UNITED STATES DISTRICT COURT
FOR THE WESTERN DISTRICT OF TEXAS
WACO DIVISION

TELEPUTERS, LLC, Plaintiff v. **ANALOG DEVICES, INC.** Defendants	**Case No. 6:23-cv-00755** **JURY TRIAL DEMANDED** **RELATED CASES**

COMPLAINT FOR PATENT INFRINGEMENT

Plaintiff Teleputers, LLC ("Plaintiff" or "Teleputers") hereby asserts the following claims for patent infringement against Analog Devices, Inc. ("Defendant" or "Analog Devices"), and alleges, on information and belief, as follows:

RELATED CASES

This case is related to the following cases:

- *Teleputers, LLC v. Renesas Electronics America, Inc., et al*, Case No. 6:20-cv-00599-ADA;
- *Teleputers, LLC v. Marvell Semiconductor, Inc., et al,* Case No. 6:20-cv-00512-ADA;
- *Teleputers, LLC v. Oracle Corporation, et al,* Case No. 6:20-cv-00600-ADA;
- *Teleputers, LLC v. Fujitsu America, Inc., et al,* Case No. 6:20-cv-00640-ADA; and
- *Teleputers, LLC v. Qualcomm Inc.., et al,* Case No. 6:23-cv-00404-ADA.

THE PARTIES

1. Teleputers, LLC is a limited liability company organized and existing under the laws of the State of New Jersey with a principal place of business in Princeton, New Jersey.

2. On information and belief, Analog Devices, Inc. is a corporation organized and existing under the laws of Massachusetts, having a place of business in this Judicial District at 3900 N Capital of Texas Hwy, Austin, Texas 78746.

JURISDICTION AND VENUE

3. This Court has original jurisdiction over the subject matter of this action pursuant to 28 U.S.C. §§ 1391 and 1400.

4. Upon information and belief, Defendant is subject to personal jurisdiction of this Court based upon it having regularly conducted business, including the acts complained of herein, within the State of Texas and this judicial district and/or deriving substantial revenue from goods and services provided to individuals in Texas and in this District.

5. Venue is proper in this District under 28 U.S.C. § 1400 because Defendant has committed acts of infringement and has regular and established places of business in this judicial district at 3900 N Capital of Texas Hwy, Austin, Texas 78746.

NOTICE OF TELEPUTERS' PATENTS

6. Teleputers is owner by assignment of U.S. Patent No. 6,922,472 ("the '472 Patent") entitled "Method and system for performing permutations using permutation instructions based on butterfly networks." A copy may be obtained at: https://patents.google.com/patent/US6922472B2/en.

7. Teleputers is owner by assignment of U.S. Patent No. 6,952,478B2 ("the '478 Patent") entitled "Method and system for performing permutations using permutation instructions based on modified omega and flip stages." A copy may be obtained at: https://patents.google.com/patent/US6952478B2/en.

8. Teleputers is owner by assignment of U.S. Patent No. 7,092,526B2 ("the '526 Patent") entitled "Method and system for performing subword permutation instructions for use in two- dimensional multimedia processing." A copy may be obtained at: https://patents.google.com/patent/US7092526B2/en.

9. Teleputers is owner by assignment of U.S. Patent No. 7,174,014B2 ("the '014 Patent" and "the Patents-in-Suit") entitled "Method and system for performing permutations with bit permutation instructions." A copy may be obtained at: https://patents.google.com/patent/US7174014B2/en.

10. Teleputers is owner by assignment of U.S. Patent No. 7,519,795B2 ("the '795 Patent") entitled "Method and system for performing permutations with bit permutation instructions." A copy may be obtained at: https://patents.google.com/patent/US7519795B2/en.

11. The foregoing Patents, namely the '014 Patent, the '526 Patent, the '478 Patent, the '472 Patent, and the '795 Patent are collectively referred to as "the Teleputers Patents."

12. The Teleputers Patents are valid, enforceable, and were duly issued in full compliance with Title 35 of the United States Code.

13. Defendants, at least by the date of this Original Complaint, are on notice of the Teleputers Patents.

THE PATENT-IN_SUIT

14. Teleputers is the lawful owner of all right, title, and interest in United States Patent No. 6,952,478 (the "'478 Patent"), entitled "Method and system for performing permutations using permutation instructions based on modified omega and flip stages," including the right to sue and to recover for infringement thereof. The '478 Patent was duly and legally issued on October 4, 2005.

15. Teleputers is the lawful owner of all right, title, and interest in United States Patent No. 7,092,526 (the "'526 Patent"), entitled "Method and system for performing subword permutation instructions for use in two-dimensional multimedia processing," including the right to sue and to recover for infringement thereof. The '526 Patent was duly and legally issued on August 15, 2006.

ACCUSED INSTRUMENTALITIES

16. On information and belief, Defendants make, use, import, sell, and/or offer for sale a multitude of products and services as systems on chips ("SoC") that employ technology supporting the infringing instructions including, but not limited to: ADSP-SC598/SC596/SC595 processors (individually and collectively, the "Accused Instrumentalities"). On information and belief, the Accused Instrumentalities are made, used, sold, offered for sale, and/or imported in the United States by Defendants.

COUNT I – INFRINGEMENT OF U.S. PATENT NO. 6,952,478

17. Teleputers repeats and realleges the allegations of each of the above paragraphs as if fully set forth herein.

18. Claim 1 of the '478 Patent recites:

1. A method of performing an arbitrary permutation of a source sequence of bits in a programmable processor comprising the steps of:

a. defining an intermediate sequence of bits that said source sequence of bits is transformed into;

b. determining a permutation instruction for transforming said source sequence of bits into said intermediate sequence of bits; and

c. repeating steps a. and b. using said determined intermediate sequence of bits from step b. as said source sequence of bits in step a. until a desired sequence of bits is obtained,

wherein the determined permutation instructions form a permutation instruction sequence.

19. On information and belief, Defendant, without license or authorization, has directly infringed and continue to infringe the '478 Patent by making, using, importing, selling, and/or, offering for sale the Accused Instrumentalities in the United States.

20. On information and belief, Defendant, with knowledge of the '478 Patent, indirectly infringes the '478 Patent by inducing others to infringe the '478 Patent. In particular, Defendant intends to induce customers to infringe the '478 Patent by encouraging customers to use the Accused Instrumentalities in a manner that results in infringement.

21. On information and belief, Defendants also induces others, including its customers, to infringe the '478 Patent by providing technical support for the use of the Accused Instrumentalities.

