XAPP599 (v1.0) September 20, 2012

Floating-Point Design with Vivado HLS

Author: James Hrica

Summary

This application note describes how the Vivado™ High-Level Synthesis (HLS) tool transforms a C/C++ design specification into a Register Transfer Level (RTL) implementation for designs that require floating-point calculations. While the basics of performing HLS on floating-point designs are reasonably straightforward, there are some more subtle aspects that merit detailed explanation. This application note presents details on the basics and advanced topics relating to design performance, area, and verification of implementing floating-point logic in Xilinx FPGAs using the Vivado HLS tool.

Introduction

Although fixed-point arithmetic logic (which is usually implemented as just integer arithmetic, perhaps with some saturation and/or rounding logic added) is generally faster and more area efficient, it is sometimes desirable to implement mathematical calculation using a floating-point numerical format. While fixed-point formats can achieve precise results (or exact, given appropriate room to grow), a given format has a very limited dynamic range, deep analysis is generally required in order to determine the bit-growth patterns throughout a complex design and many intermediate data types (of varying fixed point formats) must be introduced to achieve optimal Quality-of-Results (QoR). Floating-point formats represent real numbers in a much wider dynamic range, which allows a single data-type to be used through long sequences of calculations that is required by many algorithms. From a hardware design perspective, the cost of these features is greater area and increased latency, as the logic required to implement a given arithmetic operation is considerably more complex than for integer arithmetic.

The Vivado HLS tool supports the C/C++ float and double data-types, which are based on the single- and double-precision binary floating-point formats as defined by the *IEEE-754 Standard* [Ref 1]. For a detailed explanation of the floating-point formats and arithmetic implementation see the *IEEE-754 Standard* [Ref 1] or PG060, *LogiCORE IP Floating-Point Operator v6.1 Product Guide* [Ref 2] for a good summary. A very important consideration when designing with floating-point operations is that these numerical formats cannot represent every real number and therefore have limited precision.

This point is more subtle and complicated than it might first seem and much has been written on this topic and the user is encouraged to peruse the offered references [Ref 3], [Ref 4], and [Ref 5]. Generally speaking, the user should not expect an exact match (at the binary representation level) for results of the same calculation accomplished by different algorithms or even differing implementations (micro-architectures) of the same algorithm, even in a pure software context. Several sources for such mismatches include:

- Accumulation of rounding error, which can be sensitive to the order in which operations are evaluated
- FPU support of extended precision affect on rounding of results, for example x87 80-bit format; SIMD (SSE, etc.) instructions behave differently to x87
- Library function approximations, for example float trigonometric function
- Many floating-point literal values can only be approximately represented, even for rational numbers
- Constant propagation/folding effects

- Handling of subnormals

  Note: Subnormals are sometimes used to represent numbers smaller than the normal floating-point format can represent. For example, in the single-precision format, the smallest normal floating-point value is 2^{-126}. However when subnormals are supported, the mantissa bits are used to represent a fixed point number with a fixed exponent value of 2^{-127}. See *IEEE-754 Standard* [Ref 1] and PG060, *LogiCORE IP Floating-Point Operator v6.1 Product Guide* [Ref 2] for more details.

Some simple, but compelling, software examples are offered here to motivate attention to Validating the Results of Floating-Point Calculations.

Example 1 demonstrates that different methods (and even what appears to be the same method) of doing the same calculation can lead to slightly different answers. Example 2 helps illustrate not all numbers, even whole (integer) values, have exact representations in binary floating-point formats.

