

# A Register File with Transposed Access Mode

Yoochang Jung, Stefan G. Berg, Donglok Kim, and Yongmin Kim  
Image Computing Systems Laboratory  
University of Washington  
Box 352500  
Seattle, WA 98195-2500  
{ mrchang, sgberg, dong, ykim }@icsl.ee.washington.edu

## Abstract

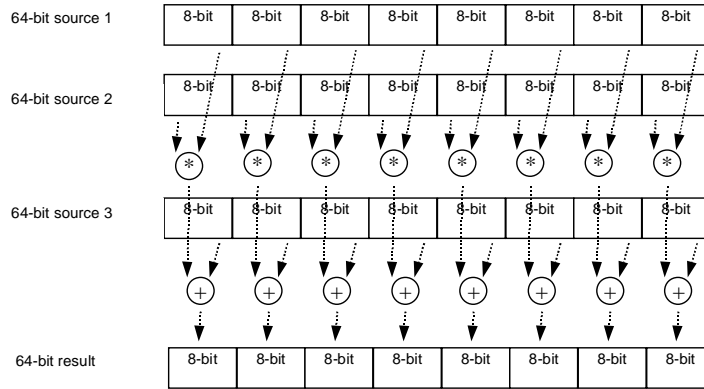
We introduce a new register file architecture that provides both row-wise and column-wise accesses, thus allowing partitioned instructions to be used in column-wise processing without transposition overhead. This feature can accelerate 2D separable image and video processing algorithms, such as 2D convolution and 2D discrete cosine transform (DCT), by eliminating the transposition steps.

**Keywords:** register file, transpose, convolution, DCT

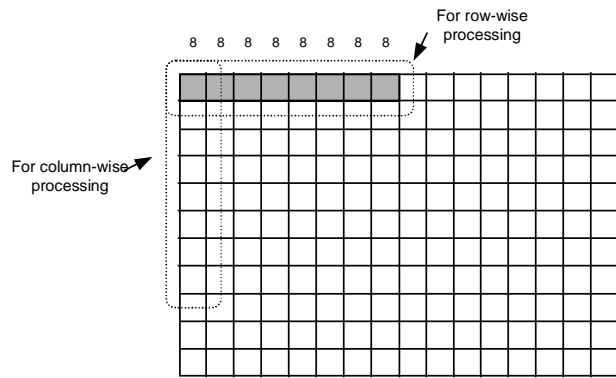
## 1. Introduction

2D convolution and 2D transforms, such as wavelet transform and discrete cosine transform (DCT), are widely used in image and video processing. To reduce the computation complexity, these algorithms are often implemented in two separable passes of 1D processing (e.g., row-wise processing followed by column-wise processing). For example, the number of multiplications of a direct  $N \times N$  2D DCT is  $N^4$ , while it is  $2N^3$  if two passes of 1D DCTs are used.

Many image and video processing algorithms handle data elements that are smaller than a register size. Mediaprocessors take advantage of this property by employing partitioned instructions that can simultaneously process multiple data elements packed into one register [1]. Figure 1 shows a partitioned-multiply-add instruction that performs eight 8-bit multiply accumulations (MAC) in parallel using a 64-bit data path. In performing the two separable passes of 1D processing, these partitioned instructions can be useful for row-wise processing. However, since the data are not stored in consecutive memory locations for column-wise processing, as shown in Figure 2, a transposition on the row-wise processing result before entering the column-wise processing and another transposition on the column-wise processing result have to be applied to utilize the partitioned instructions.

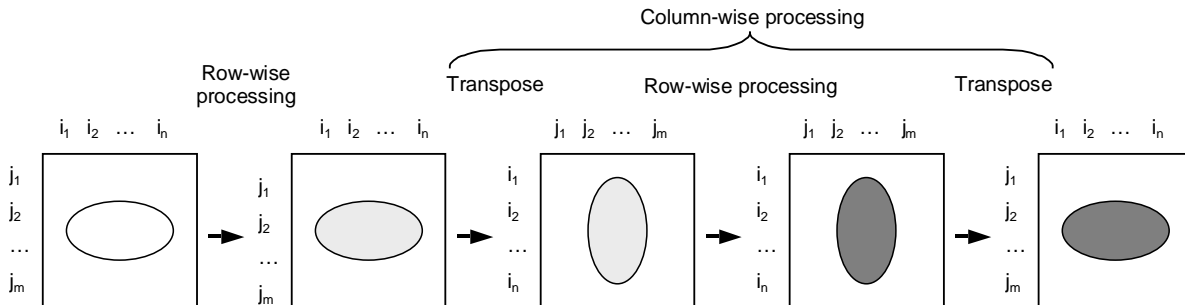


**Figure 1. An eight 8-bit partitioned-multiply-add instruction.**



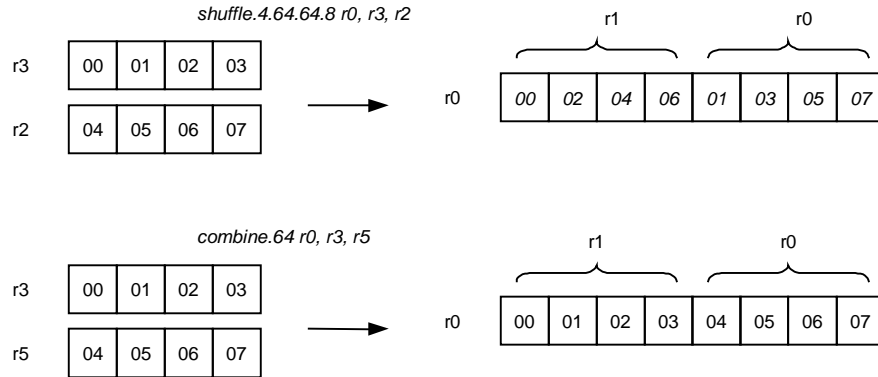
**Figure 2. A problem in column-wise processing with partitioned instructions.**

