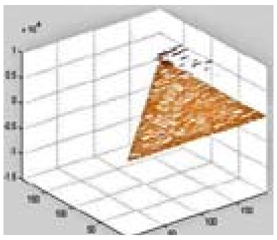


Hardware-based floating-point design flow

Michael Parker, Altera Corporation

1/17/2011 3:22 PM EST

This paper describes a new approach to efficiently implementing a floating-point algorithm in a hardware FPGA architecture that achieves extremely high rates of floating-point processing (at least 1 TeraFLOPS) in a single FPGA die, and with significantly better power efficiency than the microprocessor-based alternatives.

Floating-point processing is widely used in computing for many different applications. In most software languages, floating-point variables are denoted as "float" or double." Integer variables are also used for what is known as fixed-point processing.

Floating-point processing utilizes a format defined in IEEE 754, and is supported by microprocessor architectures. However, the IEEE 754 format is inefficient to implement in hardware, and floating-point processing is not supported in VHDL or Verilog. Newer versions, such as SystemVerilog, allow floating-point variables, but industry-standard synthesis tools do not support floating-point technology.

In embedded computing, fixed-point or integer-based representation is often used due to the simpler circuitry and lower power needed to implement fixed-point processing compared to floating-point processing. Many embedded computing or processing operations must be implemented in hardware—either in an ASIC or an FPGA.

However, due to technology limitations, hardware-based processing is virtually always done as fixed-point processing. While many applications could benefit from floating-point processing, this technology limitation forces a fixed-point implementation. If feasible, applications in wireless communications, radar, medical imaging, and motor control all could benefit from the high dynamic range afforded by floating-point processing.

Before discussing a new approach that enables floating-point implementation in hardware with performance similar to that of fixed-point processing, it is first necessary to discuss the reason why floating-point processing has not been very practical up to this point. This paper focuses on FPGAs as the hardware-processing devices, although most of the methods discussed can be applied to any hardware architecture.

After a discussion of the challenges of implementing floating-point processing, a new approach used to overcome these issues will be presented. Next, some of the key applications for using floating-point processing, involving linear algebra, are discussed, as well as the additional features needed to support these type of designs in hardware. Performance benchmarks of FPGA floating-point processing examples are also provided.

Floating-Point Issues in FPGAs

Floating-point numerical format and operations are defined by the IEEE 754 standard, but the standard's numerical representation of floating-point numbers is not hardware friendly. To begin with, the mantissa representation includes an implicit 1. Each mantissa digital representation of range [0 : 0.999..], actually maps to a value in the range of [1 : 1.999..]. Another issue is that the sign bit is treated separately, rather than using traditional twos complement signed representation.

In addition, to preserve the dynamic range of a floating-point signal, the mantissa must be normalized after every arithmetic operation. This aligns the decimal point to the far left, and adjusts the exponent accordingly. This is normally done using a barrel shifter, which shifts any number of bits in one clock cycle. Additionally, for each arithmetic operation, specific floating-point "special cases" must be checked for and flagged as they occur.

In floating-point processors, the CPU core has special circuits to perform these operations. Typical CPUs operate serially, so one or a small number of computational units are used to implement the sequence of software operations. Since CPU cores have a small number of floating-point computational units, the silicon area and complexity needed to implement the IEEE 754 standard is not burdensome, compared to the rest of buses, circuits, and memory needed to support the computational units.

Implementation in hardware, and in FPGAs in particular, is more challenging. In logic design, the standard format for signed numbers is the twos complement. FPGAs efficiently implement adders and multipliers in this representation. So the first step is to use the signed twos complement format to represent the floating-point mantissa, including the sign bit. The implicit 1 in the IEEE 754 format is not used.

With the IEEE 754 standard, normalization and de-normalization using barrel shifters is implemented at each floating-point operation. For adder or subtracter circuits, the smaller number must first be de-normalized to match the exponent of the larger. After adding and/or subtracting the two mantissas, the result must be normalized again, and the exponent adjusted. Multiplication does not require the de-normalization step, but does require normalization of the product.

Page 2

Optimal implementation of FP processing

With FPGAs, using a barrel-shifter structure for normalization requires high fan-in multiplexers for each bit location, and the routing to connect each of the possible bit inputs. This leads to very poor fitting, slow clock rates, and excessive routing. A better solution with FPGAs is to use multipliers. For a 24-bit single-precision mantissa (the signed bit is now included), the 24x24 multiplier shifts the input by multiplying by 2_N. Many FPGAs today have very high numbers of hardened multiplier circuits that operate at high clock rates.