Example 1: Different Results for the Same Calculation:

```
// Simple demo of floating point predictability problem
int main(void)
{
    float fdelta = 0.1f; // Cannot be represented exactly
    float fsum = 0.0f;

    while (fsum < 1.0f)
      fsum += fdelta;
    float fprod = 10.0f * fdelta;
    double dprod = float(10.0f * fdelta);
    cout.precision(20);
    cout << "fsum:  " << fsum << endl;
    cout << "fprod: " << fprod << endl;
    cout << "dprod: " << dprod << endl;
   return 0;
}
```

Program output:

```
fsum:  1.0000001192092895508
fprod: 1
dprod: 1.0000000149011611938
```

The first output result in Example 1 is the result of summing the approximation of 0.1 ten times, leading to accumulation of rounding errors. On each iteration, the inexact single-precision value 0.1 is added to the running sum, which is then stored in a single-precision (32-bit) register. As the (base-2) exponent of the sum grows (from –4 to 0) rounding the intermediate sum occurs four times, regardless of the internal precision of the floating point unit (FPU).

For the second value, the computation is carried out using the x87 extended precision and the result is rounded before being stored in single-precision format. For the third value, the multiplication is also done at extended precision, but it is rounded and stored in the double-precision format, leading to a different inaccuracy in the result.

Note: This code might produce different results from those shown when compiled for different machine architectures and/or with different compilers

Example 2: Even Whole Numbers Can Lose Precision

```
// Another demo of floating point predictability problem
int main(void)
{
   int i;
   float delta = 1.0f;

   for (int i = 0; i < 100000000; i++) {
      float x = (float)i + delta;
      if (x / (float)i <= 1.0f) { // (x / i) should always be > 1.0
         printf("!!! ERROR on iteration #%d !!!\n", i + 1);
         return -1;
      }
   }
   return 0;}
```

Program output:

```
!!! ERROR on iteration #16777217 !!!
```

This example shows that integer values above 16,777,216 (2^{24}) lose precision when stored in single-precision floating-point format because the significand has 24 bits and beyond that point least significant bits of `i` must be dropped when cast to float format.

The Basics of Floating-Point Design Using the Vivado HLS Tool

Native support for HLS of the basic arithmetic operators (+, -, *, /), relational operators (==, !=, <, <=, >, >=), and format conversions (e.g., integer/fixed to float and float to double) is accomplished by mapping these operations onto Xilinx LogiCORE™ IP Floating-Point Operator cores instantiated in the resultant RTL. Additionally, calls to the sqrt()family of functions (from the C/C++ standard math library), code of the form 1.0/x and 1.0/sqrt(x) are mapped onto appropriate Floating-Point Operator cores. Although these cores can be generated for custom precision floating-point types by the CORE Generator™ tool, only the single- and double-precision versions of the cores described by the *IEEE-754 Standard* are generated by the Vivado HLS tool. A potential source of (small) differences in results generated by software versus hardware based on the Floating-Point Operator cores is that these cores handle subnormal inputs by "flushing to zero", that is, they are replaced by 0.0 when encountered. For details regarding the behavior of these cores, see PG060, *LogiCORE IP Floating-Point Operator v6.1 Product Guide* [Ref 2].

Many of the other functions from the C(99)/C++ standard math library, for which there are no Floating-Point Operator cores available, are also supported by the Vivado HLS tool. For the complete lists of functions, see UG902, *Vivado Design Suite User Guide: High-Level Synthesis v2012.2* [Ref 6]. The approximation algorithms underlying many of these functions have been selected and optimized for implementation on Xilinx FPGAs and their accuracy characteristics might differ somewhat from those specified in software standards (see UG902, *Vivado Design Suite User Guide: High-Level Synthesis v2012.2* [Ref 6]).

The standard math library functions are only implemented as single- and double-precision floating-point hardware. If the user calls any of these functions with integer or fixed point arguments and/or return variables, format conversions (to/from floating point) are implemented where necessary and the calculations are carried out in floating point.

All floating-point operators and functions supported for HLS can be used in a fully pipelined context and produce one result per clock cycle, provided there is no feedback path through the operation. Although the Vivado HLS tool can schedule these operations for infrequent, sequential execution, the implementation might not be the most area efficient possible, especially with respect to FF utilization. This applies even for '/' and sqrt(), for which low throughput cores are available.