22. On information and belief, Defendant's actions satisfy each and every element of claim 1:

1. A method of performing an arbitrary permutation of a source sequence of bits in a programmable processor comprising the steps of:

The ADSP-SC598/SC596/SC595 processors are members of the ADSP-SC59x SHARC® family of products. Containing the same dual-SHARC+® DSP core architecture as the ADSP-SC594/SC592/SC591, these processors upgrade the integrated Arm core to a Cortex-A55 running at up to 1.2 GHz. The A55 processor, with FPU and Neon® DSP extensions, handles additional real-time processing tasks and manages peripherals used to interface to time-critical data in audio applications. These interfaces include Gigabit Ethernet, USB High-Speed, CAN FD, and a rich variety of other connectivity options for a flexible and simplified system design.

- **Arm Core Infrastructure**
- **1.2 GHz Arm Cortex-A55 (with Neon/FPU)**
- 32 kByte/32 kByte L1 Instr./Data Cache
- 256 kByte L2 Cache

Source: https://www.analog.com/en/products/adsp-sc596.html#product-overview

SYSTEM FEATURES
Dual-enhanced SHARC+ high performance floating-point
Cores

- Up to 1000 MHz per SHARC+ core
- 5 Mb (640 kB) L1 SRAM memory per core with parity
- (optional ability to configure as cache)
- 32-bit, 40-bit, and 64-bit floating-point support
- 32-bit fixed point
- Byte, short word, word, long word addressed
- **Arm Cortex-A55 core**
- Up to 1200 MHz/3360 DMIPS **with advanced SIMD** and
- floating-point support
- 32 kB L1 instruction cache with parity/32 kB L1 data cache
- with ECC
- 256 kB L2 cache with ECC
- Powerful DMA system with 8 MemDMAs
- On-chip memory protection

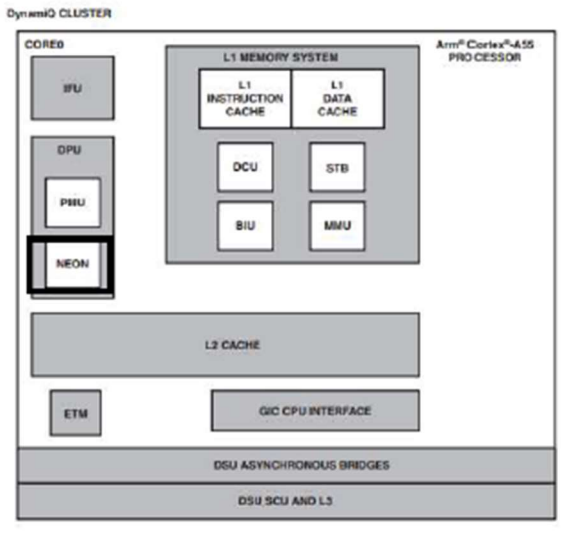


Figure 2. Arm Cortex-A55 Processor Block Diagram

Source: https://www.analog.com/media/en/technical-documentation/data-sheets/adsp-sc596-adsp-sc598.pdf

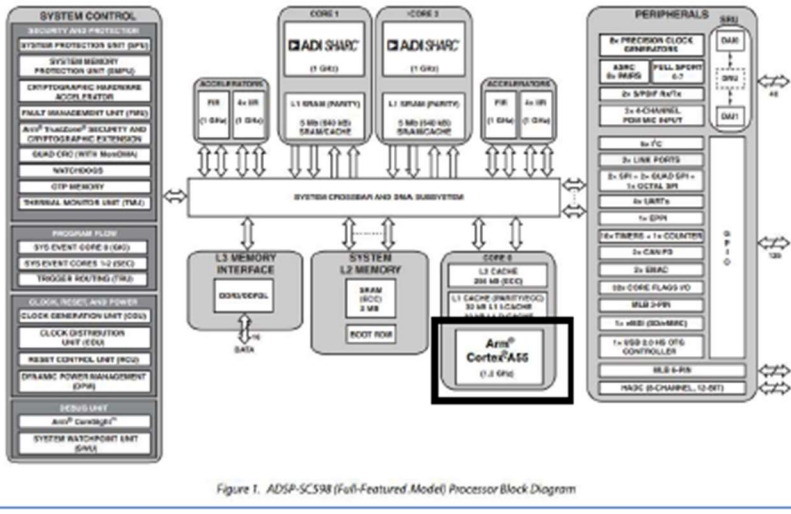


Figure 1. ADSP-SC598 (Full-Featured Model) Processor Block Diagram

Source: https://www.analog.com/media/en/technical-documentation/data-sheets/adsp-sc596-adsp-sc598.pdf

Arm Neon is an advanced single instruction multiple data (SIMD) architecture extension for the Arm Cortex-A and Arm Cortex-R series of processors with capabilities that vastly improve use cases on mobile devices, such as multimedia encoding/decoding, user interface, 2D/3D graphics and gaming.

Source:https://www.arm.com/technologies/neon#:~:text=Arm%20Neon%20is%20an%20advanced,2D%2F3D%20graphics%20and%20gaming

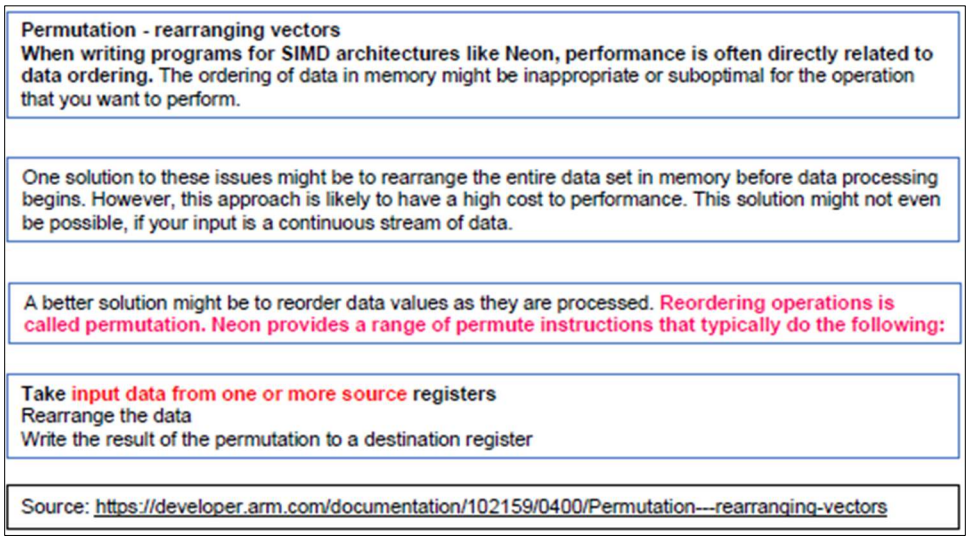
The Cortex-A55 implements the latest Armv8.2 architecture and builds on the success of its predecessor. It pushes the boundaries on performance while maintaining the same levels of power consumption as the Cortex-A53.