Figure 3 shows a typical method of implementing separable algorithms using partitioned instructions. First, we perform row-wise processing utilizing partitioned instructions, then transpose the result to prepare the data in a column to be stored consecutively in memory. Column-wise processing can now use partitioned instructions by accessing the data in the same way as row-wise processing. Finally, the result is transposed again. To transpose an  $8 \times 8$  8-bit block, it takes 112 instructions (56 loads and 56 stores) in a typical reduced instruction set computer (RISC) architecture (note that eight diagonal data elements do not need to be transposed). Some mediaprocessors feature data movement instructions that can be used to transpose a small 2D block of data more efficiently [2, 3].



**Figure 3. An implementation of separable algorithms using partitioned instructions.**

For example, the MAP1000 mediaprocessor [4] provides special instructions that can perform an  $8 \times 8$  8-bit block transposition in 40 instructions. Figure 4 shows the two instructions (*shuffle* and *combine*) that can be used to transpose an  $8 \times 8$  8-bit block. The shuffle instruction takes two 32-bit source operands and interleaves them into a 64-bit destination register, which is represented as a consecutive even-odd pair of 32-bit registers. The combine instruction generates a 64-bit result by concatenating two arbitrary 32-bit registers.



**Figure 4. Shuffle and combine instructions.**

There are two steps for transposition: shuffle and combine. In the shuffle step, one  $8 \times 8$  8-bit block is divided into four  $4 \times 4$  8-bit blocks. Two groups of shuffle instructions are used to generate four transposed local  $4 \times 4$  8-bit blocks as shown in Figure 5. The numbers 1 to 4 show how each column in the upper-left  $4 \times 4$  block is being transposed. The four transposed  $4 \times 4$  blocks are combined together to become a transposed  $8 \times 8$  block in the combine step as shown in Figure 6. In this example, we need a total of 40 instructions, i.e., 8 load, 16 shuffle, 8 combine, and 8 store instructions, to transpose an  $8 \times 8$  8-bit block of data.

In spite of these special instructions, however, transposition is still expensive. For example, the number of instructions needed for transposition in the Chen’s DCT implementation [6] on the MAP1000 takes about 32% of the total number of instructions (see Section 3.2). In this paper, we describe a register file architecture that can eliminate the instructions required for transpositions by allowing the programmers to access the registers both row-wise and column-wise directly.

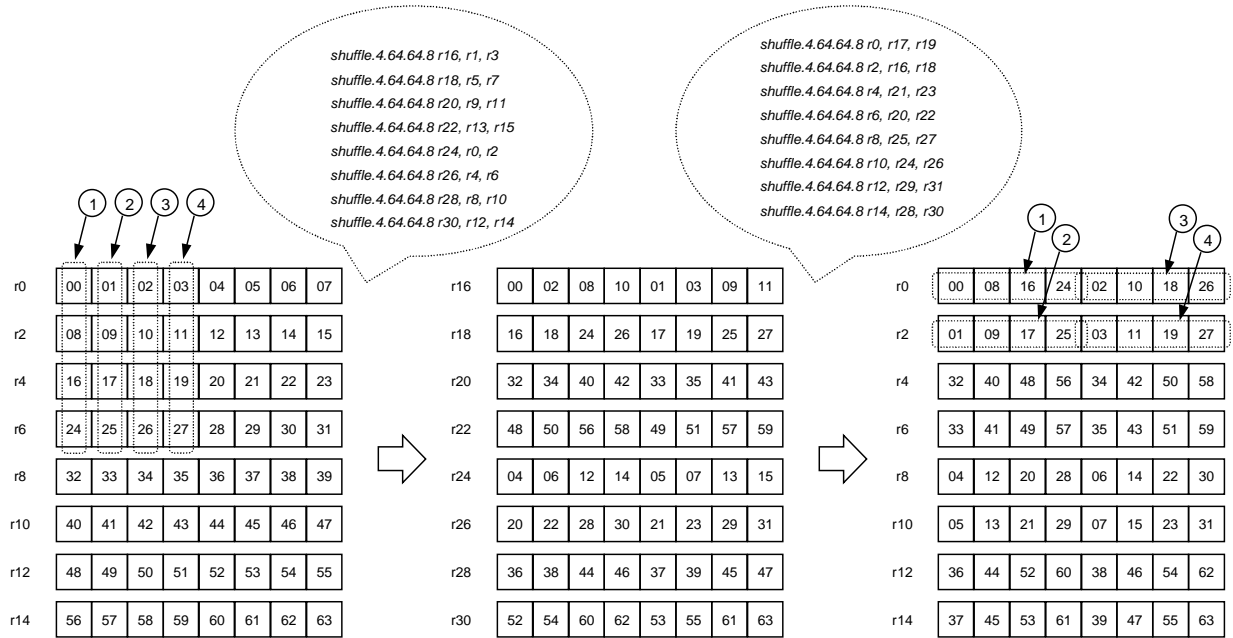


Figure 5. An 8 x 8 8-bit matrix transpose on the MAP1000 (shuffle step).

