WP491 (v1.0) March 30, 2017

Reduce Power and Cost by Converting from Floating Point to Fixed Point

By: *Ambrose Finnerty and Hervé Ratigner*

Xilinx devices and tools support a broad range of data types from binary to double precision. The scalable precision of the UltraScale™ architecture provides great flexibility for optimizing power and resource usage while meeting design performance targets.

ABSTRACT

Within market segments—including Data Center, Aerospace and Defense, 5G Wireless, and Automotive—customers have to meet challenging thermal, power, and cost requirements in applications such as ADAS, radar, and deep learning.

One extremely effective way to achieve these targets is by implementing the signal processing chain in fixed point. The variable precision support native to Xilinx FPGAs and SoCs allows customers to easily adjust to the ever evolving industry trends towards lower precision solutions.

Xilinx provides a tool flow, incorporating Vivado® High Level Synthesis (HLS), which allows customers to easily evaluate lower precision implementations of their C/C++ designs, including fixed point.

Introduction: Xilinx Data Type Support

Xilinx All Programmable devices and tools support a wide range of data types, ranging from binary to double-precision floating point. Designs implemented in fixed point will always be more efficient than their equivalent in floating point because fixed-point implementations consume fewer resources and less power. Up to 50% power and area savings are not uncommon when a design is migrated to fixed point.

The benefits of using fixed-point data types over floating point include:

- Reduction in logic resources
- Lower power
- Reduced BOM cost
- Shorter latency

Xilinx's complete range of devices can support those customers requiring the dynamic range provided by floating-point data types—offering up to 7.3TFLOPs of single-precision floating-point DSP performance.

Floating-point support is provided by Xilinx's industry-leading tool suite. Vivado® High Level Synthesis (HLS)[Ref 1] and System Generator for DSP[Ref 2] both natively support varying floating-point precisions, including half precision (FP16), single precision (FP32), and double precision (FP64); the added flexibility of custom precision is also available in System Generator. These tools have native support for variable fixed point data types as well.

Table 1: **Xilinx Tool Support Floating and Fixed Point Data Types**

Xilinx Tools	FP16	FP32	FP64	Custom FP	Fixed-Point
Vivado HLS	Y	Y	Y	N	Y
System Generator for DSP	Y(1)	Y	Y	Y	Y
Floating Point Operator IP	Y	Y	Y	Y	Y(2)

Notes:

1. System Generator for DSP has no native FP16 support but custom support allows for FP16.
2. Floating-point operator core supports conversion → fixed-to-float, float-to-fixed and varying precisions of float-to-float.

The variable precision data-type support provided with Xilinx devices and tools provides a simple and flexible solution for customers to adjust to changes in industry trends, e.g., image classification only requires INT8 or lower fixed-point compute to keep acceptable inference accuracy[Ref 3] [Ref 4].

Other devices used in computationally intensive workloads, such as GPUs, have traditionally been architected to support only single-precision floating point efficiently. These vendors are now working to redesign products to react to the changing trends. Xilinx's scalable architecture allows its customers to scale the precision of the signal processing chain to easily meet changing industry needs.

It is important that customers carefully evaluate trade-offs in power, cost, productivity, and precision when choosing to implement a floating-point vs. fixed-point signal processing chain.

Xilinx's flexible DSP48E2 slice can be used by all data types for important DSP computation. The DSP slice, in conjunction with the Xilinx tool set, provides huge benefits and flexibility when implementing new fixed-point designs or when converting existing designs from floating point to fixed point for some applications where conversion is a viable option[Ref 5].

For customers designing in C/C++, Xilinx offers Vivado HLS and support for arbitrary precision fixed-point data-types, which allows customers to easily design in fixed point or convert their existing C/C++ designs to fixed point.

Benefits of Converting Floating Point to Fixed Point

With almost all designs today, minimizing power consumption is a high priority. Most applications must meet aggressive power and thermal envelopes before they can be deployed to production.

