



WP387 (v1.0) February 16, 2011

Scaling Up to TeraFLOPs Performance with the Virtex-7 Family and High-Level Synthesis

By: Oliver Garreau and Jack Lo

FPGAs are typically associated with vast areas of programmable logic, high I/O pin count, and large fixed-point DSP arrays. FPGAs are not as well known for their floating-point processing capabilities, which are, in fact, quite impressive. Floating-point operators are mapped through a combination of programmable logic (LUT) and DSP processing elements (DSP slices).

A typical floating-point benchmark, large matrix multiplication, is described in this white paper to showcase a user-friendly C-to-Gate technology that can propel Xilinx Virtex®-7 FPGAs into TeraFLOPS territory, using powerful and flexible high level synthesis (HLS) tools.

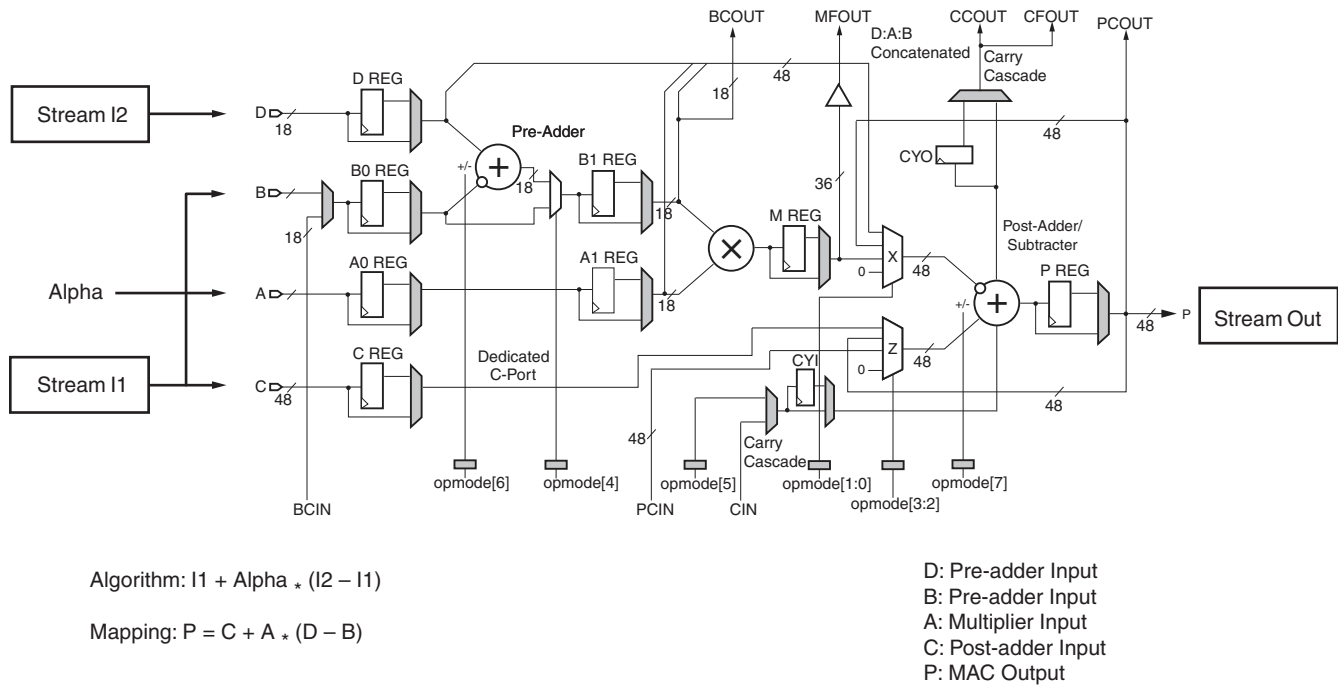
This white paper provides an overview of fixed- and floating-point DSP coding techniques, ranging from RTL to the Xilinx portfolio of IP and tools. It also describes advances with High-Level Synthesis (HLS) tools, like AutoPilot, how FPGA designs can benefit from coding in a “natural language” like C or C++, and how easily FPGAs can be programmed by a large community of software programmers.