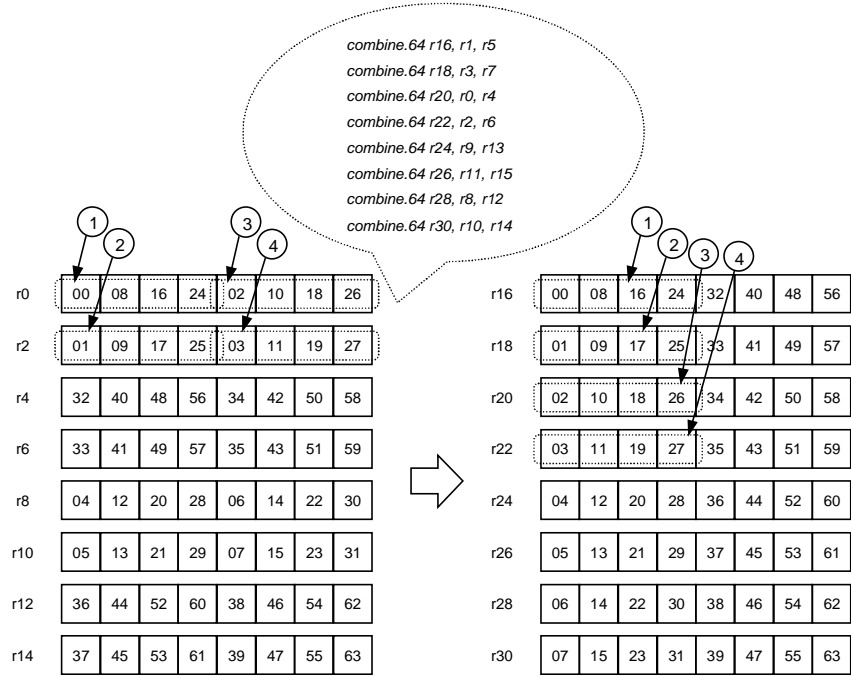
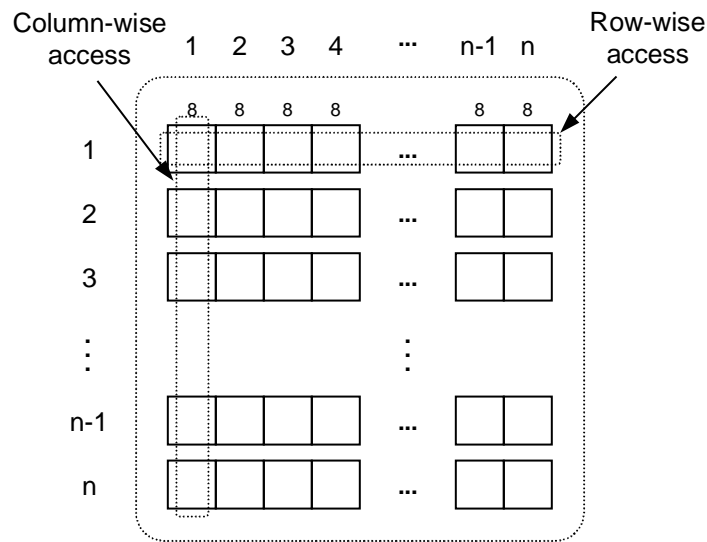


Figure 6. An 8 x 8 8-bit matrix transpose on the MAP1000 (combine step).

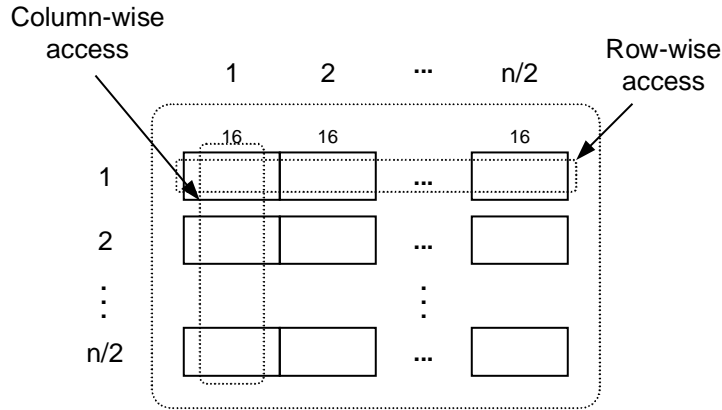
## 2. A transposable register file

Figure 7 shows a transposable register file with 8-bit partitions. Each register consists of  $n$  8 bits of data and therefore has a total width of  $n \times 8$  bits. This register file contains  $n$  registers to provide  $n \times n$  transposition. There are two access modes: normal and transposed. The register blocks are accessed row-wise in the normal access mode, while column-wise access is used in the transposed access mode. Note that we need  $n$  registers to transpose an  $n \times n$  8-bit block of data. In image and video processing, 16-bit data are also frequently used as well as 8-bit data. Figure 8 shows a transposable register file that allows 16-bit column-wise access. In this case, we need only  $n/2$  registers ( $n$  should be even) compared to  $n$  registers in the 8-bit transposable register file since there are  $n/2$  16 bits of data in a register.



**Figure 7. A transposable register file with 8-bit partitions.**

For transposition, either a normal write followed by a transposed read or a transposed write – normal read mode should be sufficient. In this paper, we show the transposed write – normal read mode since it generally requires less hardware overhead. This is because the transposition requires additional hardware for each port that supports it and there are fewer write ports than read ports on typical register files, e.g., there is one destination operand field and two or three source operand fields in typical instruction bit fields. We will therefore assume that transposition is only desired on the write ports of the register file, although read ports can be equipped with this capability, too.