It is widely accepted that designing in floating point leads to higher power usage for the design compared to lower precisions[Ref 6][Ref 7]. This remains true for FPGAs where floating-point DSP blocks have been hardened in the FPGA, and also where customers must implement a soft solution using provided DSP resources and additional FPGA resources. Floating-point implementations require larger amounts of FPGA resources than an equivalent fixed-point solution. With this higher resource usage comes higher power consumption and ultimately increased overall cost of implementing a design.

Converting a floating-point design to fixed point can help meet these challenging specifications in the following ways:

- Reduction in FPGA resources
  - Fewer DSP48E2s, look-up tables (LUTs), and flip-flops are needed when working with fixed-point data types.
  - Smaller amounts of memory are required to store fixed-point numbers.
- Lower power consumption
  - Reduction in FPGA resource usage inherently leads to lower power consumption.
- Reduced BOM cost
  - Designers can utilize the additional available resources for extra features in their application for the same cost.
  - The resource savings enable massively increased compute capabilities within the FPGA. This increased compute power benefits many applications, e.g., Machine Learning DNNs.
  - It is possible the resource savings can reduce the size of the device needed for the design.
- Latency improvements
  - Again, reducing the resources, in particular the DSP48E2 slices, when implementing the FIR brings a latency improvement in the fixed-point design.
- Comparable performance and accuracy
  - For designs and applications that do not require the dynamic range achievable with floating point, fixed-point implementations can provide comparable results and accuracy. In some cases, the results can even be improved.

In the past, converting a design from floating point to fixed point was difficult because of limited tool support. For C/C++ developers targeting Xilinx All Programmable devices, Vivado HLS can be used to reduce the challenges involved in achieving this conversion.

The benefits of performing this conversion are so great that it merits serious consideration where applicable—in particular, designs where the dynamic range and precision available with floating point are not required and the small expected loss of precision will not lead to inefficiencies in the deployed application.

Example: Convert a Floating Point FIR Filter to Fixed Point

A simple FIR filter design[Ref 8] in Vivado HLS can be used to show how converting a floating-point FIR design to a fixed-point variant leads to reduced resources and power with comparable accuracy in results.

Single Precision Floating Point FIR

In the C++ FIR function code, the top-level function instantiates the class CFir found in the FIR.h header file.

```
#include "FIR.h"


// Top-level function with class instantiated

fp_acc_t fp_FIR(fp_data_t x) {

    #pragma HLS PIPELINE

    static CFir<fp_coef_t, fp_data_t, fp_acc_t> fir1;

    return fir1(x);

}
```

The CFir class is the main FIR algorithm, which is defined in the header file FIR.h.

```
// FIR main algorithm
template<class coef_T, class data_T, class acc_T>
acc_T CFir<coef_T, data_T, acc_T>::operator()(data_T x) {
//caller uses #pragma HLS PIPELINE which makes this function pipelined as 
needed.
#pragma HLS ARRAY_PARTITION variable=c complete dim=1
#pragma HLS ARRAY_PARTITION variable=shift_reg complete dim=1
   int i;
   acc_T acc = 0;
   data_T m;

   loop: for (i = N-1; i >= 0; i--) {
      if (i == 0) {
         m = x;
         shift_reg[0] = x;
       } else {
         m = shift_reg[i-1];
         if (i != (N-1)) {
            shift_reg[i] = shift_reg[i - 1];
         }
      }
      acc += m * c[i];
   }
   return acc;
}
```

This function includes important ARRAY_PARTITION pragmas to ensure an II=1 (iteration interval of 1)[Ref 9] for all implementations of the design. The PIPELINE pragma is applied to the top-level function call as well.

These pragmas, along with the implementation of the products in parallel, followed by an adder tree for performing the accumulation, ensure minimum latency through the complete FIR function regardless of data type, while maintaining an II = 1.

In the fp_FIR function, fp_coef_t, fp_data_t and fp_acc_t are all defined as float, i.e., single-precision floating-point data type native to C++.

```
// float

typedef float fp_coef_t;
typedef float fp_data_t;
typedef float fp_acc_t;
```

The filter coefficients are loaded via an include command within the header file.

```
template<class coef_T, class data_T, class acc_T>
const coef_T CFir<coef_T, data_T, acc_T>::c[] = {
  #include "FIR_fp.inc"
};
```

The coefficients create a symmetrical FIR filter, but for this example, the pre-adder in the DSP48E2 slice is not used. If the pre-adder were used, further efficiencies will be achieved.