New architectural instructions were added to the Cortex-A55 NEON pipeline, allowing it to perform sixteen 8-bit integer operations per-cycle. These new instructions also allow eight 16-bit float operations per-cycle, and rounding double MAC instructions, beneficial for colour space conversion.

Source: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-cortex-a55-efficient-performance-from-edge-to-cloud

There are a range of NEON permutation instructions from simple reversals to arbitrary vector reconstruction. Simple permutations can be achieved using instructions that take a single cycle to issue, whereas the more complex operations are multiple cycle, and might require additional registers to be set up. As always, check your processor's Technical Reference Manual for performance details.

Source: https://documentation-service.arm.com/static/63299276e68c6809a6b41308?token=



Permutation - rearranging vectors
When writing programs for SIMD architectures like Neon, performance is often directly related to data ordering. The ordering of data in memory might be inappropriate or suboptimal for the operation that you want to perform.

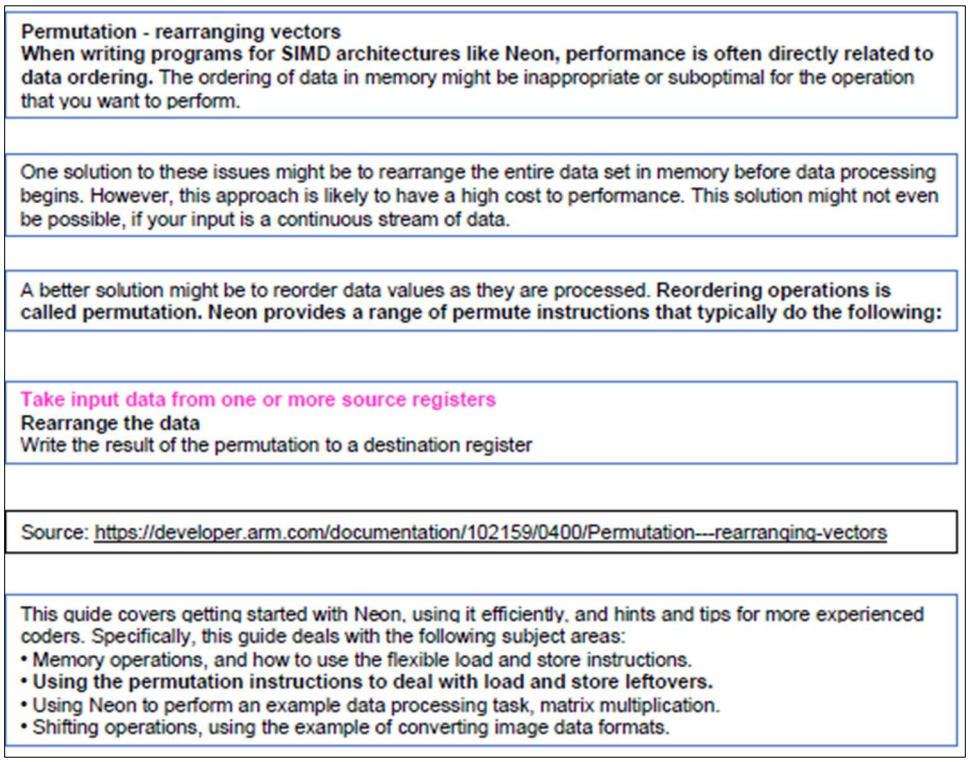
One solution to these issues might be to rearrange the entire data set in memory before data processing begins. However, this approach is likely to have a high cost to performance. This solution might not even be possible, if your input is a continuous stream of data.

A better solution might be to reorder data values as they are processed. Reordering operations is called permutation. Neon provides a range of permute instructions that typically do the following:

Take input data from one or more source registers
Rearrange the data
Write the result of the permutation to a destination register

Source: https://developer.arm.com/documentation/102159/0400/Permutation---rearranging-vectors

a. defining an intermediate sequence of bits that said source sequence of bits is transformed into;



Permutation - rearranging vectors
When writing programs for SIMD architectures like Neon, performance is often directly related to data ordering. The ordering of data in memory might be inappropriate or suboptimal for the operation that you want to perform.

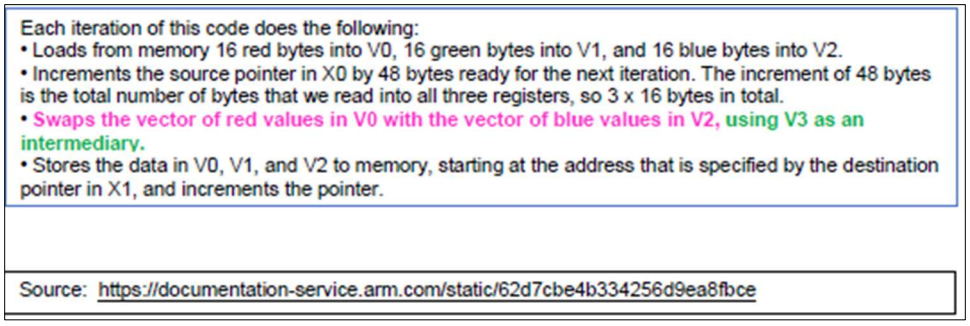
One solution to these issues might be to rearrange the entire data set in memory before data processing begins. However, this approach is likely to have a high cost to performance. This solution might not even be possible, if your input is a continuous stream of data.

A better solution might be to reorder data values as they are processed. **Reordering operations is called permutation. Neon provides a range of permute instructions that typically do the following:**

Take input data from one or more source registers
Rearrange the data
Write the result of the permutation to a destination register

Source: https://developer.arm.com/documentation/102159/0400/Permutation---rearranging-vectors

This guide covers getting started with Neon, using it efficiently, and hints and tips for more experienced coders. Specifically, this guide deals with the following subject areas:
- Memory operations, and how to use the flexible load and store instructions.
- **Using the permutation instructions to deal with load and store leftovers.**
- Using Neon to perform an example data processing task, matrix multiplication.
- Shifting operations, using the example of converting image data formats.



Each iteration of this code does the following:
- Loads from memory 16 red bytes into V0, 16 green bytes into V1, and 16 blue bytes into V2.
- Increments the source pointer in X0 by 48 bytes ready for the next iteration. The increment of 48 bytes is the total number of bytes that we read into all three registers, so 3 x 16 bytes in total.
- Swaps the vector of red values in V0 with the vector of blue values in V2, using V3 as an intermediary.
- Stores the data in V0, V1, and V2 to memory, starting at the address that is specified by the destination pointer in X1, and increments the pointer.