Using <math.h> in ANSI/ISO-C Based Projects

To use the supported standard math library functions in an ANSI/ISO-C based projects, the `math.h` header file should be included in all source file making calls to them. The base functions are intended to operate on (and return) double-precision values, for example, double sqrt(double). Single-precision versions of most functions have an 'f' appended to the function name, for example, float sqrtf(float), float sinf(float), and float ceilf(float). It is important to bear this in mind, because otherwise not only would a much larger (in FPGA resources) double-precision version be implemented even though the argument and return variables are single-precision, there are format conversions that use additional resources and add latency to the calculation.

Another consideration when working in ANSI/ISO-C is that when compiling and running the code as software (including the C test bench side during RTL co-simulation, different algorithms are used than are implemented in the HLS produced RTL. In the software, the GCC libc functions are called and on the hardware side the Vivado HLS tool math library code is used. This can lead to bit-level mismatches between the two, when both results might be quite close to the real (in the analytical sense) answer.

Example 3: Unintended Use of Double-Precision Math Function:

```
// Unintended consequences?
#include <math.h>

float top_level(float inval)
{
   return log(inval); // double-precision natural logarithm
}
```

This example leads to an RTL implementation that converts the input into double-precision format, calculates the natural logarithm at double-precision, then converts the result to single-precision for output.

Example 4: Explicit Use of Single-Precision Math Function:

```
// Be sure to use the right version of math functions...
#include <math.h>

float top_level(float inval)
{
   return logf(inval); // single-precision natural logarithm
}
```

Because the single-precision version of the logarithm function is called, that version is implemented in the RTL with no need for input/output format conversions.

Using <cmath> in C++ Projects

When designing with C++, the most straightforward way to get support for the standard math library is to include the `<cmath>` system header file in all source files that call its functions. This header file provides versions of the base (double-precision) functions overloaded to take as arguments and return single-precision (float) values in the std namespace. For the single-precision version to be used, the std namespace must be in scope, either by using the scope resolution operator (::) or by importing the entire namespace with the using directive.

Example 5: Explicit Scope Resolution:

```
// Using explicit scope resolution
#include <cmath>

float top_level(float inval)
{
   return std::log(inval); // single-precision natural logarithm
}
```

Example 6: Exposing Contents of a Namespace to File Scope:

```
// import std:: namespace
#include <cmath>
using namespace std;

float top_level(float inval)
{
   return log(inval); // single-precision natural logarithm
}
```

As with the usage of `<math.h>` in ANSI/ISO-C projects, when using functions from `<cmath>` in code to be synthesized by the Vivado HLS tool, results might differ between that code run as software versus the RTL implementation because different approximation algorithms are used. For this reason, access to the algorithms used to synthesize the RTL is provided for use in C++ modeling.

When validating changes to C++ code destined for HLS and later co-simulating the resultant RTL with a C++ based test bench, it is recommended that the HLS sources use the same math library calls and the test bench code use the C++ standard library to generate reference values. This provides an added level of verification of the HLS models and math libraries during development.

To follow this methodology, the `<hls_math.h>` Vivado HLS tool header file should be included only in any source files to be synthesized to RTL. For source files only involved in validating the HLS design, such as the test program and supporting code, the `<cmath>` system header file should be included. The HLS version of the functions in the `<hls_math.h>` header file are part of the hls:: namespace. For the HLS versions to be compiled for software modeling/validation, use hls:: scope resolution for each function call.

Note: It is not recommended to import the hls:: namespace (via 'using namespace hls') when using C++ standard math library calls, because this can lead to compilation errors during HLS. Example 7a illustrates this usage.