The following results are achieved for an 85-tap FIR filter, running the C synthesis and implementation in Vivado HLS, targeting a 400MHz clock (2.5ns clock period) on an XCVU9P-2FLGB2104 device. See Table 2.

Table 2: **Post Implementation Results for Single-Precision Floating-Point FIR**

Single-Precision Floating Point (FP32)	
F_{MAX}	500MHz
Latency (Clock Cycles)	91
Iteration Interval (II)	1
DSP48E2	423
LUTs	23,101

In this example, 423 DSP48E2s and ~23K LUTs are required to implement the single-precision floating-point FIR. The implementation results in a latency of 91 clock cycles and F_{MAX} at 500MHz (substantially above the 400MHz target).

Convert to Fixed Point FIR Filter

For the greatest DSP efficiency, the conversion to fixed point must consider the DSP slice's bus width dimensions, i.e., 27x18-bit multiplier and 48-bit accumulator. Further reducing these bus widths to the very minimum allowable within the design gives the biggest return in terms of resource and power savings.

For this FIR filter example, the following fixed point data-types are defined to match the bus sizes in the DSP48E2 slice, i.e., 18 bits coefficient with 1 integer and 17 fractional bits, 27 bits of data with 15 integer and 12 bits fractional and finally a 48-bit accumulator with 19 integer and 29 fractional bits.

```
// fixed points
#include <ap_fixed.h>


typedef ap_fixed<18,1> fx_coef_t;
typedef ap_fixed<27,15> fx_data_t;
typedef ap_fixed<48,19> fx_acc_t;
```

To use ap_fixed data types native to Vivado HLS, the `ap_fixed.h` header file must be included to define the arbitrary fixed-point data types [Ref 9].

Again, targeting a 400MHz clock (2.5ns clock period) and the XCVU9P-2FLGB2104 device, the C synthesis and implementation for the fixed-point FIR design produced the results shown in Table 3.

Table 3: **Comparing Post Implementation Results for Both Designs**

	Single-Precision Floating Point	Fixed Point	Fixed-Point Advantages
F_{MAX} (Post-Implementation)	500MHz	580MHz	16% faster
Latency	91	12	~7.5X lower
Iteration Interval (II)	1	1	–
DSP48E2	423	85	5X greater DSP efficiency
LUTs	23,106	1,973	11X greater logic efficiency

As proven by the results, paying particular attention to the latency and FPGA resource utilization provides measurable improvements.

In the UltraScale architecture, where necessary, larger bus widths can still be supported by cascading multiple DSP48E2 slices. A fixed-point design with cascading DSP48E2 slices still yields substantial improvements in resources and power compared to a floating-point implementation.

Comparing Filter Accuracy

Using the Vivado HLS block (from the Xilinx blockset) in System Generator for DSP, the two implementations of the FIR filter can be compared within the MATLAB®/Simulink® environment. See Figure 1.

