Using the Intrinsity™ FastMATH™ Processor for Hard Copy Imaging Applications

1. Introduction

Hard copy imaging devices (printers), must implement a wide range of functions to cover the gamut of applications necessary for success in the market. Printing an image today requires that the printer implement such diverse image processing functions as image compression and decompression, filtering, color space transformation, rotation, translation and scaling. Intrinsity™'s FastMATH™ processor, with its matrix unit, is capable of extremely high performance on these math-intensive functions. This paper briefly discusses four common image processing functions that are important in hard copy imaging devices:

- Discrete cosine transform (DCT) — this function and its inverse are used extensively in lossy image compression schemes, such as the JPEG compression standard
- Fast Fourier transform (FFT) — the FFT is used in image convolution and filtering functions
- Image rotation — images, in the form of pixel maps, frequently require rotation to align images or prepare rendered pages for efficient transmission to print engines
- Huffman decoding — Huffman decoding is a bottleneck in most lossy image decompression routines, such as JPEG.

2. Overview of the FastMATH Processor

The FastMATH processor is a high-performance processor designed to handle problems that require large amounts of vector- and matrix-based computation. It consists of:

- A MIPS32™ processor core with 16-Kbyte instruction and data caches
- An on-chip matrix and vector math unit, with a 4 × 4 array of 32-bit processing elements, capable of extremely fast matrix operations and parallel instruction execution in single instruction, multiple data (SIMD) mode
- An integrated 1-Mbyte L2 cache, which can be configured by halves as static random-access memory (SRAM) (if desired)
- Two bidirectional RapidIO™ ports
- An integrated 64-bit double-data-rate synchronous dynamic random-access memory (DDR-SDRAM) controller

The FastMATH processor executes the standard MIPS32 instruction set on a scalar processor, along with a set of vector and matrix instruction set extensions implemented through the MIPS® coprocessor 2 interface. It can issue and execute a core (scalar) instruction and a matrix coprocessor instruction every cycle.

The FastMATH processor includes a matrix/vector math unit that has 16 processing elements. Each element consists of a 16-entry, 32-bit register file, a 32-bit, single-cycle ALU that can perform 32-bit and paired 16-bit SIMD operations, and a 16-bit × 6-bit multiplier with two 40-bit accumulators. The elements are connected in a row/column mesh with the ability to broadcast to and receive values from the other elements in the same row or column. The result is a math coprocessor that can be viewed as having 16 matrix registers, each of which is a 4 × 4 array of 32-bit elements, each of which has a dedicated arithmetic unit capable of executing arithmetic functions, multiplication and multiply-add functions and logic functions such as `AND` and `OR`. Each matrix instruction is executed by all elements of the

matrix unit in a parallel, SIMD fashion, to provide a peak execution rate of 64 giga operations per second (GOPS) (e.g., 16 parallel multiply-accumulates at 2 GHz). Each vector instruction operates on 16 32-bit elements or 32 16-bit elements simultaneously, leading to extremely efficient implementation of vectorizable problems. In addition to standard vector arithmetic, the matrix unit implements instructions that can exploit the two-dimensional matrix interconnection, such as matrix-multiply, transpose, and block rearrangement. This unique capability allows the FastMATH processor to implement algorithms that can be expressed as a matrix. Data is supplied to the matrix unit via a 64-byte wide direct connection to the integrated L2 cache.

3. DCT Implementation on the FastMATH Processor

Because of its importance in image and video compression in the past 20 years, a plethora of algorithms for implementing the DCT have been proposed. Each has its strengths and weaknesses, depending on the hardware resources available in a particular system. Since the FastMATH processor has native support for matrix arithmetic, we selected a matrix formulation of the DCT.

3.1. The Chen Algorithm

The most efficient algorithm for use on a processor that has native hardware support for matrix-multiplication, such as the FastMATH processor, is the Chen algorithm [1]. Briefly, a two-dimensional (2-D) DCT can be performed by performing a one-dimensional (1-D) DCT on each row of the input array and then performing a 1-D DCT on each column. Without presenting the derivation, the Chen algorithm can be summarized as follows:

1. First, recast the standard vector formulation of the 1-D DCT into a form in which a matrix is multiplied by a vector.
2. Decompose the matrix recursively into submatrices, some of which are the identity matrix or its mirror or permutation matrices.

