An Innovative Processor for Parallel Processing Applications: FFT Implementation Results

Tim Olson
Intrinsity, Inc.
11612 Bee Caves Rd., Bldg. II Ste. 200
Austin, Texas 78738
512-421-2100
olson@intrinsity.com

George Yost, PhD
Intrinsity, Inc.
11612 Bee Caves Rd., Bldg. II Ste. 200
Austin, Texas 78738
512-421-2100
george@intrinsity.com

Christophe Harlé, PhD
Intrinsity, Inc.
11612 Bee Caves Rd., Bldg. II Ste. 200
Austin, Texas 78738
512-421-2100
charle@intrinsity.com
(additional authors listed at end)

ABSTRACT

The FastMATH™ processor designed by Intrinsity, Inc. is capable of scalar, vector, and matrix mathematics operations. It uses patented technology centered around dynamic logic to achieve a 2 GHz clock rate. It features a fixed-point 32-bit word length and a MIPS32®-compliant instruction set with coprocessor 2 matrix unit intrinsic instructions. It is capable of 32 giga-operations per second (GOPS) sustained, with a peak rate of 64 GOPS. It is programmable in C and other higher-level languages. It features an unusually large 1-Mbyte on-chip cache and, to support high data rates commensurate with the high processor speed, two bi-directional one gigabyte per second RapidIO™ ports. We describe its unique architecture and its advantages for mathematical and parallel (SIMD) signal processing applications.

We demonstrate the application of this processor to fast Fourier transforms (FFTs). The processor will complete a 1024-point 16-bit complex fixed-point FFT in under two microseconds (μs), performance that has been verified on the actual hardware.

Categories and Subject Descriptors

Algorithms, Embedded H/W Design, Parallel Processing

General Terms

Algorithms, Performance, Design.

Keywords

FFT, Inverse FFT, Algorithms, Signal Processing, Parallel Processing, Matrix Processing.

1. INTRODUCTION TO THE FASTMATH PROCESSOR

The FastMATH processor (Figure 1) was designed as a high-speed processor optimized for matrix and vector operations such as those encountered in adaptive signal processing. The processor extends the concept of single-instruction multiple-data (SIMD) parallel operation from the conventional vector implementations to the next dimension: full matrix operation. Matrix operation is distinguished from vector operation by inter-element row and column communication mesh paths, supporting operations such as fast single-instruction matrix transpositions, matrix block rearrangements, and full matrix multiplications as well as the usual vector operations such as element-by-element additions and multiplications.

The FastMATH processor is configured (Figure 1) with a 4×4 array of matrix processing elements coupled to a synchronous scalar unit that works in parallel. The inter-element mesh communication paths are illustrated. Each matrix processing element (Figure 2) can process data contained in its corresponding element of 16 matrix registers. It includes its own ALU and dual 40-bit multiply-accumulate registers (MACs) that accept and accumulate the results of 16-bit multiplications. It also controls its corresponding portions of each of four condition codes.

The matrix registers each hold 16 32-bit words, 64 bytes in all. From the programmer's standpoint they function as 16 complete 4×4 matrices and the architecture allows FastMATH to operate on them in SIMD mode, as well as perform certain novel inter-register rearrangements that will be described.

The instruction sequence enters the scalar unit from a 16-Kbyte L1 instruction cache. This unit processes scalar instructions and issues instructions to the matrix unit for processing in parallel. It also performs matrix register load and store operations. There is a zero-cycle load-to-use penalty for matrices. The scalar unit also accepts data from a 16-Kbyte L1 data cache as shown in Figure 1.

The 1-Mbyte L2 cache serves as the point of data coherency with the scalar unit L1 caches, with SDRAM, and with input/output. A descriptor-based DMA controller is capable of memory-to-memory and constant-to-memory transfers, memory-to-constant compares, and memory test-and-set operations. It can pre-fetch data into the L2 cache via the RapidIO interface at 1 Gbyte per second or from the SDRAM via a DDR-400 connection without intervention of the CPU. The RapidIO ports are also convenient for pipeline processing with two or more chips.