Example 7a: Test Program Uses Standard C++ Math Library:

```
// Contents of main.cpp - The C++ test bench
#include <iostream>
#include <cmath>

using namespace std;

extern float hw_cos_top(float);

int main(void)
{
   int mismatches = 0;
   for (int i = 0; i < 64; i++) {
      float test_val = float(i) * M_PI / 64.0f;
      float sw_result = cos(test_val); //float std::cos(float)
      float hw_result = hw_cos_top(test_val);
      if (sw_result != hw_result) {
         mismatches++;
         cout << "!!! Mismatch on iteration #" << i;
```

```
            cout << " -- Expected: " << sw_result;
            cout << "\t Got: " << hw_result;
            cout << "\t Delta: " << hw_result - sw_result << endl;
         }
      }
      return mismatches;
   }
```

Example 7b: The HLS Design Code Uses the hls_math Library

```
// Contents of hw_cos.cpp
#include <hls_math.h>

float hw_cos_top(float x)
{
   return hls::cos(x); // hls::cos for both C++ model and RTL co-sim
}
```

When this code is compiled and run as software (e.g., "Run C/C++ Project" in Vivado HLS GUI), the results returned by `hw_cos_top()` are the same values as the HLS generated RTL produces and the program tests for mismatches against the software reference model, i.e. `std::cos()`. If `<cmath>` had been included in `hw_cos.cpp` instead, there would be no mismatches when the C/C++ project was compiled and run as software, but there would be during RTL co-simulation.

Other Considerations

It is important not to assume that the Vivado HLS tool makes optimizations that seem obvious and trivial to human eyes. As is the case with most C/C++ software compilers, expressions involving floating-point literals (numeric constants) might not be optimized during HLS. Consider the following example code.

Example 8: Algebraically idEntical; Very Different HLS Implementations:

```
// 3 different results
void top(float *r0, float *r1, float *r2, float inval)
{
   *r0 = 0.1 * inval;   // double-precision multiplier & conversions
   *r1 = 0.1f * inval;  // single-precision multiplier
   *r2 = inval / 10.0f; // single-precision divider
}
```

If this function is synthesized to RTL, three very different circuits result for computing each of `r0`, `r1`, and `r2`. By the rules of C/C++, the literal value `0.1` represents a double-precision number that cannot be represented exactly, therefore a double-precision (double) multiplier core is instantiated, along with cores to convert `inval` to double and the product back to float (`*r0`'s type). When a single-precision (float) constant is desired, then an `f` must be appended to the literal value, for example, `0.1f`. Therefore, the value of `r1` above is the result of a single-precision multiplication between the (inexact) float representation of $0.1\overline{0}$ and `inval`. Finally, `r2` is produced by a single-precision division core, with `inval` as the numerator and `10.0f` as the denominator. The real value `10` is represented exactly in binary floating-point formats, therefore (depending on the value of `inval`), the calculation `r2` might be exact, whereas neither `r0` nor `r1` are likely to be exact.

Note: Because the order in which floating-point operations occur potentially impacts the result (for example, due to rounding at different times), multiple floating-point literals involved in an expression might not be folded together.

Example 9: Order of Operations Can Impact Constant Folding:

```
// very different implementations
void top(float *r0, float *r1, float inval)
{
   *r0 = 0.1f * 10.0f * inval; // *r0 = inval; constants eliminated
   *r1 = 0.1f * inval * 10.0f; // two double-precision multiplies
}
```

In this preceding example, because of the order of evaluation of the expression assigned to `r0`, the entire expression is recognized by the compiler to be identity and no hardware is generated. However, the same does not apply to `r1`; two multiplications are done.

Example 10: Avoid Floating Point Literals in Integer Expressions:

```
void top(int *r0, int *r1, int inval)
{
   *r0 = 0.5 * inval;
   *r1 = inval / 2;
}
```

For this example, HLS implements the logic to assign `r0` by converting `inval` to double-precision format in order to multiply it by `0.5` (a double-precision literal value), then convert it back to an integer. On the other hand, HLS optimizes multiplication and division by integer powers of two into left and right shift operations respectively, which are implemented in hardware as simple wire selections (with zero-padding or sign-extension as appropriate for the direction and type of the operand). Therefore, the logic created to assign `r1` is much more efficient while achieving the same arithmetic result.

Advanced Topics

Parallelism, Concurrency, and Resource Sharing

Because floating-point operations use considerable resources relative to integer/fixed point operations, the Vivado HLS tool utilizes those resources as efficiently as possible. Floating-Point Operator cores are often shared among multiple calls to the source operation, when data dependencies and constraints allow. To illustrate this concept, four float values are summed in the following example.

