point multiply and add saturating
- vmulsubs: fixed-point multiply and subtract saturating
- vnmuladds: negate fixed-point multiply and add saturating
- vnmulsubs: negate fixed-point multiply and subtract saturating
- v2muls: fixed point multiply with double-width result saturating
- v2muladds: fixed-point multiply with double-width result and add saturating
- v2mulsubs: fixed-point multiply with double-width result and subtract saturating
- v2nmuladds: negate fixed-point multiply with double-width result and add saturating
- v2nmulsubs: negate fixed-point multiply with double-width result and subtract saturating

[00123] In addition, the complex integer multiplies variants also include saturating fixed point variants

- vxmuls: fixed point complex multiply saturating
- vxmuladds: fixed-point complex multiply and add saturating
- vxmulsubs: fixed-point complex multiply and subtract saturating
- vxnmuladds: negate fixed-point complex multiply and add saturating
- vxnmulsubs: negate fixed-point complex multiply and subtract saturating
- v2xmuls: fixed point complex multiply with double-width result saturating
- v2xmuladds: fixed-point complex multiply with double-width result and add saturating

- v2xmulsubs: fixed-point complex multiply with double-width result and subtract saturating
- v2xnmuladds: negate fixed-point complex multiply with double-width result and add saturating

[00124] The integer reduction operations also have their fixed point equivalents. In this case, the results of the summation are saturated prior to being written to the accumulator registers. These instructions are:

- vsumred0s: sum elements and reduce saturating
- vsumreds: sum elements with accumulator and reduce saturating
- vmulred0s: sum element products and reduce saturating
- vmulreds: sum element products with accumulator and reduce saturating
- vxsumred0s: sum alternate elements and reduce saturating
- vxsumreds: sum alternate elements with accumulator and reduce saturating
- vxmulred0s: sum complex element products and reduce saturating
- vxmulreds: sum complex element products with accumulator and reduce saturating

[00125] The partial reduction operations also include fixed point equivalents. In this case, multiplies are performed using fixed point semantics, and the result is saturated prior to being written back.

- vsumns: sum groups of vector elements and saturate sums
- v2sumns: sum groups of vector elements with double-width results and saturate sums
- vdots: multiply elements using fixed point semantics, sum products, and saturate sums
- v2dots: multiply elements using fixed point semantics, sum products with double width results, and saturate sums
- vxsumns: sum groups of alternate elements and saturate sums

- v2xsumns: sum groups of alternate elements with double-width results and saturate sums
- vxdots: multiply elements using complex fixed point semantics, sum products, and saturate sums
- v2dxots: multiply elements using complex fixed point semantics, sum products with double width results, and saturate sums

Conversion

[00126] The floating point elements of a vector can be converted to the next higher size by using the vunpackf $n0,$v1,$v2 instruction. Thus, vunpack_f $n0,$v1,$v2 may convert the $n0 single-precision values in $v2 into doubles, and store the results in $v1.

[00127] The floating point elements of a vector can be converted to the next smaller size by using the vpackf $n0,$v1,$v2 instruction. Thus, vunpack_f $n0,$v1,$v2 may convert the $n0 single-precision values in $v2 into half-precision values, and store the results in $v1.

[00128] Integer elements of a vector can be converted to the next higher size by using the vunpack instruction. This instruction has several variants, including:

- sign extend
- zero extend
- place in upper half, pad right with zeroes

[00129] For example, when unpacking the byte 0xff, the three options may result in the half-words 0x00ff, 0xffff, 0xff00, respectively.

[00130] Integer elements of a vector can be converted to the next lower size by using the vpack instruction. This also has several variants, including:

- use lower half
- saturate lower half

- use upper half
- fixed point pack; this involves taking the upper half, incrementing by 1 if the highest bit of the lower half is set, saturating if necessary.

[00131] For example, when packing the half word 0xabf0, the four options produce the results of 0xf0, 0x80, 0xab, 0xac.

[00132] A vector of integer values can be converted to or from a vector of equivalent floating point values using the instructions vfcvti/vcvtif. These instructions specify the types of both the integer and the floating-point elements. Thus, vfcvti_w_s converts four byte integers to 32 bit single-precision floating point numbers, while vicvtf_d_w converts double precision floating point elements to four byte integers.

Scalar /Vector movement

[00133] A vector can be initialized to integer valued elements using the contents of an accumulator register using the vector broadcast instruction vbrdc $n0,$v1,$c2 and to floating point elements using the floating point equivalent vbrdcf. The pseudo-code for this instruction is:

```
for(i = 0; i< $n0; i++ )
    $v1[i] = $c2
```

[00134] A single integer element can be inserted into a vector using the vector append instruction, vappendc $n0,$v1,$c2. The floating-point equivalent is vappendcf. These instructions append an element $c2 at the position $n0 of $v1, and then increment $n0. The pseudo-code for this instruction is:

```
$v1[$n0] = $c2
$n0 = $n0+1
```

[00135] The increment of $n0 allows for repeated insertions into a vector without having to explicitly change the count register.

[00136] When an integer element is being copied into a vector using the vappendc/vbrdc instructions, the integer element is truncated. The vapppendcs/vbrdcs instructions saturate the value to the specified integer element width prior to writing.

[00137] A single integer element can be moved from a vector to an accumulator using the veleminc $n0,$c1,$v2 and velemdec $n0,$c1,$v2 instructions. The veleminc instruction copiesthe signed integer element in $v2 at $n0 to accumulator $c1, and then increments $n0. The velemdec instruction decrements $n0, and then copies value at the decremented position from $v2 into $c1. The pseudo-code is:

```
// veleminc
$c1 = $v[$n0]
$n0++
// velemdec
$c1 = $v[$n0-1]
$n0—
```

[00138] The operations are specified this way to allow for moving successive values from the vector register to the accumulator starting at the beginning or the end.

[00139] The floating-point equivalents are veleminсf and velemdecf.

Data Rearrangement

[00140] There are a variety of instructions that rearrange data, some using just one vector, others using two. The behavior of the integer and floating-point variants of these instructions is identical, except in the size of the values that are moved.

[00141] The vector interleave instructions, vilv_xN $n0,$v1,$v2,$v3 and its floating-point equivalent vilvf create a result vector $v1 of $n0 elements by alternately selecting N elements from $v2 and $v3. $n0 may be a multiple of 2*N. The behavior of these instructions is captured by the following pseudo-code:

```
for(i = 0; i< $n0; i += N )
    for( j = 0; j < N; j++ )
        $v1[2*i+j] += $v2[i+j]
    for( j = 0; j < N; j++ )
        $v1[2*i+N+j] += $v3[i+j]
```

[00142] Embodiments of the present disclosure also provide for vector instructions that perform the inverse operation including vodd_xN $n0,$v1,$v2 and veven_xN $n0,$v1,$v2, along with their floating-point equivalents voddf & vevenf. These instructions extract the odd and even groups of N elements. This can be demostrated in the following pseudo-code:

```
// veven
for(i = 0; i< $n0; i+= N )
    for( j = 0; j < N; j++ )
        $v1[i+j] = $v2[2*i+j]
// odd
for(i = 0; i< $n0; i+= N )
    for( j = 0; j < N; j++ )
        $v1[i+j] = $v2[2*i+j+N]
```

[00143] In one embodiment, the veven and vodd instructions are used to separate alternating groups of elements of a vector, and then to put them back together using vilv.

[00144] The trailing elements of a vector can be extracted using the vtail $n0,$v1,$v2,$n3 instruction. This instruction (and its floating-point equivalent vtailf) extracts the $n0

elements starting at $n3 from vector $v2 and puts them into $v1. The following pseudo code shows the operations in detail:

```
for(i = 0; i< $n0; i++ )
    $v1[i] = $v2[i+$n3]
```

[00145] Two vectors can be concatenated using the vconcat $n0,$v1,$v2,$v3,$n4 instruction, and its floating point equivalent vconcatf. In this instruction, the first $n4 elements of $v2 are combined with corresponding elements for $v3 to form a new vector containing $n0 elements that is written to $v1. The following pseudo code shows the operations in detail:

```
for(i = 0; i< $n4; i++ )
    $v1[i] = $v2[i]
for(i = $n4, j = 0; i< $n0; i++, j++ )
    $v1[i] = $v3[j]
```

[00146] Groups of elements in a single vector can be rearranged using the vswap_ xN $n0,$v1,$v2. This instruction splits the $n0 elements into groups of N elements, and then swaps these groups and writes them to $v1. The behavior of this instruction is undefined if $n0 is not a multiple of 2* N. The pseudo-code is:

```
for(i=0; i<$n0; i += 2*N )
    for( j = 0; j < N; j++ )
        v2[i+j] = v1[i+j+N]
        v2[i+j+N] = v1[i+j]
```

Mask Set

[00147] The vector mask registers can generally be considered as a vector of single bits.

[00148] The vector mask registers can be set by the result of various vector compare instructions. For example, insturction vcmps_cond $n0,$m1,$v2,$v3 compares elements of

the vector, assuming that the elements are signed integers of the appropriate size, and then store a 0/1 assuming the condition is false/true. The pseudo-code for this instruction is:

```
for(i = 0; i< $n0; i++ )
    $m1[i] = ( $v2[i] cond $v3[i] ) ? 1 : 0;
```

[00149] The condition, *cond* , can be one of "equal,""not equal,""greater than,""less than,""greater than or equal," and "less than or equal."

[00150] A similar instruction, vcmpu, compares vectors of integer elements as unsigned values. It includes the same 6 conditions.

[00151] The floating point instruction, vcmpf, compares vectors of floating point. In addition, it accepts two more conditions: to check if either input is a NaN, and to check if neither input is a NaN.

[00152] Embodiments may also inlcude an instruction, vclassf_class$n0,$m1,$v2 that tests the floating point class of each element of the vector $v2, and sets the corresponding in $m1 based on whether it is a member of the class(es) specified by the class field in the instruction. A floating point number can be:

- Not-a-Number (NaN); this can be further divided into quiet NaNs (qNaN) or signaling NaNs (sNaN)
- Infinity
- Zero
- Subnormal
- Normal (which is all other cases)

[00153] The class instruction modifier can allow the instruction to test for combinations of classes, as well, such as Normal +Zero, Normal+Zero+Subnormal, etc.

[00154] Embodiments may include two instructions that allow the testing of the sign and class simultaneously. The vclasspsf_classchecks for class membership as above, and also

checks for the sign bit. The mask bit is set to 1 only if the instruction is in the class being tested and is positive, except for the case where the class being tested is NaN. For this exception case, the condition is true only if the NaN is a signaling NaN (sNaN). The vclassnqf_class checks for negative/qNaN.

[00155] The vmop_op $n0,$m1,$m2,$m3 combines the $n0 bits of the vector mask registers $m2 and $m3 using the bitwise operation op, and stores the resulting bits back into $m1. The pseudo-code for this instruction is:

```
for(i = 0; i< $n0; i++ )
    $m1[i] = $m2[i] op $3[i]
```

[00156] Op is one of "and,""nand,""or,""nor,""xor,""xnor,""and-complement,"or "not and-complement."

Mask data

[00157] There are several flavors of masked data operations that move the contents of vector registers under the control of the mask bits.

[00158] The vector select instructions, vself $n0,$v1,$v2,$v3,$m4 and its integer counterpart vsel look at each of the $n0 bits of the mask register, select the corresponding element from $v2 or $v3 depending on the value of the bit, and store it in that position in $v1. Its pseudo-code is:

```
for(i = 0; i< $n0; i++ )
    $v1[i] = ( $m4[i] == 0 ) ? $v2[i] : $v3[i]
```

[00159] The vector puncture instruction vpunct_tf $n0,$v1,$v2,$m3, where tf is one of 0 or 1, steps over the $n0 integer elements of vector mask $m3 and, if it is the same as tf, appends the corresponding element of $v2 to the result vector $v0. The number of elements

appended is written back to $n0. The floating-point equivalent is vpunctf. Their behavior can be described using the pseudo-code:

```
j = 0
for(i = 0; i< $n0; i++ )
    if( $m3[i] == tf )
        $v1[j] = $v2[i]
        j++
$n0 = j
```

[00160] Another instruction is the vector vmix instruction, vmix $n0,$v1,$v2,$v3,$m4. In the vmix instruction and its floating-point counterpart vmixf, the $n0 bits of $m4 are examined. Based on its value either the next unread value of $v2 or the next unused value of $v3 is picked and added to $v1. The behavior is captured by the following pseudo-code:

```
j = 0
k = 0
for(i = 0; i< $n0; i++ )
    if( $m4[i] == 0)
        $v1[i] = $v2[j]
        j++
    else
        $v1[i] = $v3[k]
        k++
```

[00161] The combination of the vpunct and vmix instructions can be used to efficiently implement loops with conditions using vectors. For example, the pseudo code fragment belowshows one implemenation:

```
for(i = 0; i< N; i++ )
```

```
if( M[i] )
    X[i] = computationF( A[i] )
else
    X[i] = computationG( A[i] )
```

[00162] In one embodiment, the vector puncture instructions are used to separate the elements of input array A into vectors for the true and false cases, and then use the vmix instructions to combine the results from the true and false cases into the output array X. This is the same as though the code-fragment is rewritten as:

```
// vpunct_1 equivalent sequence
j = 0
for(i = 0; i< N; i++ )
    if( M[i] )
        A1[j++] = A[j]
N1 = j

// vpunct_0 equivalent sequence
j = 0
for(i = 0; i< N; i++ )
    if( !M[i] )
        A0[j++] = A[j]
N0 = j

// hopefully, now these loops can be vectorized
for(i = 0; i< N1; i++ )
    X1[i] = computationF( A1[i] )
```

```
for(i = 0; i< N0; i++ )
        X0[i] = computationG( A0[i] )

// vmix equivalent sequence
j = 0
k = 0
for(i = 0; i< N; i++ )
        if( M[i] )
                X[i] = X1[j++]
        else
                X[i] = X0[k++]
```

[00163] One case where these kinds of transformations are very useful is in implementing divergent graphics kernels.