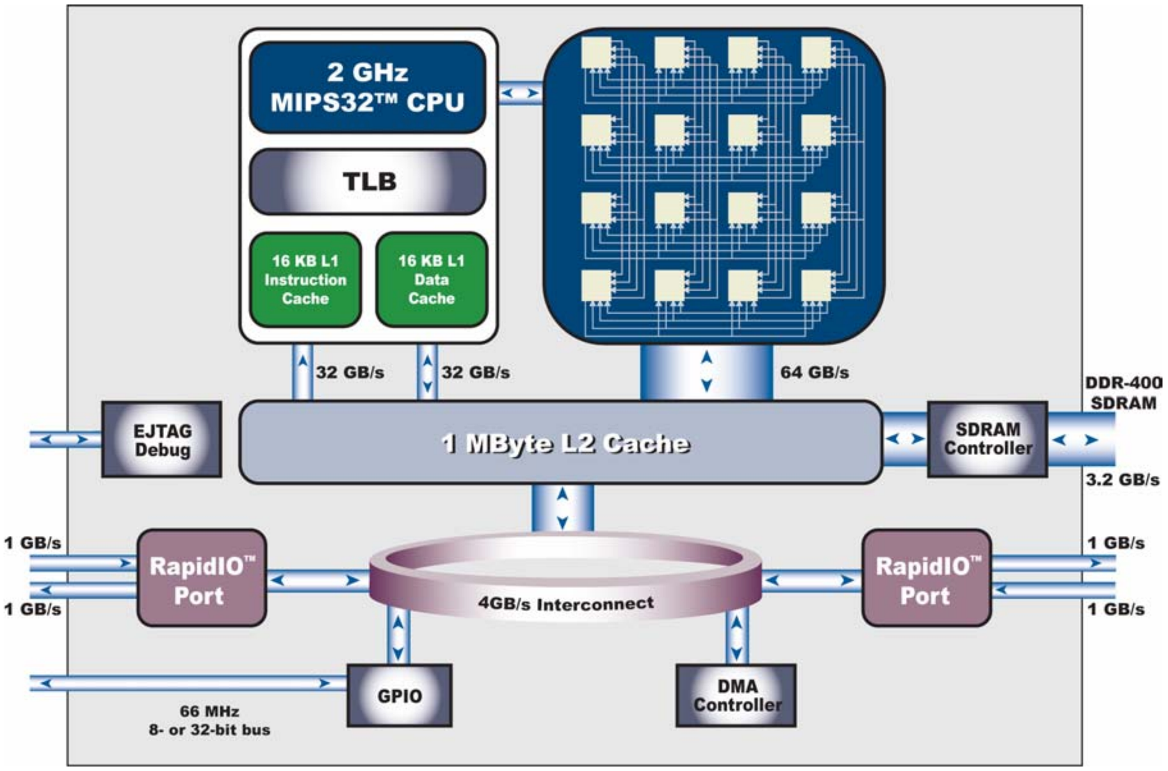

Figure 1. FastMATH logical structure

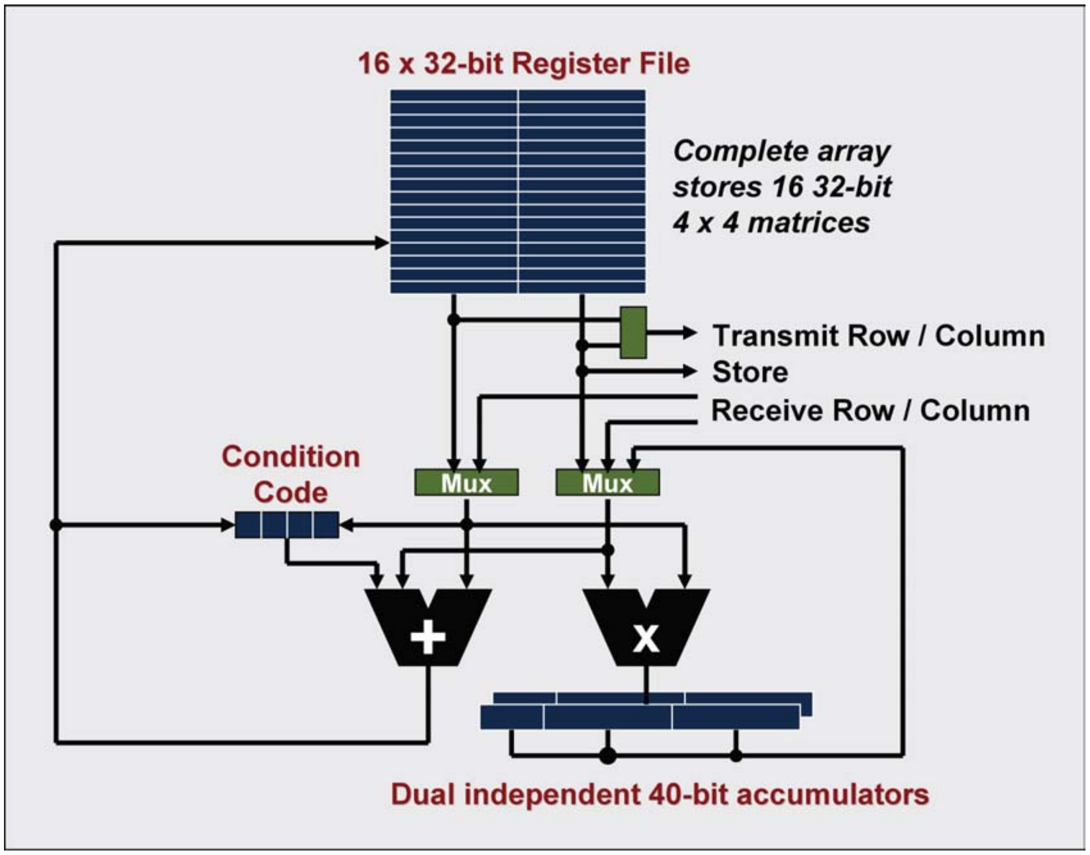
16 x 32-bit Register File
Complete array stores 16 32-bit 4 x 4 matrices
Transmit Row / Column
Store
Receive Row / Column
Condition Code
Mux
Mux
+
x
Dual independent 40-bit accumulators

Figure 2. Matrix processing element

The high-level language programming capability facilitates rapid development with standard toolchains. A C-language library of math and signal processing routines is available from Intrinsity. The matrix unit supports over 100 intrinsic instructions. These are organized into five classes as listed in Table 1.

Table 1. FastMATH matrix unit intrinsic instruction classes

Class	**Description**	**Examples**	**Matrix Unit Resources**
Memory access	Load-store matrix registers	Scalar unit load-store 64-byte matrix register	Matrix registers
ALU/logical/comparison	Element-wise additions, comparisons	16-bit, 32-bit additions and comparisons	Matrix registers, condition codes
Multiply-accumulate	Element-wise multiplications by halfword	Multiply high/low halfwords between matrices	Matrix registers, MACs, condition codes
Inter-element movement	Movement of elements within or between matrix registers	Matrix transpositions, row/column shifts, block rearrangements among matrix registers	Matrix registers, mesh interconnections
Inter-element computation	Matrix mathematics	Matrix-multiplications, sums over rows/columns	Matrix registers, MACs, condition codes, mesh interconnections