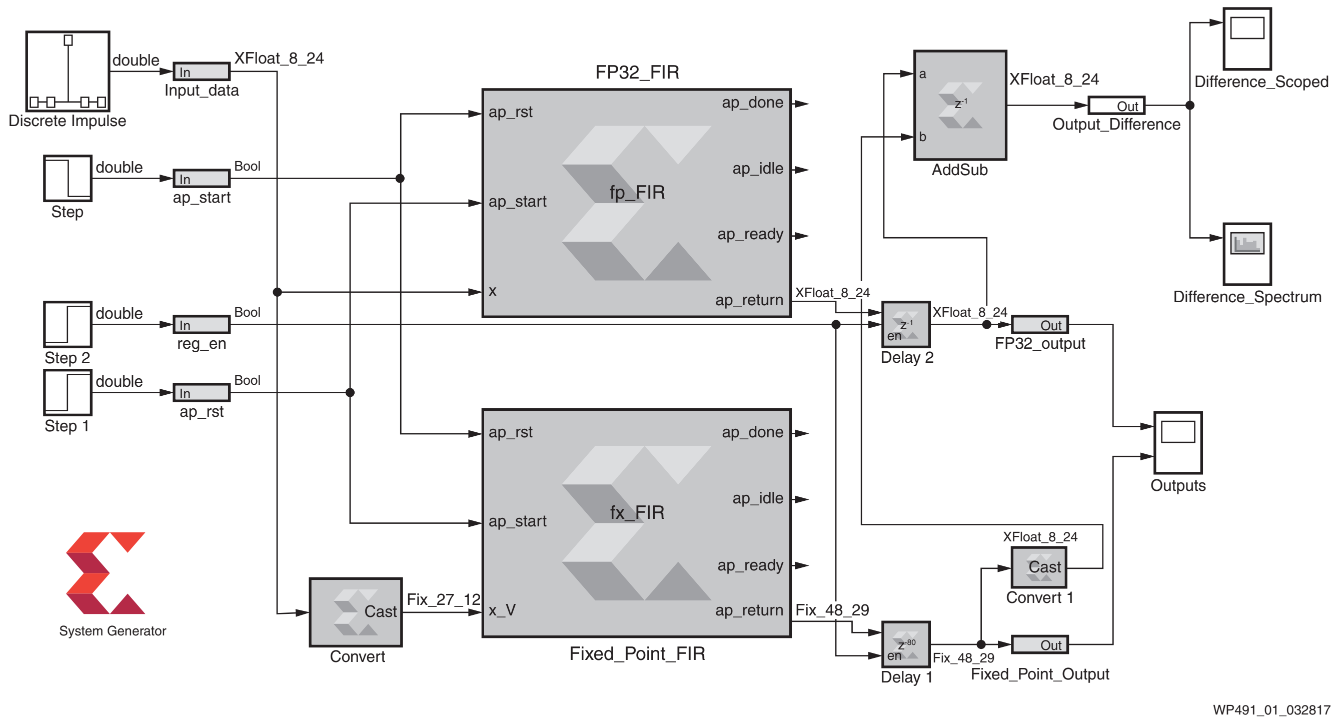

Figure 1: **System Generator for DSP Model - Using both HLS Solutions for Analysis**

The System Generator model consists of two Vivado HLS blocks, which are configured to include the single-precision floating-point (FP32) and fixed-point FIR solutions from Vivado HLS. Both blocks have the same input applied, a discrete impulse signal, and then the outputs from each FIR are compared on a Simulink scope. See Figure 2.