Search

[00164] The vector search instructions vsearchmax $n0,$n1,$c2,$v3 and vsearchmin $n0,$n1,$c2,$v3may search for the maximum or minimum valued element of the $n0 elements in $v3 within a vector and store that value in $c2 and the corresponding position of the maximum or minimum valued element in $n1. The following pseudo-code is for vsearchmax:

```
max = $v3[0]
pos = 0
for(i = 1; i< $n0; i++ )
        if( $v3[i] > max )
                max = $v3[i]
```

```
pos = i
```

[00165] Embodiments also include unsigned and floating-point equivalents for these instructions, including, for example, vsearchmaxu, vsearchminu, vsearchmaxf, and vsearchminf.

Digital Signal Processing

[00166] The variety of instructions (such as the fixed point saturation instructions) described above are useful in DSP (digital signaling processing) applications. This section describes instructions whose primary motivation is to accelerate the performance of DSP operations.

[00167] One set of common operations in DSP are filtering and correlation, including repeated dot products of a short vector with various positions within a longer vector of elements. In case of the canonical filter, dot products are performed at successive positions, as shown in the following code fragment.

```
for(i = 0; i< N; i++ )
    sum = 0;
    for( j = 0; j < TAPS; j++ )
        sum += coeff[j]*in[i+j];
    out[i] = sum;
```

[00168] The Unity ISA uses the vfilt_xN $n0,$v1,$v2,$v3,$n4 instruction (and its floating-point and saturating fixed point equivalents vfiltf & vfilts) to accelerate this. In vfilt_xN, $v3 is a vector of $n4 elements that computes $n0 dot-products with $n4 elements of $v2 starting at the beginning, and advancing in steps of size N. This behavior is described by the following pseudo-code:

```
for(i = 0; i< $n0; i++ )
```

```
sum = 0;
for( j=0; j < $n4; j++ )
        sum += $v3[j] * $v2[N*i+j]
$v1[i] = sum
```

[00169] The other common DSP operation that the Unity ISA accelerates is the radix-2 FFT (Fast Fourier Transform). The radix-2 FFT includes multiple stages, with a kernel that can be expressed as in the following code fragment:

```
for(i = 0; i< 2^M; i++ )
        X[i] = A[i] + W[i]*A[i+2^M];
        X[i+2^N] = A[i] – W[i]*A[i+2^M];
```

[00170] Note that X,A, and W in this example are all complex numbers, and the multiplication is a complex-complex multiplication. W is known as the twiddle factor.

[00171] We accelerate this kernel by using the vfftf_xN $n0,$v1,$v2,$v3, or by using its integer and fixed-point saturating equivalents vfft & vffts. These instructions divide the $n0 elements of $v2 into groups of 2*N complex numbers - i.e. 4*N elements, where pairs of elements represent the real and imaginary parts. The radix-2 FFT of size log2(N) is executed on each group using the first 2*N elements of $v3 as the twiddle factors, and the result is written to $v1. The pseudo-code for this operation is:

```
for(i = 0; i< $n0; i += 4*N )
        for( j = 0; j < 2*N; j+= 2 )
                wre,wim = $v3[j], $v3[j+1]
                are,aim = $v2[i+j+2*N], $v2[i+j+2N];
                re = wre*are – wim*aim
                im = wre*aim + wim*are
                $v1[i+j]      = $v1[i+j] + re
```

$v1[i+j+1] = $v1[i+j+1] + im

$v1[i+j+2N] = $v[i+j] – re

$v1[i+j+2N+1] = $v1[i+j+1] – im

[00172] The behavior of the instruction is undefined if $n0 is not a multiple of 4N. The multiplication in vffts is done using fixed-point semantics. In general, N may be restricted to powers of 2.

[00173] When the 2M is 1 in the example above, W[0] = 1, and the multiply is unnecessary. the vfft2f $n0,$v1,$v2 instruction, and its counterparts vfft2 & vfft2s may take advantage of this fact. The pseudo-code for this instruction is:

for(i = 0; i< $n0; i += 4)

$v1[i+0] = $v2[i+0] + $v2[i+2]

$v1[i+1] = $v2[i+1] + $v2[i+3]

$v1[i+2] = $v2[i+0] - $v2[i+2]

$v1[i+3] = $v2[i+1] - $v2[i+3]

[00174] After (or before) an FFT, the data may need to be reordered in a pattern called bit reversal. In bit reversal of length M, if we write out the index of a value as being at position bM-1bM-2…b1bo, then it can get moved to position b0b1…bM-2bM-1; i.e. to the index with its bits reversed, hence the name. The vswapbr_M_xN $n0,$v1,$v2 instruction and its floating point equivalent vswapbrf can be used to do this transform. They group the elements of $v2 into groups of size N, then arrange the groups into clusters of N*2M. The groups within these clusters are swapped based on bit reversal. The behavior of these instructions is undefined if $n0 is not a multiple of N*2M. The pseudo-code is:

for(i = 0; i< $n0; i += $N*2^M$)

for(j = 0; j < 2^M; j++)

for(k = 0; k < N; k++)

$$\$v1[i + j*N + k] = \$v2[\ i + \text{bitreverse}_M(j)*N + k\ ]$$

Graphics

[00175] Graphics code involves vector and matrix arithmetic, mostly on small 4x4 matrices and 4-vectors, and mostly using single-precision floating point. Instructions that can compute the dot-product of 4 vectors (vdot_x4) are discussed in previsous sections.

[00176] The vmulmmf_xN $n0,$v1,$v2,$v3 instruction assumes that $v2 and$ v3 are a collection of NxN square matrices that are multiplied together and the results are stored in $v1. $n0 may be a multiple of N2. The pseudo-code is:

```
for(i = 0; i< $n0; i += N^2 )
    for( j = 0; j < N; j++ )
        for( k = 0; k < N; k++ )
            sum = 0;
            for( l = 0; l < N; l++ )
                sum += $v2[i+j*N+l] + $v3[i+l*N+k]
            $v1[i+j*N+k] = sum;
```

[00177] The vmulmvf_ord_xN $n0,$v1,$v2,$v3 instruction assumes that $v2 is a collection of N vectors and $v3 is an NxN matrix. This instruction multiplies each of the vectors with the matrix, and writes the $n0 results into $v1. $n0 may be a multiple of N. The order of the matrix, ord, can be specified to be either column major or row-major. The order of the matrix represents how the elements of the matrix are stored in the vector $v3. For row-order, the pseudo-code is:

```
for(i = 0; i< $n0; i += N )
    for( j = 0; j < N; j++ )
        sum = 0;
        for( k = 0; k < N; k++ )
```

```
                sum += $v2[i+k] + $v3[i+j*N+k]
            $v1[i+j] = sum;
```

[00178] For column order, the pseudo code may read $v3[i+j+N*k].

[00179] Another vector operation that occurs frequently is vector normalization. In vector normalization, each N-vector is scaled so that its length is 1. This is done using the vnormalf_xN $n0,$v1,$v2 instruction. This divides the $n0 elements of $v2 into groups of N and divides them. The pseudo-code is:

```
for(i = 0; i< $n0; i += N )
    sum = 0;
    for( k = 0; k < N; k++ )
        sum += $v2[i+k] * $v2[i+k]
    scale = 1/√sum
    for( k = 0; k < N; k++ )
        $v1[i+k] += $v2[i+k] * scale
```

[00180] This can be a fairly expensive operation to compute. Because exact precision is generally not needed in graphics code,some embodiments may use an instruction of vector normalization approximation,vnormalaf,which is like the vnormalf instruction except that it produces an approximation to the actual normalized value.

[00181] Another operation that is defined in the Unity ISA to support graphics is to rearrange the data in a vector to implement a matrix transpose of a square matrix. The instruction vswaptrf_xN $n0,$v1,$v2 treats the $n0 elements of $v2 as a collection of NxN square matrices. Each of these is then transposed and the result written to $v1. The pseudo-code for this is:

```
for(i = 0; i< $n0; i += N^2 )
    for( j = 0; j < N; j++ )
```

```
for( k = 0; k < N; k++ )
    $v1[i+j*N+k] = v2[i+k*N+j]
```

[00182] The integer equivalents of the instructions described in this section are vmulmm, vmulmv, vnormal, vnormala, and vswaptr.

Vector Load/Store

[00183] Vectors are moved from memory to vector registers and back using vector loads and stores. There are different ways of specifying the memory address. In the Unity ISA, the base address is provided by an address register, denoted $a. When adapting the instructions described here to another architecture, the mechanism by which the base address is generated may need to be changed to fit in with the characteristics of that architecture.

[00184] The basic load/store moves a contiguous block of memory to/from a vector register. The instructions ldv $n0,$v1,$a2 and stv $n0,$v1,$a2 move the $n0 integer elements in $v1 from/to a contiguous block of memory starting at $a2. Floating point vectors are moved using the ldvf/stvf instructions. The pseudo code for the ldv operation is:

```
for(i = 0; i< $n0; i ++ )
    $v1[i] = *($a2+i*SIZE) // SIZE = #bytes of element
```

[00185] Strided load/stores move non-contiguous blocks of memory. Here the address for each element are computed based on the stride specified in $n3 in the instructions ldstv $n0,$v1,$a2,$n3 and ststv $n0,$v1,$a2,$n3. The pseudo-code for the ststdv instruction is:

```
for(i = 0; i< $n0; i ++ )
    *($a2+i*SIZE*$n3)= $v1[i]
```

[00186] Indexed load/stores ldixv $n0,$v1,$a2,$v3 and stixv $n0,$v1,$a2,$v3 use the elements of vector $v3 to provide offsets into the base address $a2 to provide the address for

each of the $n0 elements of $v1 that are to be moved from/to memory. The pseudo-code for the ldixv instruction is:

```
for(i = 0; i< $n0; i ++ )
    $v1[i] = *($a2+$v3[i]*SIZE)
```

[00187] The vector offset elements could be of any type. According to an embodiment of the Unity ISA, the vector offset elements are assumed to be of half-word precision.

[00188] The strided and indexed loads/stores may have a floating-point equivalent, ldstvf, ststvf, ldixvf, and stixvf.

Variations

Length register

[00189] Embodiments of the present disclosure may use an explicit length register $n0. Every vector instruction may include an explicit length field. Alternatively, selected encoding scheme may be used where the length registers are sometimes or always derived from the other information in the instruction, including, for example:

- The target register
- Either of the source registers
- The instruction opcode

[00190] Alternatively, exactly one length register may be selected to have all instructions use that length register. Other count registers may be used to deal with instructions, such as vtail, where multiple counts are needed.

[00191] Another alternative is to have a notion of an active length register. In this model, there is an instruction (say vactive $n0) that makes one of the vector count registers $n0 as the implicit input to all subsequent vector instructions till the next execution of a vactive

instruction. Note that this introduces more states into the processor, to keep track of the currently active count register.

Typing

[00192] In the architecture having been described, the types of the elements of vectors read by and produced by all vector instructions are specified. If the input type of the elements does not match the type expected by the vector instructions, the result is undefined.

[00193] A novel alternative model relies on keeping track of the types of the vector elements dynamically; every time a vector is written, the type of the elements is recorded. In this model, when a vector instruction is exectued, the recorded type of each input is compared against the expected type. If they are not the same, the input is implicitly converted to the expected input type, using some set of rules. For example, when assuming that the instruction being executed is a vaddf_d (i.e. vector add double-precision), if one of the input is a vector of single-precision floating-point elements,these single-precision floating-point elements may be converted to doubles before being used.

[00194] In this novel model, the conversion rules can either be fixed, or can be configured dynamically probably by using a control register.

[00195] Yet another alterative model dispenses with the typing for most vector instructions. In this model, a vector instruction generally does not specify any types. Instead, when it executes, it examines the recorded types of its inputs and infers both the operation and output types. Consider in this model what happens if, for example, a vadd instruction were executed:

- It reads its inputs
- If both the inputs were floating-point vectors, it would perform a floating-point add operation; if they were integers, it would perform an integer add operation.

- The output vectors type is set to the type of the input.

[00196] If the types of the input are not the same, based on the architecture definition, the resulting behavior can be undefined, or it can result in implicit conversions of values.

[00197] In this approach, some instructions may still need to specify the output types of the vectors,including those instructions where there is not enough information to infer an output type (such as vector loads) and those instructions that explicitly change the types of vectors.

Scalar registers

[00198] The disclosure so far has specified the use of accumulator and count registers that are presumably specific to the vector unit. But, other scalar register types may be used in place of these registers. For example, general purpose/integer registers may be used to specify counts, and floating-point registers may be used as the target for floating point reduction operations.

Count register vs. Constant

[00199] In various instructions, such as the group counts _xN in vilv etc, constant values are specified as part of the instruction. Alternatively, count registers may be used. Thus, instead of vilv_xN $n0,$v1,$v2,$v3, vilv $n0,$v1,$v2,$v3,$n4 may be used. When $n4 contains the value N, vilv $n0,$v1,$v2,$v3,$n4 provides identical behavior.

Others

[00200] The instructions described may be encoded in different manners other than described above, including:

- In some cases, the description may leave the range of the various fields, such as the group counts _x*N*invilv, vsumn, vdotas undefined. An architecture can choose varying subsets of values.
- In other cases, such as vpunct, a single register (e.g., $n0 in the case of vpunct) is chosen to be both an input and an output. An alternative encoding may use separate registers for this purpose – so instead of vpunct_*tf*$n0,$v1,$v2,$m3,the instruction may be vpunct_tf $n0,$n1,$v2,$v3,$m4 with $n1being the output.
- The other direction is also possible. For example, while vmuladd $n0,$v1,$v2,$v3,$v4 with $v1 the output, and $v4 as the adder input is used, vmuladd $n0,$v1,$v2,$v3 may also be used, where $v1 is both used as the adder input and used as the output target.
- The disclosure has explicitly enumerated all the registers used by an instruction. For example, in vxmulred $n0,$c1,$c2,$v3,$v4, two target accumulator registers $c1 and $c2 are explicitly named. In some embodiments, register pairs could be used, where only one of the registers is named and the other is implicitly derived. In another embodiment, the targets is fixed, so that the instruction always writes to the same two accumulators.
- In an instruction where vectors of different lengths and/or other counts are involved, the vector count register(s) specify some of the values, and the other counts are derived. It is possible to have encodings where different counts are explicitly specified. For example, in the vsumn_x*N* $n0,$v1,$v2 instruction, $n0 specifies the length of the input $v2, and the length of the output $v1 is derived to be $n0/N. Alternatively, $n0 may be used to specify the length of the output $v1, and to derive the length of the input as $n0*N.