All matrix unit multiplications are by halfword and accumulate in the MACs. A sequence of multiply-add operations can be pipelined in a single cycle each. The MACs, with 40 bits of capacity, can accumulate the results of such repeated multiplications.

The halfword math capability is ideal for complex 16-bit operands. If, for example, the real part of the data is stored in the upper halfword and the imaginary part in the lower halfword, then it is straightforward for the two sets of MACs to accumulate, in parallel, the real and imaginary parts of the resultant of a sequence of operations.

The inter-register block rearrangement instructions (Figure 3) allow large matrices to be loaded, 16 32-bit elements per row at a time, and then rearranged across registers to permit SIMD operations on either 4 or 16 parallel streams. The figure illustrates four separate data streams in memory or cache, which are first loaded into four separate matrix registers; each register then contains 16 32-bit elements from one user. Next, a single instruction will reorganize the data among the four registers such that each register will contain four elements of four users. Finally, other instructions can configure the data in all 16 registers, as partially illustrated in the last row, to place a single element from each of 16 users into each register. A complete reassembly of 16 32-bit data streams into 16 registers in this way can be done in 70 cycles, or 35 nanoseconds. These block rearrangement instructions can also be used as part of the process to interleave separated real and imaginary 16-bit data streams into adjoining halfwords for complex math as described above.