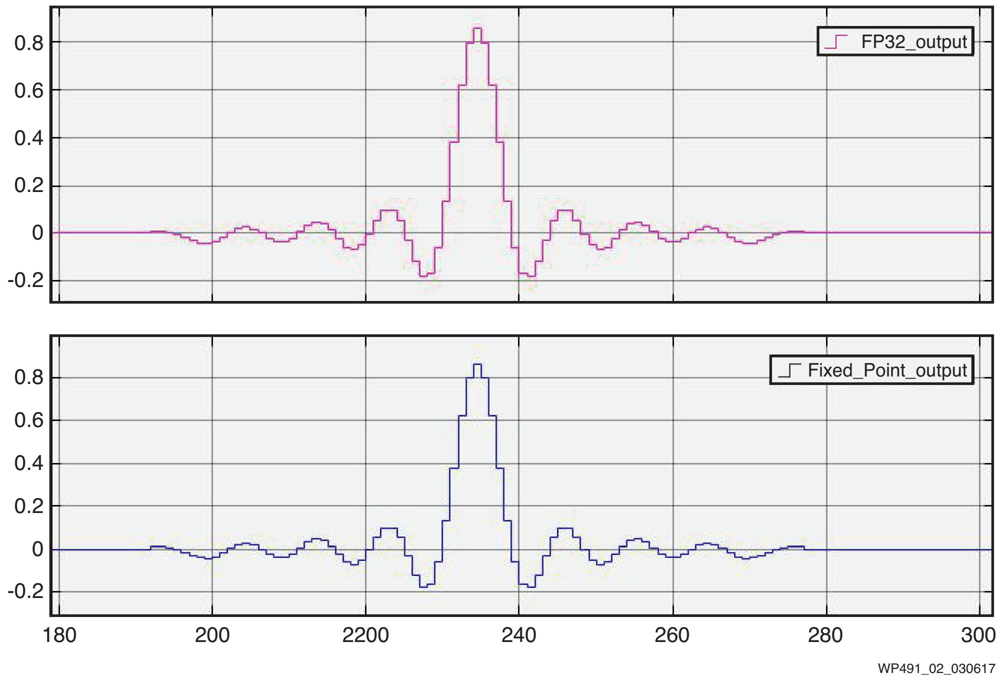

Figure 2: **Outputs from Both HLS Designs in System Generator**

To easily compare the outputs, it was necessary to delay the fixed-point result to align by the difference in latency between the two solutions.

As expected, both FIR filters produce almost identical results with minimal difference.

To further analyze the signals, both outputs were subtracted from each other. The resulting signal showed a very small loss of precision, in the range of -100dBm to -160dBm, on the spectrum analysis plot shown in Figure 3.

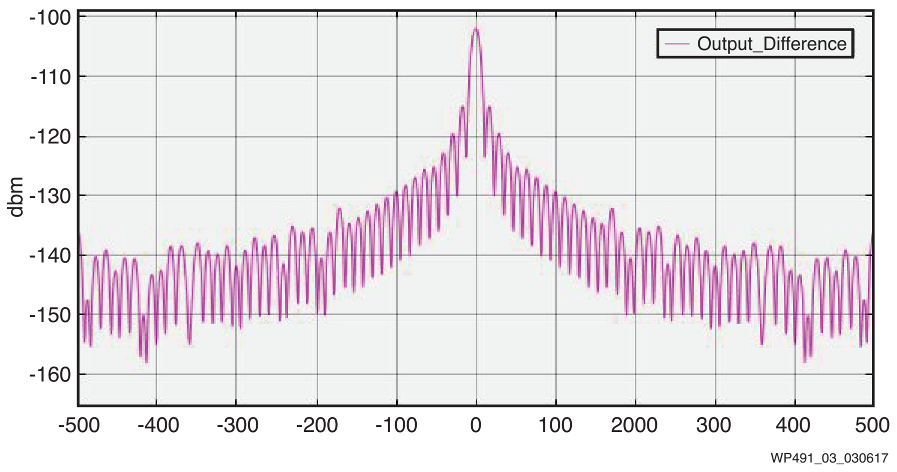

Figure 3: **dB Plot of the Difference between Both Outputs**

Highlighting the Key Benefits

When comparing the results from the original single-precision floating-point FIR filter to the converted fixed-point FIR filter, the fixed-point design shows both resource reduction and latency improvements, while maintaining or improving design F_{MAX}. See Figure 4.

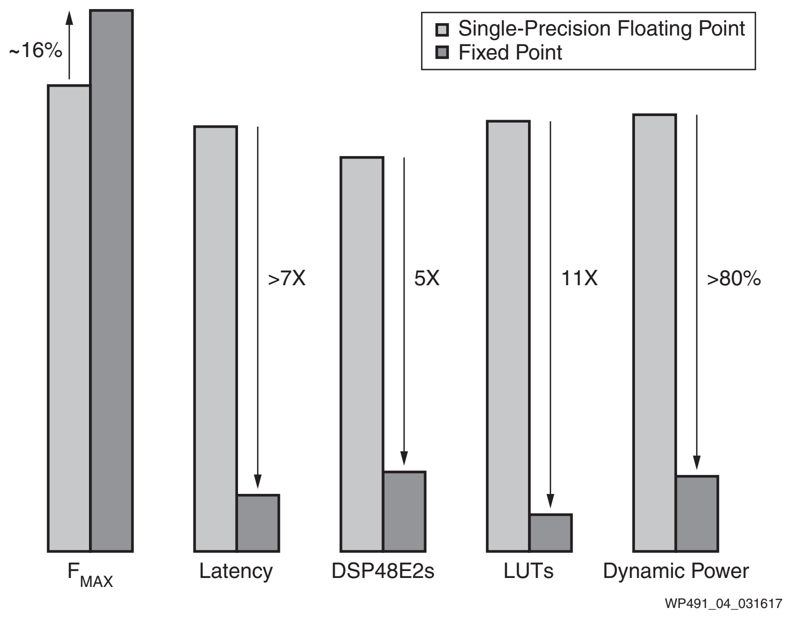

Figure 4: **Fixed Point - Similar Performance with Reduced Latency, Resources, and Power**

Large Reductions of FPGA Resource Utilization

The fixed-point FIR in this example is more than 5X smaller than the original floating-point FIR.