Source: https://documentation-service.arm.com/static/62d7cbe4b334256d9ea8fbce

b. determining a permutation instruction for transforming said source sequence of bits into said intermediate sequence of bits; and

Permutation - Neon instructions
Neon provides several different kinds of permute instruction to perform different operations:

Move instructions
Reverse instructions
Extraction instructions
Transpose instructions
Interleave instructions
Lookup table instructions

Source: https://developer.arm.com/documentation/102159/0400/Permutation---Neon-instructions?lang=en

This guide covers getting started with Neon, using it efficiently, and hints and tips for more experienced coders. Specifically, this guide deals with the following subject areas:
• Memory operations, and how to use the flexible load and store instructions.
• **Using the permutation instructions to deal with load and store leftovers.**
• Using Neon to perform an example data processing task, matrix multiplication.
• Shifting operations, using the example of converting image data formats.

Source: https://documentation-service.arm.com/static/62d7cbe4b334256d9ea8fbce

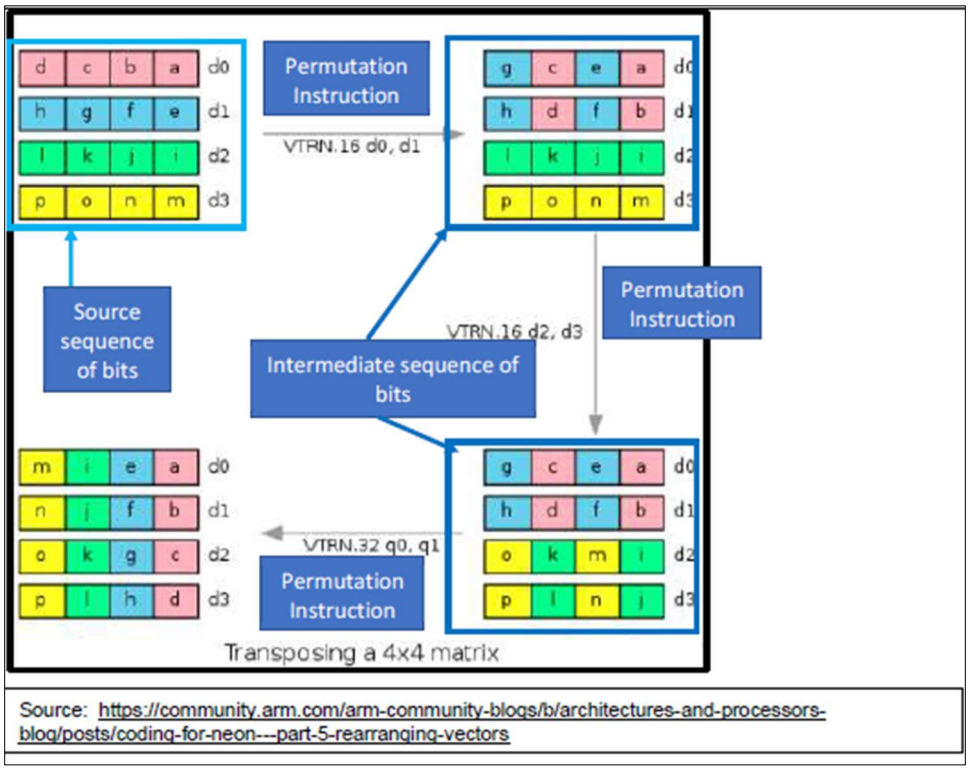
Each iteration of this code does the following:
• Loads from memory 16 red bytes into V0, 16 green bytes into V1, and 16 blue bytes into V2.
• Increments the source pointer in X0 by 48 bytes ready for the next iteration. The increment of 48 bytes is the total number of bytes that we read into all three registers, so 3 x 16 bytes in total.
• **Swaps the vector of red values in V0 with the vector of blue values in V2, using V3 as an intermediary.**
• Stores the data in V0, V1, and V2 to memory, starting at the address that is specified by the destination pointer in X1, and increments the pointer.

Source: https://documentation-service.arm.com/static/62d7cbe4b334256d9ea8fbce

Instructions
Neon provides a range of permutation instructions, from basic reversals to arbitrary vector reconstruction. Simple permutations can be achieved using instructions that take a single cycle to issue, **whereas the more complex operations use multiple cycles, and may require additional registers to be set up.** As always, benchmark or profile your code regularly, and check your processor's Technical Reference Manual (Cortex-A8, Cortex-A9) for performance details.

VTRN: Transpose
VTRN transposes 8, 16 or 32-bit elements between a pair of vectors. It treats the elements of the vectors as 2x2 matrices, and transposes each matrix

Use multiple VTRN instructions to transpose larger matrices. For example, a 4x4 matrix consisting of 16-bit elements can be transposed using three VTRN instructions.



Source: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/coding-for-neon---part-5-rearranging-vectors

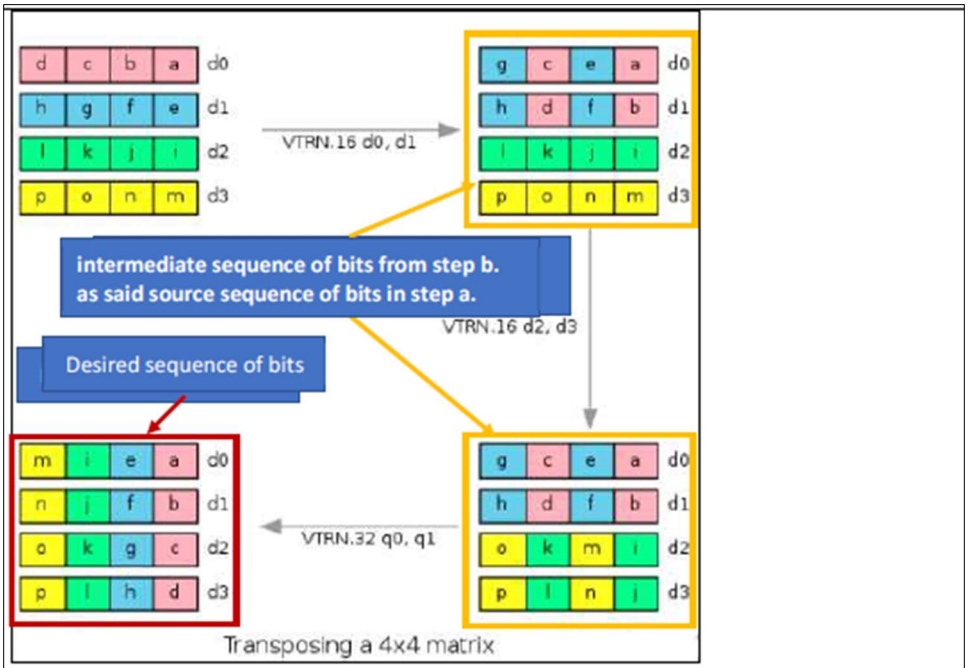
c. repeating steps a. and b. using said determined intermediate sequence of bits from step b. as said source sequence of bits in step a. until a desired sequence of bits is obtained,

Instructions
Neon provides a range of permutation instructions, from basic reversals to arbitrary vector reconstruction. Simple permutations can be achieved using instructions that take a single cycle to issue, whereas the more complex operations use multiple cycles, **and may require additional registers to be set up.** As always, benchmark or profile your code regularly, and check your processor's Technical Reference Manual (Cortex-A8, Cortex-A9) for performance details.