Traditional DSP Design with Xilinx FPGAs: RTL Language for Fixed- and Floating-Point DSP

A very common operation in DSP is the multiply-accumulate (MACC):

$$a \leq a + (b \times c)$$

where b and c are the input vectors and a is the internal accumulator. See [Figure 1](#). Xilinx FPGAs support this hardware operation for fixed-point arithmetic, but they also support floating-point via dedicated libraries.



WP387_01_011311

Figure 1: DSP48A1 Slice Configured for Alpha-Blending

The traditional approach to coding an FPGA is to create RTL code in a text format and insert the source files in a design project within the Xilinx design environment (ISE® Design Suite). In this approach, customers will need to be proficient in Verilog or VHDL and have a solid understanding of FPGA block architecture. The hardware designer can achieve a very fine level of control of the hardware design at the gate level and is responsible for the whole data flow and pipeline management, including the pipelines balancing, whereas HLS FPGA tools can tremendously assist the designer with pipeline management. However, the DSP block detailed architecture, as shown in [Figure 1](#), can be overwhelming to software programmers, which is why Xilinx offers powerful DSP development software tools and DSP libraries. The most recent tools, like HLS, which process high-level language like C and C++, increase the level of hardware abstraction and make FPGAs more accessible to software designers. As an example of pipeline management, a fixed-point project can be defined, shown schematically in [Figure 1](#). This example is a simple alpha-blending algorithm merging two color video streams into one with an 8-bit per color component. The alpha coefficient is also coded over 8 bits. One DSP slice is used for each color component, so three slices are required to process 24-bit colors.