The decomposition is useful, since it provides a simplification of operations: dot products of the rows of the identity matrices do not require the full multiplication and addition of all terms since all but one term is zero and the remaining term has the value 1.

Thus, a single 1-D DCT can be performed on a single row by a matrix-vector multiplication. For the 8×8 case, a further optimization can be achieved by the recognizing that all eight rows of an 8×8 input matrix can be transformed by augmenting the input vector into an input matrix. Thus, a single 8×8 matrix multiplication can perform eight 1-D DCTs. Lastly, it is recognized that performing 1-D DCTs on the columns of this result is the same as transposing the matrix (interchanging rows and columns) and performing the eight 1-D DCTs on the rows, which are the columns of the original input.

3.2. Implementation on the FastMATH Processor

The FastMATH processor has 16 matrix registers. Each register is a 4×4 array of 32-bit elements. Each 32-bit element can be treated as two 16-bit halfwords. Arithmetic can be performed on these halfwords using the high-order halfwords, low-order halfwords, or any combination as inputs. The matrix-multiply instruction can multiply two 4×4 matrices in seven clock cycles and another one can be issued every four cycles. There is also a transpose instruction that can transpose a 4×4 matrix with the same timing. These two instructions form the basis of the DCT routine.

The routine is performed as follows:

1. The 8×8 matrix of input values are stored as 16-bit real values. Since each matrix register can hold eight 16-bit values in row (two halfwords per element) and has four rows, each register holds a single 4×8 matrix of 16-bit real values. Thus, the 8×8 input matrix is stored into two matrix registers.
2. A few vector adds and subtracts form the input vectors.
3. Four matrix-multiplications perform eight 1-D DCTs on the rows of the input matrix.
4. Two transpose instructions transpose the results of step 3, interchanging the rows and columns.

5. Four matrix-multiplications perform eight 1-D DCTs on the columns (now the rows) of the input matrix
6. Re-arrange the output data.

If there are a large number of DCTs to be performed, this implementation of the 8 × 8 2-D DCT requires 64 clock cycles on the FastMATH processor. At a clock frequency of 2 GHz, this is equivalent to 31.6 million 8 × 8 DCTs per second.

4. The Fast Fourier Transform

The fast Fourier transform (FFT) is one of the most fundamental transformations in signal processing, but it is not often used in imaging. The reason for its lack of popularity is not its lack of utility, but its high computation cost. In hard copy imaging, there are many cases in which an image may need to be filtered. One way to accomplish this is by convolving the image with a filter function by transforming the image from the spatial domain to the frequency domain, multiplying by a frequency domain filter, and then transforming the image back into the spatial domain. With the advent of the availability of the FastMATH processor, this becomes feasible.

4.1. The 2-D FFT

Like the DCT described above, the familiar one-dimensional FFT can be generalized to higher dimensions. A two-dimensional (2-D) FFT can be performed by applying the 1-D forward FFT in place on each row and then applying the 1-D FFT on each column of that output. The most efficient way to accomplish this on the FastMATH processor is by adopting a different strategy for each 'direction'. Performing the FFT on the rows is fairly straightforward because of the pattern of memory accesses needed to address the rows. Normally, the matrix would be transposed and the FFT applied again to the rows (columns). However, if there are 16 or more columns, it is more efficient to vectorize the problem and calculate 16 FFTs (one on each column) simultaneously, without paying the penalty of the transpose.

4.2. Radix-4 FFT

The radix-4 FFT is the most efficient choice for implementation on the FastMATH processor because it affords the most opportunity for parallelism and because it is well balanced between access bandwidth and computation. The FastMATH processor can be fully utilized on the radix-4 butterfly. The basic radix-4 butterfly has four inputs (complex numbers), three complex multiplicands (twiddle factors), and produces four complex outputs. Each output consists of various combinations of the sum and difference of the inputs multiplied by the twiddle factors.