Another technique used to minimize the amount of normalization and de-normalization is to increase the size of the mantissa. This allows the decimal place to move a few positions before normalization is required, such as in a multiplier product. This is easily accomplished in an FPGA, as shown in **Figure 1 below**.

For most linear algebra functions, such as vector sums, vector dot-products, and cross products, a 27-bit mantissa reduces normalization frequency by over 50%. For more non-linear functions, such as trigonometric, division, square root, a larger mantissa is needed. A 36-bit mantissa works well in these cases. The FPGA must support 27x27 and 36x36 multipliers. For example, one recently announced FPGA offers over 2000 multipliers configured as 27x27, or over 1000 multipliers configured as 36x36.

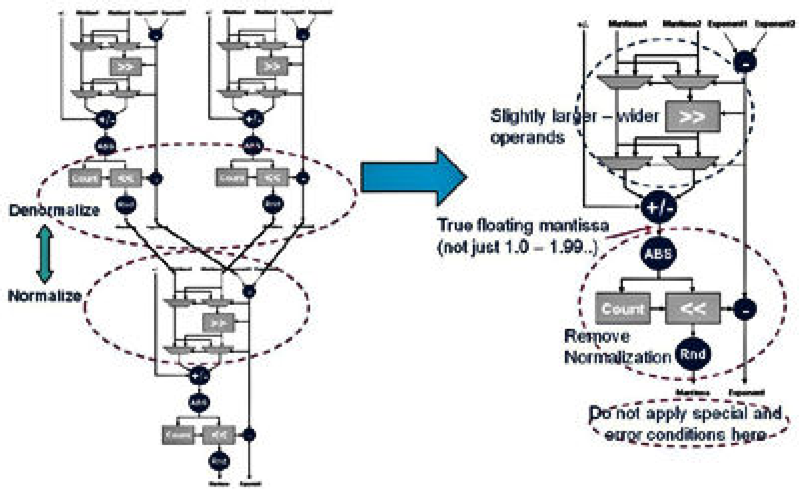
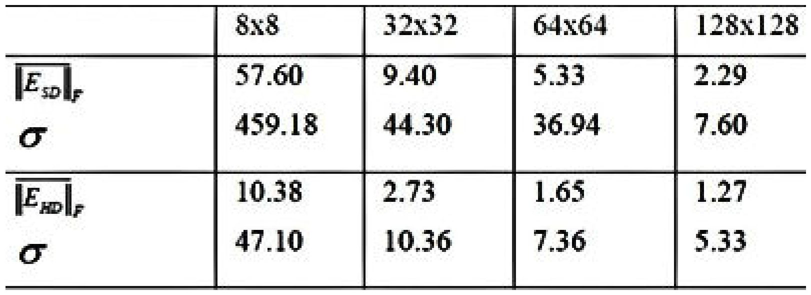

Denormalize
Normalize
Slightly larger – wider operands
True floating mantissa (not just 1.0 – 1.99..)
Count
<<
>>
+/-
ABS
Rnd
Remove Normalization
Do not apply special and error conditions here

Figure 1. New Floating-Point Approach (*To view larger image, **[click here](#)**)*

These techniques are used to build a high-performance floating-point datapath within the FPGA. Because IEEE 754 representation is still necessary to comply with floating-point processing, the floating-point circuits must support this interface at the boundaries of each datapath, such as a fast Fourier transform (FFT), a matrix inversion, or sine function.

This floating-point approach has been found to yield more accurate results than if IEEE 754 compliance is performed at each operator. The additional mantissa bits provide better numerical accuracy, while the elimination of barrel shifters permits high clock-rate performance, as shown in **Table 1 below.**



	8x8	32x32	64x64	128x128
$\|E_{SD}\|_F$	57.60	9.40	5.33	2.29
σ	459.18	44.30	36.94	7.60
$\|E_{HD}\|_F$	10.38	2.73	1.65	1.27
σ	47.10	10.36	7.36	5.33

Table 1. FPGA Floating-Point Precision Results

Table 1 lists the mean, the standard deviation, and the Frobenious norm where the SD subscript refers to IEEE 754-based single-precision architecture in comparison with the reference double-precision architecture, and the HD subscript refers to the hardware-based single-precision architecture in comparison with the reference double-precision architecture.