Instruction Issue

[00201] Thissection concerns issuesrelating to implementing a variable-length long vector-register architecture, such as the Unity instruction set architecture, specifically related to instruction issue – namely, the selection of one or more instructions from instructions that are available and the dispatch of the selected instructions for execution. The issues are in the context of various implementation styles. The styles considered range from straightforward in-order, non-renaming, non-pipelined implementations to more performance-oriented out-of-order, renaming, deeply pipelined implementations.

[00202] One defining characteristic of long-vector architectures is that the number of elements in a vector, and therefore the number of operations required to be performed on that vector, exceeds the number of available functional units that can perform those operations. In each cycle, only a subset of the vector operations can start being processed. Consequently, a vector instruction may, in general, require multiple cycles to complete.

Scalar Instruction issue

[00203] Consider the 6-stage processor execution pipeline shown in **Figure 3**. Implementations of a processor pipeline can be simpler by combining various stages (e.g. decode and issue), or more complex (e.g. multiple instruction fetch and execute stages). The actual pipeline is a function of the process technology, the desired power/area/performance, and the instruction set architecture being implemented.

[00204] Consider the situation to execute two operations: mul $r0,$r1,$r2 and add $r0,$r4,$0. The first instruction:

- reads the registers $r1 and $r2 from the register file in the RF (register fetch) stage
- multiplies these values read in the EX (execution) stage
- writes the results back to register $r0 in the WB (write-back) stage

[00205] The second instruction reads the results of the multiplications and register $r4, adds these values, and writes the results back to $r0.

[00206] In a naive processor implementation, the ADD instruction cannot issue till the contents of the multiply have been written to the register file. This means that the ADDinstruction cannot execute immediately after the multiply. Instead, the ADD instructionneeds to wait for two extra cycles. Alternatively, if the register file is a write-through register file, or if it includes bypass logic, the register file of the ADD can overlap with the WB of the mul, and the add is held up for only one cycle. If the processor includes logic to bypass the results of the execution stage so that the next instruction can use it, then the ADD instruction can issue back-to-back with the mul instruction.

[00207] The instruction issue logic 204 is responsible for coordinating the issue of instructions so that they execute successfully. This includes making sure that the appropriate values are read, as shown above. It also ensures that more instructions do not try to access a hardware resource that are not supported. Some examples are:

- a processor with 1 multiplier and 1 adder. The instruction issue logic may ensure that 2 adds are not sent for execution
- a processor with a 4-stage multiplier that is not fully pipeline, and can only accept instructions every other cycle. The instruction issue logic may ensure that 2 multiplies are not sent for execution back-to-back
- a processor with 1 write port on the register-file, but multiple functional units with a varying number of execute stages. The instruction issue logic may ensure that the instructions are sent for execution in such a fashion that no two instructions complete in the same cycle.

[00208] Instruction issue can generally be divided into two classes; in-order and out-of-order. In in-order instruction issue, instructions are sent for execution in the order that they

are fetched from memory. If an instruction cannot be issued, all succeeding instructions are blocked from issue till that instruction is issued. In out-of-order instruction issue, a pool of instructions is maintained. Instructions are picked from this pool and issued for execution. If an instruction cannot be issued, but a subsequent one can, then it is possible that the subsequent instruction is executed first.

[00209] Another organization of the processor pipeline is shown in **Figure 4**. This pipeline organization is generally found in out-of-order processors. Instructions read their input values from the register file and enter a issue structure. The issue structure is part of the memory used to hold instructions while the issue logic is the logic circuitry used to select from the instructions stored in the issue structure for sending to execution. In one embodiment, both the issue structure and the issue logic is part of the issue block 204 as shonw in Figure 2. Note that some input values may not be available in the register file because the instructions that produce them have not finished execution, or are themselves waiting in the issue structure. The issue logicexamines the pool of instructions in the issue structure and, from the ones that have all their input values available and that can be executed given the available resources, picks one or more instructions for execution. When an instruction execution completes, the execution results are written back to the register file. These results are also sent to the issue structure, where any instruction that is waiting for these results copies these results, and may now be ready for issue.

Variable-length long vector register instructions

[00210] A vector register is a register capable of holding a number of values such as, for example, multiple single-precision floating point numbers or multiple bytes. These registers may be identified as $vN.A vector instruction is one which specifies operations on the vector registers, generally applying the same operation to the individual elements of the

vectors. A variable length instruction is one where the instruction specifies the number of elements that may be processed using a register, generally known as the vector-length register and denoted as $nN. For example, the instruction vadd_f $n0,$v1,$v2,$v3 has the behavior similar to the following pseudo-code:

```
for( j = 0; j < $n0; j++ )
    $v1[j] = $v2[j] + $v3[j]
```

[00211] Long vector instructions are ones where the instructions may take multiple execution cycles. For example, if the vectors can hold up to 64 floating-point numbers and there are only 8 floating point adders, the execution of the vector instruction may take at least 8 cycles to execute a maximum length vector. However, it is possible that $n0 specifies a much smaller vector. In that case, it is possible to complete the instruction using fewer execution cycles (including 0, if $n0 is 0).

Vector instruction issue

[00212] It is possible to implement vector instruction issue in a fashion similar to scalar instruction issue – the input vector registers are read in their entirety along with the vector length register, and the execution unit executes over multiple cycles, operating on subsets of the inputs, producing the corresponding results, and then writing the result to the output register. In this approach, there needs for a state machine or sequencer that may identify the subset that is to be read and written by the execution units in each of the cycles. This sequencer can always run for the maximum number of cycles, ignoring the vector-length, or, preferably, can run for a number of a cycles based on the contents of the vector length register. The behavior is captured in the following pseudo-code for the execution of vadd_f $n0,$v1,$v2,$v3:

```
// RF stage
```

```
for( j = 0; j < VECTOR_LENGTH; j++ )
        in0[j] = $v2[j];
for( j = 0; j < VECTOR_LENGTH; j++ )
        in1[j] = $v3[j];
// multiple EX stages
for( j =0; j < $n0; j += NUM_ADDERS )
        for( k = 0; k < NUM_ADDERS; k++ )
                out[j+k] = in0[j+k] + in1[j+k]
// WB stage
for( j = 0; j < VECTOR_LENGTH; j++ )
        $v1[j] = out[j];
```

[00213] This approach has the drawback that it requires that the entirety of the register file be read, intermediate input and output pipeline registers that are very wide, since they have to match the size of the register file, and have to have very wide multiplexers to select the subset of inputs the execution units can operate on.

[00214] An alternative approach is to read and write those subsets of the vector file in each cycle that is needed by the execution stage. This approach is illustrated with the following pseudo code:

```
for( j = 0; j < $n0; j += NUM_ADDERS )
        // RF stage
        for( k = 0; k < NUM_ADDERS; k++ )
                in0[k] = $v2[j+k]
        for( k = 0; k < NUM_ADDERS; k++ )
                in1[k] = $v3[j+k];
        // EX stage
```

```
for( k = 0; k < NUM_ADDERS; k++ )
        out[k] = in0[k] + in1[k];
// WB stage
for( k = 0; k < NUM_ADDERS; k++ )
        $v3[k] = out[k]
```

[00215] This illustrative pseudo codeomits several details, including pipeline overlap and the handling of the situation where $n0 is not a multiple of NUM_ADDERS, whose handling can be implemented in a suitable manner.

[00216] This approach may also include a sequencer. In one embodiment, the sequencer may be stand-alone. In an alternative embodiment, the sequencor may be incorporated into the instruction issue logic. When incorporated, all the details of instruction issue and sequencing are incorporated into one block.

[00217] One of the inputs used to efficiently execute a vector instruction is the contents of the vector length register. The contents of the vector length register suggestthe need to read the vector length register prior to issue. However, for efficiency reasons, it is desirable to read the vector length register after issue (otherwise the contents of the vector length register need to be held in the issue structure). Thus the preferred pipeline organization for the vector unit is the one similar to that shown in **Figure 5**.

Sub-vector instruction issue

[00218] As mentioned above, an instruction cannot be issued until its input values are available (or would be available, via bypassing). In long vector execution, during each execution cycle, the vector instruction only operates on a subset of the vector at a time. Therefore, the execution does not depend on the entire vector being available, but only a

subset. For a higher performance implementation, it is preferable that the instruction issue logic treat vector instructions at the sub-vector level.

[00219] Consider the case where the vector unit has two sets of execution units – 8 multipliers and 8 adders. Assume that the instructions vmul_f $n0,$v0,$v1,$v2 and vadd_f $n0,$v3,$v4,$v0 need to be executed, and that the vector lengths are the same and the results of the vector multiply are used by the vector add. Further, assume that both types of execution units take 1 execution cycle, and that $n0 is 32 . If the instruction issue logic waits for all elements of $v0 to be written before starting the vadd, as shown in **Figure 6**, it can take 12 cycles to execute.

[00220] Alternatively, if instruction issue only waits for the appropriate sub-vector of $v0, it can partially overlap the execution of the vadd with the vmul as shown in **Figure 7**, resulting in a total time of 9 cycles.

[00221] If the instruction-issue logic is out-of-order, then sub-vector issue can be adapted to allow sub-vectors of instructions to be issued for execution out-of-order with respect to each other. This can include the reordering of different sub-vectors belonging to a single instruction.

[00222] In one embodiment, the vector instruction may be split into sub-instructions each of which deala with a specific sub-vector and inserts each of this sub-instructions into the issue structure, to be dealt with independently by the instruction issue logic.This may impact the rest of the pipeline, specifically including the instruction completion and retire logic.

Implementations

[00223] The implementations are based on a design target that includes the ability to overlap execution between separate vector instructions and a small degree of out-of-order. In particular, thetarget application does not require the extreme performance enabled by

splitting instructions into sub-instructions. Instead, only one sub-vector is a candidate for execution at any one time, generally the smallest unexecuted sub-vector.

[00224] In the implementation, all registers are renamed, removing the possibility of write-write collisions in the vector register file. Further,it is assumed that the vector register file has 12 physical registers of 256 bytes which are organized into 8 sub-vectors of 32 bytes.

[00225] The instruction issue structure as shown in **Figure 8** maintains a queue of slots for potential vector instructions, sorted in instruction fetch order. When an instruction is added to the issue structure, it is assigned the next available slot at the tail of the queue. The previous stages of the pipeline are set up to guarantee that there is always a slot available. If there is no slot available, instruction fetch may have stalled. When the instruction is added, the valid bit is set to 1, and the tail of the queue is incremented. The At-Count is set to 0.

[00226] Prior to an instruction being initially inserted into the instruction issue queue, it reads the vector length registers. If the value in that vector length register is not valid, the instruction is not considered to be ready for scheduling and is marked pending. Every time a value is written back to a vector length register, the register number is compared against the vector length register number in each instruction in the issue slot that has the Pending bit set. If they match, the value is copied into the slot for that register.

[00227] When the length becomes available, the left count is set up to the number of execution cycles required. This is generally the length/8. Thus, this is a count of the number of sub-vectors remaining for that instruction. When the left-count reaches zero, the instruction is removed from the queue, and the queue is compressed. At the time the values for that sub-vector are written to the register file, the instruction completion/retire logic is informed of the completion of that instruction.

[00228] When an instruction is inserted into the issue structure, a bit-mask corresponding to each of the sub-vectors is initialized. In our case, the bit-vector is 96 bits long. This bit-

mask corresponds to the sub-vectors that need be available in order for the instruction to be issued against the first sub-vector. Generally, for an N-input instruction, it has N bits set, corresponding to the first sub-vector read for each input register. However, there may be some instructions that read some number of values larger than a sub-vector from one argument, and then repeatedly apply them against sub-vectors from the next argument. In that case, multiple bits corresponding to the sub-vectors for the first argument, and 1 bit for the first sub-vector for the second argument may be set.

[00229] The issue logic maintains a similar bit-mask of the unavailable sub-vectors. This works in co-operation with the rename logic and the register-file write-back logic. Every time a physical vector register is allocated to an instruction, all 8 of the sub-vectors corresponding to that vector register are marked as unavailable (i.e. set to 1). When an instruction writes back to a sub-vector, the corresponding bit is cleared.

[00230] Every cycle, the issue logic examines all instruction in the slots. For each slot, it checks if

- the slot is valid
- the vector length is not pending
- the vector count is not zero
- the bitwise-and of the sub-vector read bit-mask and the unavailable sub-vector bitmask is all zeroes

[00231] It selects the oldest instruction (i.e. the one closest to the head) in the queue that satisfies these conditions for execution.

[00232] When an instruction is sent for execution, its count is decremented, and the sub-vector read-mask is adjusted. Generally, this involves shifting the some of the bits by 1.

[00233] When the vector count is 0, and the instruction is removed from the queue, the unavailable bits for its output vector register are all cleared. This takes care of the situation where the vector length is such that not all sub-vectors are written.

Vector Register Implementation

[00234] This section considers the issues reltating to implementing a variable-length long vector-register architecture, such as the Unity instruction set architecture, specifically related to the register file. This section examines these issues in the context of various implementationstyles. The styles considered range from straightforward in-order, non-renaming, non-pipelined implementations to more performance-oriented out-of-order, renaming, deeply pipelined implementations.

[00235] One defining characteristic of long-vector architectures is that the number of elements in a vector, and therefore the number of operations required to be performed on that vector, exceeds the number of available functional units that can perform those operations. In each cycle, only a subset of the vector operations can start being processed. Consequently, a vector instruction can, in general, require multiple cycles to complete.

Conventions

[00236] There are assumed to be 4 kinds of registers that are specific to the vector unit, including:

- Vector registers, written as $v*N*, which contains vectors of elements, such as bytes or floating-point single-precision numbers;
- Vector count registers, written as $n*N*, which contain the count of the number of elements that are to be operated on;

- Vector mask registers, written as $m*N*, which contain vectors of single bits, generally used as the output of vector compares, or to condition the element-wise behavior of vector instruction; and
- Vector accumulator registers, written as $c*N*, which contain are used to hold the scalar results of vector reduction operations such as dot-products, and are used in other situations where a scalar input/output is needed for vector operations.