Example 11: Multiple Operations Use Single Core:

```
// How many adder cores?
void top (float *r, float a, float b, float c, float d)
{
   *r = a + b + c + d;
}
```

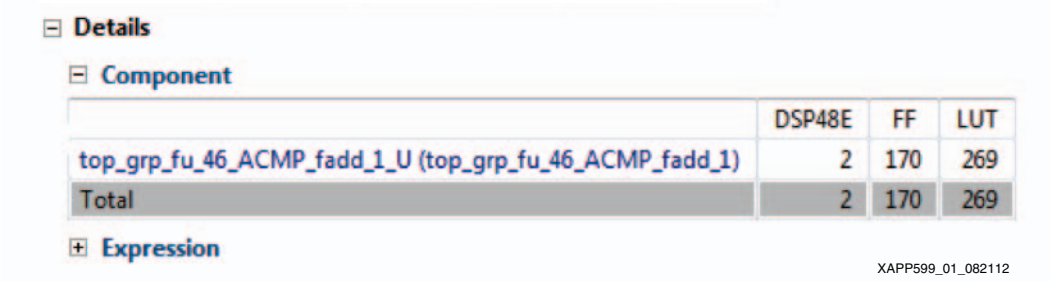

Figure 1: **Vivado HLS Report: A Single Adder Core Instantiated**

Sometimes, when data bandwidth allows, it might be desirable to do more work in a given time by doing many operations concurrently, which would otherwise be scheduled sequentially. In the following example, the values in the result array are generated by summing the elements of two source arrays in a pipelined loop. Vivado HLS maps top-level array arguments to memory

interfaces and therefore, the number of accesses per cycle are limited, e.g., two per cycle for dual-port RAM, one per cycle for FIFO, etc.

Example 12: Independent Sums:

```
// Independent sums, but I/O only allows throughput of one result per cycle
void top (float r0[32], float a[32], float b[32])
{
#pragma HLS interface ap_fifo port=a,b,r0
   for (int i = 0; i < 32; i++) {
#pragma HLS pipeline
      r0[i] = a[i] + b[i];
   }
}
```

By default, the Vivado HLS tool schedules this loop to iterate 32 times and implement a single adder core. Provided input data is available continuously and the output FIFO never gets full, the resulting RTL block requires 32 cycles, plus a few to flush the adder's pipeline. This is essentially as fast as possible, given the I/O data rate. If on the other hand the data rates are increased, then HLS techniques can be used to increase the processing rate as well. Extending the previous example, the I/O bandwidth is increased by doubling the width of the interfaces, using the Vivado HLS tool's array reshape directive. To increase the processing rate, the loop is partially unrolled by a factor of two to match the increase in bandwidth.

Example 13: Independent Sums:

```
// Independent sums, with increased I/O bandwidth -> high throughput and
area
void top (float r0[32], float a[32], float b[32])
{
#pragma HLS interface ap_fifo port=a,b,r0
#pragma HLS array_reshape cyclic factor=2 variable=a,b,r0
   for (int i = 0; i < 32; i++) {
#pragma HLS pipeline
#pragma HLS unroll factor=2
      r0[i] = a[i] + b[i];
   }
}
```

With these added directives, the Vivado HLS tool synthesizes RTL that has two adder pipelines working concurrently over half as many iterations to produce two output samples per iteration. This is possible because each calculation is completely independent and the order in which the addition operations occur cannot affect the accuracy of the results. However, when more complex calculations occur, perhaps as a result of a chain of dependent floating-point operations, the Vivado HLS tool cannot rearrange the order of those operations, which can result in less concurrency and/or sharing than expected. Furthermore, when there is feedback or recurrence in a pipelined datapath, increasing design throughput via concurrency might require some manual restructuring of the source code.

A detailed example of how the Vivado HLS tool deals with feedback/recurrence through floating-point operations is presented next followed by a discussion of how to improve performance in such cases.