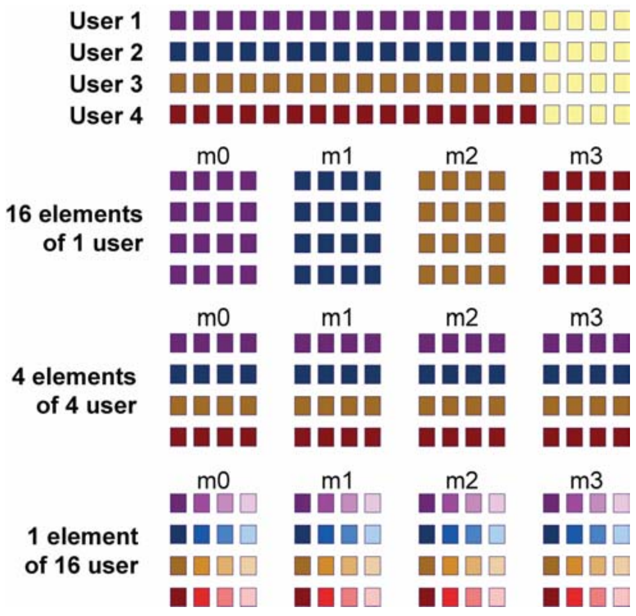

Figure 3. Matrix register block rearrangement functionality

To convert ordinary serial C language code into efficient parallelized C language code for the FastMATH processor one looks for situations in which 16 operations can be performed in parallel, with the same instruction operating on multiple data values. As an example, a single stream of data might require the same sequence of operations on each data element. If this is not the situation, then it may be possible, as illustrated in Figure 3, to operate in parallel on 16 different streams of data to again achieve very efficient operation. The inter-element movement instructions are very useful in formatting data into structures suitable for parallel computation.

2. IMPLEMENTATION OF FAST FOURIER TRANSFORMS

The Fourier transform is well known in the literature [1]; we will not describe it here. An FFT algorithm can perform a Fourier transform in considerably less time than a brute-force, element-by-element multiplication. In addition to direct Fourier transform problems, many other problems can be solved indirectly with their use, and the FFT is the method of choice for a wide variety of applications. Examples include vector correlation or convolution calculations (including finite-impulse-response filters), where one will take an FFT of the two N-vectors involved, multiply the resultant coefficients element-by-element, and take an inverse FFT of the final vector to obtain the results. N correlations/convolutions will result. In spite of the increased number of steps, the indirect FFT solution is often much faster than N direct vector multiplications. The FFT is a core function in many modern signal-processing applications.

The FFT algorithm is amenable to parallelization. The entire set of data points, which may be large, need to be multiplied by an equal number of factors, commonly known as "twiddle factors". In the FastMATH processor these can be loaded and processed 64 bytes at a time. Thus, for data consisting of 16-bit real and imaginary parts, 16 data elements are processed in parallel.

Radix-2 and radix-4 FFT algorithms (requiring that the number of points N is an integer power of 2 or 4, respectively) have been coded in the C language as part of the FastMATH library. These algorithms produce results in which the output frequency coefficients are out of order; hence a full implementation will require a re-ordering phase ("decimation in time" if done at the beginning or "decimation in frequency" if done at the end) known as address bit-reversal. Library routine performance for complex forward FFTs, coded for decimation in frequency but without the actual address bit-reversal step, is shown in Figure 4. These results assume that both the data and the code are resident in cache. As shown in Figure 4, a 1024-point FFT with complex 16-bit input data and twiddle factors can be done in under 2 μs; a 16 K-point FFT can be done in a little under 50 μs. These results are from the Intrinsity cycle-accurate simulator; the 1024-point FFT timing has been confirmed on the actual hardware.