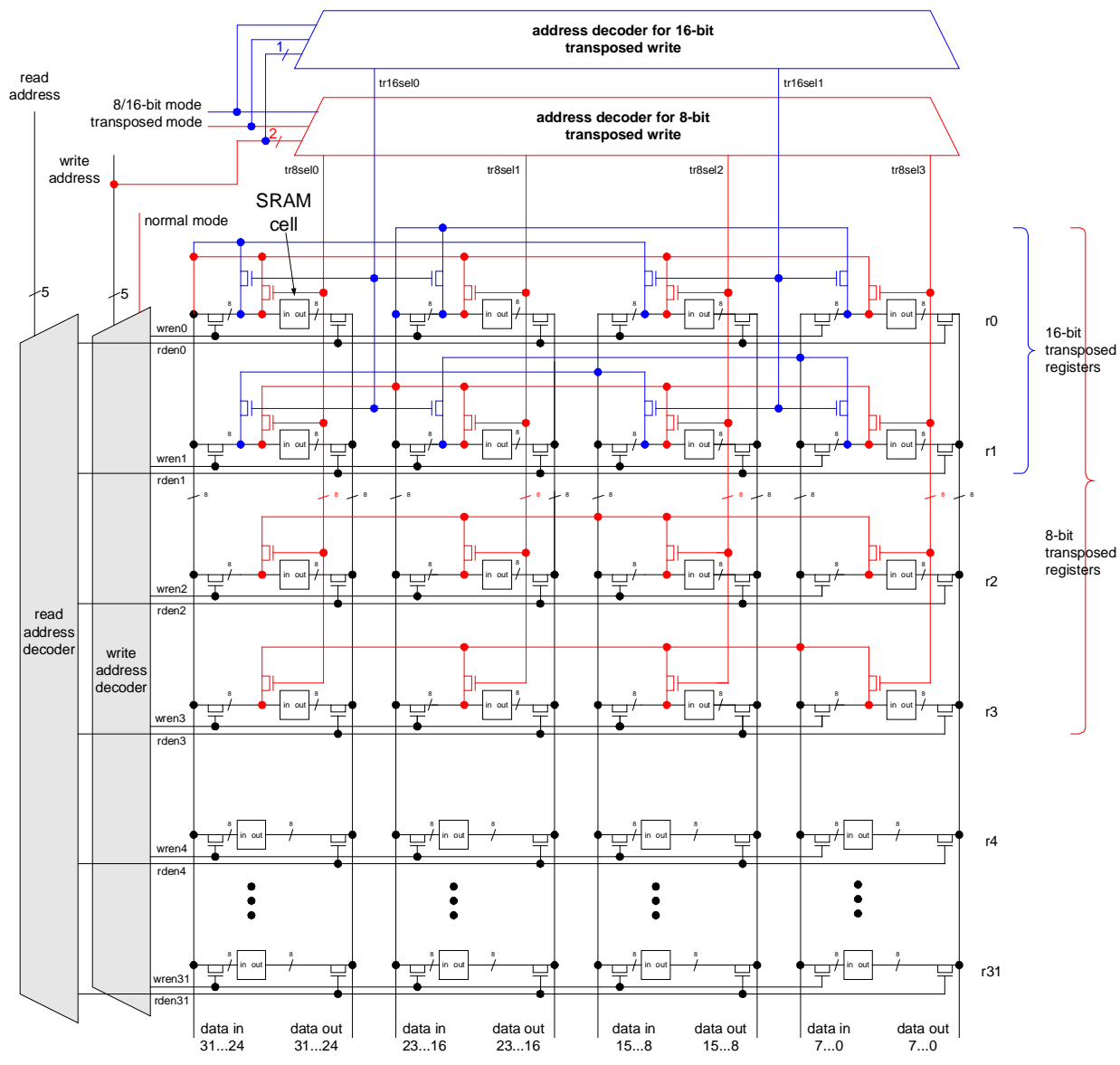
**Figure 8. A transposable register file with 16-bit data.**

Figure 9 shows an implementation of the transposable register file utilizing the transposed-write mode. The register file has thirty-two 32-bit registers. For 16-bit data transposition, the first two registers ( $r_0$  and  $r_1$ ) are used. For 8-bit data transposition, the first four registers ( $r_0 \dots r_3$ ) are used. This design extends naturally to wider-bit register files, but for illustrative purposes, we describe a 32-bit transposable register file only in this paper.

The implementation shown in Figure 9 is based on a typical design for a register file with separate read and write ports [5]. To reduce the complexity of the figure, we have combined eight bits into one SRAM cell. A row of four 8-bit SRAM cells forms a register with register 0 at the top. When reading from register 3, for example, the fourth row of SRAM cells will be selected by  $rden_3$  through the read address decoder while all other rows remain inactive. The register contents will appear on the data-out wires at the bottom of the figure.

The two decoders at the top of the figure add the 8-bit and 16-bit transposed write capability. We describe the 8-bit transpose operation first. The key enabling component for the 8-bit transposed write mode is its address decoder at the top. Instead of selecting a row of SRAM cells, each of its enable lines ( $tr8sel_0 \dots tr8sel_3$ ) selects a column of SRAM cells that contain the transposed values. The data-in values in the selected column are transferred to the corresponding input to the SRAM cells. For example, when accessing the transposed register  $r_0$  (consisting of the four vertically-aligned SRAM cells at the far left), data-in 31...24 will be driven to the top-most SRAM cell, data-in 23...16 to the second SRAM cell, data-in 15...8 to the third SRAM cell and data-in 7...0 to the last SRAM cell.