Floating-Point Verification

Floating-point results cannot be verified by comparing bit for bit, as is typical in fixed-point arithmetic. The reason is that floating-point operations are not associative, which can be proved easily by writing a program in C or MATLAB to sum up a selection of floating-point numbers.

Summing the same set of numbers in the opposite order will result in a few different LSBs. To verify the floating-point designs, the designer must replace the bit-by-bit matching of results typically used in fixed-point data processing with a tolerance-based method that compares the hardware results to the simulation model results.

The results of an *R* matrix calculation in a QR decomposition are shown **Figure 2 below**, using a three-dimensional plot to show the difference between the MATLAB-computed results and the hardware-computed results using an FPGA-based floating-point toolflow. Notice the errors are in the 10_{-6} range, which affects only smallest LSBs in the single-precision mantissa.

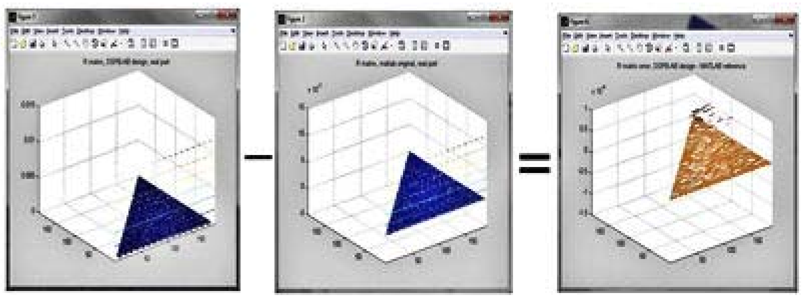


Figure 2. R Matrix Error Plot *(To view larger image, **click here**)*

To verify the accuracy of the non-IEEE 754 approach, matrix inversion was performed using single-precision floating-point processing. The matrix-inversion function was implemented using the FPGA and tested across different-size input matrices. These results were also computed using single-precision with an IEEE 754-based Pentium processor. Then a reference result was computed on the processor, using IEEE 754 double-precision floating-point processing, which provides near-perfect results relative to single-precision.

Comparing both the IEEE 754 single-precision results and the single-precision hardware results, and computing norm and the differences, shows that the hardware implementation gives a more accurate result than the IEEE 754 approach, due to the extra mantissa precision used in the intermediate calculations.

FPGA FP design-flow methodology

Hardware, including FPGAs, is typically designed using an HDL, either Verilog or VHDL. These languages are fairly low level, requiring the designer to specify all data widths at each stage and specify each register level, and does not support the synthesis of floating-point arithmetic operations.

To implement the approach above using HDL would be very arduous and has greatly discouraged the use of floating-point processing in hardware-based designs. Therefore, a high-level toolflow is needed to implement these floating-point techniques.

The design environment chosen in this case is Simulink, a widely used product from The Mathworks. Simulink is model based, which allows the designer to easily describe the data flow and parallelism in the design, traditionally a challenge when using software language.

Compared to HDL, Simulink provides a high level of design description, allowing the designer to describe the algorithm flow behaviorally, without needing to insert pipeline registers or know the details of the FPGA hardware, and to easily switch between fixed-point processing and single- and double-precision floating-point variables. This level of abstraction provides the opportunity for an automated tool to optimize the RTL generation, including the floating-point synthesis.

An additional advantage of this choice is that the system can be simulated in the Simulink and MATLAB domains, and the same testbench used in system-level simulation is later used to verify the FPGA-based implementation. The automated back-end synthesis tool running under Simulink is called DSP Builder. It performs all the required floating-point optimizations to produce an efficient RTL representation. A simple circuit representation is shown in **Figure 3 below.**

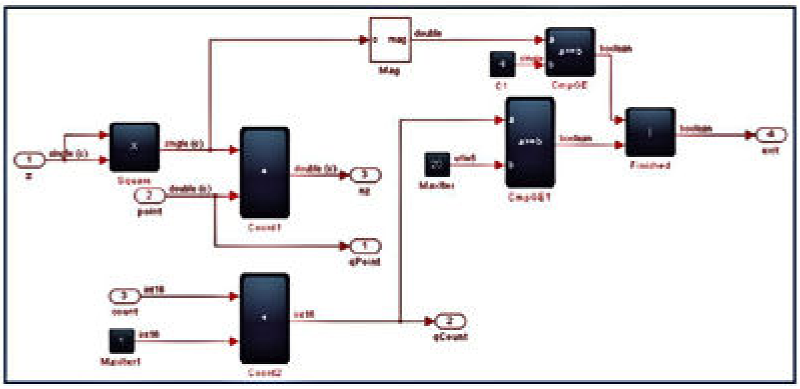


Figure 3. Simple DSP Builder Floating-Point Processing Example (**To view larger image**, [click here](#)**)

FPGAs, with their hardware architecture, distributed multipliers, and memory blocks, are ideal for high-bandwidth parallel processing. The distributed nature of the programmable logic, hardened blocks, and I/Os minimize the occurrence of bottlenecks in the processing flow. The embedded memory blocks store the intermediate data results, and the extensive I/O options of FPGAs provide easy interconnects to the larger system, whether they are processors, data convertors, or other FPGAs.