The most efficient way to perform complex arithmetic on 16-bit operands is to interleave the data. That is, a 16-bit real component and a 16-bit imaginary component are packed into a single 32-bit value. This is shown in Figure 1.

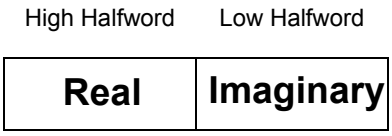

Figure 1: Interleaved Complex data (16-bit values)

The FastMATH processor has native support for this interleaved complex data type. It has a full set of 16-bit instructions including signed and unsigned addition, subtraction, multiply, multiply-add, multiply-subtract. The result of the 16 × 16 multiply-add is placed in a 40-bit accumulator. This allows 256 multiply-add or multiply operations before any saturation or scaling needs to be applied. All of these are SIMD instructions that simultaneously compute 16 results for each instruction. To support complex arithmetic on the packed format, each instruction has variants to operate on the high halfword (most significant 16 bits), the low halfword (least significant 16 bits), and all combinations. For instance, there are instructions to multiply the high halfword of one source operand by the low halfword of the other source operand. This allows a complex-multiply to be accomplished in only four instructions. The data flow diagrams for `MultiplyAddLowLow`, `MultiplyAddHighHigh`, and `MultiplyAddHighLow` instructions are shown in Figure 2.

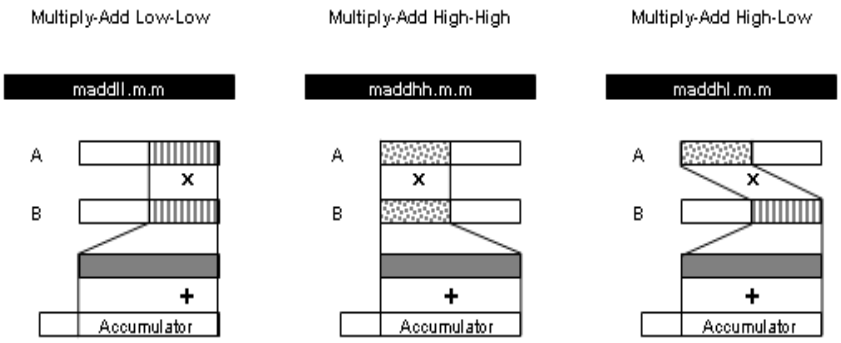

Figure 2: Data flow for `MultiplyAdd` instructions

The last two stages of the FFT use the `BLOCK4` and `BLOCK4V` instructions (described in Section 5, "Image Rotation"). These instructions allow the rearrangement of data to exploit interelement operations.

4.3. Performance

The full 2-D FFT is performed by first executing a 'horizontal' 1-D FFT on each row of the image, and then executing the vectorized 'vertical' FFT on the columns. The horizontal FFT takes 884 cycles per row and the vertical FFT takes 12,998 cycles for 16 columns. This results in a total of 434,272 cycles.

5. Image Rotation

The rotation of images is one of the fundamental functions needed in printer applications. Compositing images, such as in the PostScript® language, often requires that the images be translated, scaled and rotated. At times, it is convenient to render a page in a given orientation, and then transfer it to the print engine in a different orientation. This requires that large images be rotated. A common problem is rotation by 90 degrees. The matrix unit of the FastMATH processor allows very efficient manipulation of two-dimensional data. Since any rotation can be reduced to a series of rotations, this allows the efficient implementation of rotation on the processor.

5.1. Rotation Algorithm

The rotation of an image by 90 degrees is a special case of arbitrary rotation and can be separately optimized. The basic idea is to break the image into blocks, load each block into matrix registers, and then use the matrix unit instructions to rotate the blocks. The blocks are then stored into the proper (rotated) location in memory.

5.2. Implementation on the FastMATH Processor

The matrix unit within the FastMATH processor has 16 processing element arranged in a square 4×4 array. The elements are interconnected through a two-dimensional bus mesh that allows for rapid transfer of data within elements. The unit is capable of moving data from any element to any other element (or, indeed, all elements) within a row or column in a single clock cycle. Complex data movement functions are built using this mesh. For example, a matrix transpose can be accomplished in a total of seven cycles and a new transpose instruction can be started in only four cycles.