The 16-bit transposed mode spans only two registers because only two 16-bit partitions can fit in a 32-bit register. Two SRAM cells are combined to form a single 16-bit partition. Similar to the 8-bit access mode, there is a separate address decoder at the top of the register file. This address decoder can only select one of the two 16-bit transposed registers, i.e., either the two left halves of  $r_0$  and  $r_1$  (via  $tr16sel_0$ ) or the two right halves of  $r_0$  and  $r_1$  (via  $tr16sel_1$ ). Therefore, the input data to the SRAM cells are correctly transferred from the corresponding data-in wires.



**Figure 9. A 32-bit transposable register file architecture.**

Registers r4 through r31 in Figure 9 are normal registers and have no transpose capability. However, it might be desirable to extend the transposition capability. For example, when there are multiple data blocks that need to be transposed, two separate register blocks for alternating data load and computation (called double buffering) can be used to better pipeline the computation. To provide this capability, registers r2 and r3 could be made 16-bit transposable and registers r4 to r7 could also be made 8-bit transposable.

Note also that a 64-bit wide register file (used in Section 3) can be designed by extending the register size and doubling the number of registers required for the transposition. In other words, 8-bit transposed access would require eight registers while 16-bit transposed access would require four registers if the register file is extended to 64 bits.

### 3. Examples

In this section, we show three examples to see the effect of the transposable register file. We use MAP1000 instructions in these examples. Image transposition is a common example that can be used in many 2D separable algorithms. An  $8 \times 8$  16-bit block DCT example shows that the two separate transposition steps can be completely hidden when all the data fit in the register file. A 2D separable convolution example shows that this architecture is useful for the algorithms that can be processed in a block-wise fashion.

#### 3.1 Image transpose

Figure 10 shows an example of  $8 \times 8$  8-bit block transposition without the transposable register file. *bsld.64* is a 64-bit load instruction in big endian, which takes three parameters, i.e., a destination register, a pointer register and an offset that is added to the pointer register to generate the effective address. The unit of the offset is 64 bits, e.g., the offset of 1 (line 3) indicates the second 64-bit data from the pointer. *bsst.64* is a 64-bit store instruction in big endian where the first parameter works as a source register.

```
// load eight 64-bit data, i.e., an 8x8 8-bit block.
01: bsld.64 r0, r32, 0; /* r0 = *r32 */
02: bsld.64 r2, r32, 1; /* r2 = *(r32 + 8) */
03: bsld.64 r4, r32, 2; /* r4 = *(r32 + 8 * 2) */
04: bsld.64 r6, r32, 3; /* r6 = *(r32 + 8 * 3) */
05: bsld.64 r8, r32, 4; /* r8 = *(r32 + 8 * 4) */
06: bsld.64 r10, r32, 5; /* r10 = *(r32 + 8 * 5) */
07: bsld.64 r12, r32, 6; /* r12 = *(r32 + 8 * 6) */
08: bsld.64 r14, r32, 7; /* r14 = *(r32 + 8 * 7) */

09-32: // transpose the block using
        // 16 shuffle and 8 combine instructions. (See Figures 5 and 6)

// store back the result
33: bsst.64 r16, r33, 0; /* *r33 = r16 */
34: bsst.64 r18, r33, 1; /* *(r33 + 8) = r18 */
35: bsst.64 r20, r33, 2; /* *(r33 + 8 * 2) = r20 */
36: bsst.64 r22, r33, 3; /* *(r33 + 8 * 3) = r22 */
37: bsst.64 r24, r33, 4; /* *(r33 + 8 * 4) = r24 */
38: bsst.64 r26, r33, 5; /* *(r33 + 8 * 5) = r26 */
39: bsst.64 r28, r33, 6; /* *(r33 + 8 * 6) = r28 */
40: bsst.64 r30, r33, 7; /* *(r33 + 8 * 7) = r30 */
```

**Figure 10. An example of  $8 \times 8$  8-bit block transposition without the transposable register file.**

We load a data block that consists of eight 64-bit memory words, transpose the block using 16 shuffle and 8 combine instructions, and then store the result back to memory. Figure 11 shows the same  $8 \times 8$  block transposition with the transposable register file. We assume that we can access the register file in transposed mode when a postfix *t* is used, e.g., *r0.t*. In this example, 16 instructions are used for transposition compared to the 40 instructions used in Figure 10. Note