The following code involves a floating-point accumulation in a pipelined region.

Example 14: Dependency through an Operation:

```
// Floating point accumulator
float top(float x[32])
{
#pragma HLS interface ap_fifo port=x
   float acc = 0;
   for (int i = 0; i < 32; i++) {
#pragma HLS pipeline
```

```
        acc += x[i];
    }
    return acc;
}
```

Because this form of accumulation results in a recurrence and the latency for floating-point addition is generally greater than one cycle, this pipeline cannot achieve a throughput of one accumulation per cycle.

For example, if the floating-point adder has a latency of four cycles the pipeline initiation interval is also four cycles (this can be inferred from the Vivado HLS tool's synthesis report shown in see Figure 2), due to the dependency requiring that each accumulation complete before another can start. Therefore, the best throughput that can be achieved is one accumulation every four cycles. The accumulation loop iterates 32 times, taking four cycles per trip, leading to a total of 128 cycles plus a few to flush the pipeline.

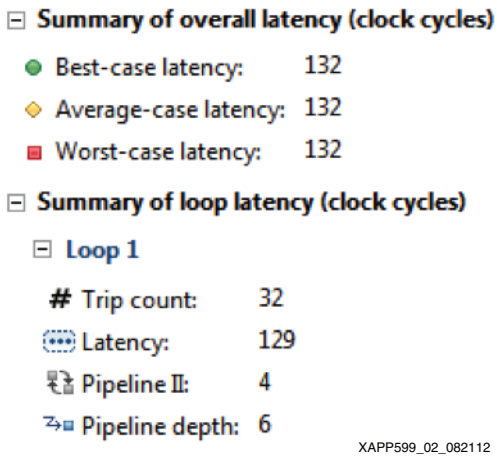

Figure 2: **Vivado HLS Report: Pipeline II = 4 Indicates a Bottleneck**

A higher performance alternative might be to interleave four partial accumulations onto the same adder core, each completing every four cycles, thereby reducing the time to accomplish the 32 addition operations. However, the Vivado HLS tool cannot infer such an optimization from the code provided above because it requires a change to the order of operations for the accumulation; if each partial accumulation takes every fourth element of `x[]` as input, the order of the individual sums changes, which could lead to a different result.

This limitation can be worked around by minor modifications to the source code to make more explicit the designer's intent. The following example code introduces an array, `acc_part[4]`, to store the partial sums, which is subsequently summed, and the main accumulation loop is partially unrolled.

Example 15: Explicit Reordering of Operations for Better Performance:

```
// Floating point accumulator
float top(float x[32])
{
#pragma HLS interface ap_fifo port=x
   float acc_part[4] = {0.0f, 0.0f, 0.0f, 0.0f};
   for (int i = 0; i < 32; i += 4) { // Manually unroll by 4
      for (int j = 0; j < 4; j++) { // Partial accumulations
#pragma HLS pipeline
         acc_part[j] += x[i + j];
   }
   for (int i = 1; i < 4; i++) { // Final accumulation
#pragma HLS unroll
      acc_part[0] += acc_part[i];
```

```
        }
        return acc_part[0];
    }
```

With this code structure, the Vivado HLS tool recognizes that it can schedule the four partial accumulations onto a single adder core on alternating cycles, which is a more efficient use of resources (see Figure 3). The subsequent final accumulation might also use the same adder core, depending on other factors. Now the main accumulation loop is done in eight iterations (32/4), each taking four cycles to produce the four partial accumulations. The same amount of work is completed in much less time with a small increase in FPGA resources. The final accumulation loop, while using the same adder core, adds extra cycles, but the number is fixed and small relative to the savings from unrolling the main accumulation loop, especially when the data set is large. This final accumulation step could be further optimized, but with diminishing returns relative to performance versus area.