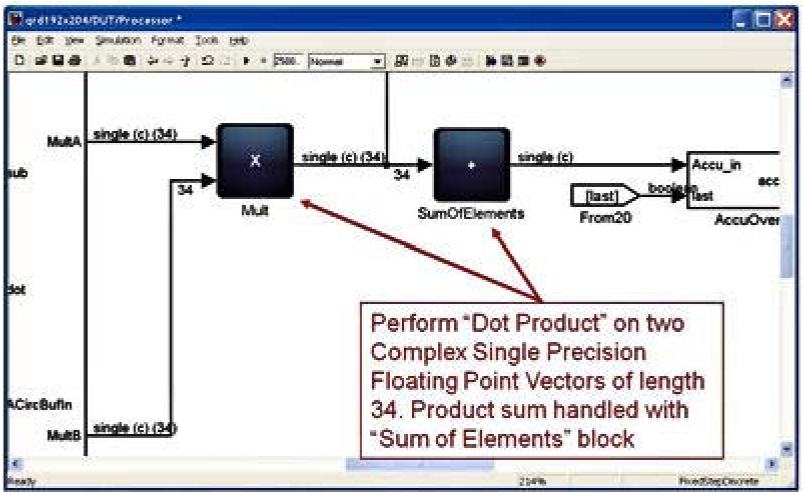
Page 3

Vector Processing

Many designs requiring the dynamic range of floating-point processing are based on linear algebra. Using linear algebra to solve problems is typical for multi-input and multidimensional systems, such as radar, medical imaging and wireless systems. For this reason, it is important to support vector processing of complex (quadrature) data.

Vector processing is ideally suited for parallel processing. Processing serially dramatically reduces throughput and increase latency. Due to the inherent parallelism, hardware implementation is well suited to vector processing. However, vector processing must be representable and synthesizable in the design entry process.

Figure 4 below shows a simple dot products example where vectors are denoted by single lines, and the vector length or number of elements is displayed, all of which are implemented in a complex, single-precision floating-point numerical representation. The vector length is a parameter set in a top-level constant file, and the example uses a special block, SumOfElements, that acts as an accumulator to add together all the partial products.



Perform "Dot Product" on two Complex Single Precision Floating Point Vectors of length 34. Product sum handled with "Sum of Elements" block

Figure 4. DSP Builder Floating-Point Dot Product Example

Managing Data Flow in Hardware

Simple flow diagrams can implement some algorithms, but other algorithms require more complex control. For example, matrix multiplication, which is a series of vector dot products, requires indexing of the rows and columns. Software languages have structures to implement looping, and the same is needed in hardware flow. For this reason, a for loop block has been added to the Simulink library, as shown in **Figure 5 below**.

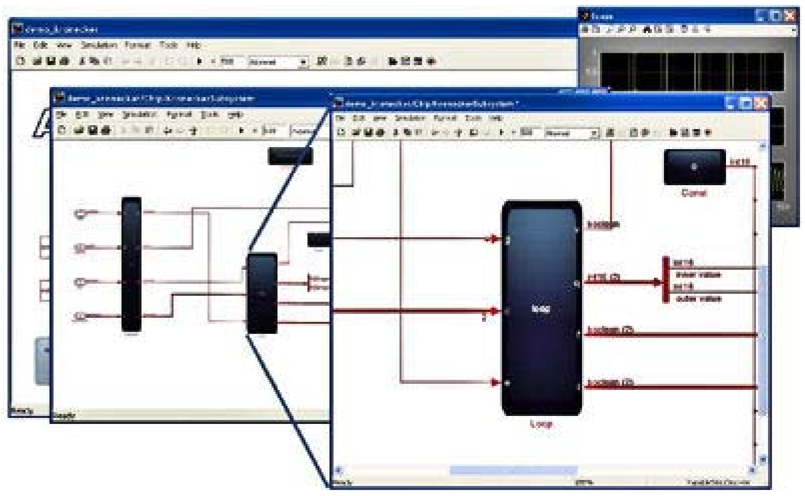


Figure 5. DSP Builder For-Loop Block (*To view larger image*, ***click here)***

In addition, multiple for loops nest, just as in a software environment, to build indexing counters and control signals, as shown in **Figure 6 below**. This capability is critical in many applications, including the indexing often needed for linear algebra processing.