that, with the transposable register file, we can eliminate the 24 shuffle and combine instructions since transposition is done as the data are loaded into the register file.

```

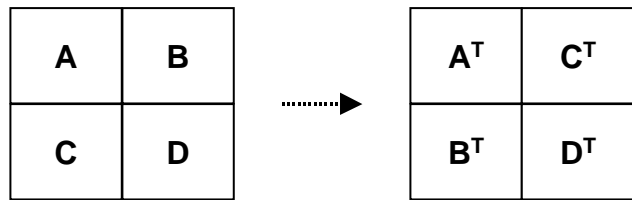
// load eight 64-bit data, i.e., an 8x8 8-bit block.
01: bsld.64 r0, r32, 0; /* r0 = *r32 */
02: bsld.64 r2, r32, 1; /* r2 = *(r32 + 8) */
03: bsld.64 r4, r32, 2; /* r4 = *(r32 + 8 * 2) */
04: bsld.64 r6, r32, 3; /* r6 = *(r32 + 8 * 3) */
05: bsld.64 r8, r32, 4; /* r8 = *(r32 + 8 * 4) */
06: bsld.64 r10, r32, 5; /* r10 = *(r32 + 8 * 5) */
07: bsld.64 r12, r32, 6; /* r12 = *(r32 + 8 * 6) */
08: bsld.64 r14, r32, 7; /* r14 = *(r32 + 8 * 7) */

// read out the data in transpose mode and store them
09: bsst.64 r0.t, r33, 0; /* *r33 = r0 */
10: bsst.64 r2.t, r33, 1; /* *(r33 + 8) = r2 */
11: bsst.64 r4.t, r33, 2; /* *(r33 + 8 * 2) = r4 */
12: bsst.64 r6.t, r33, 3; /* *(r33 + 8 * 3) = r6 */
13: bsst.64 r8.t, r33, 4; /* *(r33 + 8 * 4) = r8 */
14: bsst.64 r10.t, r33, 5; /* *(r33 + 8 * 5) = r10 */
15: bsst.64 r12.t, r33, 6; /* *(r33 + 8 * 6) = r12 */
16: bsst.64 r14.t, r33, 7; /* *(r33 + 8 * 7) = r14 */

```

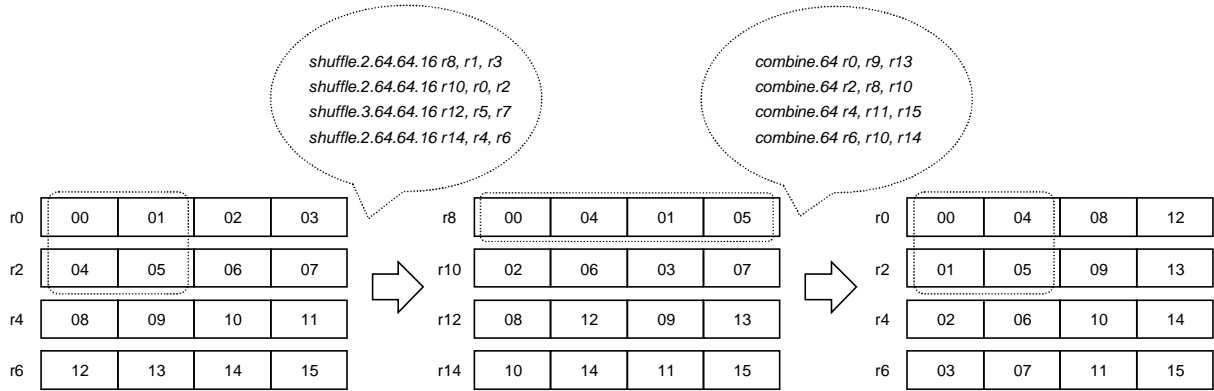
**Figure 11. An example of 8 x 8 8-bit block transposition with the transposable register file.**

To transpose an image whose size does not fit in the register file, we can divide the image into several smaller blocks so that each block can fit in the register file, then transpose each block, and rearrange them as shown in Figure 12.



**Figure 12. Block-wise transposition.**

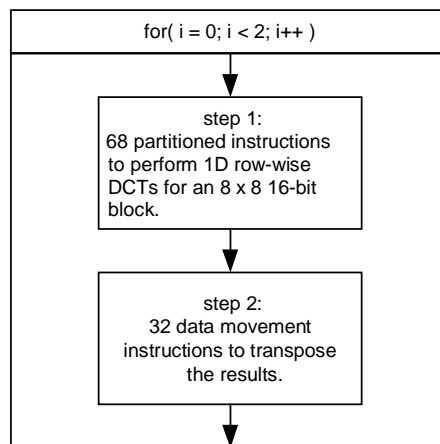
In case of 16-bit data, It takes 4 shuffle and 4 combine instructions on the MAP1000 as shown in Figure 13 to transpose a 4 x 4 16-bit block in addition to 4 load and 4 store instructions. Again, with the transposable register file, the eight instructions used for shuffle and combine can be eliminated.



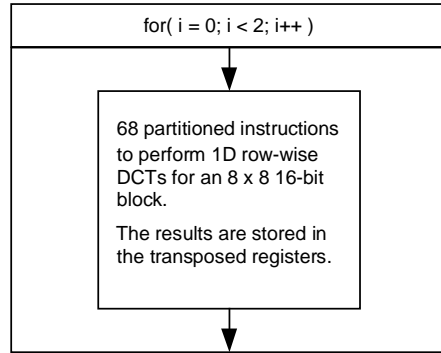
**Figure 13. 4 x 4 16-bit matrix transposition on the MAP1000.**

### 3.2 8x8 16-bit block DCT

Figure 14 shows an 8 x 8 16-bit 2D DCT implementation using the Chen's algorithm [6] without using the transposable register file. Since a 2D DCT is separable, we can compute a 2D DCT by performing two sets of 1D row-wise DCTs and two transpositions. In the first iteration ( $i=0$ ), we perform 1D DCTs for the 8 x 8 16-bit input data (step 1), and then transpose the result (step 2). Next ( $i=1$ ), we repeat these steps again. We use 16-bit partitioned-multiply-add/sub instructions (shown in Figure 1) to perform four 8-point 1D DCTs at the same time in the 64-bit data path, taking 68 instructions to process the flow graph of Chen's algorithm. For column-wise processing, we need two 8 x 8 16-bit transpositions. Since one 4 x 4 16-bit transposition takes 8 instructions as shown in Figure 13, 32 instructions are used for an 8 x 8 16-bit transposition. As a result, a total of  $(68 + 32) * 2 = 200$  instructions are required to compute one 2D 8 x 8 16-bit DCT. In this example, two transpositions take 64 instructions out of the total of 200 instructions required for an 8 x 8 2D DCT. With the transposable register file, we can eliminate the step 2 (transposition) as shown in Figure 15, thus it would take a total of 136 instructions to perform an 8 x 8 16-bit 2D DCT.