The bus widths were chosen for optimal mapping to the DSP48E2 slice in hardware. This allows each multiply to be completed in one DSP48E2 slice and in parallel for each of the 85 coefficients. This reduces the DSP48E2 slice usage to 20% of the floating-point solution.

Huge savings (~90%) are also made with respect to the LUTs in the FPGA fabric, because in the fixed-point implementation, no additional LUTs are required to build the floating-point operations.

If a design had 10 such FIR filters, the estimated power will scale with the design. Table 4 shows the XCVU9P FPGA resource utilization for both the single-precision and fixed-point implementations of a 10 FIR filter design. There is a marked difference when comparing the single-precision floating-point resources to the fixed-point implementation in this design.

Table 4: **Resources Utilized for 10 FIR Filters with Both Data-Type Solutions**

	DSP48E2		LUT	
	Resource Count	Device Utilization	Resource Count	Device Utilization
Single-Precision Floating Point	4,230	62%	231,060	20%
Fixed Point	850	12%	19,730	2%

The substantial resource savings provide multiple benefits with far-reaching consequences for the designer with respect to design feature set, design power, design performance, and design cost.

Substantial Power Savings Achieved

The substantial resource savings lead to a related reduction in the amount of power consumed.

Comparing power estimates for both implementations of the single FIR filter example described in this white paper, the fixed-point FIR uses 1.4W less power. In both cases, the static power for the device is just over 3W and the total power for the individual single-precision floating-point FIR design is 4.7W. This shows >80% savings in dynamic power for this design with the fixed-point FIR using 3.3W.

Looking at the 10 FIR filter design, the power estimates for both implementations can be checked using Xilinx Power Estimator (XPE) and the resources calculated in Table 4. The savings are compared in Figure 5.

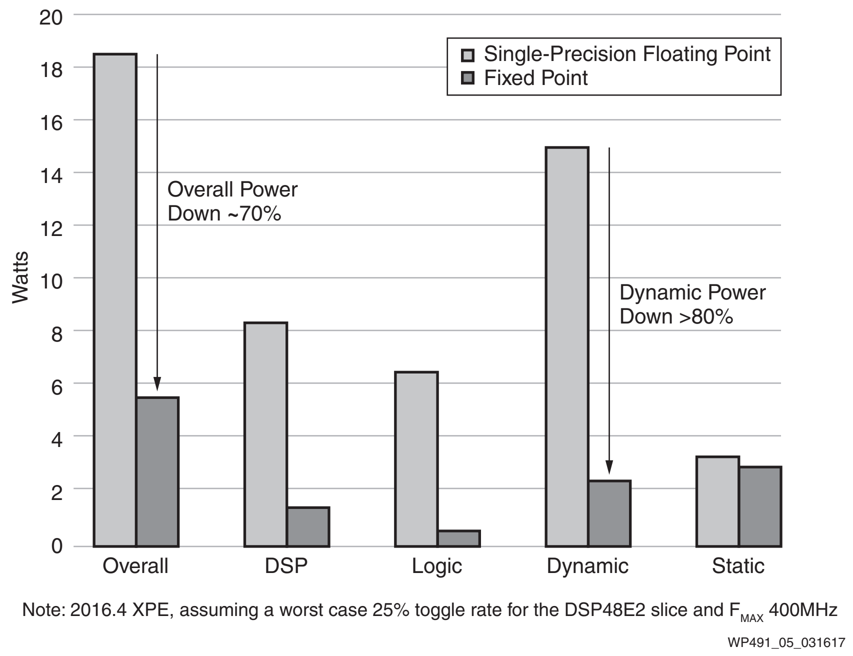

Figure 5: **10 FIR Filter Example: Massive Power Savings with Fixed Point**

In this 10 FIR filter example, a 70% overall power savings can be achieved when the design is converted to use fixed-point data types. For designs with large amounts of floating-point signal processing that use large amounts of FPGA resources, huge power savings can be achieved by converting some or all of the floating-point signal processing chain to fixed point.