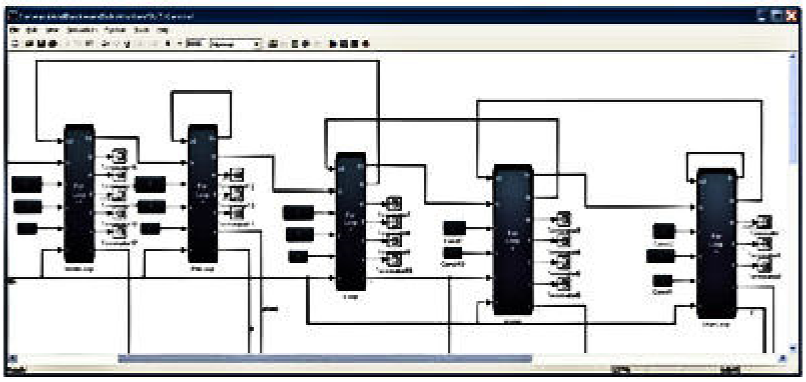


Figure 6. DSP Builder Nested For-Loop Blocks (*To view larger image*, ***click here)***

Solving systems of multiple equations or inverting matrices often requires back substitution, where each unknown is solved iteratively, one equation at time. The code for this solution is as follows:

```
for (uint8 countA=0; countA<16; countA++)

 {

for (uint8 countB=0; countB<=countA; countB++) {

qc1 = countA;

qc2 = countB;

        }
```

}

Figure 7 below shows how it is necessary to index across both vertical and horizontal elements. Using nested for-loop blocks allows complex hardware control functions to manage data flow.

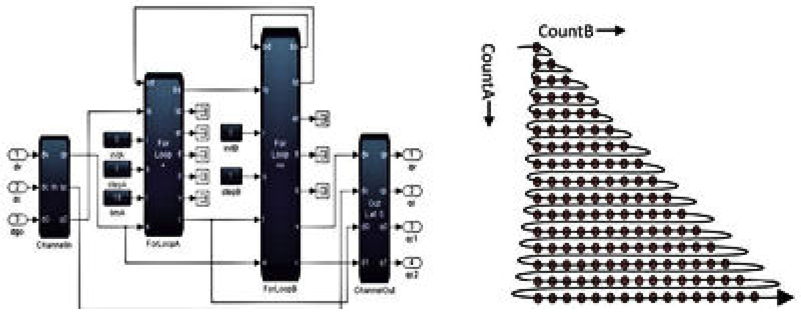


Figure 7. DSP Builder Back Substitution Diagram*(To view larger image, [**click here**])*

Many algorithms requiring the dynamic range of floating-point processing are iterative. One such example is the recursive infinite impulse response (IIR) filter function. One of the challenges is to design the data flow in such a manner as to avoid stalls in the hardware blocks, and to parallelize the critical path as much as possible for maximum throughput. Additionally, to provide for fast clock rates, adequate delay registers must be placed in the feedback path.

In the example shown in **Figure 8 below**, the IIR bi-quad filter is constructed just as in a textbook diagram. However, the toolflow has the capability to create multichannel designs. This example initially has four filter channels, though the design depicts the operations of only a single channel. Next, the design is increased to 20 channels.

The channel-in and channel-out blocks denote the boundaries of this function, and the user can parameterize the function to have as many channels as necessary. The tool creates the scheduling logic to manage the *N* channels. In this example, using 20 channels requires each channel to have a delay of 20 registers between each feedforward and feedback multiplier.

The tool automatically distributes these registers throughout the circuit to function both as algorithmic delay registers and as pipeline registers, thereby achieving a high circuit f_{MAX}. Also, should the input data type be changed from real to complex, the tool automatically implements the adders and multipliers to handle the complex arithmetic.