[00237] The implementation of the register file both with and without renaming is examined. In a direct implementation, i.e. one without renaming, there is a one to one mapping between architected and physical registers. In a renamed implementation, the two are different. In the contexts where it is desirable to distinguish between an architected register and a physical register of the same type, lower and upper cases are used. Thus, in an implementation with renaming, if vector register number three is mapped to the physical register 11, we shall use $v3 and $V11, respectively.

Direct

[00238] In the absence of renaming, there is a 1-to-1 mapping between architected and physical registers. Thus, an instruction such as vaddf $n0,$v0,$v1,$v2 may read physical registers $V1 and $V2 and write physical register $V0. This section examines the issues that arise in handling exceptions in a direct implementation.

Imprecise Exceptions

[00239] Consider the case where the execution of one of the sequence of operations during the execution of a vector instruction can cause an exception, such as a divide-by-zero when dividing two elements while doing a vector floating-point divide, or an overflow/underflow when adding two elements during a vector floating point add.

[00240] One scenario is that no element of the vector output register can have been written before an exception is written. This is conceivably the case of a deeply-pipelined operation such as floating-point divide. If assuming a 20 stage floating-point divide pipeline which detects divide-by-zeroes in the first stage, 4 divide units, and a maximum of 64 element vector, then when the divide-by-zero exception of the last element in the vector is detected, the first element of the vector is still several stages away from completion, so the entire instruction can be aborted before the output register is modified.

[00241] The other, more likely, scenario is one where the exception occurs at a point where some of the outputs of the vector instruction have been written but not all. In that case, at the point where the exception occurs, the output register includes a mixture of new values, produced by the vector instruction and old values. This situation where the state of the processor at the point of an exception is intermediate between the execution of an instruction and its completion is known as an imprecise exception. Such an exception, including the methods for fixing up the exception and resuming, may be handled using any suitable method.

[00242] One corner case when dealing with precise exceptions arises when the same register is used both as an input and as an output, and there is a possibility that the input value is needed even after the output has been overwritten. Consider the Unity ISA instruction vmulmvf $n0,$v1,$v2,$v3. This instruction assumes that its second input, $v3, holds an N-by-N matrix and its first input $v2 holds multiple N-vectors. It repeatedly multiplies each of the N-vectors in $v2 with the N-by-N vector in $v3, and writes the results to $v1. Now, if the same register is used for both output and the matrix, as in vmulmvf $no,$v1,$v2,$v1, the matrix may be overwritten by the results.

[00243] One solution is to have an implementation that buffers the matrix in an internal register prior to beginning the computation, and have all matrix-vector multiplies use that

internal buffer register for the matrix input. The issue with this solution is that if one of the intermediate computations takes an exception, at this point, the original value of $v1 has been (partially) overwritten. To be able to restart after fixing up the instruction, one would have to be able to recover the original matrix, conceivably by exposing the internal buffer register.

[00244] Another solution is to prohibit such instructions, either globally (so that no output register can ever be the same as an input), or in selected cases (so, allow the output of vmulmvf to be the same as the vector input, but not the matrix input).

[00245] In general, allowing imprecise exceptions has several undesirable consequences, including

- very complex exception recovery code
- addition of internal state to aid in recovery
- exposure of implementation specific internal state

Precise Exceptions

[00246] By contrast, on a processor with precise exceptions, at the point that the exception handler is entered after an exception caused by an instruction, the processor state appears as though the instruction had never been executed.

[00247] In the context of long vector instructions, one method for implementing precise exceptions is to have the instruction temporarily to write any results to an intermediate register until all elements have been processed, and then copy them to the output register. If an exception occurs in the middle, the copy is discarded, leaving the output register with its original contents.

[00248] Alternatively, one could copy the original contents of the output register to an internal register. If an exception occurs, the internal register is copied back to the internal

register. If the execution succeeds, then the internal register contents can be discarded. The copying of the original contents of the output register to the internal register can be done in bulk prior to the execution of the instruction, or more lazily as the contents of the output register are overwritten. If the second approach is used, only the overwritten parts of the output register are copied back from the internal register if an exception occurs.

[00249] One advantage of imprecise exceptions in vector instructions arises in a situation where one wants to redo the vector instruction after handling the exception. With precise exceptions, it is necessary to start at the first element and redo any previously done work. With imprecise exceptions, after handling the exception, it is generally possible to start at the element that caused the exception, saving the need to redo all the previous work. This may potentially be a large savings.

Renaming

Renaming vector registers

[00250] Embodiments of the present disclosure may adapt the technique of register renaming to long vector registers. In register renaming, an architected register, i.e. the registers named in an instruction, are mapped to physical registers. There are more physical registers than architected registers. Before an instruction is executed, its input registers are mapped to physical registers, and its output registers are mapped to a new, unused, physical register. The physical registers may be recycled according to known techniques to provide free physical registers.

[00251] As an example, assume that the architected and physical registers have a 1-1 relationship, so that $v0 is mapped to $V0 , $v1 to $V1 etc. Further, assume that there are 8 architected and 12 physical registers. **Figure 9** shows the mapping before and after executing a sequence of instructions. First, instruction vadd $n0,$v0,$v1,$v2 is executed,

mapping architected register $v1 to physical register $V1 and $v2 to $V2, and allocating a new register to $v0, say $V8. So, the instruction that is executed is equivalent to vadd $n0,$V8,$V1,$V2 (ignoring any renaming for $n0). Next, the instruction vmul $n0,$v1,$v2,$v0 is executed. Now, $v2 is still mapped to $V2, but architected $v0 is actually physical $V8. A new register for $v1 is allocated, say $V9. The instruction that gets executed is vmul $n0,$V9,$V2,$V8. Finally, vsub $n0,$v0,$v1,$v0 is executed. $v0 and $v1 at this point are mapped to $V8 and $V9, and $v0 is now allocated a new register, say $V10, resulting in the effective instruction vsub $n0,$V10,$V9,$V8.

Exception Handling

[00252] Obviously, with renaming, obtaining precise exceptions is easy to accomplish. With renaming, the output register is an otherwise unused register that is physically different from the input register, even if it has the same architected register names. When an exception occurs while executing a vector instruction, the original state can be obtained by rolling back the mapping to its value before the exception occurred.

[00253] In current register allocation schemes, when an exception occurs, the new output register is freedbecause the results are no longer useful. However, in the case of vector registers, the partially completed results may be valuable, either in debugging, to help diagnose the cause of the exception, or to avoid having to re-compute these partial results after restart on an exception. In one embodiment, these partial results are preseveredto expose them to the exception handling mechanism. This can be done in one of several ways:

- Copy the contents of the output register to a different register, presumably a special-purpose non-renamed register, and then proceed to free the register as usual
- Prevent the register from being freed immediately, and provide a means for the exception handling code to access its contents. One way of doing this is to have

another architected register name that the mapping logic may map to this register. There also exists a mechanism to free the register once the exception processing and recovery logic is done with its contents.

Name reuse

[00254] In current register allocation schemes, every output register is allocated a new, unused, physical register. However, there are situations in which this is not desirable. For instance, consider the Unity ISA instruction vappendc $n0,$v1,$c2 that inserts the contents of $c2 into the vector register $v1 at position $n0 so that $v1 is both an input and an output. If the output physical register for $v1 is different than the input physical register, the elements of the input physical register (other than the one at $n0) are copied to the outputphysical register. This can be quite expensive. To modify one element in the vector, it is desirable to effectively copy the vector.

[00255] Embodiments of the present disclosure include a renaming mechanism that does not always create a new mapping for output registers, but instead preserves the mapping for some instructions. In the context of long-vector register implementations, this is useful for instructions where the output register is also an input register, and only a subset of the values of the input are modified.

Register Banks

Segmentation

[00256] The total memory required to implement a large-register-vector is going to be quite large. For example, if there are eight physical vector registers each with 256 bytes, the memory needed is 4K bytes. However, given that a vector instruction is to complete over multiple cycles, not all contents of a vector registers are needed at once.

[00257] In one embodiment, each register is split into segments so that a vector instruction typically processes at most a segment's worth of data at a time. For example, a 256-byte register can broken into 8 segments of 32 bytes each. A single-precision floating point add instruction vaddf _s $n0,$v0,$v1,$v2 can read the 8 single precision numbers in the first

segment of $v1 and $v2, add them together, and write the results back to the first segment of $v0. Then the second segment and so on are similarly processed. If $n0 is smaller than the maximum value (64 in this case), it might not be necessary to read all segments. For instance if $n0 is 8, only the first segments of $v1 and $v2 need to be read and processed.

[00258] If the number of functional units matches the number required to process acomplete segment, the processing may read the input's segments, and start the functional units. If the number of functional units is less than that number, multiple cycles may be needed to consume the entirety of the segments. The implementation can:

- Reread the segment, choosing different subsets, until the entire segment has been processed, and/or
- Buffer the segment in an internal register, and read repeatedly from that internal register until it has been fully processed.

[00259] If the number of functional units exceeds the number required to process a complete segment and it is desired to keep them busy, the implementation may need to read and write multiple segments.

[00260] Another case where it is necessary to read/write multiple segments is in the case of those vector instructions which operate simultaneously on non-contiguous elements from the same register, where it is possible for the non-contiguous elementsto come from different segments of the register. For instance, in the Unity ISA, the instruction vfftf_s_x8 $n0,$v1,$v2,$v3 specifies that the implementation perform a radix-2 FFT over 8 elements of $v2, where the elements are complex single precision floating point numbers. In this case, to perform the first FFT, the instruction may need to read the bytes [0…7] and bytes [64…71] simultaneously. Thus, to perform this operation, the instruction needs to read two 32 byte segments.

Memory Arrays

[00261] One way of implementing these register files is to use memory arrays. A memory array may includea number of rows, each containing a certain number of bits. When arow is addressed, the bits of that row can be read or written to. The number of bits in a row is called the width of the array.

[00262] A memory array can permit multiple rows to be read and/or written in the same cycle. An array that supports N simultaneous accesses in the same cycle is called an N-ported array. Some ports can be restricted to doing either a read or a write. For example, a so-called "dual ported array" can permit at most one row to be read and one row to be written in the same cycle. So, the "dual ported array" includes one read port and one write port. A "two port array," by contrast, has two ports that can be used for either a read or a write. Thus, a "two port array" can do two reads, or two writes, or a read and a write in the same cycle. As the number of ports increases while keeping the number of rows and width constant, the size of the array increases and the performance goes down.

[00263] There is a tradeoff between the width of a memory array and its power/performance. At a certain design point, for a specific number of rows, there is a maximum width that can be achieved. A number of smaller memory arrays may be grouped together and accessed in parallel to to build what appears to be a wider memory array. For the purposes of discussion, it is assumed that it is possible to build arrays of a desired width desired either directly or by grouping smaller arrays.

Organization

[00264] In one embodiment, the register file is to use a memory array with one segment per row that is effectively one segment wide. Thus, with for a register file with 12registers

that have 4 segments of 32 bytes, we could implement it as a memory array with 48 rows of 256 bits (48x32bytes).

[00265] An alternative implementation technique is to use multiple memory arrays. Thus, the example above could be implemented with 2 memory arrays, each containing 24 rows, or 3 memory arrays with 16 rows. Because each of these arrays contain fewer rows, the multiple memory array provides for the advantages of faster speed and lower power.

Read Ports

[00266] If the instruction-set architecture includes instructions that can read three segments and write 1 segment, such as vector multiply add vmuladd $n0,$v1,$v2,$v3, for full performance, the single array implementationmay need a 3-read, 1-write port register file. An interesting peculiarity of this kind of vector instruction is thatthe instruction reads the same segments of all three registers. The instructioncan first read the first segment of all three registers, then the second, etc. A multi-bank implementation where the ith segment of registers are distributed to different arrays so that no more than two registers have the same segment in the same array can cut the read port requirement down to two. This may require, at a minimum, that the number of arrays be half of the number of registers.

[00267] Embodiments of the present disclosure can organize the 48 segments as shown in the **Figure 10**. The notation $vN.I means the Ith segment of vector register N. Each column corresponds to a separate memory array, and each row to a row within that array. The number of rows is twice the number of segments.

[00268] As shown in **Figure 10**, there are only 2 instances of each segment in any column. So, to read the inputs to a vmulred, at most 2 read ports is needed. This assumes that all the input registers are different. If the instruction specifies the same register twice (or three

times), such as vmulred $n0,$v1,$v9,$v9,$v3, the hardware may recognize this case access only two registers (or one) and replicated it as needed.

[00269] Another case where the straightforward register file implementation may require the register file to need 3 read ports is when the instruction needs to access multiple segments of a register, as well as some other segment of another register (as in the vfft example above).

[00270] Again, multiple arrays may be used to reduce the read port requirement to two. As long as all the segments of a register are in different arrays, different segments of the register can be read in the same cycle without increasing the number of register ports. This requires that the number of banks be at least as large as the number of segments. Note that in the example above this condition is satisfied.

[00271] In the FFT case, the segments that are to be accessed simultaneously are a power of two apart. Assuming thatthere are eight segments and depending on the size of the FFT, the instruction may access segment I and segments I+1, I+2 or I+4 simultaneously. As long as there are at least three memory banks and the number of memory arrays is not a power of 2,it is possible to arrange the segments so that this power of two access pattern never requires two accesses to different rows of the same memory bank for the same register.

Write ports

[00272] It may be possible to combine the write port(s) with the read ports, and decrease the overall number of ports in the memory arrays by implementing a register file in one of several ways.

[00273] In one embodiment, the register file is implemented by stalling instruction execution to all writes to occur so that when an instruction is about to write, it prevents any register reads by the current or any other instruction, and delays the read by a cycle.

[00274] A variation of this approach is when the memory arrays have 2 read ports and a single read/write port. In that case, the stall only occurs when attempting to issue a 3 input instruction at the same time as a write. In that case, the 3-input instruction is delayed. There is no impact on 2 input instructions.