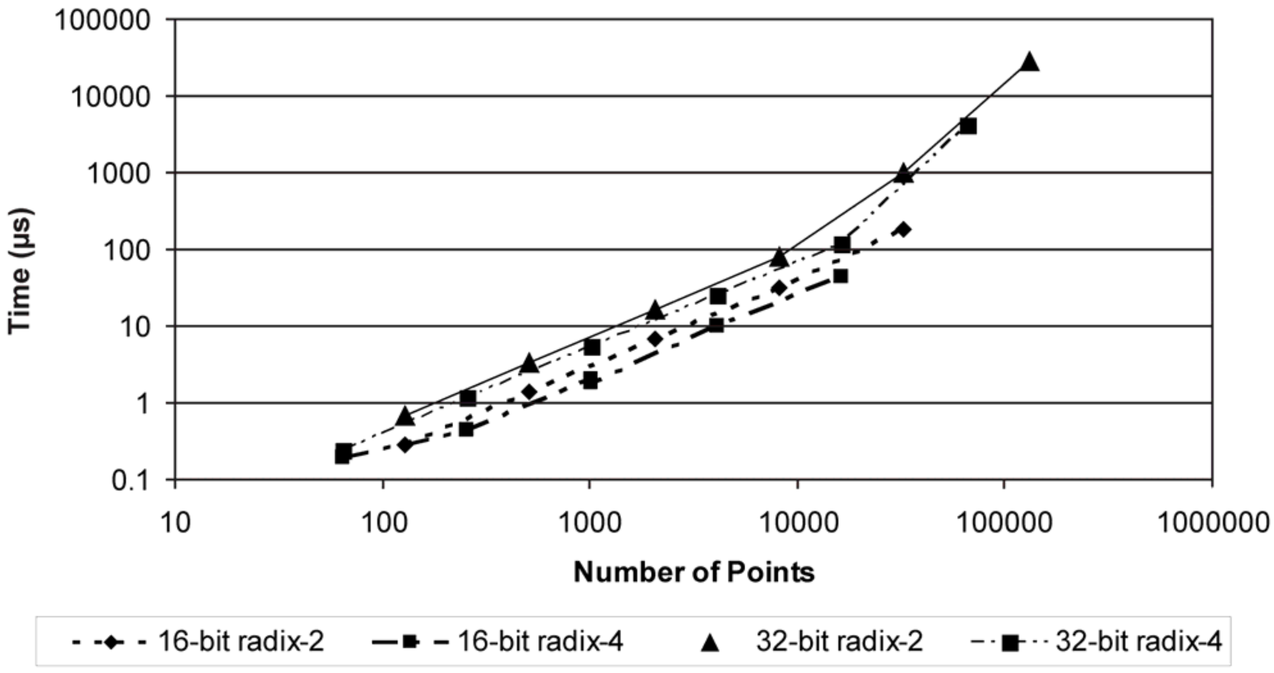

Figure 4. Forward FFT performance on 16-bit complex data

The bit-reversal stage is not included here because in many applications it is not needed. For example, the correlation/convolution function mentioned before, as well as many other applications of FFTs, require a forward FFT followed later by an inverse FFT. If the forward and inverse stages are properly coordinated the two bit-reversals cancel each other and can therefore be eliminated. When address bit-reversal is needed it will typically add 10% to 20% more cycles.

It should be noted that performance is always dependent upon the exact toolchain being used and the degree to which hand optimization is applied. If additional speed is required, faster performance can sometimes be achieved by certain optimization steps; for most problems this effort is not required.

To illustrate the C implementation of an FFT and introduce the practice of FastMATH coding, we include the code for one of the butterfly loops (a key component of the algorithm; so named because of the diagram that describes it [1]) of a radix-2 FFT in Figure 5 (see next page). This loop illustrates the calculation of one of the outer (block) loops; inner loops may be parallelized differently for greatest efficiency. Note that there is explicit code for overflow protection.

This code loads two matrix registers with 16 words each of complex input data (that may be the output of the previous butterfly stage) and a single register with the appropriate twiddle factors. These data are loaded from memory but may be cache-resident. If the input data matrices are denoted A and B and the matrix of twiddle factors is labeled W, then the radix-2 butterfly step requires two complex results, $(A + B)$ and $(A - B) \times W$. This code demonstrates that calculation.