Two 8-bit vectors represent the two video streams, one pixel at a time, and are fed to three of the DSP slice pipelines (B, C, and D). The Alpha coefficient is fed to the A pipeline, and the blended video stream is coming out of the P pipeline. For example, the matching and summarized code for the direct RTL implementation (VHDL), optimized for the FPGA DSP slice in Spartan®-6 FPGAs (DSP48A1) is:

```
entity AlphaBlender is
  port (
    clk      : in  std_logic;
    ce       : in  std_logic;
    alpha    : in  std_logic_vector(7 downto 0);
    pixel1   : in  std_logic_vector(7 downto 0);
    pixel2   : in  std_logic_vector(7 downto 0);
    pixelout : out std_logic_vector(7 downto 0)
  );
end AlphaBlender;

begin

  bin <= ext(pixel1 & AlphaPadding, DspDataWidth);
  cin <= ext(pixel1 & AlphaPadding & AlphaPadding, DspCascadeWidth);
  din <= ext(pixel2 & AlphaPadding, DspDataWidth);
  ain <= ext(alpha, DspDataWidth);

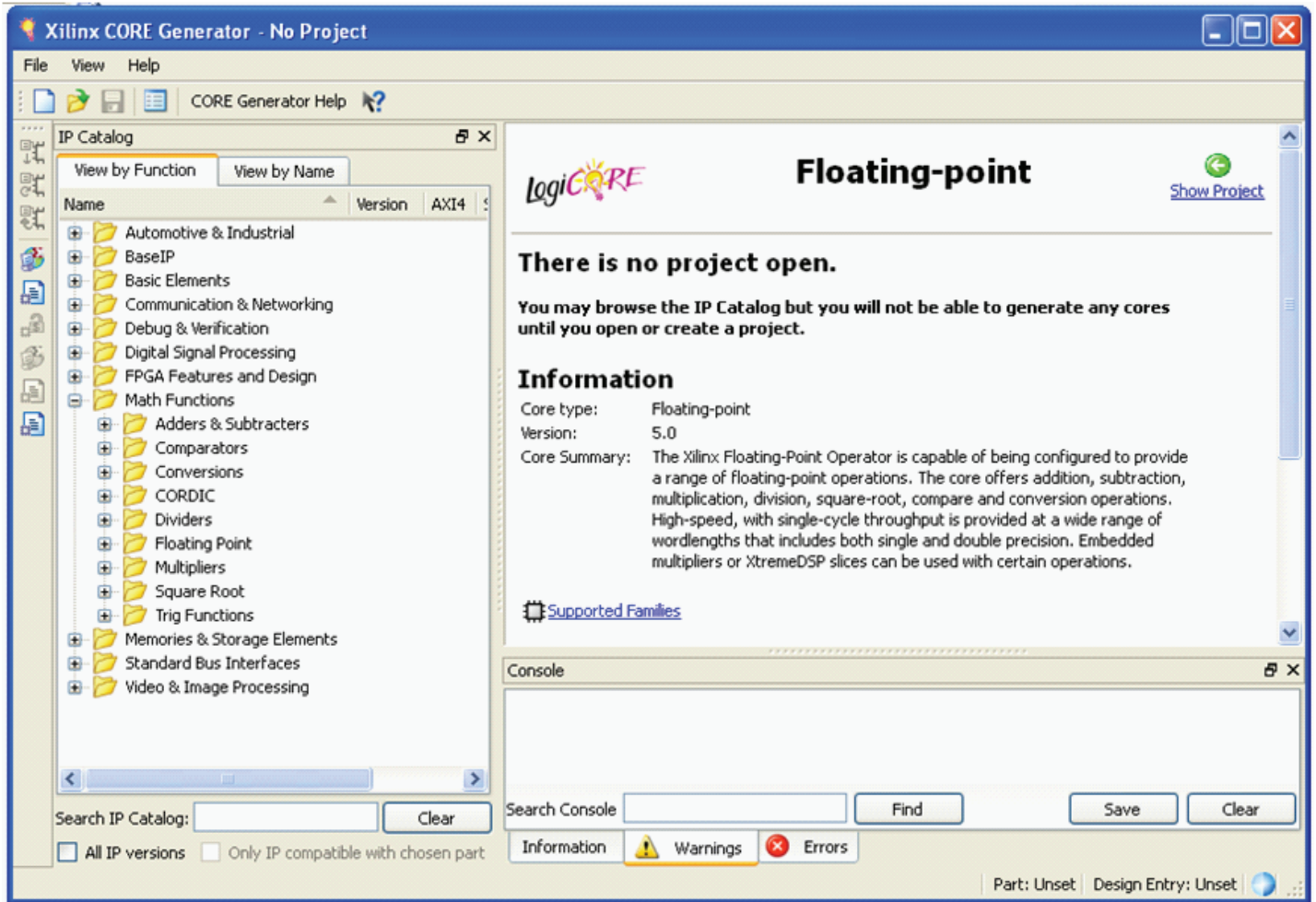
  addmultadd : process (clk)
  begin
    if rising_edge(clk) then
      ain_r1 <= ain;
      ain_r2 <= ain_r1;
      bin_r  <= bin;
      din_r  <= din;
      cin_r1 <= cin;
      cin_r2 <= cin_r1;
      cin_r3 <= cin;
      preadd_r <= din_r - bin_r;
      mult_r  <= preadd_r * ain_r2;
      sum_r   <= mult_r + cin_r3;
    end if;
  end process addmultadd;

  pixelout <= sum_r(2*AlphaWidth+PixelWidth-1 downto 2*AlphaWidth);

end behavioral;
```