[00275] Embodiments of the present disclosure may combine the stall approach with multiple banks. In this case, the memories may have 1 read port and 1 read/write port. If an instruction is attempting to read 2 rows from the same bankand at the same time, a value is being written to that bank, the instruction may get delayed by a cycle.

[00276] It is also possible to control write-port conflicts, on an implementation with register renaming, by having the register renaming logic examine the physical registers corresponding to the inputs, and then allocating a physical register that does not conflict with them.

[00277] For example of the 12 register in 4 segments, assume that the segment/bank layout is as in the example, and assume, for the purpose of this example, that all instructions take one cycle. For an instruction such as vmuladd, this means that segment 1 of the 3 inputs is being read at the same time as segment 0 of the output is being written. There can be at most one array whose ports are both being used for reads. If the output register is allocated such that its 0th segment is not in that port, there may be no collision. Assume that the instruction is vmulred $n0,$v7,$v0,$v1,$v3 and that $v0,$v1,$v3 are mapped to $V0,$V1 and $V3. $V0 and $V3 read from the memory array every cycle. In the second cycle, they read from bank 1. As long as the register rename logic does not map $v0 to $V6 or $V9, there is no collision between the writes and the reads.

Properties

Type

[00278] An architecture may specify that behavior of an instruction as undefined if the type of the elements last written to a register does not match the type that is expected by the instruction.

[00279] One option is to ignore this situation, and interpret the contents of the register as though they were elements of the expected type. This option can result in less expensive hardware, but makes it harder to detect certain kinds of bugs in programs.

[00280] Another option is to detect this mismatch and take action, preferably by raising an exception. In this case, the type information is stored with each register. For the purposes of context switching, this information may be accessible by the code that saves and restores register state, so that it can also save and restore the type information.

Length

[00281] An architecture may define that behavior of an instruction to be undefined if the number of elements that was last written to a register is less than the number that is to be read by the instruction.

[00282] One option is to detect this mismatch, and take corrective action, preferably by raising an exception. In this case, the length information is stored with each register. For the purposes of context switching, this information may be accessible by the code that saves and restores register state, so it can also save and restore the length information.

[00283] An alternative approach is to ignore the length mismatch, and use the contents of the register. In particular, the value of the elements beyond those written by the last instruction to write the register may depend on the values written by previous instructions to

write the register. These may not be predictable in the situation where these values were written by instructions other than the current program, leading to unacceptably variable results.

[00284] Another alternative approach to have an instruction to overwrite the entire register every time the register is written, setting the elements that would not otherwise be written by an instruction to a default value. This default value can be a NaN or some other predefined value, or it can be the contents of a configuration register.

[00285] Alternatively, a length is stored with each register (as described above), usingthe stored lengthto detect attempts to read beyond the elements written. At this point, the value returned could be the default value.

[00286] A hybrid approach is to break a vector register into a certain number of sections, and keep track of the number of sections written by an instruction, preferably using a bit-mask. When a vector instruction writes to a vector register, the number of sections written is recorded. If the last section is only partially written, then the trailing elements of that section are set to the default value.

[00287] Note that in this approach, the context switch code does not need to save any length registers. The context switch code, when reading the register, may attempt to read the entire register. This includes the elements last written to this register, and possibly some trailing elements containing default values. Then, when the vector register is restored, all of it can be overwritten, but the same set of trailing elements can contain default values.

Variations

[00288] Althoughexemplary instructions with at most 3 inputs and 1 output are discussed in this disclosure, the same techniques can be applied, without limitation, to instructions with more inputs and outputs. Similarly, although the disclosure describes to cut the port

requirement down from 3 read+1write port to 1 read+1 read/write port in a vector register file by multi-banking, similar techniques can be used to achieve savings in other contexts.

Implementation

[00289] The implementation is based on a design target that includes the need to overlap execution between separate vector instructions and a small degree of out-of-order, supporting precise exceptions and minimizing the implementation cost.

[00290] One embodiment may include the implementation of a renamed register file that names 8 architected registers to 12 physical registers. The physical registers are 256 bytes long, divided into 8 sub-vectors of 32 byte each. The registers are implemented using 6 memory banks, built out of smaller arrays so that they are effectively of size 16x512bytes.

[00291] In one embodiment, the type or length tags are not kept on each register. But, returning a default value for undefined parts of the register file is supported.

[00292] The memory banks are built using four 16x128b arrays, with various control connected in common, and separate data signals. This is shown in **Figure 11**. Each of these banks includes a read/write port A and a read-only port B. The read-enable for port A (RENA), write-enable for port A (WENA), the row address for port A (ADRA), the read-enable for port B (RENB), write-enable for port A (WENB), the row address for port B (ADRB), and the clock (CLK) are common to all memory arrays. The write data input for port A (DA) and the two read data outputs for ports A & B (QA,QB) are distributed across all the memories arrays, as shown in **Figure 11**.

[00293] Six memory banks formed from these arrays are combined as shown in **Figure 12**. The decode logic can receive up to 3 reads and 1 write requests for subvectors every cycle,resulting in different subvectors being enabled for reading and writing. The decode logic may convert these requests into the read and write enables as well as addresses for the

2 ports on each of the 6 banks. Thus, up to 4 banks can be activated in a cycle. Each of the 3 outputs is connected using a 512byte wide 7-to-1 mux to the 6 banks plus to the default value as described below.

[00294] The logic for mapping subvector N in register R is:

```
if( R < 6 )
        bank = (R*8+N)%6
        addr = (R*8+N)/6
else
        bank = (R*8+N+1)%6
        addr = (R*8+N)/6 + 8
```

[00295] In addition, embodiments provide for a default value for undefined subvectors. There is an array of 96 bits used to indicate if the corresponding sub-vector has valid contents. It is implemented using flops that are set to 1 if the corresponding sub-vector was last written during an operation. It is cleared on chip reset, and when the register is first allocated during renaming. When a sub-vector is written, its bit is set. When a sub-vector is read, and its bit is 0, the value provided is read from the default value register. This is a 64 bit register, whose contents can be replicated to 512 bits.

Implementation Variations

[00296] Implementations can differ from each other in the fine-grain micro choices of the exact pipeline organization and the technology picked, etc. However, this section examines coarser variations in the nature of the implementation as descibed above section including possible implementation variations such as:

- different vector register sizes,

- different vector registers for different types, possibly with different number of elements,
- different mixtures of functional units,
- different unimplemented vector instructions (*e.g.* instructions that are implemented entirely in software),
- partial implementation (*e.g.* some instruction variants are implemented in hardware, but others cause interrupts and are implemented in software).

Vector-unit less implementation

[00297] One possibility for an extremely low cost implementation is one that omits the vector unit, causing an interrupt to be taken every time a vector instruction is selected for execution. In this case, the vector-unit is implemented exclusively in software, and all vector instructions are executed via emulation.

Multi-threaded implementation

[00298] There are many styles of multi-threaded implementation, including round-robin, hyper-threading, symmetric multi-threading, etc. In the context of the vector unit, the techniques of multi-threading apply:

- There is a copy of the architected register file for each of the threads that are supported by the processor;
- In an in-order multi-threaded processor, there is a separate register file for each thread;
- In multi-threaded processor with renaming, each thread may have its own pool of registers for renaming, or there may be a common pool and registers are renamed from that pool;

- All threads share using the same functional units for executing their vector instructions.

Asymmetric multi-threading

[00299] Embodiments of the present disclosure include asymmetric multi-threading this is a version of multi-threading in which not all threads have equal access to resources in the processor. In the specific context of an implementation with a vector processor, this would mean that only some threads would have access to the vector unit.

[00300] In a vector unit, a considerable area is consumed by the vector register file. In a multi-threaded implementation, each additional thread increases the number of required registers, causing the area of the register file to grow. Larger area increases costs and can impact cycle time.

[00301] Further, not all threads require the vector unit. For example, in the case where there is a 2-way multi-threaded processor in which one thread handles I/O interrupts, and the other thread is performing number crunching, the interrupt handling thread does not need any vector processing. In this case, it is fine with only one thread having accessing to the vector unit.Consequently, if the embodiment includes a multi-threaded processor in which some threads have access to the vector unit, and others do not, it only needs to implement enough vector register states to satisfy the requirements of the threads that do have access, thereby saving on the register area. For instance, if there are a 3-threaded processor where only 1 thread was allowed access to the vector unit, the implementation only needs to have vector registers for only 1 thread as opposed to 3, resulting in a large savings. If it turned out that only one program at a time needed to use the vector unit, this saving comes with no loss in performance.

Static vs. Dynamic Asymmetric Multi-threading

[00302] In the case of static multi-threading, certain hardware threads have access to the vector unit, and others do not. The allocation of hardware threads is fixed by the hardware. If a program running on a thread with no access to the vector unit wants to start executing vector instructions, it is swapped out of the processor, and then swapped back in onto a thread that does have vector access.

[00303] In the case of dynamic multi-threading, the processor can be configured to allow different threads to the vector unit so that at any one time only a subset of all threads have access to the vector unit, but that subset can be changed. In this particular case, if a program running on a thread with no access to the vector unit wants to start executing vector instructions, the hardware can be reconfigured to allow that thread access to the vector unit.

[00304] In the case of dynamic multi-threading, the program can be assigned to access to the vector unit directly if no other program is using a vector unit, or it can be made to wait till a vector unit becomes available when a program that is using the vector unit releases it, or some other thread can be forced to release its vector unit.

[00305] Generally, when a program releases a vector unit, this is similar to being swapped out during a context switch. The vector registers being used by the program are saved, generally by the operating system, in an area that is specific to that program. When a program acquires a vector unit, this is similar to being swapped in during a context switch. The vector registers are loaded, generally by the operating system, from an area specific to that program.

Generalization

[00306] This section describes asymmetric multi-threading as applied to the vector unit, in which only a subset of threads are allowed simultaneous access to the vector unit, thereby

allowing the implementation to only hold the vector state for those threads, as opposed to holding the vector state for all threads. Given the size of the vector registers and the area of the vector register files, this results in a substantial savings.

[00307] However, this idea has more general applicability. For example, in any multi-threaded processor, it is possible to allow threads to have asymmetric access to processor resources, and thereby save the cost of having all threads be able to access all resources. For example, in a multi-threaded implementation for an embedded processor, it may make sense to do the same for the floating point unit – say, have 4 threads, but only one with access to the floating point unit, and therefore only have one set of floating point registers.

[00308] In an example, a vector (computer) processor 100 is provided. The computer processor may comprise a vector unit 108, 200 comprising a vector register file 206 comprising at least one vector register that may hold a varying number of elements. The computer processor 100 may further comprise processing logic configured to operate on the varying number of elements in the vector register file 206 using one or more instructions that produce results with elements of widths different than that of the input elements. The computer processor 100 may be implemented as a monolithic integrated circuit.

[00309] In an example, the computer processor 100 may further comprise a vector length register file 208 comprising at least one register, where theat least one register of the vector length register file 208 may be used to specify the number of elements on which the processing logic operates.

[00310] In an example, the processing logic may be further configured to: read two vector registers from the vector register file 206, treat the contents of the two vector registers as two input vectors of input integer elements of a bit-size, element-wise multiply the elements of the first vector register with the corresponding elements of the second register to produce an output vector of output integer elements, the size of each output integer element having a

size that is twice the number of bits of a corresponding input integer element, and store the output vector of integer elements to a vector register in the vector register file 206. In an example, the size of each of the input integer elements may beone byte, and the size of each of the output integer elements may be two bytes. In an example, the size of each of the input integer elements may be two bytes, and the size of each of the output integer elements may be four bytes. In an example, the size of each of the input integer elements may be four bytes, and the size of each of the output integer elements may be eight bytes.

[00311] In an example, the processing logic may be further configured to: read a vector register from the vector register file 206, treat the contents of the vector register as an input vector of input integer elements of a smaller bit-size, convert the input vector of input integer elements into an output vector of output integer elements of a larger bit-size, and store the output vector of output integer elements to a vector register in the vector register file 206. In an example, the computer processor 100 converting from a smaller bit-size to a larger bit-size may comprise the computer processor appending zeroes to upper bits of each of the input integer elements. In an example,the computer processor 100 converting from a smaller bit-size to a larger bit-size may comprise the computer processor 100 appending zeroes to the lower bits of each of the input integer elements. In an example, the computer processor 100 converting from a smaller bit-size to a larger bit-size may comprise the computer processor 100 appending copies of the highest-bit of the input element to the upper bits of the integer element for each of the input integer elements.

[00312] In an example, the size of each of the input integer elements may be one byte, and the size of each of the output integer elementsmay be two bytes. In an example, the size of each of the input integer elements may be two bytes, and the size of each of the output integer elements may be four bytes. In an example, the size of each of the input integer

elements may be four bytes, and the size of each of the output integer elementsmay be eight bytes.

[00313] In an example, the processing logic may be further configured to: read a vector register from the vector register file 206, treat the contents of the vector register as an input vector of input integer elements of larger bit-size,convert the contents of the vector register into output integer elements of a smaller bit-size, and store the resulting output vector of output integer elements of a smaller bit-size to a vector register in the vector register file 206. In an example, the computer processor 100 converting from input integer elements of a larger bit-size to corresponding output integer elements of a smaller bit-size may comprise the computer processor 100 discarding the high bits of each of the input integer elements. In an example, the computer processor 100 converting from input integer elements of a larger bit-size to corresponding output integer elements of a smaller bit-size may comprise the computer processor 100 discarding the low bits of each of the input integer elements.

[00314] In an example, the computer processor 100 converting from input integer elements of a larger bit-size to corresponding output integer elements of a smaller bit-size may comprises the computer processor 100 representing an integer value in the smaller bit-size, converting the integer value to the largest number representable at the smaller bit-size if the original integer value is larger than the largest number representable, or converting the integer value to the smallest number representable if the original integer value is smaller than the smallest number representable.

[00315] In an example, the bit-size of each of the input integer elements may be two bytes and the bit-size of each of the output integer elements may be one byte. In an example, the bit-size of each of the input integer elements may be four bytes and the bit-size of each of the output integer elements may betwo bytes. In an example, the bit-size of each of the

input integer element may be eight bytes and the bit-size of each of the output integer elements may befour bytes.