3. CONCLUSIONS

Intrinsity, Inc. has designed an innovative processor that consists of a 2 GHz MIPS32-based scalar engine and a matrix engine that accepts MIPS coprocessor 2 intrinsic instructions. Matrix processing proceeds in parallel with the scalar processing. The matrix engine facilitates full matrix mathematics, as well as SIMD operations on large arrays of data. The machine is programmable in standard C language with extensions, enabling use of standard toolchains. This processor has been applied to the problem of fast Fourier transforms, a key algorithm in many signal processing applications. This algorithm is well-suited to a SIMD machine. With the fast cycle speed coupled with the unique architecture a 1024-point complex FFT with 16 bits of precision in the real and imaginary parts can be done in under 2 μs.

4. REFERENCES

[1] Vijay K. Madisetti and Douglas B. Williams, eds., *The Digital Signal Processing Handbook* (CRC Press LLC, Boca Raton, FL,1998).

5. ADDITIONAL AUTHORS

Veeraragahavan Anantha, PhD
Intrinsity, Inc.
11612 Bee Caves Rd., Bldg. II, Ste. 200
Austin, Texas 78738
512-421-2100
anantha@intrinsity.com

Greg Crabtree
Intrinsity, Inc.
11612 Bee Caves Rd., Bldg. II, Ste. 200
Austin, Texas 78738
512-421-2100
gcrabtree@intrinsity.com

```
//Generic 16-bit complex radix-2 DIF butterfly.
//Fixed-point arithmetic, assuming data is stored as (S.15).
//Butterfly calculation computes two complex results from input matrices
//A and B: (A + B) and (A - B)*twiddle factors.
//High halfwords contain real parts, low halfwords imaginary parts.
//Matrix registers mInA, mInB are input data, loaded from pointer p.
//Matrix register mW contains (S.15) twiddle factors, loaded from q.
//mDiff, mSum, mRe, mIm are temporary matrix registers.
//mAcRe, mAcIm are MACs to accumulate real, imaginary parts of results.

//Initialize input data.
//Value of N depends on stage of butterfly in calculation.
mInA = p[0];
mInB = p[N];
//Scale input to prevent overflow: right-shift real and imaginary parts
//2 bits, sign-extended.
mInA = matrixShiftRightArithmeticHalfwordImmed(mInA, 2);
mInB = matrixShiftRightArithmeticHalfwordImmed(mInB, 2);
for (i=N; i>0; --i)  {
    //Load twiddle factors.
    mW = wp[0];
    //Begin calculation of (mInA - mInB)*mW.
    //Compute (mInA - mInB).
    mDiff = matrixSubFromHalfword(mInA, mInB);
    //Calculate complex sum (A + B).
    mSum = matrixAddHalfword(mInA, mInB);
    //Shift each result halfword 1 bit, sign-extended, to scale both real and
    //imaginary parts to prevent future overflows.
    mOutA = matrixShiftRightArithmeticHalfwordImmed(mSum, 1);
    //Multiply real part of (mInA - mInB) by real part of mW.
    mAcRe = matrixMulHighHigh(mW, mDiff);
    //Multiply and subtract imaginary parts. to complete real part of (mInA - mInB)*mW.
    mAcRe = matrixMulSubLowLow(mAcRe, mW, mDiff);
    //Compute real*imaginary.
    mAcIm = matrixMulHighLow(mW, mDiff);
    //Compute im*re and accumulate, to complete imaginary part of
    //(mInA - mInB)*mW.
    mAcIm = matrixMulAddHighLow(mAcIm, mDiff, mW);
    //Product (mInA - mInB)*mW will have 30 significant bits plus sign.
    //Move 32-bit real and imaginary results from MACs.
    mRe = matrixMoveFromLow(mAcRe);
    mIm = matrixMoveFromLow(mAcIm);
    //Pack most significant 16-bit real and imaginary parts into output
    //matrix register.
    mOutB = matrixPackHighHalfword(mIm, mRe);
    //Store results.
    q[0] = mOutA;
    ++p;
    mInA = p[0];
    mInB = p[N];
    mInA = matrixShiftRightArithmeticHalfwordImmed(mInA, 2);
    mInB = matrixShiftRightArithmeticHalfwordImmed(mInB, 2);
    q[N] = mOutB;
    ++q;
    ++wp;
}
```

Figure 5. Coding for radix-2 FFT butterfly loop