Hybrid Floating Point Technique Yields 1.2 Gigasample Per Second 32 to 2048 point Floating Point FFT in a single FPGA.

Raymond J. Andraka, P.E., President, Andraka Consulting Group, Inc. mailto:ray@andraka.com

Abstract

Hardware Digital Signal Processing, especially hardware targeted to FPGAs, has traditionally been done using fixed point arithmetic, mainly due to the high cost associated with implementing floating point arithmetic. That cost comes in the form of increased circuit complexity. The increase circuit complexity usually also degrades maximum clock performance. Certain applications demand the dynamic range offered by floating point hardware, and yet require the speeds and circuit density usually associated with fixed point hardware. The Fourier transform is one DSP building block that frequently requires floating point dynamic range.

Textbook construction of a pipelined floating point FFT engine capable of continuous input entails dozens of floating point adders and multipliers. The complexity of those circuits quickly exceeds the resources available on a single FPGA.

This paper describes a technique that is a hybrid of fixed point and floating point operations designed to significantly reduce the overhead for floating point. The results are illustrated with an FFT processor that performs 32, 64, 128, 256, 512, 1024 and 2048 point Fourier transforms with IEEE single precision floating point inputs and outputs. The design achieves sufficient density to realize a continuous complex data rate of 1.2 Gigasamples per second data throughput using a single Virtex4-SX55-10 device.

Introduction

The challenge of fitting the high data rate Fourier transform processor on a single FPGA is approached along two avenues: First, the algorithm itself is examined and optimized specifically to minimize the hardware footprint. Second, the hybrid techniques alluded to earlier are applied to reduce the reliance on the hardware extensions needed for floating point implementation.

Algorithm

The Fast Fourier Transform can be factored in a variety of different ways, each of which results in a different algorithm. The most common factorization is the Cooley-Tukey algorithm[1], which factors each N point transform into a pair of N/2 point transforms combined with a "butterfly" operation. The factorization is recursively applied on the reduced transforms, eventually ending up with an algorithm that performs a series of butterfly computations to arrive at the Fourier Transform. Each butterfly operation consists of a two point transform and a phase rotation (a complex multiply by a phasor, often called "twiddle factor". In a software implementation, the regularity of the algorithm and data sequencing leads to a loop structure with very little overhead. The number of multiplies and floating point operations is of little consequence because all the operations are performed serially by a single floating point unit.

There are other factorizations of the Fourier Transform that can result in a more efficient hardware design. The Winograd FFT algorithm[2] is particularly interesting for hardware implementations because it is a factorization to minimize the number of multiplies needed to perform the transform. The Winograd algorithm performs the transform as a complex matrix multiply factored into three consecutively applied matrix multiplies. All the elements in the first and third matrix are from the set {0,-1,1, j, –j}, which means those matrix multiplies are performed using a sequence of add operations. The center matrix is a diagonal matrix containing only real constants. The price paid for the reduced number of operations is a rather irregular addressing sequence, which makes it inefficient to perform with a microprocessor. In a hardware implementation however, the irregular addressing is easier to deal with because of the parallelism offered by a hardware solution. A 16 point Winograt FFT decomposes into three layers of add/subtract operations on each side of a real multiplier[3] (Figure 1), which translates into 74 real adds and only 18 real multiplies[4]. This represents about one third of the hardware of a Cooley-Tukey implementation.

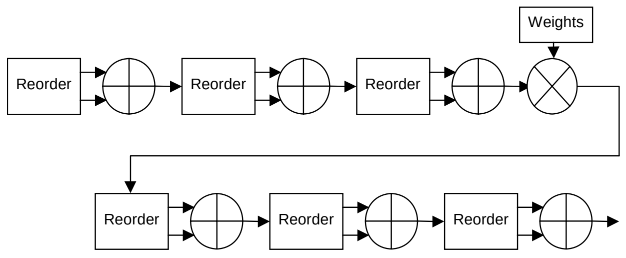

Figure 1. 16 point winograd FFT

The Winograd FFT is awkward to factor for sizes larger than 16 point. The Mixed Radix algorithm[5] for cascading smaller FFTs is used to achieve the larger FFT sizes from Winograd kernels that perform 4, 8 and 16 point FFTs. The mixed radix algorithm arranges the data into a matrix, performing a small FFT down each column, phase rotating the results and then an additional small FFT across each row. The hardware required to do this is a phase rotator (complex multiply and twiddle factor table) and a reorder memory.

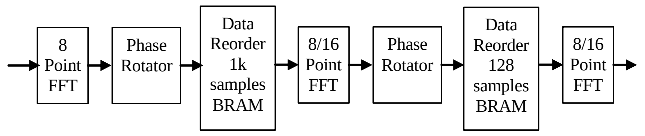

Figure 2. Combination of small Winograd kernels using mixed radix algorithm to achieve larger FFTs.

Hybrid Floating Point Implementation

Traditional floating point implementations treat each add or multiply operation as a stand-alone floating point operation requiring normalized inputs and outputs. When these basic operations are assembled into more complicated operators, the intermediate normalize/denormalize operations are often unnecessary and represent a considerable amount of wasted hardware. With the hybrid approach, we take larger pieces of the algorithm and treat them as fixed point blocks that operate on the mantissas of the input data. The data is typically de-normalized so that all the data within a block shares a common exponent. The exponent is passed around the fixed point operation. The output of the fixed point operation is re-normalized and the common exponent is added to the individual exponents arising out of the re-normalization. The key is to select the size of the fixed point composite operation so that the result of the operation has a narrow enough dynamic range so as to not require a large number of additional bits to avoid overflow.