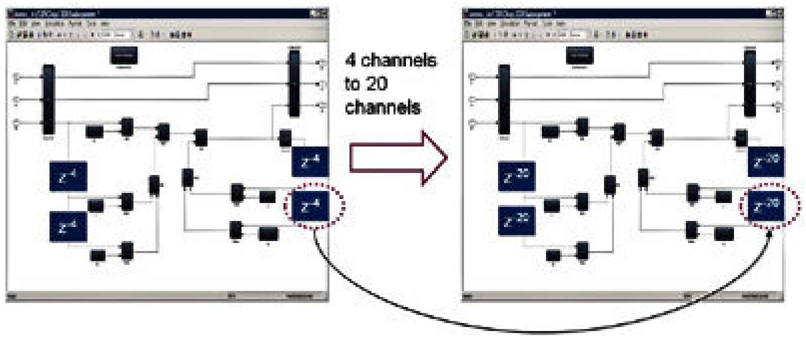


***Figure 8. IIR Filter** (To view larger image, **[click here]**)*

More complicated algorithms require more complicated implementations. An example of such a dataflow is depicted in **Figure 9 below** in the recursive part of the Mandelbrot implementation, which computes $z_{n+1} = z_n^2 + c$, then uses the feedback FIFO buffer to delay feedback data until needed to stream data without stalling In this case, additional latency in the recursive path is needed to allow enough register levels to efficiently implement the logic. This is accomplished through the use of FIFO buffers.

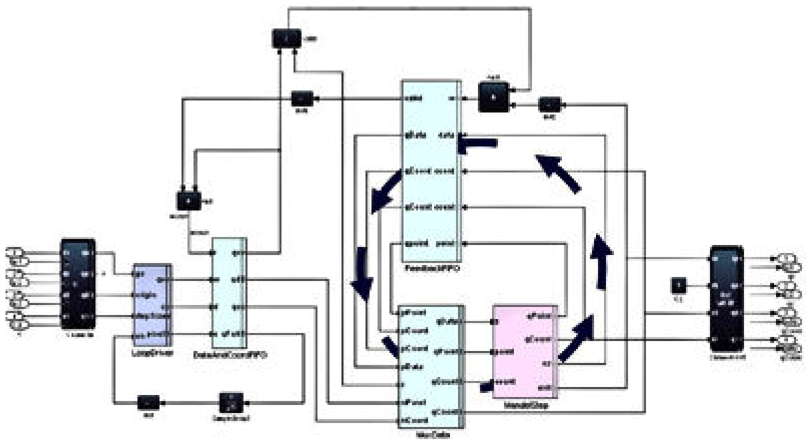


*Figure 9. Floating-Point Mandelbrot Design Example (*To view larger image, ***click here****)*

Library Support Requirements

In a software programming environment, the math.h utility is provided, which includes the following functions:

SIN, COS, TAN, ASIN, ACOS, ATAN, EXP, LOG, LOG10, POW(x,y), LDEXP, FLOOR, CEIL , FABS, SQRT, DIVIDE, and **1/SQRT**. Given the very common use of these functions in many signal-processing algorithms, these functions are also provided in this hardware flow (***Figure 10 below***).