VTRN: Transpose
VTRN transposes 8, 16 or 32-bit elements between a pair of vectors. It treats the elements of the vectors as 2x2 matrices, and transposes each matrix

Use multiple VTRN instructions to transpose larger matrices. For example, a 4x4 matrix consisting of 16-bit elements can be transposed using three VTRN instructions.

Source: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/coding-for-neon---part-5-rearranging-vectors



Source: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/coding-for-neon---part-5-rearranging-vectors

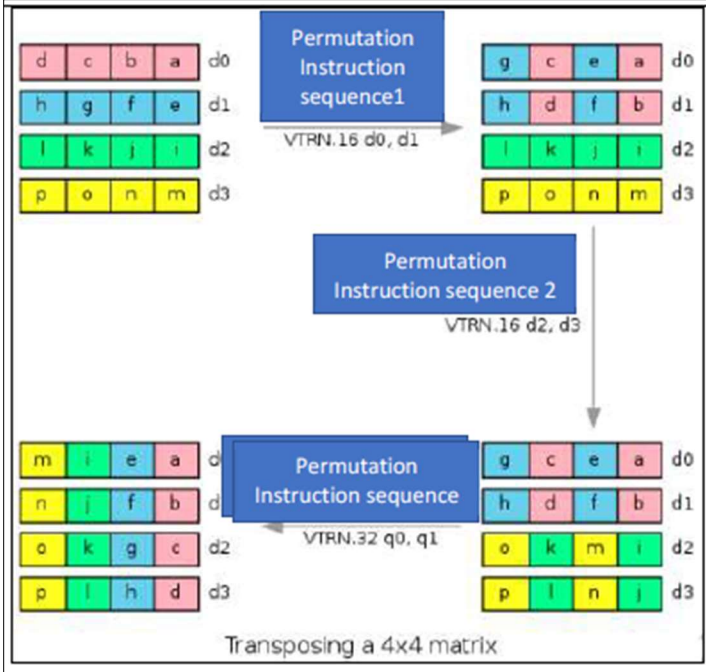
wherein the determined permutation instructions form a permutation instruction sequence.

Instructions
Neon provides a range of permutation instructions, from basic reversals to arbitrary vector reconstruction. Simple permutations can be achieved using instructions that take a single cycle to issue, **whereas the more complex operations use multiple cycles, and may require additional registers to be set up.** As always, benchmark or profile your code regularly, and check your processor's Technical Reference Manual (Cortex-A8, Cortex-A9) for performance details.

VTRN: Transpose
VTRN transposes 8, 16 or 32-bit elements between a pair of vectors. It treats the elements of the vectors as 2x2 matrices, and transposes each matrix

Use multiple VTRN instructions to transpose larger matrices. For example, a 4x4 matrix consisting of 16-bit elements can be transposed using three VTRN instructions.

Source: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/coding-for-neon---part-5-rearranging-vectors



Source: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/coding-for-neon---part-5-rearranging-vectors

COUNT II – INFRINGEMENT OF U.S. PATENT NO. 7,092,526

23. Teleputers repeats and realleges the allegations of each of the above paragraphs as if fully set forth herein.

24. Claim 1 of the '526 Patent recites:

1. A method for permuting two dimensional (2-D) data in a programmable processor comprising the steps of:

decomposing said two dimensional data into at least one atomic element said two dimensional data being located in at least one source register said at least one atomic element of said two dimensional data is a 2×2 matrix and said two dimensional data is decomposed into data elements in said matrix;

determining at least one permutation instruction for rearrangement of said data in said atomic element;

said data elements being rearranged by said at least one permutation instruction, each of said data elements representing a subword having one or more bits; and

applying said permutation instructions to said subwords and placing said permutated subwords into a destination register.

25. On information and belief, Defendant, without license or authorization, has directly infringed and continue to infringe the '526 Patent by making, using, importing, selling, and/or, offering for sale the Accused Instrumentalities in the United States.

26. On information and belief, Defendant, with knowledge of the '526 Patent, indirectly infringes the '526 Patent by inducing others to infringe the '526 Patent. In particular, Defendant intends to induce customers to infringe the '526 Patent by encouraging customers to use the Accused Instrumentalities in a manner that results in infringement.

27. On information and belief, Defendants also induces others, including its customers, to infringe the '526 Patent by providing technical support for the use of the Accused Instrumentalities.

28. On information and belief, Defendant's actions satisfy each and every element of claim 1:

1. A method for permuting two dimensional (2-D) data in a programmable processor comprising the steps of:

The ADSP-SC598/SC596/SC595 processors are members of the ADSP-SC59x SHARC® family of products. Containing the same dual-SHARC+® DSP core architecture as the ADSP-SC594/SC592/SC591, these processors upgrade the integrated Arm core to a Cortex-A55 running at up to 1.2 GHz. The A55 processor, with FPU and Neon® DSP extensions, handles additional real-time processing tasks and manages peripherals used to interface to time-critical data in audio applications. These interfaces include Gigabit Ethernet, USB High-Speed, CAN FD, and a rich variety of other connectivity options for a flexible and simplified system design.

- **ARM Cortex-A7 dual-core with high-speed memory bus**:
- 32 KB I / 32 KB D, 256 KB L2 cache
- 3040 DMIPS at 800 MHz for dual-core
- **VFP v4 FPU and NEON SIMD extension support**
- TrustZone and virtualisation extensions
- CoreSight for debug/trace

Source: https://www.analog.com/en/products/adsp-sc596.html#product-overview

SYSTEM FEATURES
Dual-enhanced SHARC+ high performance floating-point
Cores

- Up to 1000 MHz per SHARC+ core
- 5 Mb (640 kB) L1 SRAM memory per core with parity
- (optional ability to configure as cache)
- 32-bit, 40-bit, and 64-bit floating-point support
- 32-bit fixed point
- Byte, short word, word, long word addressed
- **Arm Cortex-A55 core**
- Up to 1200 MHz/3360 DMIPS **with advanced SIMD** and
- floating-point support
- 32 kB L1 instruction cache with parity/32 kB L1 data cache
- with ECC
- 256 kB L2 cache with ECC
- Powerful DMA system with 8 MemDMAs