There are two specialized instructions that re-arrange the data within multiple matrix registers simultaneously: `BLOCK4` and `BLOCK4V`. These instruction can be used to rotate a 16×16 pixel image of 32-bit pixels, completely within the matrix unit. This primitive can then be used to rotate larger images.

5.2.1. The BLOCK4 Instruction

The `BLOCK4` instruction uses the mesh interconnect in the matrix unit to perform a four-way shuffle on the rows of four matrix registers. An example is given in Figure 3 below.

	Before	After
M0:	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	00 01 02 03 10 11 12 13 20 21 22 23 30 31 32 33
M1:	10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F	04 05 06 07 14 15 16 17 24 25 26 27 34 35 36 37

	Before	After
M2:	20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F	08 09 0A 0B 18 19 1A 1B 28 29 2A 2B 38 39 3A 3B
M3:	30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F	0C 0D 0E 0F 1C 1D 1E 1F 2C 2D 2E 2F 3C 3D 3E 3F

Figure 3: Data before and after `block4.m $m0`

The operation is as follows:

1. The source operand is a register number. The four consecutive registers starting with this number participate in the operation. The four registers are the source and the destination of the operation. Assume we start with matrix register zero (`$m0`).
2. The first row of `$m0` is placed in the first row of `$m0` (same register). The second row of `$m0` is placed in the first row of `$m1` (the next higher register number). The third row of `$m0` is placed in the first row of `$m2`, and the fourth row of `$m0` is placed in the first row of `$m3`.
3. The process is repeated starting with `$m1`. The first row of `$m1` is placed in the second row of `$m0`. The second row of `$m1` is placed in the second row of `$m1` (same register). The third row of `$m1` is placed in the second row of `$m2`, and the fourth row of `$m1` is placed in the second row of `$m3`.
4. This process is repeated using `$m2` and `$m3` as the source registers.

5.2.2. The BLOCK4V Instruction

The `BLOCK4V` instruction is similar to the `BLOCK4` instruction, except that it operates on the columns of the matrices and the registers it affects are different. The `BLOCK4` instruction uses the mesh interconnect in the matrix unit to perform a four-way shuffle on the columns of four matrix registers. The matrix registers affected begin with the register number of the source operand and then use every fourth register. For example, `block4v $m0` will use `$m0`, `$m4`, `$m8`, and `$m12`. An example is show in Figure 4 below.

	Before	After
M0:	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	00 10 20 30 04 14 24 34 08 18 28 38 0C 1C 2C 3C
M4:	10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F	01 11 21 31 05 15 25 35 09 19 29 39 0D 1D 2D 3D

	Before	After
M8:	20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F	02 12 22 32 06 16 26 36 0A 1A 2A 3A 0E 1E 2E 3E
M12:	30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F	03 13 23 33 07 17 27 37 0B 1B 2B 3B 0F 1F 2F 3F

Figure 4: Data before and after `block4v.m $m0`

The last ingredient in the rotate is the matrix load-and-store instructions. The matrix load instruction will load 64 bytes from memory and place them in a matrix register. The load will place the first 4 bytes into element (`0,0`), the next into element (`0,1`) and so on, effectively 'folding' a 1-D dimensional vector in memory into a 4 × 4 matrix of 32-bit elements. The matrix `STORE` instruction does the reverse, taking each element and placing it into consecutive memory locations.

5.3. Rotation

A rotation of a pixel array (image) can be performed by first performing a transpose and then reversing the order of the columns of the array. This is shown in Figure 5.