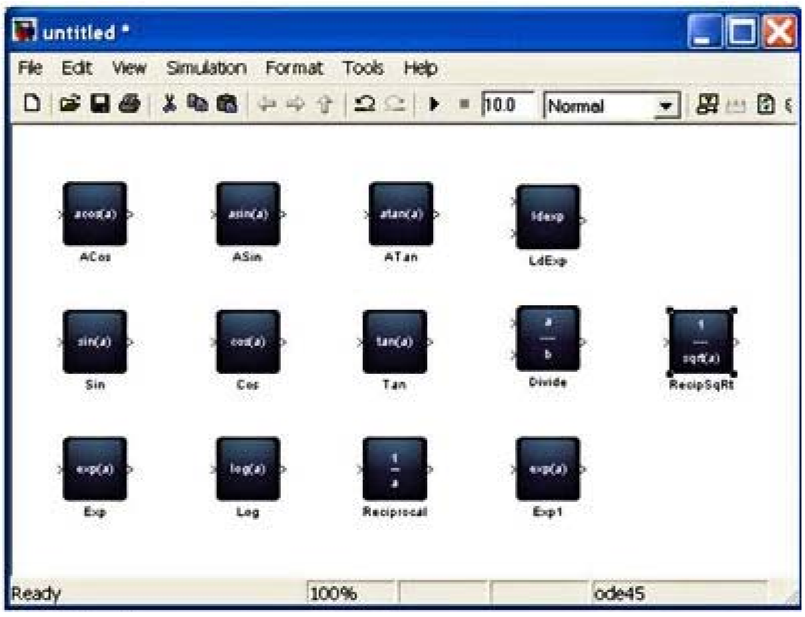


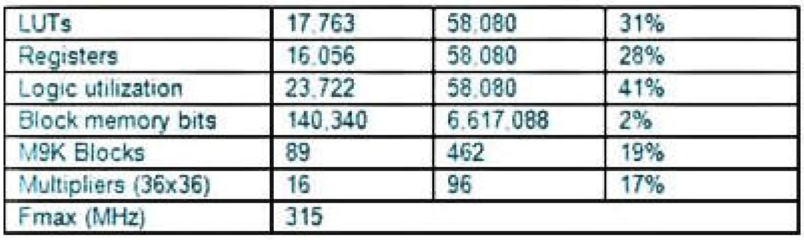
Figure 10. Math.h Functions

In an FPGA implementation, parallelism is used to reduce the latency normally associated with such function. Computation of a trigonometric function in a processor easily takes over 100 cycles. For FPGA implementations, the use of multiplier-based algorithms and use of several multipliers typically results in latency and resources usage of three to five times what a basic floating-point add or multiply requires. This means that the hardware designer need not restructure the implementation to minimize use of divide, trigonometric, square root, or other functions.

Matrix inversion is a critical function in many floating-point applications. Therefore, efficient implementation of common algorithms such as QR decomposition, LU decomposition, and Choleski is needed. This requires a toolflow to support the features discussed and very high degrees of parallelism.

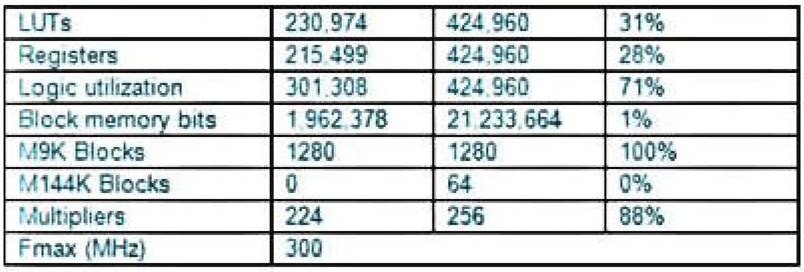
FPGA Floating-Point Benchmarks

Single or very few floating-point multiply-add circuits have been benchmarked at high clock rates. However, performance then tends to fall off very fast, as the traditional IEEE 754 implementation imposes unsustainable routing requirements in the FPGA. As the floating-point performance benchmarks for FFT and matrix multiplication in **Table 2** and **Table 3 below** demonstrate, the new tool flow is able to reduce routing resources to a sustainable level by using non-IEEE 754 numerical representation and the techniques previously described.



LUTs	17,763	58,080	31%
Registers	16,056	58,080	28%
Logic utilization	23,722	58,080	41%
Block memory bits	140,340	6,617,088	2%
M9K Blocks	89	462	19%
Multipliers (36x36)	16	96	17%
Fmax (MHz)	315		

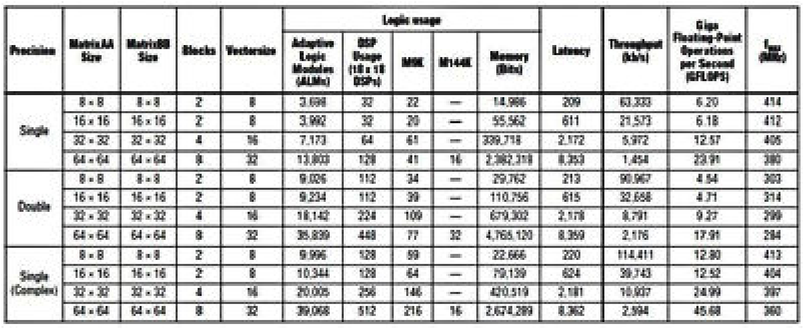
14 single precision FFTs (1024 pt) in Stratix IV 4SGX70 device



LUTs	230,974	424,960	31%
Registers	215,499	424,960	28%
Logic utilization	301,308	424,960	71%
Block memory bits	1,962,378	21,233,664	1%
M9K Blocks	1280	1280	100%
M144K Blocks	0	64	0%
Multipliers	224	256	88%
Fmax (MHz)	300		

Table 2. FFT Benchmarks

Additionally, the multiplier-to-logic ratios for these implementations are reasonable given the resource mixes of many FPGAs today. However, the benchmarks shown use FPGAs with 36x36 multipliers (composed of four 18x18 multipliers), and a large reduction in multiplier usage is expected for the newer FPGAs with native 27x27 multipliers.