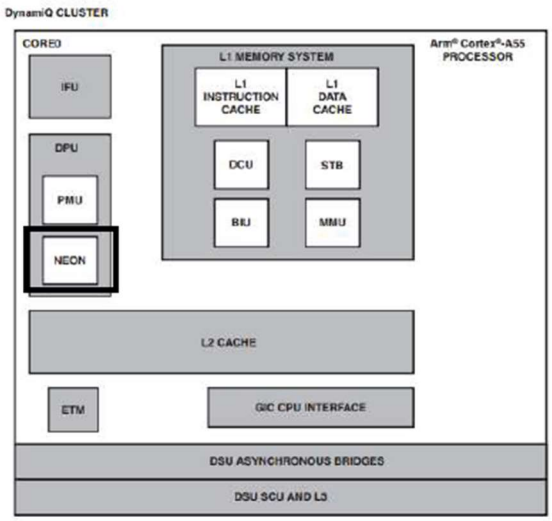


Figure 2. Arm Cortex-A55 Processor Block Diagram

Source: https://www.analog.com/media/en/technical-documentation/data-sheets/adsp-sc596-adsp-sc598.pdf

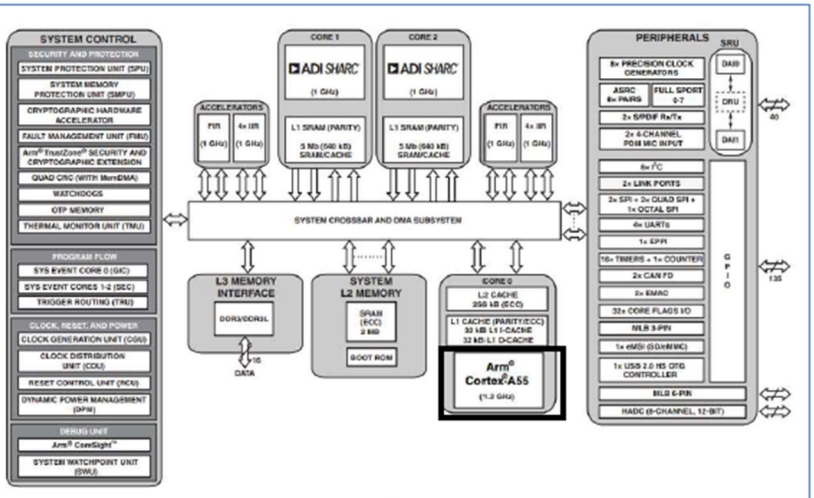


Figure 1. ADSP-SC598 (Full-Featured Model) Processor Block Diagram

Source: https://www.analog.com/media/en/technical-documentation/data-sheets/adsp-sc596-adsp-sc598.pdf

Arm Neon is an advanced single instruction multiple data (SIMD) architecture extension for the Arm Cortex-A and Arm Cortex-R series of processors with capabilities that vastly improve use cases on mobile devices, such as multimedia encoding/decoding, user interface, 2D/3D graphics and gaming.

Source:https://www.arm.com/technologies/neon#:~:text=Arm%20Neon%20is%20an%20advanced,2D%2F3D%20graphics%20and%20gaming

The Cortex-A55 implements the latest Armv8.2 architecture and builds on the success of its predecessor. It pushes the boundaries on performance while maintaining the same levels of power consumption as the Cortex-A53.

New architectural instructions were added to the Cortex-A55 NEON pipeline, allowing it to perform sixteen 8-bit integer operations per-cycle. These new instructions also allow eight 16-bit float operations per-cycle, and rounding double MAC instructions, beneficial for colour space conversion.

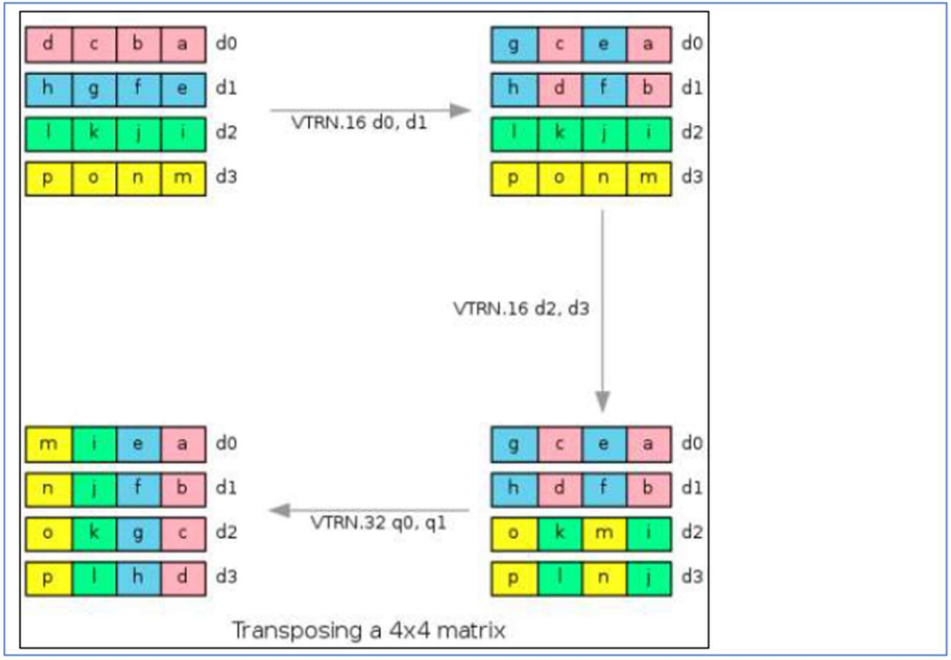
Source: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-cortex-a55-efficient-performance-from-edge-to-cloud

There are a range of NEON permutation instructions from simple reversals to arbitrary vector reconstruction. Simple permutations can be achieved using instructions that take a single cycle to issue, whereas the more complex operations are multiple cycle, and might require additional registers to be set up. As always, check your processor's Technical Reference Manual for performance details.

Source: https://documentation-service.arm.com/static/63299276e68c6809a6b41308?token=

VTRN: Transpose
VTRN transposes 8, 16 or 32-bit elements between a pair of vectors. It treats the elements of the vectors as 2x2 matrices, and transposes each matrix.

Use multiple VTRN instructions to transpose larger matrices. For example, a 4x4 matrix consisting of 16-bit elements can be transposed using three VTRN instructions.