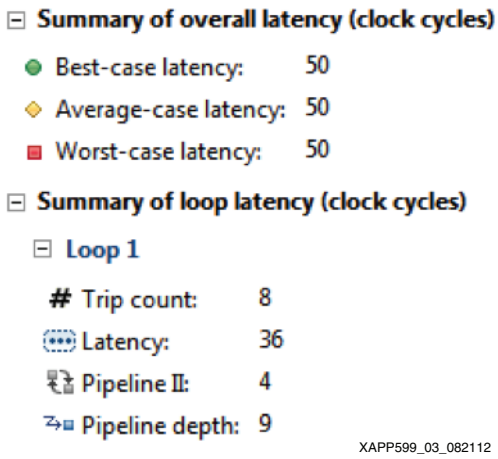

Figure 3: **Vivado HLS Report: Pipeline II = 4 Still, But Trip Count Reduced**

When greater I/O bandwidth is available, larger unrolling factors can be specified in order to bring more arithmetic cores to bear. If in the previous example, two `x[]` elements were available per clock cycle, the unrolling factor could be increased to 8, in which case two adder cores would be implemented and eight partial accumulations done per cycle. The exact operator latency can be impacted by the target device selection and user timing constraints. Generally, it is necessary to run HLS and do some performance analysis of a simpler base case (e.g., Example 14) to determine the optimal amount of unrolling.

Controlling Implementation Resources

The Xilinx LogiCORE IP Floating-Point Operator cores allow control over the DSP48 utilization of some of the supported operations. For example, the multiplier core has four variants that trade logic (LUT) resources for DSP48 usage (see PG060, *LogiCORE IP Floating-Point Operator v6.1 Product Guide* [Ref 2] for details). Normally, the Vivado HLS tool automatically determines which type of core to use based on performance constraints. The Vivado HLS tool RESOURCE directive can be used to override automatic selection and specify which type of Floating-Point Operator cores to use for a given instance of an operation. For example, for the code presented in Example 14, the adder would generally be implemented using the "Full usage" core, using two DSP48E1 resources on a Kintex™-7 FPGA, as shown in the “Component” section of the synthesis report (see Figure 4).

Details

Component

	DSP48E	FF	LUT
top_grp_fu_192_ACMP_fadd_1_U (top_grp_fu_192_ACMP_fadd_1)	2	227	403
Total	2	227	403

XAPP599_04_082112

Figure 4: **Vivado HLS Report: Default Adder Core Uses Two DSP48E Resources**

In the following example code, the addition operation is forced to be mapped onto a FAddSub_nodsp core, as shown in Figure 5.

Example 16: Using the RESOURCE Directive to Specify Floating-Point Operator Cores Variant:

```
// Floating point accumulator
float top(float x[32])
{
#pragma HLS interface ap_fifo port=x
   float acc = 0;
   for (int i = 0; i < 32; i++) {
#pragma HLS pipeline
#pragma HLS resource variable=acc core=FAddSub_nodsp
      acc += x[i];
   }
   return acc;
}
```

Details

Component

	DSP48E	FF	LUT
top_grp_fu_62_ACMP_fadd_1_U (top_grp_fu_62_ACMP_fadd_1)	0	168	630
Total	0	168	630

XAPP599_05_082112

Figure 5: **Vivado HLS Report: Now the Adder Uses No DSP48E Resources**

See UG902, *Vivado Design Suite User Guide: High-Level Synthesis v2012.2* [Ref 6] for full details on the usage of the RESOURCE directive and list of available cores.

Validating the Results of Floating-Point Calculations

There are many reasons to expect bit-level (or greater) mismatches between floating-point results of the same calculation done by different means. For example, different approximation algorithms, re-ordering of operations leading to rounding differences, and handling of subnormal (which are flushed to zero by the Floating-Point Operator cores).

In general, the result of comparing two floating-point values, especially for equality, can be misleading. Two values being compared might differ by just one "unit in the last place" (ULP; the least significant bit in their binary format), which can represent an extremely small relative error, yet the '==' operator returns false. For example, using the single precision float format, if both operands are non-zero (and non-subnormal) values, a difference of one ULP represents a relative error on the order of 0.00001%. For this reason, it is good practice to avoid using the '==' and '!=' operators to compare floating-point numbers. Instead, it is recommended to check that the values are "close enough", perhaps by introducing an acceptable error threshold.