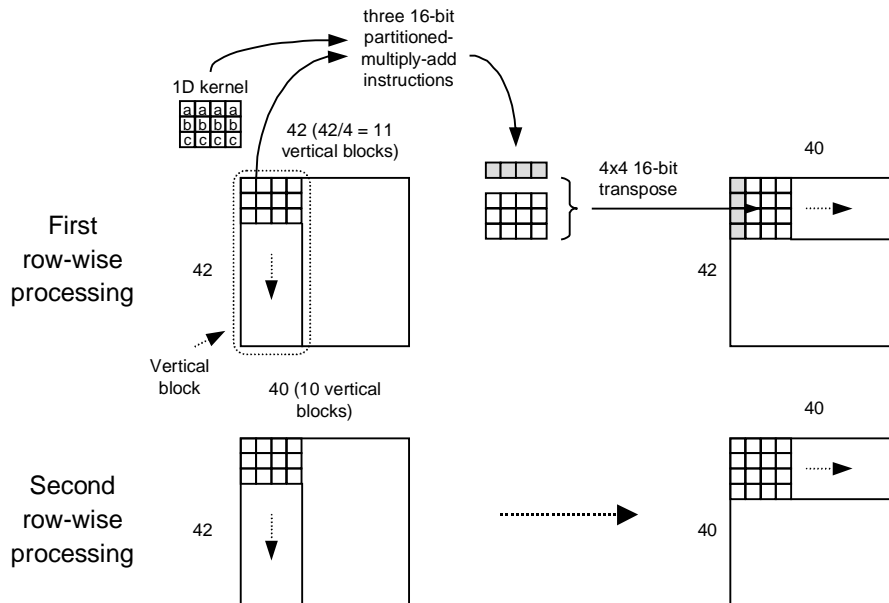
**Figure 14. 8 x 8 16-bit block DCT implementation without the transposable register file.**



**Figure 15. 8 x 8 16-bit block DCT implementation with the transposable register file.**

### 3.3 2D separable convolution

2D convolution kernels are not necessarily separable. However, there are several useful 2D kernels that can be decomposed into two vectors, e.g., Boxcar, Gaussian, and Daubechies wavelet. In this example, we perform 2D separable convolution on a  $42 \times 42$  16-bit block of data with a kernel length of 3. Figure 16 shows the details of the processing where the three kernel coefficients for the row-wise processing are a, b, and c. Three 16-bit partitioned-multiply-add instructions (shown in Figure 1) applied between the kernel and the three vertical input data generate four 16-bit results in the shaded block as shown in Figure 16. After four rows of such block results are generated, the rows are transposed and stored into local memory horizontally. Note that the  $42 \times 42$  input block becomes a  $42 \times 40$  block after the first row-wise processing and a  $40 \times 40$  block after the second processing, respectively.



**Figure 16. An example of block-wise 2D separable convolution.**

Since we process four 16-bit data in parallel using partitioned instructions, there are  $42 / 4 = 11$  vertical blocks in the first row-wise processing and  $40 / 4 = 10$  vertical blocks in the second one. Let us consider the number of instructions required for one vertical block only. Since there are 40 outputs and the tap length is 3, we need 120 partitioned-multiply-add instructions in addition to 42 *load* and 40 *store* instructions. Moreover, 40 *shuffle* and 40 *combine* instructions are required to transpose the ten  $4 \times 4$  blocks. Therefore, it takes a total of  $(42 + 40 + 120 + 40 + 40) * 11 = 3102$  instructions to process the 11 vertical blocks in the first row-wise processing. The second pass requires  $(42 + 40 + 120 + 40 + 40) * 10 = 2820$  instructions since there are 10 vertical blocks only. With the transposable register file, the 40 *shuffle* and 40 *combine* instructions can be completely hidden if we write the results to the register file in the transpose mode before we store them into memory. This case, it takes  $(42 + 40 + 120) * (11 + 10) = 4242$  instructions compared to the  $(3102 + 2820) = 5922$  instructions used in the implementation without the transposable register file, thus reducing the number of instructions by 28%.