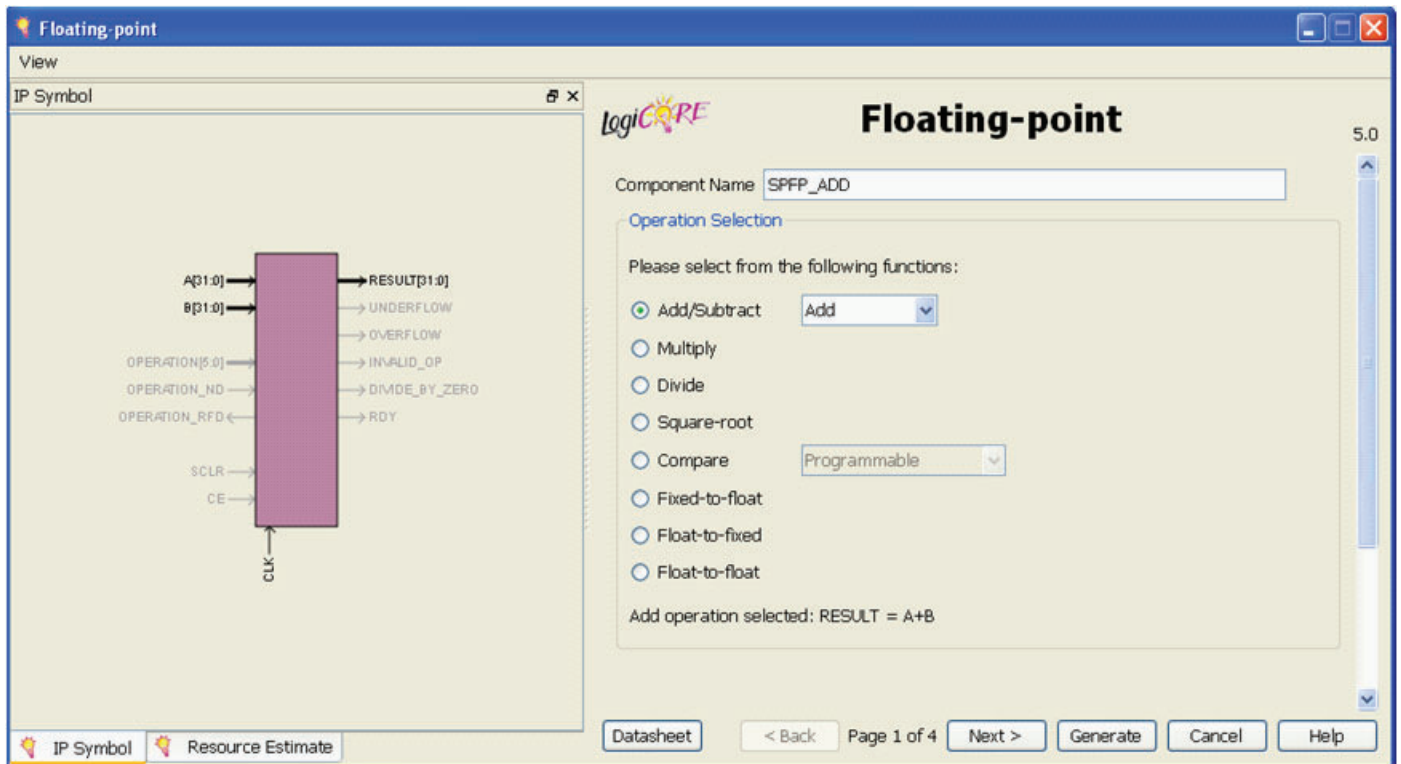
RTL design provides a high degree of control of the hardware implementation; the maximum level of performance can be obtained when the designer is familiar with the processing unit architecture. To optimize the learning curve, Xilinx started to unify the DSP slice architectures in Spartan-6 and Virtex-6 families and achieved a full unification in the 7 series FPGAs. This unification also allows a seamless device migration. To maximize the throughput of the DSP slice, the designer must design according to the pipeline structures of the DSP slice and must virtually rebuild that structure in RTL. In other words, the RTL designer is in charge of the pipeline balancing whereas HLS FPGA tools, like AutoPilot from AutoESL Design Technologies, can assist with pipelining.