BOM Cost Reductions

Converting to a fixed-point implementation of a floating-point design greatly reduces the FPGA resource utilization. This large reduction in FPGA resources can enable potential bill of material (BOM) cost reductions. These can be realized in three ways.

1. The application feature set can be augmented by utilizing the now newly available FPGA resources.
2. The overall compute capabilities of the FPGA can be substantially increased due to the massive FPGA resource reduction and any improvement in F_{MAX} through the datapath.
3. The design can potentially move to a smaller Xilinx FPGA because fewer FPGA resources are now required.

Comparable Accuracy & Precision

By comparing the outputs of both implementations of the single FIR filter design, the fixed-point implementation delivers comparable accuracy in the filter, with just -100dBm to -160dBm loss in precision while benefiting from the power and cost savings achieved.

However, with the fixed-point implementation, the same dynamic range is not possible, which can lead to a perceived loss of precision within a design. For many designs, this is not a problem because only a minimum standard of precision is required. Similar to the single FIR example, these types of designs are ideal for converting to fixed point.

For designs with values that require greater precision, intermediate values in the signal processing chain can sometimes be converted from floating point to fixed point. This approach enables the designer to convert certain portions of the design to fixed point—but not all. Ultimately, this enables the designer to maintain the dynamic range where required and help assure precision is maintained for the datapath while leveraging some of the benefits of a fixed-point implementation.

Latency Improvements

For the single FIR design example, the latency improves through the filter to 12 clock cycles for the fixed-point implementation versus 91 clock cycles for the floating-point design. As the resources reduce, in particular the DSP48E2 slices reduce, an improvement in latency can be expected.

Along with the latency improvement, an F_{MAX} improvement might be achieved as with the single FIR example, where after implementation, a 16% improvement in FMAX was realized.

Conclusion

Xilinx All Programmable devices and tools support a variety of data types, including multiple precisions of floating point and fixed point. Designs in floating point use more resources and higher power than the same design in fixed point, regardless of whether one is targeting an FPGA or other architectures, e.g., GPU.

Industry trends show a clear shift away from using floating-point data types for some applications, e.g., deep learning inference workloads are using INT8 or lower precision where possible.

With thermal and power envelopes becoming more and more difficult to meet in today's challenging design environments, designers must evaluate all possible avenues available to them to reduce power consumption. One such option is to convert floating-point designs to fixed point.

For those working in C/C++, Xilinx tools like Vivado HLS can help ease the conversion process.

A designer must fully investigate the trade-offs associated with converting to fixed point data types and fully understand the huge benefits that this effort provides.

Staying in floating point can offer an easier path to market, but it is expensive. Investing the time and effort to convert to fixed point gives massive benefits in terms of lowering resources, cost, and power, with minimal loss of performance.

For more information, go to the DSP page on Xilinx.com:
https://www.xilinx.com/products/technology/dsp.html

References

1. Xilinx Landing Page *Vivado High-Level Synthesis*
2. Xilinx Landing Page *System Generator for DSP*
3. Gysel et al, *Hardware-oriented Approximation of Convolutional Neural Networks*, ICLR 2016
4. Han et al, *Deep Compression: Compressing Deep Neural Networks With Pruning, Trained Quantization And Huffman Coding*, ICLR 2016
5. *Deep Learning with Int8 Optimization on Xilinx Devices (WP486)*
6. Gupta et al, *Deep Learning with Limited Numerical Precision*
7. Ying Fai Tong et al, *Reducing Power by Optimizing the Necessary Precision/Range of Floating-Point Arithmetic*
8. Github, location for the *design files*
9. Xilinx Software Manual, *Vivado Design Suite User Guide, High-Level Synthesis*

Revision History

The following table shows the revision history for this document:

Date	Version	Description of Revisions
03/30/2017	1.0	Initial Xilinx release.

Disclaimer

Automotive Applications Disclaimer