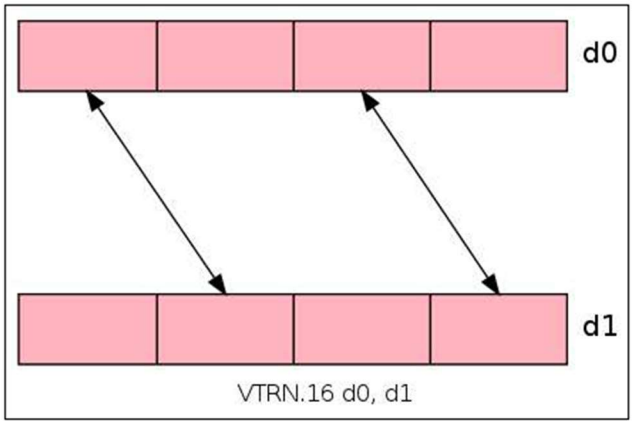


Source: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/coding-for-neon---part-5-rearranging-vectors

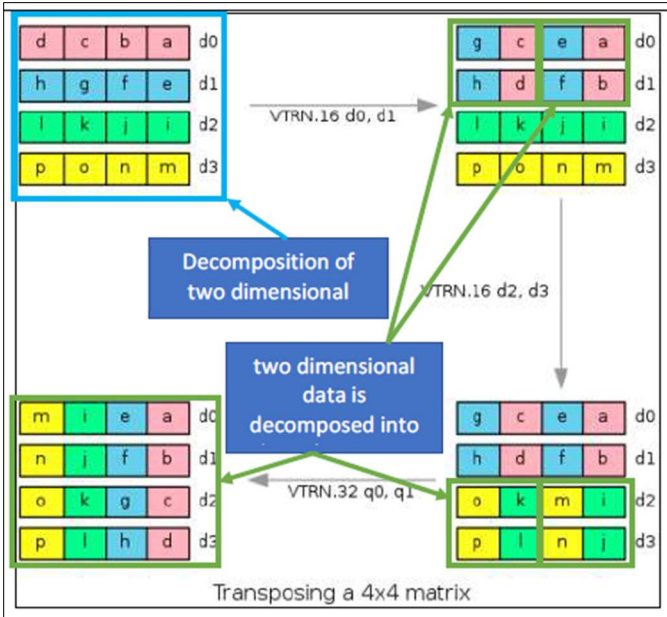
decomposing said two dimensional data into at least one atomic element said two dimensional data being located in at least one source register said at least one atomic element of said two dimensional data is a 2×2 matrix and said two dimensional data is decomposed into data elements in said matrix;

When writing code for Neon, you may find that sometimes, the data in your registers are not quite in the correct format for your algorithm. You may need to rearrange the elements in your vectors so that subsequent arithmetic can add the correct parts together, or perhaps the data passed to your function is in a strange format, and must be reordered before your speedy SIMD code can handle it.

VTRN: Transpose
VTRN transposes 8, 16 or 32-bit elements between a pair of vectors. It treats the elements of the vectors as 2x2 matrices, and transposes each matrix.



Use multiple VTRN instructions to transpose larger matrices. For example, a 4x4 matrix consisting of 16-bit elements can be transposed using three VTRN instructions.



Source: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/coding-for-neon---part-5-rearranging-vectors

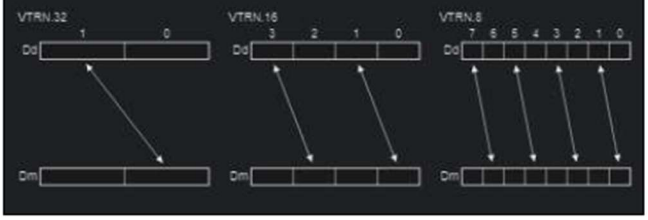
determining at least one permutation instruction for rearrangement of said data in said atomic element;

VTRN
Vector Transpose **treats the** elements of its operand vectors **as** elements of 2 × 2 matrices, **and** transposes the matrices.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

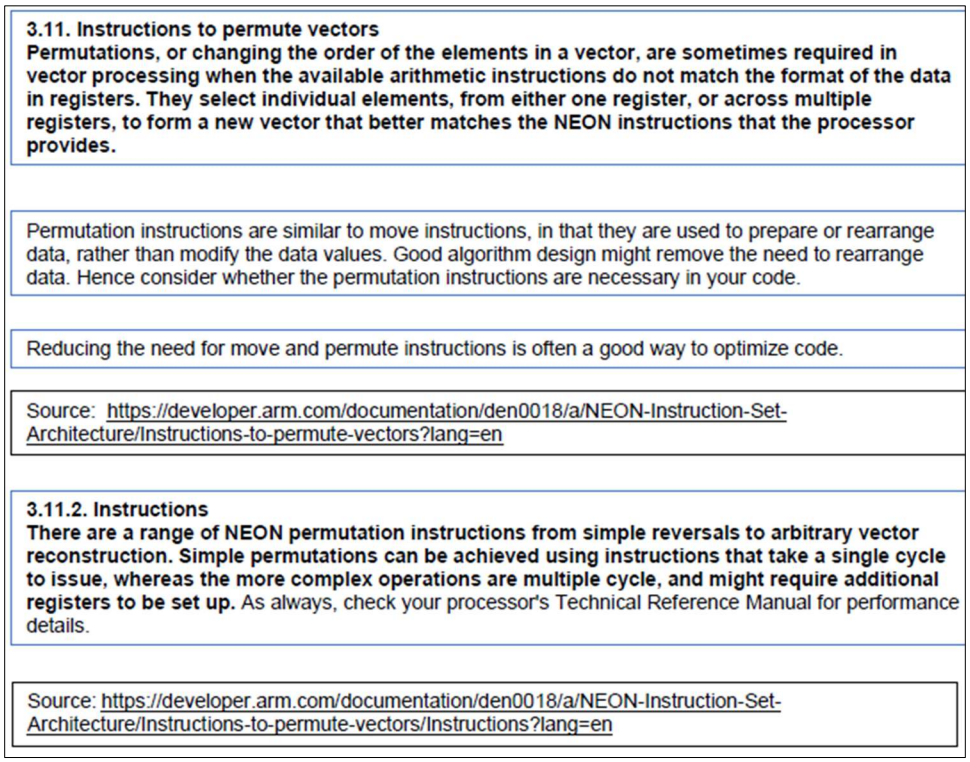
Figure 8.7 shows the operation of doubleword VTRN. Quadword VTRN performs the same operation as doubleword VTRN twice, once on the upper halves of the quadword vectors, and once on the lower halves

Figure 8.7. VTRN doubleword operation



Source: https://developer.arm.com/documentation/ddi0406/cb/Application-Level-Architecture/Instruction-Details/Alphabetical-list-of-instructions/VTRN

said data elements being rearranged by said at least one permutation instruction, each of said data elements representing a subword having one or more bits; and



3.11. Instructions to permute vectors
Permutations, or changing the order of the elements in a vector, are sometimes required in vector processing when the available arithmetic instructions do not match the format of the data in registers. They select individual elements, from either one register, or across multiple registers, to form a new vector that better matches the NEON instructions that the processor provides.