Xilinx delivers basic floating-point cores through CORE Generator™ software. These cores can be instantiated with a variety of different parameters, as shown in [Figure 2](#) and [Figure 3](#). Typically, these cores are more heavily pipelined than fixed-point arithmetic to achieve high clock speeds.



WP387_02_010411

Figure 2: CORE Generator Software Floating-Point Interface



WP387_03_010411

Figure 3: CORE Generator Software Floating-Point Interface — Configuration of the FP Operator

Model Based Design Flow for Floating-Point: Xilinx System Generator

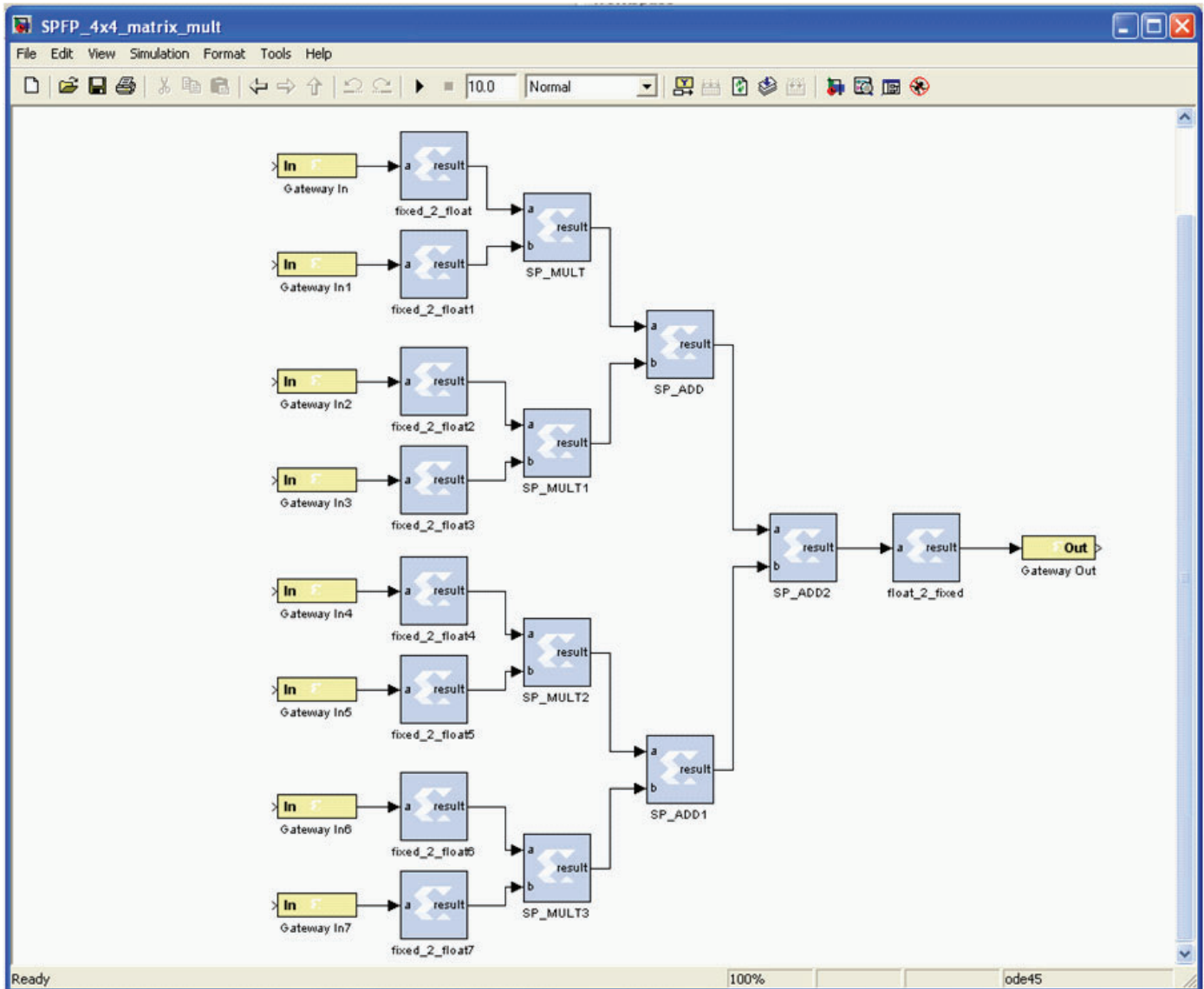
After the various FP operators (FPO) are created, the designer assembles them and creates the top level of the DSP engine with a Xilinx tool called System Generator. With System Generator for DSP, floating-point IP such as the Xilinx floating-point operator can be brought into System Generator using the black box HDL import flow. The System Generator black box block allows VHDL, Verilog, and EDIF to be brought into a design. The black box block behaves like other System Generator blocks; it is wired into the design, participates in simulations, and is compiled into hardware. When System Generator compiles a black box block, it automatically wires the imported module and associated files into the surrounding netlist. System Generator simulates black boxes by automatically launching an HDL simulator, generating additional HDL as needed (analogous to an HDL testbench), compiling HDL, scheduling simulation events, and handling the exchange of data between Simulink® software and the HDL simulator.

This is called HDL co-simulation. Black box HDL can be co-simulated with Simulink software, a MathWorks, Inc. tool, using the System Generator interface to either ModelSim (simulation software from Model Technology, Inc.) or the ISE Simulator.

As a design example, the following algorithm represents the test example for a 4×4 FP Matrix Multiplication (shown in Equation 1). If $R(i, j)$, also called a dot product, is the result matrix and M and N are the matrixes to multiply, any component of the R matrix should be processed this way:

$$R(i,j) = M(i,1) \bullet N(1,j) + M(i,2) \bullet N(2,j) + M(i,3) \bullet N(3,j) + M(i,4) \bullet N(4,j) \quad \text{Equation 1}$$

Figure 4 shows a design of a 4 x 4 matrix multiplication for a given matrix component, after being captured in Simulink.