## 4. Discussion

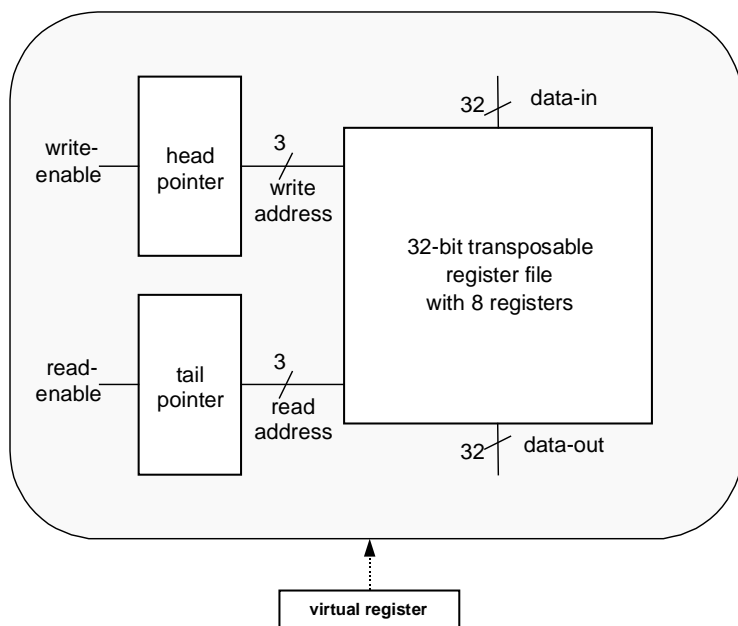
Table 1 compares the number of instructions taken in the examples in Section 3. The reduction in the number of instructions due to the transposable register file is significant with the factors ranging 1.4 to 2.5 compared to the regular methods.

**Table 1. Comparison of the number of instructions**

Examples (tight-loop only)	Without transposable register file	With transposable register file	Ratio
8x8 8-bit transpose	40	16	2.5 : 1
4x4 16-bit transpose	16	8	2.0 : 1
8x8 16-bit Chen's DCT	200	136	1.47 : 1
42x42 16-bit 2D convolution with a kernel length of 3	5922	4242	1.40 : 1

There are three aspects to the hardware cost of our technique of transposing registers. First, each transposable access port requires its own decoder and additional wiring within the register file structure. Second, the ability to transpose on different data widths, such as 8 bits and 16 bits, requires separate transposable access ports for each data width. Finally, the instruction set architecture will need to be able to address additional registers in its instruction to include the newly added transposed registers. Our choice of supporting transposition on the write port only can reduce this cost effectively, because there are typically fewer write ports than read ports. With only one destination field in an instruction, the instruction width would need to be increased by only a single bit to accommodate the additional four 8-bit transposed and two 16-bit transposed registers.

A transposable register file can hide the time-consuming transposition steps in many fundamental algorithms and allow us to explore various implementation methods. However, it increases the need for more number of registers when the data width increases since the required number of registers is proportional to the number of partitions in a register. For example, in a 64-bit architecture, we need 8 registers to transpose an  $8 \times 8$  8-bit data block. These eight registers cannot be used for other purposes until the transposition is completed. In another example, 32 registers are needed in a 256-bit architecture, which can transpose a  $32 \times 32$  8-bit data block. These 32 registers cannot be used for other purposes during transposition. In addition, when there is a long latency between writing and reading the register file caused by the processor pipeline, we cannot start reading the transposed data right after issuing an instruction that writes a result to the register file. Double buffering by providing more registers to resolve this latency issue will also increase the register pressure. To address these drawbacks with minimal impact on the instruction set architecture, we can consider a queue-based transposable register file as shown in Figure 17.



**Figure 17. A queue-based transposable register file.**

In this example, there are eight 32-bit registers in the queue. The queue is mapped into one virtual register that can be used as a source or destination operand in the instructions. We assume that transposition is done through the write port. The destination and source registers are selected by separate head and tail pointers, respectively. When a datum is written to the virtual register, it is stored in the register pointed to by the head pointer, and the head pointer is incremented by 1. Similarly, when a datum is read from the virtual register, the value stored in the register pointed to by the tail pointer is returned and the tail pointer is incremented by 1. The two pointers are circular so that they become 0 when they are incremented beyond seven. The reason that we have eight registers in Figure 17, i.e., twice more than we need, is to provide double buffering. In a normal transposable register file, it is difficult to double the number of transposable registers for double buffering since the register pressure increases as well. The queue-based transposable register file can lower this register pressure at the cost of sacrificing the random access

capability. In many cases, however, this sequential access to the 2D array being transposed is not a problem since we can schedule the order of output.

## 5. Conclusion

We have presented a transposable register file architecture for reducing the number of instructions for transposing a block of data. This technique allows us to access the data in a register file in both row-wise and column-wise directions. It requires no extra instructions in accessing the transposed data in the register file. For example, the results of row-wise processing can be stored in a transposed fashion and directly used as source operands in the subsequent column-wise processing. The transposable register file requires many registers during transpositions especially on wide data path architecture, thus could increase the register pressure. A queue-based transposable register file was proposed to lower this register pressure. We have shown three examples that demonstrate the effect of the transposable register file in image and video computing algorithms, which resulted in the reduced number of instructions by factors ranging 1.4 to 2.5 compared to the regular implementation methods.

## References

1. S. Rathnam and G. Slavenburg, "Processing the new world of interactive media," *IEEE Signal Processing Magazine*, vol. 15, no. 2, pp. 108-117, 1998.
2. B. Eitan, B. Nissenbaum, M. Feder, "Method for performing an inverse cosine transfer function for use with multimedia information," US Patent #5754457, 1998.
3. N. Sidwell, "System and method for restructuring data strings," US Patent #5822619, 1998.
4. C. Basoglu, R. Gove, K. Kojima, and J. O'Donnell, "A single-chip processor for media applications: the MAP1000," *International Journal of Imaging Systems and Technology*, vol. 10, pp. 96-106, 1999.
5. N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, 2<sup>nd</sup> Ed., Addison-Wesley, Reading, MA, 1993.
6. K. Rao and J. Hwang, *Techniques & Standards for Image, Video, and Audio Coding*. Prentice-Hall, Englewood Cliffs, NJ, 1996.