[00316] In an example, the processing logic may be further configured to: read a vector register from the vector register file 206, treat the contents of the vector register as an input vector of input floating-point elements of a smaller bit-size, convert the input floating-point elements of the vector register into an output vector of output floating-point numbers of a larger bit-size, and store the resulting output vector of output floating-point elements of a larger bit-size to a vector register in the vector register file 206. In an example, the size of each of the input floating-point elements may be two bytes, and the size of each of the output floating-point elements may be four bytes. In an example, the size of each of the input floating-point elements may be four bytes, and the size of the each of the output floating-point elements may be eight bytes.

[00317] In an example, the processing logic may be further configured to: read a vector register from the vector register file 206, treat the contents of the vector register as an input vector of input floating-point elements of a larger bit-size, convert the contents of the vector register into floating-point elements of a smaller bit-size, and store the resulting output vector of output floating-point elements of a smaller bit-size to a vector register in the vector register file 206. In an example, the bit-size of each of the input floating-point elements may be four bytes and the bit-size of each of the output floating-point elements may be two bytes. In an example, the bit-size of each of the input floating-point elements may be eight bytes and the bit-size of each of the output floating-point elements may be four bytes.

[00318] In an example, the computer processor 100 may be further configured to control rounding and saturation modes. In an example, the value of a smaller floating-point number may be obtained from a larger floating-point number by rounding and saturation defined by

the rounding and saturation modes. In an example, the rounding and saturation modes may be defined as per an IEEE standard.

[00319] In the foregoing description, numerous details are set forth. It will be apparent, however, to one of ordinary skill in the art having the benefit of this disclosure, that the present disclosure may be practiced without these specific details. In some instances, well-known structures and devices are shown in block diagram form, rather than in detail, in order to avoid obscuring the present disclosure.

[00320] Some portions of the detailed description have been presented in terms of algorithms and symbolic representations of operations on data bits within a computer memory. These algorithmic descriptions and representations are the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. An algorithm is here, and generally, conceived to be a self-consistent sequence of steps leading to a desired result. The steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like.

[00321] It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise as apparent from the following discussion, it is appreciated that throughout the description, discussions utilizing terms such as "segmenting", "analyzing", "determining", "enabling", “identifying,” "modifying" or the like, refer to the actions and processes of a computer system, or similar electronic computing device, that manipulates and transforms data represented as physical (e.g., electronic)

quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission or display devices.

[00322] The disclosure also relates to an apparatus for performing the operations herein. This apparatus may be specially constructed for the required purposes, or it may include a general purpose computer selectively activated or reconfigured by a computer program stored in the computer. Such a computer program may be stored in a computer readable storage medium, such as, but not limited to, any type of disk including floppy disks, optical disks, CD-ROMs, and magnetic-optical disks, read-only memories (ROMs), random access memories (RAMs), EPROMs, EEPROMs, magnetic or optical cards, or any type of media suitable for storing electronic instructions.

[00323] The words "example" or "exemplary" are used herein to mean serving as an example, instance, or illustration. Any aspect or design described herein as "example' or "exemplary" is not necessarily to be construed as preferred or advantageous over other aspects or designs. Rather, use of the words "example" or "exemplary" is intended to present concepts in a concrete fashion. As used in this application, the term "or" is intended to mean an inclusive "or" rather than an exclusive "or". That is, unless specified otherwise, or clear from context, "X includes A or B" is intended to mean any of the natural inclusive permutations. That is, if X includes A; X includes B; or X includes both A and B, then "X includes A or B" is satisfied under any of the foregoing instances. In addition, the articles "a" and "an" as used in this application and the appended claims should generally be construed to mean "one or more" unless specified otherwise or clear from context to be directed to a singular form. Moreover, use of the term "an embodiment" or "one embodiment" or "an implementation" or "one implementation" throughout is not intended to mean the same embodiment or implementation unless described as such.

[00324] Reference throughout this specification to "one embodiment" or "an embodiment" means that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one embodiment. Thus, the appearances of the phrase "in one embodiment" or "in an embodiment" in various places throughout this specification are not necessarily all referring to the same embodiment. In addition, the term "or" is intended to mean an inclusive "or" rather than an exclusive "or."

[00325] It is to be understood that the above description is intended to be illustrative, and not restrictive. Many other implementations will be apparent to those of skill in the art upon reading and understanding the above description. The scope of the disclosure should, therefore, be determined with reference to the appended claims, along with the full scope of equivalents to which such claims are entitled.

CLAIMS

What is claimed is:

1. A computer processor, comprising:

a vector unit comprising:

a vector register file comprising at least one register to hold a varying number of elements; and

processing logic configured to operate on the varying number of elements in the vector register file using one or more instructions that produce results with elements of widths different than that of the input elements.

2. The computer processor of 1, wherein the computer processor is implemented as a monolithic integrated circuit.

3. The computer processor of claim 1 or 2, further comprising a vector length register file comprising at least one register, where theat least one register of the vector length register file is used to specify the number of elements on which the processing logic operates.

4. The computer processor of any of claims 1 to 3, wherein the processing logic is further configured to:

read two vector registers from the vector register file;

treat the contents of the two vector registers as two input vectors of input integer elements of a bit-size;

element-wise multiply the elements of the first vector register with the corresponding elements of the second register to produce an output vector of output integer elements, the size of each output integer element having a size that is twice the number of bits of a corresponding input integer element; and

store the output vector of integer elements to a vector register in the vector register file

5. The computer processor of claim 4, wherein the size of each of the input integer elements is one byte, and the size of each of the output integer elements is two bytes.

6. The computer processor of claim 4 or 5, wherein the size of each of the input integer elements is two bytes, and the size of each of the output integer elements is four bytes.

7. The computer processor of any of claims 4 to 6, wherein the size of each of the input integer elements is four bytes, and the size of each of the output integer elements is eight bytes.

8. The computer processor of claim 1, wherein the processing logic is further configured to:

read a vector register from the vector register file;

treat the contents of the vector register as an input vector of input integer elements of a smaller bit-size;

convert the input vector of input integer elements into an output vector of output integer elements of a larger bit-size; and

store the output vector of output integer elements to a vector register in the vector register file.

9. The computer processor of claim 8, wherein converting from a smaller bit-size to a larger bit-size comprises appending zeroes to upper bits of each of the input integer elements.

10. The computer processor of claim 8 or 9, wherein converting from a smaller bit-size to a larger bit-size comprises appending zeroes to the lower bits of each of the input integer elements.

11. The computer processor of any of claims 8 to 10, wherein converting from a smaller bit-size to a larger bit-size comprises appending copies of the highest-bit of the input element to the upper bits of the integer element for each of the input integer elements.

12. The computer processor of any of claims 8 to 11, wherein the size of each of the input integer elements is one byte, and the size of each of the output integer elements is two bytes.

13. The computer processor of any of claims 8 to 12, wherein the size of each of the input integer elements is two bytes, and the size of each of the output integer elements is four bytes.

14. The computer processor of claim any of claims 8 to 13, wherein the size of each of the input integer elements is four bytes, and the size of each of the output integer elements is eight bytes.

15. The computer processor of claim 1, wherein the processing logic is further configured to: read a vector register from the vector register file;

treat the contents of the vector register as an input vector of input integer elements of larger bit-size;

convert the contents of the vector register into output integer elements of a smaller bit-size; and

store the resulting output vector of output integer elements of a smaller bit-size to a vector register in the vector register file.

16. The computing processor of claim 15, wherein converting from input integer elements of a larger bit-size to corresponding output integer elements of a smaller bit-size comprises discarding the high bits of each of the input integer elements.

17. The computing processor of claim 15 or 16, wherein converting from input integer elements of a larger bit-size to corresponding output integer elements of a smaller bit-size comprises discarding the low bits of each of the input integer elements.

18. The computing processor of any of claims 15 to 17, wherein converting from input integer elements of a larger bit-size to corresponding output integer elements of a smaller bit-size comprises:

representing an integer value in the smaller bit-size;

converting the integer value to the largest number representable at the smaller bit-size if the original integer value is larger than the largest number representable; or

converting the integer value to the smallest number representable if the original integer value is smaller than the smallest number representable.

19. The computing processor of claim any of claims 15 to 18, wherein the bit-size of each of the input integer elements is two bytes and the bit-size of each of the output integer elements is one byte.

20. The computing processor of claim any of claims 15 to 19, wherein the bit-size of each of the input integer elements is four bytes and the bit-size of each of the output integer elements is two bytes.

21. The computing processor of claim any of claims 15 to 120, wherein the bit-size of each of the input integer element is eight bytes and the bit-size of each of the output integer elements is four bytes.

22. The computer processor of claim 1, wherein the processing logic is further configured to:

read a vector register from the vector register file;

treat the contents of the vector register as an input vector of input floating-point elements of a smaller bit-size;

convert the input floating-point elements of the vector register into an output vector of output floating-point numbers of a larger bit-size; and

store the resulting output vector of output floating-point elements of a larger bit-size to a vector register in the vector register file.

23. The computer processor of claim 22, wherein the size of each of the input floating-point elements is two bytes, and the size of each of the output floating-point elements is four bytes.

24. The computer processor of claim 22 or 23, wherein the size of each of the input floating-point elements is four bytes, and the size of the each of the output floating-point elements is eight bytes.

25. The computer processor of claim 1, wherein the processing logic is further configured to:

read a vector register from the vector register file;

treat the contents of the vector register as an input vector of input floating-point elements of a larger bit-size;

convert the contents of the vector register into floating-point elements of a smaller bit-size; and

store the resulting output vector of output floating-point elements of a smaller bit-size to a vector register in the vector register file.

26. The computing processor of claim 25, wherein the bit-size of each of the input floating-point elements is four bytes and the bit-size of each of the output floating-point elements is two bytes.

27. The computing processor of claim 25 or 26, wherein the bit-size of each of the input floating-point elements is eight bytes and the bit-size of each of the output floating-point elements is four bytes.

28. The computer processor of any of claims 25 to 27, further comprising controlling rounding and saturation modes.

29. The computer processor of claim 28, wherein the value of a smaller floating-point number is obtained from a larger floating-point number by rounding and saturation defined by the rounding and saturation modes.

30. The computer processor of claim 25, wherein the rounding and saturation modes are defined as per an IEEE standard.

31. A method, comprising:

holding, by a vector unit of a computer processor comprisinga vector register file comprising at least one register, a varying number of elements; and

operating, by processing logic of the computer processor, on the varying number of elements in the vector register file using one or more instructions that produce results with elements of widths different than that of the input elements.

32. The method of claim 31, wherein the computer processor is implemented as a monolithic integrated circuit.

33. The method of claim 31, further comprising a vector length register file comprising at least one register, wherein the at least one register of the vector length register file is used to specify the number of elements on which the processing logic operates.

34. The method of claim 31 or 32, further comprising:

reading, by the processing logic, two vector registers from the vector register file;

treating the contents of the two vector registers as two input vectors of input integer elements of a bit-size;

element-wise multiplying the elements of the first vector register with the corresponding elements of the second register to produce an output vector of output integer elements, the size of each output integer element having a size that is twice the number of bits of a corresponding input integer element; and

storing the output vector of integer elements to a vector register in the vector register file

35. The method of claim 34, wherein the size of each of the input integer elements is one byte, and the size of each of the output integer elements is two bytes.

36. The method of claim 34 or 35, wherein the size of each of the input integer elements is two bytes, and the size of each of the output integer elements is four bytes.

37. The method of claim any of claims 34 to 36, where the size of each of the input integer elements is four bytes, and the size of each of the output integer elements is eight bytes.

38. The method of claim 31, further comprising:

reading, by the processing logic, a vector register from the vector register file;

treating the contents of the vector register as an input vector of input integer elements of a smaller bit-size;

converting the input vector of input integer elements into an output vector of output integer elements of a larger bit-size; and

storing the output vector of output integer elements to a vector register in the vector register file.

39. The method of claim 38, wherein converting from a smaller bit-size to a larger bit-size comprises appending zeroes to upper bits of each of the input integer elements.

40. The method of claim 38 or 39, wherein converting from a smaller bit-size to a larger bit-size comprises appending zeroes to the lower bits of each of the input integer elements.

41. The method of any of claims 38 to 40, wherein converting from a smaller bit-size to a larger bit-size comprises appending copies of the highest-bit of the input element to the upper bits of the integer element for each of the input integer elements.

42. The method of claims 38 to 41, wherein the size of each of the input integer elements is one byte, and the size of each of the output integer elements is two bytes.

43. The method of claims 38 to 42, where the size of each of the input integer elements is two bytes, and the size of each of the output integer elements is four bytes.

44. The method of claims 38 to 43, where the size of each of the input integer elements is four bytes, and the size of each of the output integer elements is eight bytes.

45. The method of claim 31, further comprising:

reading, by the processing logic, a vector register from the vector register file;

treating the contents of the vector register as an input vector of input integer elements of larger bit-size;

converting the contents of the vector register into output integer elements of a smaller bit-size; and

storing the resulting output vector of output integer elements of a smaller bit-size to a vector register in the vector register file.

46. The method of claim 45, wherein converting from input integer elements of a larger bit-size to corresponding output integer elements of a smaller bit-size comprises discarding the high bits of each of the input integer elements.

47. The method of claim 45 or 46, wherein converting from input integer elements of a larger bit-size to corresponding output integer elements of a smaller bit-size comprises discarding the low bits of each of the input integer elements.

48. The method of any of claims 45 to 47, wherein converting from input integer elements of a larger bit-size to corresponding output integer elements of a smaller bit-size comprises:

representing an integer value in the smaller bit-size;

converting the integer value to the largest number representable at the smaller bit-size if the original integer value is larger than the largest number representable; or

converting the integer value to the smallest number representable if the original integer value is smaller than the smallest number representable.

49. The method of any of claims 45 to 48, wherein the bit-size of each of the input integer elements is two bytes and the bit-size of each of the output integer elements is one byte.

50. The method of any of claims 45 to 49, wherein the bit-size of each of the input integer elements is four bytes and the bit-size of each of the output integer elements is two bytes.