For most cases, setting an acceptable ULP or relative error level works well and is preferred over absolute error (or "epsilon") thresholds. However, when one of the values being compared is exactly zero (0.0), this method breaks down. If one of the values being compared can take on

a zero value (or has a constant), then an absolute error threshold should be used instead. The following example code presents a method that can be used to compare two floating-point numbers for approximate equality, allowing the user to set both ULP and absolute error limits. This function is intended for use in the C/C++ "test bench" code for validating modifications to HLS source code and verifying during the Vivado HLS tool's RTL co-simulation. Similar techniques can be used in code destined for HLS implementation as well.

Example 17: C Code to Test Floating Point Values for Approximate Equivalence:

```
// Create a union based type for easy access to binary representation
typedef union {
   float fval;
   unsigned int rawbits;
} float_union_t;

bool approx_eqf(float x, float y, int ulp_err_lim, float abs_err_lim)
{
   float_union_t lx, ly;
   lx.fval = x;
   ly.fval = y;
   // ULP based comparison is likely to be meaningless when x or y
   // is exactly zero or their signs differ, so test against an
   // absolute error threshold this test also handles (-0.0 == +0.0),
   // which should return true.
   // N.B. that the abs_err_lim must be chosen wisely, based on
   // knowledge of calculations/algorithms that lead up to the
   // comparison. There is no substitute for proper error analysis
   // when accuracy of results matter.
   if (((x == 0.0f) ^ (y == 0.0f)) || (__signbit(x) != __signbit(y))) {
#ifndef NDEBUG
      if (x != y) { // (-0.0 == +0.0) so warning not printed for that case
        printf("\nWARNING: Comparing floating point value against zero ");
         printf("or values w/ differing signs.  ");
         printf("Absolute error limit has been used.\n");
      }
#endif
      return fabs(x - y) <= fabs(abs_err_lim);
   }
   // Do ULP base comparison for all other cases
   return abs((int)lx.rawbits - (int)ly.rawbits) <= ulp_err_lim;
}
```

There is no single answer as to at what level to set the ULP and absolute error thresholds should be set, as it design dependent. The level at which the ULP and absolute error threshold settings should be set are design dependent. A complex algorithm can have the potential to accumulate many ULPs of inaccuracy in its output, relative to a reference result. Other relational operators can also be prone to misleading results. For example, when testing whether one value is less (or greater) than another and the difference is only a few ULPs, is it reasonable to resolve the comparison in this case? The function presented above could be used in conjunction with a less-than/greater-than comparison to flag results that might be ambiguous.

Summary

The ability to easily implement floating-point arithmetic hardware (RTL code) from C/C++ source code on Xilinx FPGAs is a powerful feature of the Vivado HLS tool. However, use of floating-point math is not as straightforward as it might seem, whether viewing from a software, hardware, or mixed perspective. The imprecise nature of the *IEEE-754 Standard* binary floating-point formats can make it difficult to interpret computed results. Additionally, care must be taken, both at the C/C++ source code level and when applying HLS optimization directives, to get the intended QoR, whether with respect to FPGA resource utilization or design performance.

References

The following are references for this doc:

1. *ANSI/IEEE, IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-2008. IEEE-754
2. PG060, *LogiCORE IP Floating-Point Operator v6.1 Product Guide*
3. http://randomascii.wordpress.com/category/floating-point/
4. http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html
5. http://www.lahey.com/float.htm
6. UG902, *Vivado Design Suite User Guide: High-Level Synthesis v2012.2*
7. UG871, *Vivado Design Suite Tutorial: High-Level Synthesis v2012.2*
8. Vivado High-Level Synthesis product page
9. Vivado Video Tutorials
10. Floating-Point Operator product page

Revision History

The following table shows the revision history for this document.

Date	Version	Description of Revisions
09/20/12	1.0	Initial Xilinx release.

Notice of Disclaimer

Automotive Applications Disclaimer