The FFT design uses Winograd FFT blocks that compute 4, 8 or 16 point FFTs. Considering the properties of the FFT, these blocks have a maximum gain of 16, which is accommodated by allowing for a 4 bit growth. These FFT blocks then make an ideal candidate for a fixed point function block in this hybrid floating point approach. The design denormalizes each of the complex samples into the small FFT so that the largest sample is left justified in the 31 bits presented to the FFT. The remaining samples are right shifted by the difference between their exponent and the exponent of that largest sample. A new maximum is obtained for each 4, 8 or 16 point set so that the FFT is always performed at the maximum permissible precision. The FFT design internally expands to 35 bits, and presents 35 bits at the output so that overflow is not possible regardless of the inputs. Each complex pair out of the FFT kernel is then re-normalized producing a complex pair with a common exponent for the I and Q components. The larger component is left justified. . Note that since the FFT algorithm necessarily involves addition, no precision is lost using this method as compared with performing the entire FFT with floating point arithmetic! This is because the smaller intermediate results wind up getting rounded in a full floating point implementation anyway.

The phase rotations are also accomplished with fixed point hardware (a 35x35 bit complex multiply), but since these are multiplications they are very similar to floating point multiplies. The input from the FFT kernel is an IQ pair with a common exponent with the larger component left justified. The twiddle factor is a 35 bit fixed point sine and cosine pair, whose combined magnitude is unity. Since the selected number representation has a common exponent for the IQ pair, the result of the operation has the same common exponent. The worst case rotations are ones that start with the vector on an axis and rotate it 45 degrees or end with it on an axis after a 45 degree rotation. In those cases, there is a one bit growth or shrink of the larger component, requiring a simple +/- one bit shift to re-normalize (and the attendant increment or decrement to the exponent) after the rotation. The inputs and outputs from this floating point 4/8/16 point kernel (Figure 3) are floating point IQ pairs with a common 8 bit exponent. The mantissas are 31 bits sign-magnitude. This internal pair format is used rather than separate exponents because it reduces the storage for the intermediate results, makes the normalization easier, and loses nothing compared to independent I and Q. The internal representation is converted from and to IEEE single precision format at the input and output of the FPGA.

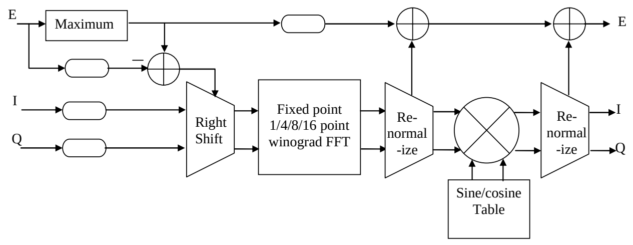

Figure 3. Hybrid floating point 4/8/16 point FFT kernel.

FPGA implementation

The design was implemented in a Xilinx Virtex4 XCV4SX55-10 device. All of the arithmetic is confined to the DSP48 slices so that the design is not slowed down by the relatively slow carry chains in the FPGA fabric. Careful implementation has yielded a design whose maximum clock rate is the DSP48's maximum clock rate of 400 MHz. The 32-2048 point FFT described here operates at up to 400 megasamples per second using a 400 MHz clock, and occupies less than 30% of the FPGA. Three instances, along with a round robin controller, QDR memory interfaces (for the data source and sink), and buffering fit within the device. By sequencing the FFT starts, the three threads achieve a composite 1.2 Gigasample per second throughput. Using a hybrid floating point approach made a single FPGA implementation possible where it would not have been with a conventional approach.

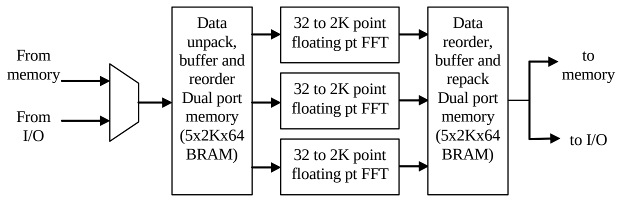
From memory
From I/O
Data unpack, buffer and reorder Dual port memory (5x2Kx64 BRAM)
32 to 2K point floating pt FFT
32 to 2K point floating pt FFT
32 to 2K point floating pt FFT
Data reorder, buffer and repack Dual port memory (5x2Kx64 BRAM)
to memory
to I/O

Figure 4. Three units in one FPGA work in parallel to achieve the 1.2 GS/sec data rate

References

[1] Cooley, J. W. and Tukey, J. W., "*An Algorithm for the Machine Computation of Complex Fourier Series, Mathematics of Computation*", Vol. 19, pp. 297-301, April 1965.

[2] Winograd, S., "*On Computing the Discrete Fourier Transform*," Mathematics of Computation, Vol. 32, No. 141, pp. 175-199 (1978).

[3] Smith, W. W. and Smith, J. M., "*Handbook of Real-Time Fast Fourier Transforms*", IEEE press, New York, 1995., pp 128-136

[4] Blahut, R. "*Fast algorithms for digital signal processing*", Addison-Wesley, 1985, pp 424-425.

[5] Smith, W. W. and Smith, J. M., "*Handbook of Real-Time Fast Fourier Transforms*", IEEE press, New York, 1995., pp 207-241