WP387_04_010411

Figure 4: 4 x 4 Matrix Multiplication in Simulink

Figure 4 includes the operators to compute one element of the result matrix (i.e, one dot product).

After this elementary matrix operator is built, it can be:

- Built 16 times over to represent the whole matrix multiplication, producing the most parallel implementation and the lowest latency, or
- *Scheduled* 16 times over for a sequential implementation, producing the lowest FPGA resource utilization

Other trade-offs can be found between latency and resource utilization, although it is the responsibility of the DSP designer to implement any design mitigation.

Based on [Equation 1](#), a standard 4x4 multiplication requires only three additions and four multiplications per dot product, so only seven elementary FP operators are required (three adders and four multipliers).

Note: The implementation in [Figure 4](#) shows 16 operators because nine additional operators are needed to convert a fixed-point test vector to a FP test vector. This is needed only for simulation purposes in Simulink with ISE Design Suite 12.x.

This coding technique allows the design of fairly simple FP projects with a relatively flat architecture. The MATLAB/Simulink environment is efficient; the DSP designer can reuse vast libraries of components, processors, and generators. It allows the exploration of several implementations in parallel and, in the case of this FP exercise, allows comparing the performance of algorithms coded in floating point against their counterparts coded in fixed point.

This design approach uses a model-based methodology with pre-existing floating-point operators. It allows only several levels of architectural complexity and remains a manual entry process. For larger or more complex projects with deeper hierarchies, the designer might want to use higher-level tools, like HLS tools, that offer coding languages like C, C++, or SystemC.

High-Level Synthesis Tools Suitable for FP Projects: Natural Language

To take advantage of the vast DSP resources that FPGAs offer and accelerate the design process for larger FP applications, HLS tools can be used for design entry and verification. FP operations are typically defined in C/C++, and modern HLS tools offer a powerful way of compiling that C code into RTL. HLS tool performance has expanded tremendously in recent years with product offerings from Synopsys, AutoESL Design Technologies, Mentor Graphics, and many other vendors. The ability of HLS tools to capture and manage large or complex designs becomes more important as overall design complexity and available FPGA resources continue to increase.

Some key benefits of HLS are:

- C/C++ coding and simulation environment
- Shorter development times due to ability to iterate and test design in C
- Ability to rearchitect RTL design by changing implementation parameters
- Ability to retarget design to multiple Xilinx FPGA families
- Maintainability of design for future Xilinx FPGA products
- Ability to scale the problem or algorithm to any size due to a highly parameterized coding capability.

To illustrate this point, the process of designing a larger 16x16 FP matrix multiplier can be examined. A description of this algorithm in C++ language is shown this code example:

```

template<int A_ROWS, int A_COLS, int B_COLS >
void matrix_multiply (float a_in[A_ROWS][A_COLS],
                    float b_in[A_COLS][B_COLS],
                    float c_out[A_ROWS][B_COLS])
{
    int index_a, index_b, index_c;
    float tmp_a, tmp_b, tmp_mult, sum_mult;

    // matrix multiplication of a A*B matrix
    for (index_a = 0; index_a < A_ROWS; index_a++) {
        for (index_b = 0; index_b < B_COLS; index_b++) {
            sum_mult = 0.0f;
            for (index_c = 0; index_c < A_COLS; index_c++) {
                tmp_a = a_in[index_a][index_c];
                tmp_b = b_in[index_c][index_b];

                tmp_mult = tmp_a * tmp_b;

                sum_mult += tmp_mult;
            }
            c_out[index_a][index_b] = sum_mult;
        }
    }
}

```

This design can be tested and debugged in C and verified using a C testbench. This same design source can then be pushed through an HLS tool (such as AutoPilot) to generate an RTL implementation in minutes. The result of this initial design that targets a Virtex®-6 XC6VLX240T-FF1156-3 FPGA is shown in [Table 1](#).