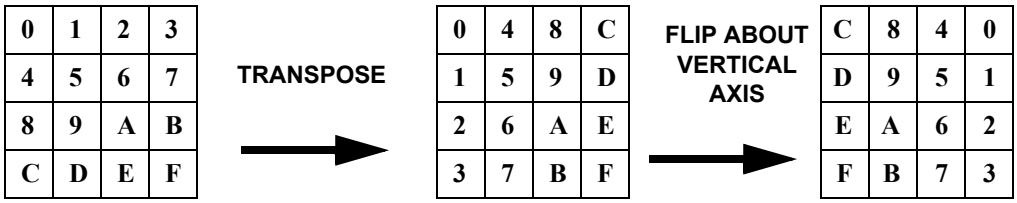

Figure 5: Rotation by transposition and flip

To do this on the FastMATH processor, the algorithm is as follows:

1. Load all 16 matrix registers with a 16 × 16 sub-block of the whole image. The load instructions calculate the effective address using a 'stride'. A stride is the length of a row of pixel (in bytes). The actual matrix register numbers into which the rows of the 16 × 16 sub-block are loaded is specific to the operation of the `BLOCK4` and `BLOCK4V` instructions and provides part of the transpose and interchange column movement.
2. A `BLOCK4` instruction is performed on each set of four registers. This shuffles the rows.
3. A `BLOCK4V` instruction is performed on each set of four registers. This shuffles the columns.
4. The 16 matrix registers are stored back to memory in an order that completes the column reversal.
5. The entire image is rotated by performing the 16 × 16 rotate on sub-blocks and storing the sub-blocks back at the 'rotated' position in memory.

5.4. Performance

This algorithm can be programmed into 72 instructions that take 142 cycles to read, rotate, and store the sub-block. For a large array, the loop count can be unrolled to a depth of 4, and the loop overhead to perform the sub-block decomposition would be negligible. This means that a larger array would take linear time to complete. For example, a 256 × 256 array of 32-bit pixels would take approximately 142 × (16 × 16) = 36,352 cycles (18.2 microseconds at 2-GHz clock frequency). For these larger arrays, the direct memory access (DMA) unit would be used to fill a pre-fetch buffer to hide the latency of system memory (SDRAM).

6. Huffman Decoding

Huffman compression is a mainstay of many compression standards. In the JPEG image compression standard, the image is transformed, in 8 × 8 blocks, using the discrete cosine transform (DCT). Then Huffman compression is used to efficiently and losslessly compress the coefficient output of the DCT. Huffman compression is a process of assigning variable-length codes to input data based on expected or measured probability of each datum. Typically, the codes are built into a binary tree with the data values in the leaves. Decompression is thus a process of reading the compressed stream bit by bit and traversing the tree until a leaf is found. Since this is an inherently serial process, it is not obviously amenable to parallelization. In this case, we use the optimized version of Huffman decode written in the C language by the Independent JPEG Group (IJG). It was compiled using an in-house Intrinsity version of the gcc compiler. The code does not use the matrix unit and thus is a measure of the efficiency of the scalar pipeline.

6.1. Performance

The IJG JPEG implementation was compiled for the FastMATH processor using the Intrinsity version of the gcc compiler. The subroutine `decode_mcu`, which is responsible for the Huffman decode of the DC and AC coefficients was profiled. A reference image was decoded and the total number of cycles was measured. There were 150 calls to the `decode_mcu` routine, taking a total of 929,871 cycles. Each minimum code unit (MCU) holds 8 × 8 × 6 symbols × 150 MCUs = 578,600 symbols for an average of 16.1 cycles per symbol.

7. Acronyms

1-D	one-dimensional
2-D	two-dimensional
DCT	discrete cosine transform
DDR-SDRAM	double-data-rate synchronous dynamic random-access memory
DMA	direct memory access
FFT	fast Fourier transform
IJG	Independent JPEG Group
JPEG	Joint Photographics Experts Group
MCU	minimum code unit
SIMD	single instruction, multiple data

8. References

[1] W. C. Chen, C. H. Smith and S. C. Fralick, “A Fast Computational Algorithm for the Discrete Cosine Transform”, IEEE Transactions on Communications, Vol COM 25 No 9, pp 1004-1009, Sept 1977.

9. Revision History

Revision	Date	Brief Description
1.0	2/5/03	Initial publication
1.1	2/17/03	Added acronyms section; replaced logo graphic
1.2	3/26/03	New format and logo with tag line