51. The method of claim any of claims 45 to 51, wherein the bit-size of each of the input integer element is eight bytes and the bit-size of each of the output integer elements is four bytes.

52. The method of claim 31, further comprising:

reading, by the processing logic, a vector register from the vector register file;

treating the contents of the vector register as an input vector of input floating-point elements of a smaller bit-size;

converting the input floating-point elements of the vector register into an output vector of output floating-point numbers of a larger bit-size; and

storing the resulting output vector of output floating-point elements of a larger bit-size to a vector register in the vector register file.

53. The method of claim 52, wherein the size of each of the input floating-point elements is two bytes, and the size of each of the output floating-point elements is four bytes.

54. The method of claim 52 or 53, wherein the size of each of the input floating-point elements is four bytes, and the size of the each of the output floating-point elements is eight bytes.

55. The method of claim 31, further comprising:

reading, by the processing logic, a vector register from the vector register file;

treating the contents of the vector register as an input vector of input floating-point elements of a larger bit-size;

converting the contents of the vector register into floating-point elements of a smaller bit-size; and

storing the resulting output vector of output floating-point elements of a smaller bit-size to a vector register in the vector register file.

56. The method of claim 55, wherein the bit-size of each of the input floating-point elements is four bytes and the bit-size of each of the output floating-point elements is two bytes.

57. The method of claim 55 or 56, wherein the bit-size of each of the input floating-point elements is eight bytes and the bit-size of each of the output floating-point elements is four bytes.

58. The method of any of claims 55 to 57, further comprising controlling rounding and saturation modes.

59. The method of claim 58, wherein the value of a smaller floating-point number is obtained from a larger floating-point number by rounding and saturation defined by the rounding and saturation modes.

60. The method of claim 58 or 59, wherein the rounding and saturation modes are defined as per an IEEE standard.

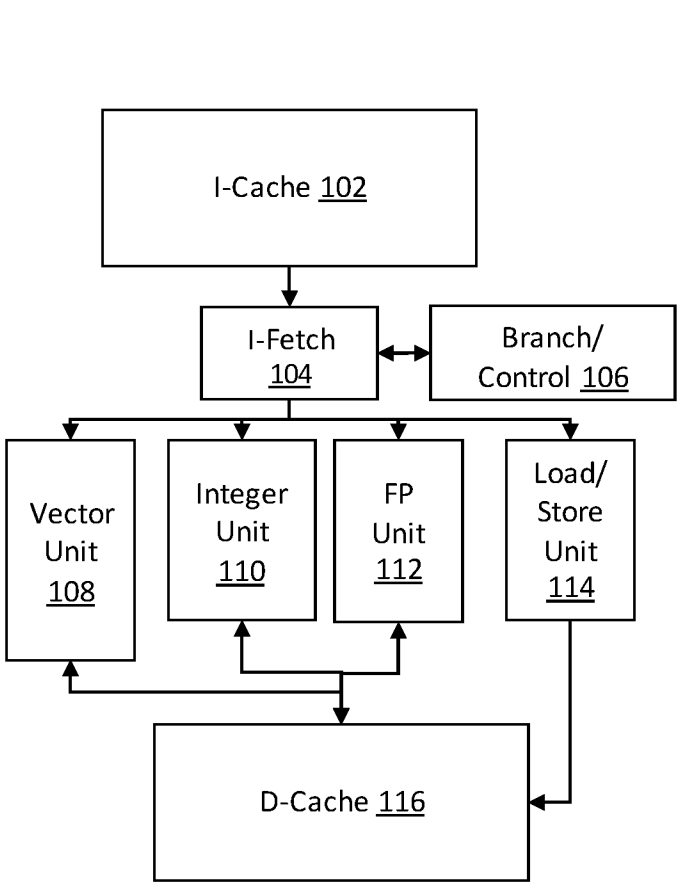

100
I-Cache 102
I-Fetch 104
Branch/ Control 106
Vector Unit 108
Integer Unit 110
FP Unit 112
Load/ Store Unit 114
D-Cache 116

Figure 1

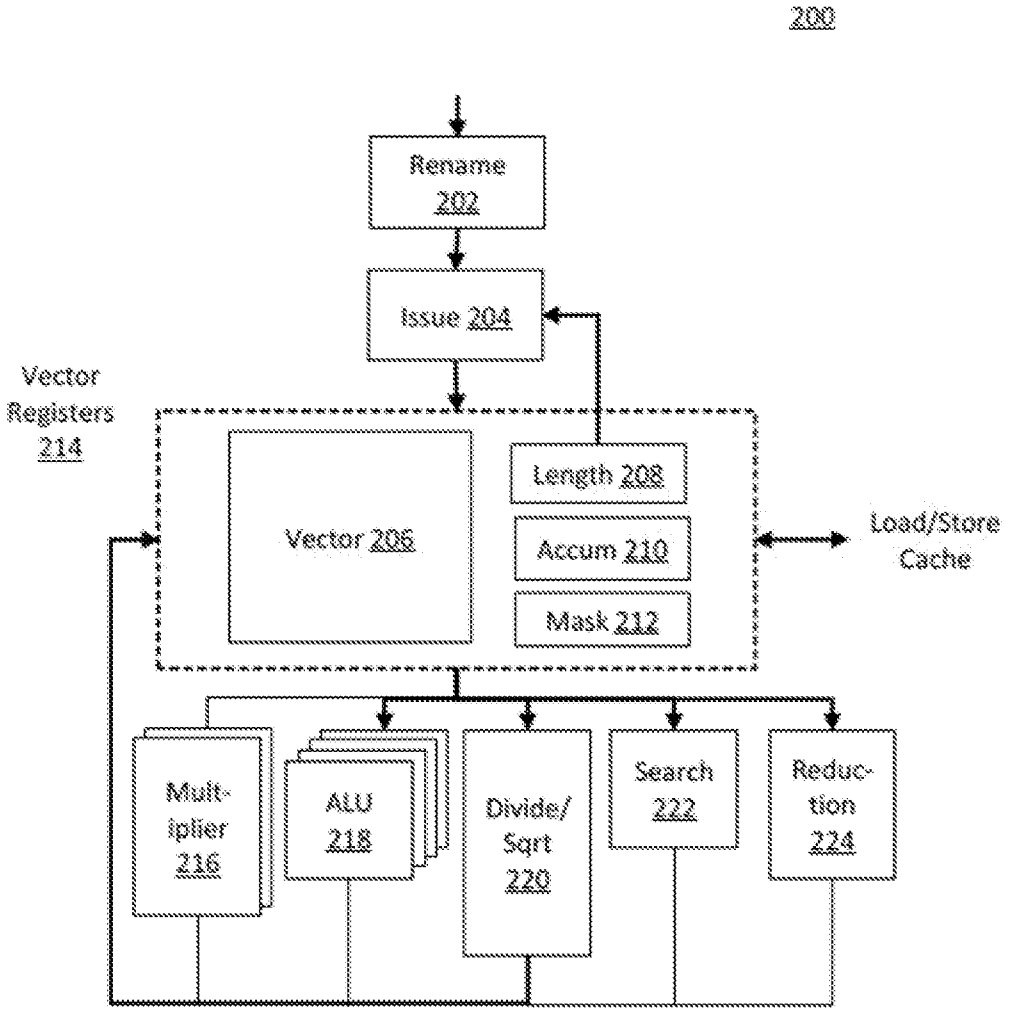

200
Rename 202
Issue 204
Vector Registers 214
Vector 206
Length 208
Accum 210
Mask 212
Load/Store Cache
Mult-iplier 216
ALU 218
Divide/ Sqrt 220
Search 222
Reduc-tion 224

Figure 2



Figure 3

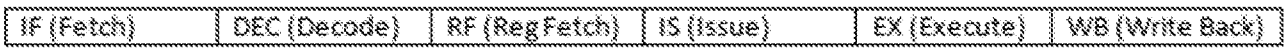


Figure 4

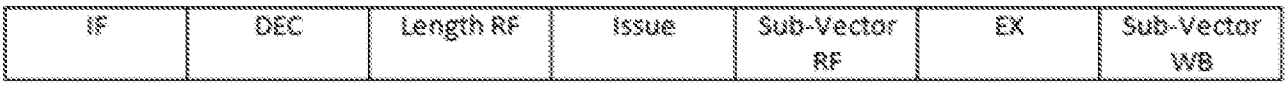


Figure 5

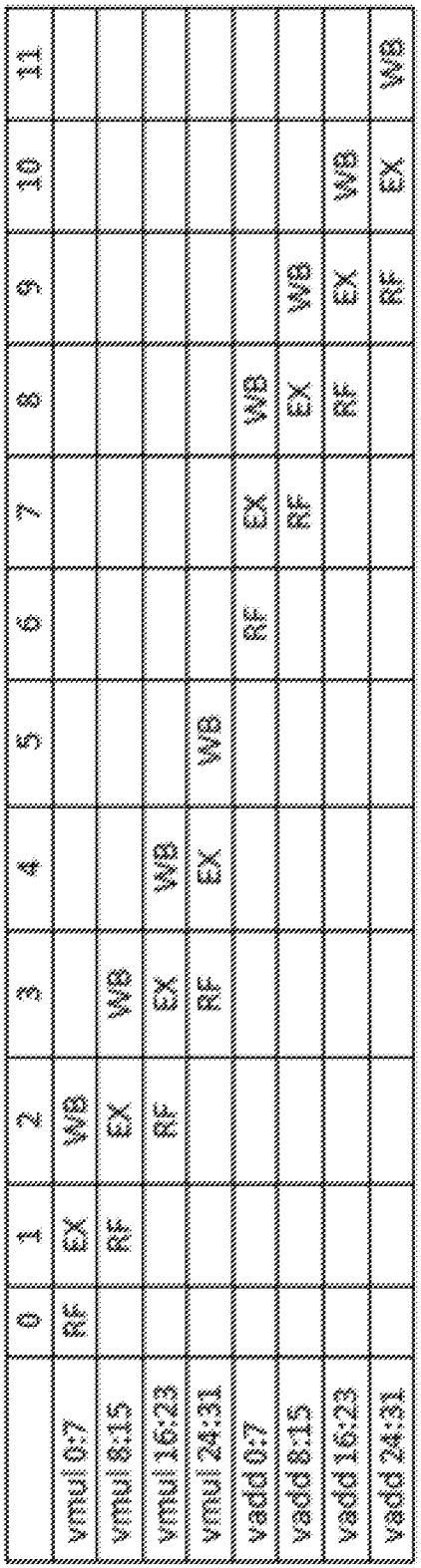

0 1 2 3 4 5 6 7 8 9 10 11
vmul 0:7 RF EX WB
vmul 8:15 RF EX WB
vmul 16:23 RF EX WB
vmul 24:31 RF EX WB
vadd 0:7 RF EX WB
vadd 8:15 RF EX WB
vadd 16:23 RF EX WB
vadd 24:31 RF EX WB

Figure 6

	0	1	2	3	4	5	6	7	8
vmul 0:7	RF	EX	WB						
vmul 8:15		RF	EX	WB					
vmul 16:23			RF	EX	WB				
vmul 24:31				RF	EX	WB			
vadd 0:7				RF	EX	WB			
vadd 8:15					RF	EX	WB		
vadd 16:23						RF	EX	WB	
vadd 24:31							RF	EX	WB

Figure 7

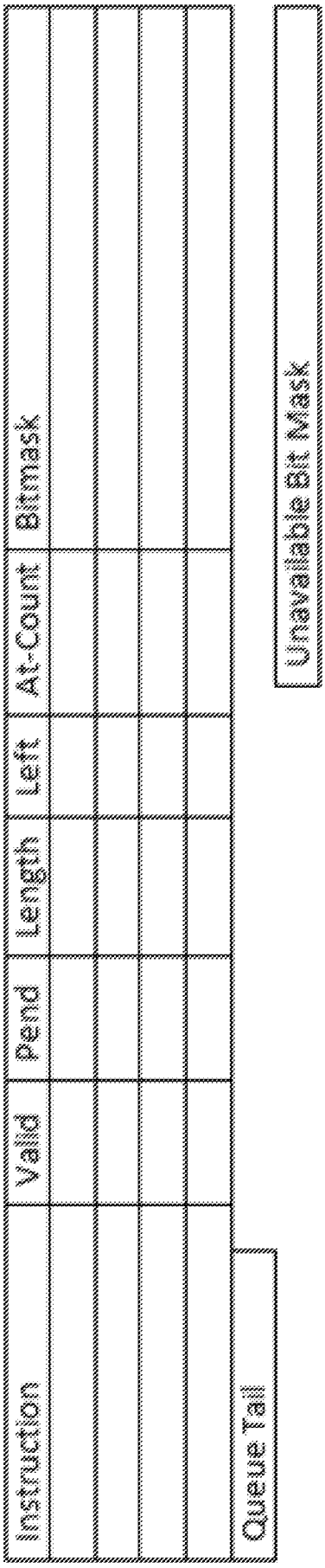

Instruction
Valid
Pend
Length
Left
At-Count
Bitmask
Queue Tail
Unavailable Bit Mask

Figure 8

Architected	map $v0	map $v1	map $v2	...	Effective
Initial	$V0	$V1	$V2		
vadd $n0,$v0,$v1,$v2	$V8				vadd $n0,$V8,$V1,$V2
	$V8	$V1	$V2		
vmul $n0,$v1,$v2,$v0		$V9			vmul $n0,$V9,$V2,$V8
	$V8	$V9	$V2		
vsub $n0,$v0,$v1,$v0	$V10				vsub $n0,$V10,$V9,V8
	$V10	$V9	$V2		