Table 1: Initial Results for 16x16 Multiply

	Initial Design	Units
Latency	103,200	cycles
DSPs	3	-
LUTs	650	-
Slices	190	-
F _{MAX}	551	MHz
Throughput	187.3	µs
FPO core ⁽¹⁾	1	-

Notes:

1. Number of FPO cores used per each matrix multiplication

At this point, the designer can optimize this design and reduce the latency of the function further. This is done by examining the C code and inserting directives for controlling how different parts of the design are implemented. In particular, controlling the behavior of loops has a direct effect on the design's performance. Adding a simple loop initiation interval directive (`#pragma`) in the innermost loop, for example, can reduce the overall latency by directing the inner loop to run in a pipelined fashion rather than serially. Following is the code with the pragma directive inserted in the correct location:


```

template<int A_ROWS, int A_COLS, int B_COLS >
void matrix_multiply (float a_in[A_ROWS][A_COLS],
                     float b_in[A_COLS][B_COLS],
                     float c_out[A_ROWS][B_COLS])
{
    int index_a, index_b, index_c;
    float tmp_a, tmp_b, tmp_mult, sum_mult;

    // matrix multiplication of a A*B matrix
    for (index_a = 0; index_a < A_ROWS; index_a++) {
        for (index_b = 0; index_b < B_COLS; index_b++) {
            sum_mult = 0.0f;
            for (index_c = 0; index_c < A_COLS; index_c++) {
#pragma AP PIPELINE II=14
                tmp_a = a_in[index_a][index_c];
                tmp_b = b_in[index_c][index_b];

                tmp_mult = tmp_a * tmp_b;

                sum_mult += tmp_mult;
            }
            c_out[index_a][index_b] = sum_mult;
        }
    }
}

```

Tailoring the Design to Optimize FPGA Density

"AP PIPELINE II" of 14, which specifies the initiation interval (II) in this new pragma directive, reduces the latency to nearly half that of the initial design at the cost of increased resource utilization (most notably, a larger number of flip-flops to implement the pipeline).

Additional optimizations can be made to further refine the design. For example, moving the pipeline directive from the innermost loop to the one above it effectively unrolls the inner loop and pipelines it. This move reduces the latency dramatically by a factor of 26 at the cost of area. Still more optimizations can be made by controlling the partitioning of the memories used to store/access the input matrices, allowing them to be accessed in parallel and moving the addition operation to its own separate parallel loop.

The results of these various optimizations are shown in [Table 2](#).

Table 2: Final Optimization Results for 16 x 16 FP Multiply

ISE Design Suite 12.3	Initial	II = 14 ⁽¹⁾	II = 8 ⁽²⁾	Parallel			Units
				II = 4 ⁽²⁾	II = 2 ⁽²⁾	II = 1 ⁽²⁾	
Latency	103,200	61,185	2,279	1,094	581	325	cycles
DSPs	3	3	6	12	24	48	-
LUTs	650	693	3,223	4,026	6,166	9,588	-
Slices	190	259	1,182	1,535	2,101	3,536	-
F _{MAX}	551	516	403	451	416	362	MHz
Throughput	187.3	118.6	5.7	2.4	1.4	0.9	μs
FPO Core	1	1	2	4	8	16	-

Notes:

1. Refers to the innermost loop.
2. Refers to the loop above the innermost loop.

The iteration time for each of these designs is in the order of minutes (compared to hours potentially caused by RTL code changes), and each design generates a completely unique organization of pipelines and control logic to optimize for a target frequency and latency requirement. This illustrates the power of HLS tools in design exploration and verification.

A typical design flow involves the following steps:

1. The application functionality is captured in C and its behavior is verified through C simulations.
2. The design is compiled and implemented using the HLS tool, and the latency results are reviewed.
3. The design is optimized via directives and is recompiled, and the results are reviewed. This step is iterated until results are achieved or no further optimization is feasible.
4. Optionally, the code can be rearchitected in C and verified for correctness in C. The design needs to be recompiled, and the latency results are reviewed.
5. [Step 3](#) or [Step 4](#) should be iterated until results are achieved or no further optimization is feasible.
6. The final generated RTL is integrated into the FPGA design.

Compiling the design might also involve running the FPGA implementation tools to ensure that the estimated performance from the HLS tool matches that of the FPGA.

For the majority of DSP designs, HLS design is a powerful way to describe, test, and implement a design on an FPGA, with design results rivaling those of hand-coded RTL in a fraction of the development time.

Performance to Expect in Upcoming 7 Series FPGAs

This section includes an overview of FP processing capability of Xilinx FPGAs while implementing the HLS design flow and reports the benchmark data of a real-valued floating-point operator (matrix multiplication of two 16 x 16 matrixes) processing data in single-precision IEEE floating point (32-bit).