Precision	MatrixAA Size	MatrixBB Size	Blocks	Vectorsize	Logic usage					Latency	Throughput (kb/s)	Giga Floating-Point Operations per Second (GFLOPS)	f_{MAX} (MHz)
					Adaptive Logic Modules (ALMs)	DSP Usage (18 x 18 DSPs)	M9K	M144K	Memory (Bits)				
Single	8 × 8	8 × 8	2	8	3,698	32	22	—	14,986	209	63,333	6.20	414
	16 × 16	16 × 16	2	8	3,992	32	20	—	55,562	611	21,573	6.18	412
	32 × 32	32 × 32	4	16	7,173	64	61	—	339,718	2,172	5,972	12.57	405
	64 × 64	64 × 64	8	32	13,803	128	41	16	2,382,318	8,353	1,454	23.91	380
Double	8 × 8	8 × 8	2	8	9,026	112	34	—	29,762	213	90,967	4.54	303
	16 × 16	16 × 16	2	8	9,234	112	39	—	110,756	615	32,658	4.71	314
	32 × 32	32 × 32	4	16	18,142	224	109	—	679,302	2,178	8,791	9.27	299
	64 × 64	64 × 64	8	32	35,839	448	77	32	4,765,120	8,359	2,176	17.91	284
Single (Complex)	8 × 8	8 × 8	2	8	9,996	128	59	—	22,666	220	114,411	12.80	413
	16 × 16	16 × 16	2	8	10,344	128	64	—	79,139	624	39,743	12.52	404
	32 × 32	32 × 32	4	16	20,005	256	146	—	420,519	2,181	10,937	24.99	397
	64 × 64	64 × 64	8	32	39,068	512	216	16	2,674,289	8,362	2,594	45.68	360

Table 3. Matrix Multipler Benchmarks (*To view larger image****, click here)***

QR decomposition was also implemented, using a complex, single-precision matrix of 200x192. The performance achieved and resources used are as follows:

Algorithmic requirements: $m*(k^2+k)$ complex multiplies per QRD - $200 * (192^2+192) = 7{,}411{,}200$

System requirements: QRD performed in less than 1 ms - Clock rate of 250 MHz

Parallelism: 7,411,200 complex multipliers / (1 ms * 250 MHz) = 29.6 complex multipliers in parallel are needed

[*Chosen: 32 parallel complex multipliers = 128 real multipliers (either 27x27 or 36x36)*]

FPGA resource requirements: 138 multipliers (27x27 or 36x36), 120K registers, 96K FPGA look-up tables (LUTs), 8-Mb memory.

Additional circuit blocks were used to implement both forward and back substitution, meeting the performance requirements in a radar application. This particular design used a vector of 34. By updating the design to support a vector to 200, a throughput increase of five times is achievable at the expense of more FPGA resources. This ability to easily trade resources for throughput is a key advantage of implementing these types of application in hardware.

Another matrix inversion design using the Choleski algorithm was built using this toolflow. The complex, single-precision 256x256 matrix operated with a latency of less than 1 ms, or over 1,000 matrix inversions per second.

Conclusion

This paper demonstrates that by using new floating-point processing techniques, high-performance implementation of large designs on large FPGAs can be readily implemented. The throughput and clock rates are comparable to that of fixed-point implementations, but provide the floating-point benefit of high dynamic range and eliminate most numerical processing issues. Traditional IEEE 754-compatible interfaces are supported to allow easy integration, and support common test benches and simulation environments between system and hardware design teams.

In order to synthesize floating-point circuits, the designer must give up the detailed circuit description using Verilog or VHDL. Instead, a model-based design description using The Mathworks's Simulink environment is used to provide a sufficient level of abstraction for automated synthesis of floating-point datapaths.

This environment also provides the ability to incorporate other features that allow high productivity floating-point design flow, such as vector representation and support, auto-pipelining, mathematical functions, and to leverage the extensive amount of hard multipliers available in today's FPGAs.

*As senior DSP technical marketing manager, **Michael Parker** is responsible for DSP-related IP at **Altera**, and is also involved in optimizing FPGA architecture planning for DSP applications. Mr. Parker joined Altera in January 2007, and has over 20 years of DSP wireless engineering design experience with Alvarion, Soma Networks, TCSI, Stanford Telecom, and several startup companies.*