Figure 9

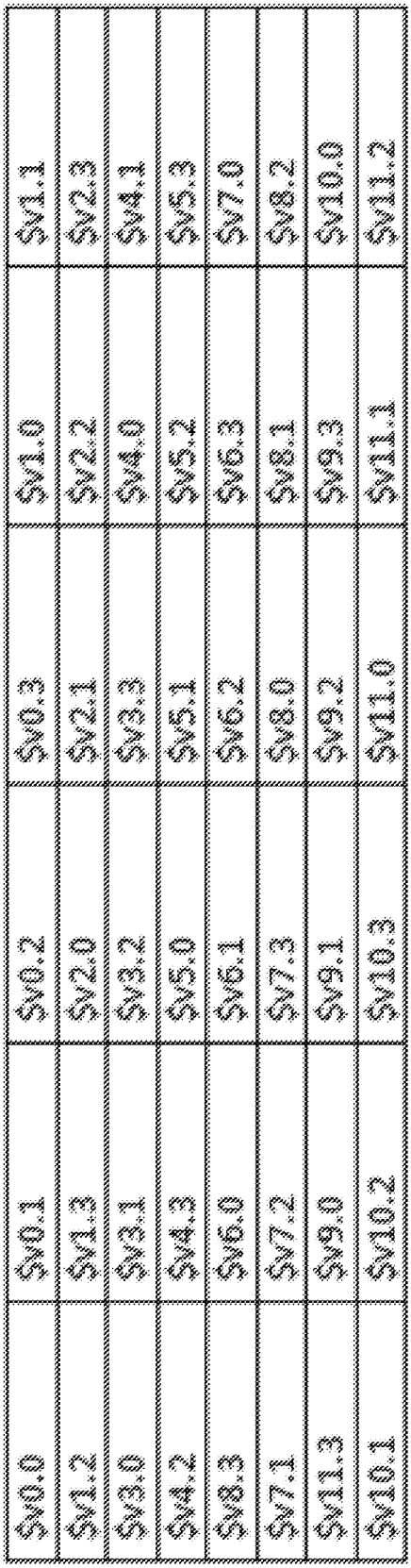

Sv0.0 Sv0.1 Sv0.2 Sv0.3 Sv1.0 Sv1.1
Sv1.2 Sv1.3 Sv2.0 Sv2.1 Sv2.2 Sv2.3
Sv3.0 Sv3.1 Sv3.2 Sv3.3 Sv4.0 Sv4.1
Sv4.2 Sv4.3 Sv5.0 Sv5.1 Sv5.2 Sv5.3
Sv8.3 Sv6.0 Sv6.1 Sv6.2 Sv6.3 Sv7.0
Sv7.1 Sv7.2 Sv7.3 Sv8.0 Sv8.1 Sv8.2
Sv11.3 Sv9.0 Sv9.1 Sv9.2 Sv9.3 Sv10.0
Sv10.1 Sv10.2 Sv10.3 Sv11.0 Sv11.1 Sv11.2

Figure 10

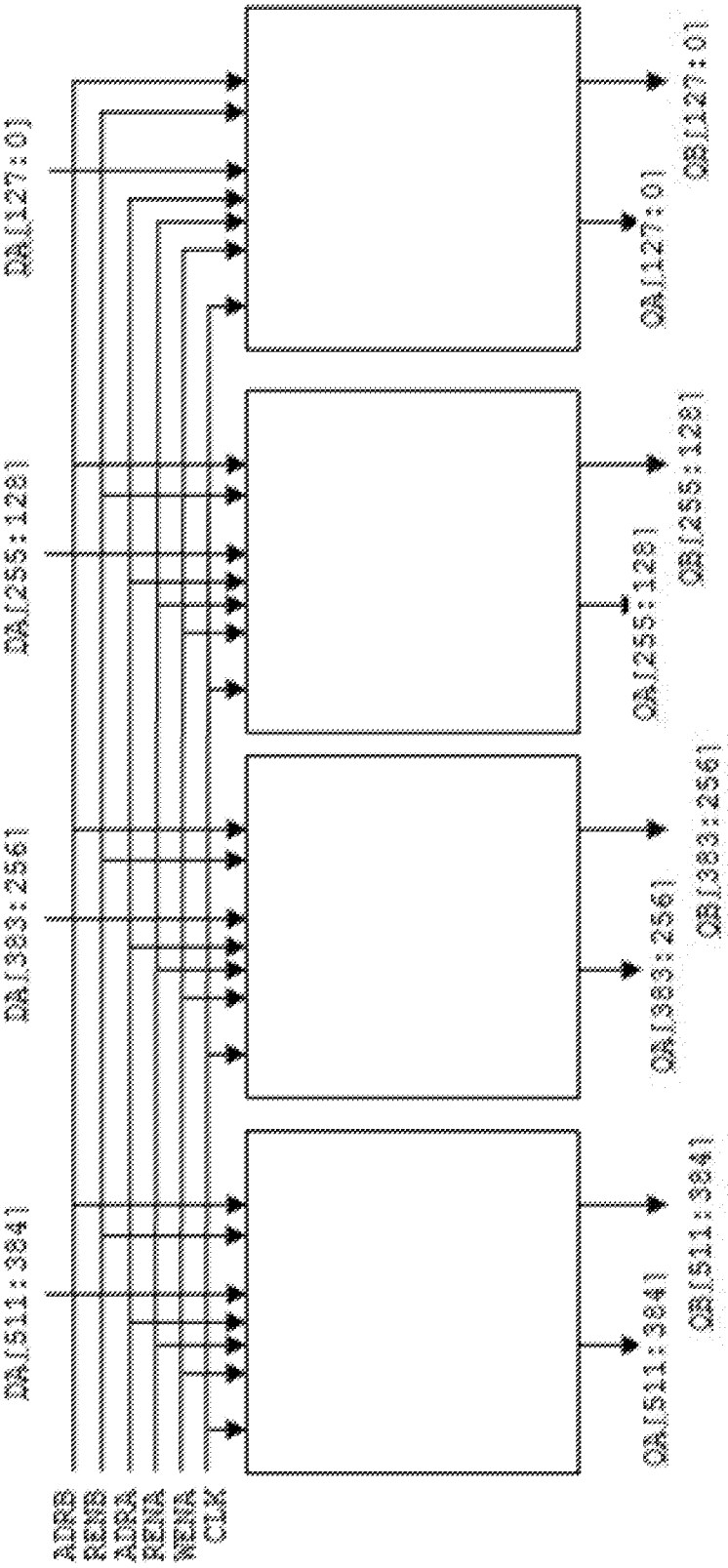

ADDR
REQB
ADDA
REQA
WENA
CLK
DA[511:384]
DA[383:256]
DA[255:128]
DA[127:0]
QA[511:384]
QA[383:256]
QA[255:128]
QA[127:0]
QB[511:384]
QB[383:256]
QB[255:128]
QB[127:0]

Figure 11

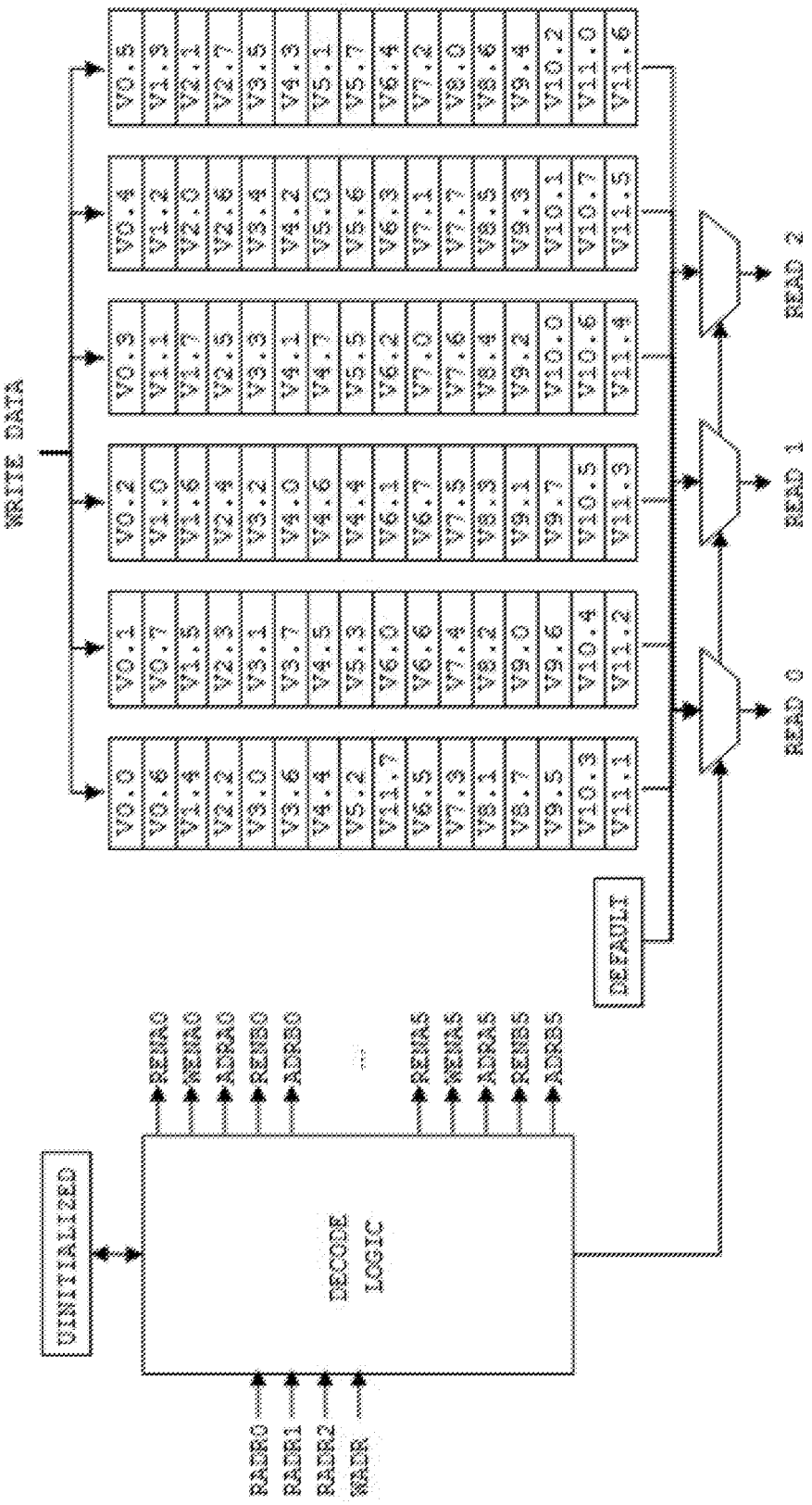

WRITE DATA
READ 2
READ 1
READ 0
DEFAULT
UNINITIALIZED
DECODE LOGIC
RADR0
RADR1
RADR2
WADR

Figure 12

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US16/14288

A. CLASSIFICATION OF SUBJECT MATTER
IPC(8) - G06F 9/30, 9/38; H03M 7/40 (2016.01)
CPC - G06F 9/3885, 9/3877, 9/30112
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC(8) - G06F 9/30, 9/38; H03M 7/40 (2016.01)
CPC - G06F 9/3885, 9/30025, 9/30036, 9/3877, 9/30112, 9/30014; G06T 1/20; H03M 7/4093

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
PatSeer (US, EP, WO, JP, DE, GB, CN, FR, KR, ES, AU, IN, CA, Other Countries (INPADOC), RU, AT, CH, TH, BR, PH); Google/Google Scholar; IEEE/IEEEXplore; EBSCO Non-Patent Prior Art Source
KEYWORDS: vector, processor, variable, length, elements, operation, different-width, shorter, narrower, wider, longer

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X --- Y	US 2005/0028070 A1 (LIN, C., et al.) 03 February 2005; abstract; figures 4, 8, 13, 15; paragraphs [0009], [0054], [0059], [0062], [0071], [0073], [0076], [0082], [0087]-[0089]	1-3, 31-35 --- 8-10, 15-17, 22-27, 30 38-40, 45-47, 52-57
Y	WO 2013/095637 A9 (INTEL CORPORATION) 27 June 2013; abstract; page 1, lines 6-7, page 20, lines 3-10, page 22, lines 28-34, page 26, lines 18 to page 27, line 3, page 28, lines 11-20, page 35, lines 26 to 32, page 38, lines 22-34; figures 5A, 6A, 9B	8-10, 15-17, 22-27, 38-40, 45-47, 52-57
Y	US 6058465 A (NGUYEN L.) 02 May 2000; abstract; column 2, lines 1-3, column 34, lines 24-39	30

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

* Special categories of cited documents:
"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed
"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
15 March 2016 (15.03.2016)	31 MAR 2016
Name and mailing address of the ISA/ Mail Stop PCT, Attn: ISA/US, Commissioner for Patents P.O. Box 1450, Alexandria, Virginia 22313-1450 Facsimile No. 571-273-8300	Authorized officer Shane Thomas PCT Helpdesk: 571-272-4300 PCT OSP: 571-272-7774

Form PCT/ISA/210 (second sheet) (January 2015)

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US16/14288

Box No. II Observations where certain claims were found unsearchable (Continuation of item 2 of first sheet)

This international search report has not been established in respect of certain claims under Article 17(2)(a) for the following reasons:

1. ☐ Claims Nos.:
because they relate to subject matter not required to be searched by this Authority, namely:

2. ☐ Claims Nos.:
because they relate to parts of the international application that do not comply with the prescribed requirements to such an extent that no meaningful international search can be carried out, specifically:

3. ☒ Claims Nos.: 4-7, 11-14, 18-21, 28-29, 36-37, 41-44, 48-51, 58-60
because they are dependent claims and are not drafted in accordance with the second and third sentences of Rule 6.4(a).

Box No. III Observations where unity of invention is lacking (Continuation of item 3 of first sheet)

This International Searching Authority found multiple inventions in this international application, as follows:

1. ☐ As all required additional search fees were timely paid by the applicant, this international search report covers all searchable claims.

2. ☐ As all searchable claims could be searched without effort justifying additional fees, this Authority did not invite payment of additional fees.

3. ☐ As only some of the required additional search fees were timely paid by the applicant, this international search report covers only those claims for which fees were paid, specifically claims Nos.:

4. ☐ No required additional search fees were timely paid by the applicant. Consequently, this international search report is restricted to the invention first mentioned in the claims; it is covered by claims Nos.:

Remark on Protest

☐ The additional search fees were accompanied by the applicant's protest and, where applicable, the payment of a protest fee.

☐ The additional search fees were accompanied by the applicant's protest but the applicable protest fee was not paid within the time limit specified in the invitation.

☐ No protest accompanied the payment of additional search fees.