The first two columns are showcasing two existing Virtex-6 devices; the next two columns are showcasing the two largest devices of the upcoming Virtex-7 FPGA family. For each target, the place and route tools were set up for the fastest speed grade (-3) and DSP resources were maximized.

The performance data in GFlops are peak numbers and make the assumption that data pipes are properly designed to feed a sustained data stream to the FP operators.

Table 3 shows the benchmark results after being synthesized with AutoPilot and IDS 12.4.

One 16 x 16 floating-point matrix multiplication equates to 7,936 single-precision FLOPS.

II = 1 (AutoESL setup),
Design Suite 12.4,
AutoPilot 2010.a.3

Table 3: Summary of Features in Virtex-6 and Virtex-7 Devices, Maximum Performance

Specific Device	Virtex-6 Family		Virtex-7 Family	
	XC6VLX240T	XC6VSX475T	XC7V2000T	XC7VX850T
Speed Grade (Software 12.4)	-3	-3	-3	-3
DSP slice count	768	2,016	2,160	3,960
FPO frequency (MHz)	450	450	500	500
DSP slice usage/matrix multiplication	48	48	48	48
Matrix multiplication per device	16	42	45	82
Matrix multiplication frequency (MHz)	1.59	1.59	1.76	1.76
Performance (GFLOP) ⁽¹⁾	201	527	628	1,145

Notes:

1. Performance is computed by multiplying the number of FLOPS per matrix multiplication, times the number of matrix multiplication per device, times the frequency of the matrix multiplication.

In this example and benchmark based on a single-precision 16x16 matrix multiplication synthesized from C, the performance in the largest Virtex-6 device exceeds 0.5 TeraFLOPs, and in the largest Virtex-7 device exceeds 1.1 TeraFLOPs.

Conclusion

Various methods of designing DSP projects in both fixed point and floating point have been described. This underscores the great flexibility of FPGAs to address the multiple precisions needed for DSP, as well as the ability to address several data representations, whether fixed at 18 or 25 bits, or floating from 32 to 64 bits.

Coding methods are also flexible and include low-level operator instantiation in RTL, a block design approach with IP generation tools, high-level language synthesis in C and SystemC, and even object-oriented programming with C++.

The HLS method has become an effective approach to coding large DSP projects in a “natural language” like C. This should appeal to a class of software designers more accustomed to high-level languages like C/C++, but without sacrificing the

performance levels that were obtained with RTL languages. A tool like Auto Pilot allows a DSP designer to quickly explore a range of automatically optimized DSP architectures.

The simple floating-point example based on a matrix multiplication shows that, as an HLS project, it enables a peak theoretical performance of *0.5 TeraFLOPs* (Single Precision) for the largest Virtex-6 FPGA, and beyond *1.0 TeraFLOPs* (Single Precision) for the largest Virtex-7 FPGA. It also offers fast design and optimization cycles, providing a shorter time to market.

In summary, DSP designers using Xilinx FPGAs and HLS tools can achieve outstanding floating-point performance levels with greater ease and efficiency through the use of familiar high-level language coding and tools.

Additional Information

1. [Product Selection Guide](#)
2. [UG369](#), *Virtex-6 FPGA DSP48E1 Slice User Guide*
3. [DS335](#), *Floating-Point Operator Data Sheet*
4. [UG640](#), *System Generator for DSP User Guide*

Revision History

The following table shows the revision history for this document:

Date	Version	Description of Revisions
02/16/11	1.0	Initial Xilinx release.

Notice of Disclaimer

The information disclosed to you hereunder (the "Information") is provided "AS-IS" with no warranty of any kind, express or implied. Xilinx does not assume any liability arising from your use of the Information. You are responsible for obtaining any rights you may require for your use of this Information. Xilinx reserves the right to make changes, at any time, to the Information without notice and at its sole discretion. Xilinx assumes no obligation to correct any errors contained in the Information or to advise you of any corrections or updates. Xilinx expressly disclaims any liability in connection with technical support or assistance that may be provided to you in connection with the Information. XILINX MAKES NO OTHER WARRANTIES, WHETHER EXPRESS, IMPLIED, OR STATUTORY, REGARDING THE INFORMATION, INCLUDING ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT OF THIRD-PARTY RIGHTS.