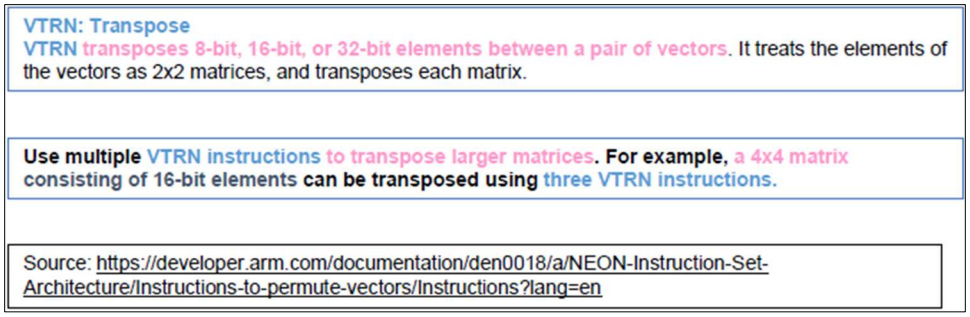
Permutation instructions are similar to move instructions, in that they are used to prepare or rearrange data, rather than modify the data values. Good algorithm design might remove the need to rearrange data. Hence consider whether the permutation instructions are necessary in your code.

Reducing the need for move and permute instructions is often a good way to optimize code.

Source: https://developer.arm.com/documentation/den0018/a/NEON-Instruction-Set-Architecture/Instructions-to-permute-vectors?lang=en

3.11.2. Instructions
There are a range of NEON permutation instructions from simple reversals to arbitrary vector reconstruction. Simple permutations can be achieved using instructions that take a single cycle to issue, whereas the more complex operations are multiple cycle, and might require additional registers to be set up. As always, check your processor's Technical Reference Manual for performance details.

Source: https://developer.arm.com/documentation/den0018/a/NEON-Instruction-Set-Architecture/Instructions-to-permute-vectors/Instructions?lang=en



VTRN: Transpose
VTRN transposes 8-bit, 16-bit, or 32-bit elements between a pair of vectors. It treats the elements of the vectors as 2x2 matrices, and transposes each matrix.

Use multiple VTRN instructions to transpose larger matrices. For example, a 4x4 matrix consisting of 16-bit elements can be transposed using three VTRN instructions.

Source: https://developer.arm.com/documentation/den0018/a/NEON-Instruction-Set-Architecture/Instructions-to-permute-vectors/Instructions?lang=en

applying said permutation instructions to said subwords and placing said permutated subwords into a destination register.

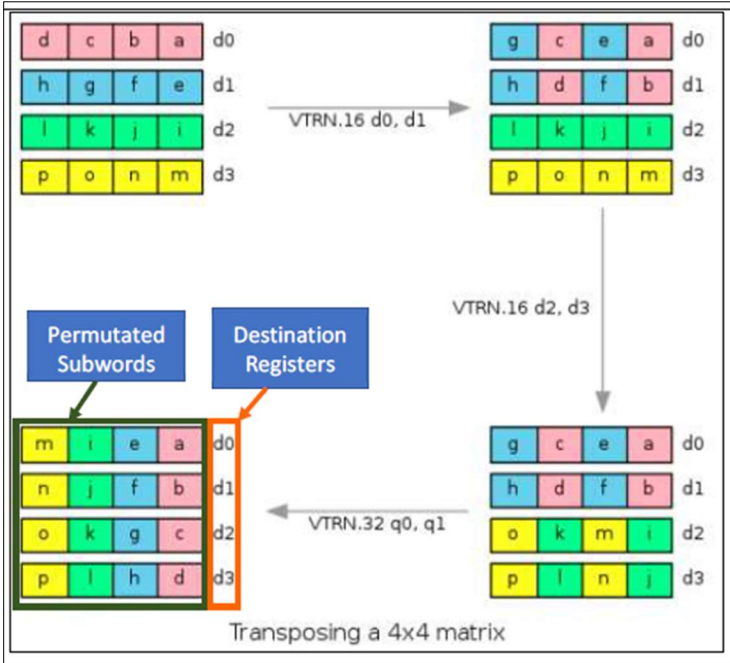
When writing code for Neon, you may find that sometimes, **the data in your registers are not quite in the correct format for your algorithm. You may need to rearrange the elements in your vectors so that subsequent arithmetic can add the correct parts together, or perhaps the data passed to your function is in a strange format, and must be reordered before your speedy SIMD code can handle it.**

Neon provides a range of permutation instructions, from basic reversals to arbitrary vector reconstruction. Simple permutations can be achieved using instructions that take a single cycle to issue, **whereas the more complex operations use multiple cycles, and may require additional registers to be set up.** As always, benchmark or profile your code regularly, and check your processor's Technical Reference Manual (Cortex-A8, Cortex-A9) for performance details.

VTRN: Transpose
VTRN **transposes 8, 16 or 32-bit elements between a pair of vectors. It treats the elements of the vectors as 2x2 matrices, and transposes each matrix.**

Use multiple VTRN instructions to transpose larger matrices. For example, a 4x4 matrix consisting of 16-bit elements can be transposed using three VTRN instructions.

Source: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/coding-for-neon---part-5-rearranging-vectors



Source: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/coding-for-neon---part-5-rearranging-vectors

29. Teleputers is entitled to recover from Defendant the damages as a result of Defendant's infringement of the '526 patent in an amount subject to proof at trial, which, by law, cannot be less than a reasonable royalty, together with interest and costs as fixed by this Court under 35 U.S.C. § 284.

PRAYER FOR RELIEF

WHEREFORE, Plaintiff respectfully requests the Court enter judgment against Defendant:

WHEREFORE, Teleputers requests that this Court enter judgment against Defendant as follows:

A. An adjudication that Defendant has infringed the '478 and '526 patents;

B. An award of damages to be paid by Defendant adequate to compensate Teleputers for Defendant's past infringement of the '478 and '526 patents and any continuing or future infringement through the date such judgment is entered, including interest, costs, expenses and an accounting of all infringing acts including, but not limited to, those acts not presented at trial;

C. A declaration that this case is exceptional under 35 U.S.C. § 285, and an award of Teleputer's reasonable attorneys' fees; and

D. An award to Teleputers of such further relief at law or in equity as the Court deems just and proper.

JURY DEMAND

Plaintiff demands trial by jury, Under Fed. R. Civ. P. 38.

Dated: November 9, 2023

Respectfully Submitted

/s/ Raymond W. Mort, III

Raymond W. Mort, III
Texas State Bar No. 00791308
raymort@austinlaw.com

THE MORT LAW FIRM, PLLC
501 Congress Ave, Suite 150
Austin, Texas 78701
Tel/Fax: (512) 865-7950

ATTORNEYS FOR PLAINTIFF